CICS JVM Server Application Isolation

Masterarbeit

Lehrstuhl für Technische Informatik
Wilhelm-Schickard-Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät
Universität Tübingen

Dipl.-Ing. (FH) Robert Harbach

Betreuer:

Prof. Dr.-Ing. Wilhelm G. Spruth

Tag der Anmeldung: 1. Oktober 2011
Tag der Abgabe: 30. März 2012

Executive Summary

The Java Virtual Machine (JVM) is able to host multiple simultaneously executing transaction programs. Since a JVM is not capable to provide full application isolation however, the common practice found in most Java application servers implies to run one JVM per program. Due to the fact, that this implies an overhead of computational resources, several, most often proprietary, solutions for application isolation in JVMs have been proposed in the past.

This paper outlines isolation properties of a new Java Virtual Machine (JVM) type, called a JVM Server, which implements the OSGi framework, and is integrated into the System z architecture Customer Control Information System (CICS). It is shown that this approach resolves two application isolation issues: The CICS environment enables transaction safe execution of tasks. In addition, inclusion of the OSGi Framework provides namespace isolation based on class loading.

However, several issues remain unsolved. They arise from the use of Java system classes, OSGi specific functions, resource exhaustion, and the use of the Java Native Interface (JNI).

This paper proposes solutions to these isolation exposures. Some employ functions of the OSGi security layer. Others utilise existing CICS services for program control which would have to be implemented differently on other Java Transaction Servers.

Erklärung

Hiermit bestätige ich, dass ich diese Masterarbeit selbstständig und nur mit Hilfe der angegebenen Quellen und Hilfsmittel verfasst habe. Die vorliegende Arbeit ist in gleicher oder ähnlicher Form noch in keinem anderen Prüfungsverfahren eingereicht worden.

Tübingen, den 30. März 2012

Robert Harbach

Acknowledgments

First of all I would like to thank my mother Galina and my sister Anna for supporting me incredibly during my entire study period and my father Alexander for doing the same as long as he was able to.

At second I would like to thank Professor Spruth for supervising this project and moreover, for his constant encouragement. In addition, I would like to thank Dr. Walter Lange for helping me enormously during my Master studies as well as Prof. Roland Kiefer, Prof. Dr. Johannes Maucher and Prof. Walter Kriha from the Stuttgart Media University (HdM) for being an inspiration not only from a technical point of view right from the beginning of my studies.

My gratitude goes also to the specialists from IBM; especially to Uwe Denneler, Ulrich Seelbach, Philipp Breitbach, Wilfried Van Hecke, Tobias Leicher and Isabel Arnold because of their outstanding help regarding all technical topics.

Last but not least I would like to thank Daniel Glück and Robert Hasselaar for reviewing this document as well as all remaining classmates I had the chance to work and spend time with during my studies.

Contents

Chapter 1

Introduction

1.1 Overview

Enterprises and governmental organizations nowadays implement complex Information Technology (IT) infrastructures. In large enterprises these infrastructures often contain thousands of servers such as firewall, web, application and business intelligence servers. The servers are responsible for miscellaneous tasks as securing networks and applications, routing and switching as well as hosting (web-)applications. Despite the variety of tasks, however, in almost all large enterprises the data required for the servers is stored and managed centrally by one or more Mainframe Computers, that are usually based on the z/OS operating system. Due to the crucial dependence of enterprises on their data inventory and resulting security considerations, these Mainframes are usually placed in different locations. Moreover, applications that access the enterprise data inventory in most cases include transactional characteristics. In general, an application represents a transaction if it follows ACID (Atomicity, Consistency, Isolation, Durability) properties [GR93, ch. 1].

In many cases transactions are "enterprise critical" applications. We define these as applications, whose incorrect execution leads to intolerable consequences. An example is the faulty rounding of the last decimal in a floating number of a financial transaction. In that particular case, the impact of the error needs to be identified and corrected, which might require manual interaction for days. Therefore, in most cases the transactions are controlled by a middleware, referred to as a transaction monitor or transaction server. The most widespread transaction servers used in enterprise IT infrastructures are Tuxedo (BEA, today Oracle), CICS (IBM), MTS (Microsoft) and SAP R/3. Due to the growing popularity of the Java programming language also several Java specific transaction servers have been introduced within the last 12 years. Some examples are Weblogic (BEA), WebSphere (IBM) and NetWeaver (SAP).

1.2 Problem Definition

A transactional application usually consists of two parts: the business logic and the presentation logic. While the business logic carries out the actual func-

tionality of an application, the presentation logic is responsible for displaying particular results in a somewhat user-friendly manner on the monitor. Hence, in the Model-View-Controller (MVC) design pattern, the business logic represents the Model and the presentation logic the View [HKS04, ch. 9.2.3].

Since the introduction of the Java programming language in 1995, a certain interest regarding the application of Java for enterprise purposes is present. For this aim an extension of the Java programming language, the Java Enterprise Edition (JEE), has been introduced in 1999. Although Java has been widely adopted thereupon for the implementation of the presentation logic, its application for the business logic in enterprise critical software is still not common.

This arises from several reasons. One are certainly the performance issues, that result from features as the garbage collection and from the fact that Java applications require the Java Virtual Machine (JVM), that serves as a middleware layer between applications and the operating system, for their execution. Since, however, enterprise environments usually include the necessary computational resources for compensating such issues, performance drops do not represent a criterion for exclusion for the implementation of enterprise critical software in Java. The most important reason for the moderate advance of Java for enterprise purposes represents its insufficient implementation of isolation (the I in ACID) properties, that are commonly known already since introduction of the JEE standard. Czajkowski and Daynés for instance state:

> "The existing application isolation mechanisms, such as class loaders, do not guarantee that two arbitrary applications executing in the same instance of the JVM will not interfere with one another. Such interference can occur in many places. For instance, mutable parts of classes can leak object references and can allow one application to prevent the others from invoking certain methods. The internalized strings introduce shared, easy to capture monitors. Sharing event and finalization queues and their associated handling threads can block or hinder the execution of some application. Monopolizing of computational resources, such as heap memory, by one application can starve the others." [CD01, ch. 1].

While Sandén outlines that

> "Java gives the virtuoso thread programmer considerable freedom, but it also presents many pitfalls for less experienced programmers who can create complex programs that fail in baffling ways."[San04]

1.3 Solution Approaches

In order to resolve the isolation problem in Java, several approaches have been introduced. Most of these, however, have been discard.

The general problem related to application isolation in JVMs arises from missing functionalities of the VM such as absent resource management. This implies, that the JVM is not designed for hosting multiple applications in parallel.

Therefore, the most common approaches found in application servers requires to run each application within a separate JVM. This however, leads to an overhead of computational resources needed, especially in terms of JVM creation maintenance and destruction. Hence, several approaches towards enabling a multi application environment in a single JVM have been carried out in previous researches. A significant approach for solving most isolation issues, was found to be the Application Isolation API [Jav06], that has been implemented into the Multi-Tasking Virtual Machine [CD01]. This approach, however, has never been implemented into any official Java Standard Edition (SE) or EE releases. Another interesting approach was represented by the Persistent Reusable Virtual Machine (PRVM) [IBM01] provided for CICS, that included a mechanism for resetting the state of the JVM prior to certain program executions and therefore, could have been used by multiple applications in serial. This implied a tradeoff between economic resource usage and application isolation. The PRVM however, has been replaced by a technology called the JVM Servers, that is an OSGi based JVM, where several applications, in form of bundles, are running in parallel.

This paper verifies several existing issues related to application isolation in JVMs and outlines solution approaches. Since some of these approaches are based on features provided by the middleware, that is hosting the JVM, not all approaches are applicable to environments other than the one used within scope of our investigations. Our analysis is based on a JVM version that is implemented into the *JVM Server* and is available since 2011 for the CICS version 4.2.

The Customer Control Information System (CICS), represents a transaction monitor with application hosting capabilities. The most important reasons for using CICS as an application server are, besides its popularity and its widespread usage, its various services, that among others enable transaction safe execution of tasks. Moreover, CICS serves as a middleware layer and therefore enables to use several programs regardless the programming language used for their creation together, that implies advanced inter application communication.

1.4 Document Structure

The remaining chapters of this paper are organized as follows. Chapter 2 shows an overview of the technologies used along with their properties, that are important within the scope of investigations carried out in chapter 3. In addition, an introduction of JVM integration in CICS including a architectural overview and a detailed introduction to a recently introduced technology referred to as JVM Servers are provided.
Chapter 3 includes a detailed discussion of closed and open issues related to application isolation in JVMs
Chapter 4 outlines approaches to solving open issues mostly by using built-in functionalities provided for the JVM or the CICS environment. A conclusion and a summary of solutions for open issues are provided in chapter 5.
The Appendix shows the source code used to exploit open issues and lists the content of compact disc (CD) attached to the hard copy of this document.

Chapter 2

Technologies Used

This chapter introduces the major technologies that are important within the scope of this paper. Moreover, a detailed introduction including an architectural overview of JVM Servers is provided.

2.1 Properties of the Java Programming Language

Java is an interpreted, object oriented programming language introduced in 1995, "and was [originally] designed for use in embedded consumer-electronic applications" [GJSB05, Preface]. Due to the many characteristics beneficial for developers and users, Java rapidly emerged as one of the most popular programming languages. Java's major characteristics related to the investigations discussed in chapter 3 are introduced briefly in the following. For a detailed introduction to Java refer to [GJSB05], [LY99] and [GM96].

2.1.1 The Java Runtime Environment

One of Java's most renowned characteristics is represented by the Java Virtual Machine (JVM). It defines an interface between applications, containing the program code, the Java API, containing the libraries, and the operating system. Together with all system classes, libraries and the JVM, this interface is called the Java Runtime Environment (JRE).
This section outlines the basic responsibilities as well as the basics of the storage management of the JRE that were found to be important for the investigations that were carried out in chapter 3.

Responsibilities of the JRE

Apart from acting as a middleware layer, the JRE is responsible for all tasks related to application execution. A non exclusive list of these tasks is presented below.

- **Compilation & Interpretation:** Management of the compilation of classes, including **class loading & linking** [LY99] of required classes and control of the interpretation process, since Java programs are are compiled into java bytecode and interpreted on demand (refer figure 2.1).

- **Garbage collection:** Initialization of the garbage collection thread for the removal of unused objects and references.
- **Memory:** Management of all memory parts such as the heap and the stack.
- **Security:** Management of most security features, some of which are discussed in chapter 2.1.3.
- **Thread management**
- **Exception handling**

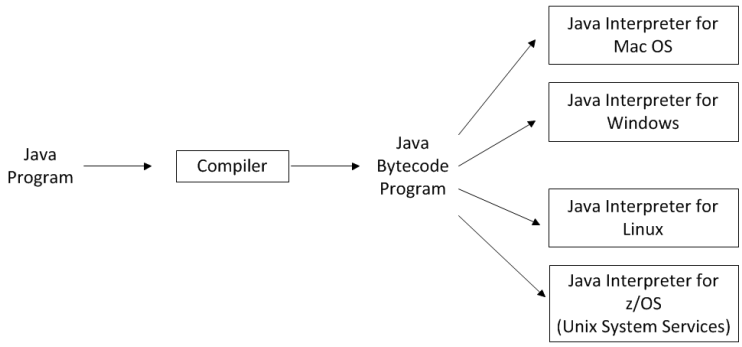


Figure 2.1: Java application execution. Modified after [Eck11, ch. 1.3].

Memory Management

As described in [LY99, ch. 3.5], the JRE memory is divided into six "runtime data areas":

- Java Virtual Machine Stack, that "holds local variables and partial results, and plays a part in method invocation and return". [LY99, ch. 3.5.2].
- The Heap, that represents the "runtime data area from which memory for all class instances and arrays is allocated". [LY99, ch. 3.5.3].
- The PC (program counter) Register, that "contains the address of the Java virtual machine instruction currently being executed". [LY99, ch. 3.5.1].
- The Runtime Constant Pool, that "contains several kinds of constants, ranging from numeric literals known at compile time to method and field references that must be resolved at run time". [LY99, ch. 3.5.5].
- The Method Area, that "stores per-class structures such as the runtime constant pool, field and method data, and the code for methods and constructors, including the special methods [...] used in class and instance initialization and interface type initialization." [LY99, ch. 3.5.4].
- The Native Method Area, that "colloquially called "C stacks," to support native methods, methods written in a language other than the Java programming language". [LY99, ch. 3.5.6].

Lindholm et. al. [LY99, ch. 3.5] also mention that while the heap and the method area are "shared among all Java virtual machine threads", the pc register and the JVM stacks are created for each thread.

Note that although the above mentioned data areas are defined by the JVM specification, actual implementations of the JVM may define different data areas. Within the scope of this paper, however, only the stacks, the heap and the method area, which are contained within IBM's JVM implementation, play an important role.

Since Java applications always remain under the control of the JRE, they imply high security and portability. Due to the on demand interpretation of code however, Java applications are known for their performance issues. Moreover, the JVM implies isolation issues if it runs multiple applications at the same time (refer chapter 3).

2.1.2 Threads in Java

Like most other programming languages Java provides the explicit creation of user threads, which is achieved by using the Thread API. Threads in general, are a concept of operating systems that enable parallel processing on multi processor systems and quasi parallel processing on single processor systems by dividing a process into several units, known as threads, that can be dispatched by a scheduler. Since the JVM includes a hardware architecture, it intends to serve as a operating system to Java applications. Therefore, the JVM includes the thread concept as well.
An important property of threads that belong to the same process is their sharing of the same address space and the same storage within the memory [Tan03, ch. 2] (refer figure 2.2). Since all threads active within a JVM belong to the same process, the JVM process, they are sharing the heap and the method area (refer chapter 2.1.1).
Note that even without the explicit creation of threads, Java applications include at least one thread initialized by the main method [OW99, ch. 1].

Silberschatz [Sil05, ch. 4.2] describes that there are three different multithreading models differentiated by their mapping of user threads to kernel threads:

- **The many-to-one model**: maps all user threads to exactly one kernel thread enabling high efficiency but also disabling multiprocessing [Sil05, ch. 4.2.1]
- **The one-to-one model**: maps each user thread to a different kernel thread, enabling real multiprocessing but also implying an administrational overhead for the creation of kernel threads resulting in performance drops [Sil05, ch. 4.2.2]
- **The many-to-many model**: can be described as a combination of the previously mentioned models. According to [Sil05, ch. 4.2.3] "the many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads" and hence, represents a tradeoff in terms of

performance and multiprocessing between the many-to-one and the one-to-one models

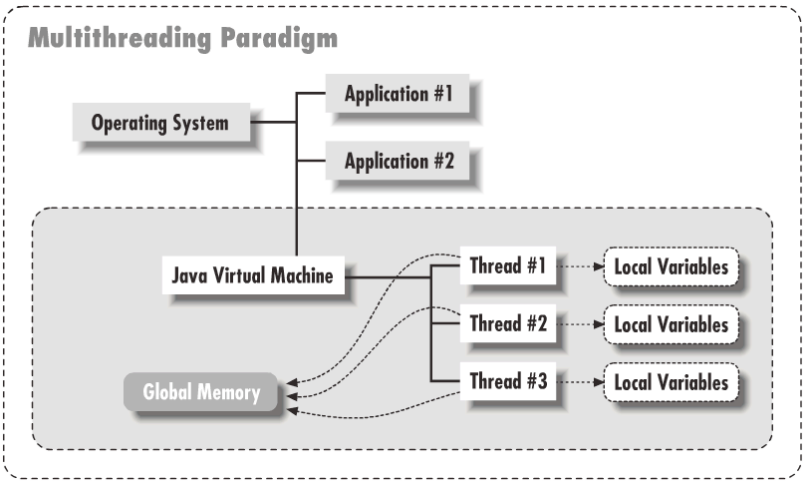


Figure 2.2: Multithreading in Java. Source: [OW99, ch. 1.2.2].

Silberschatz [Sil05, ch. 4.3] further describes that there are three libraries serving as an interface for kernel thread creation: the *pthread* library, used by UNIX operating systems, the *win32* thread library, used by Microsoft Windows operating systems and the *Java Thread API* used for explicit creation of threads within Java applications. Note that the mapping model of Java threads is based on the operating system the JVM is running on [Sil05, ch. 4.3]. As discussed in chapter 2.4.3, in JVM Servers Java threads are managed quite uniquely compared to most other operating systems due to the reason that a JVM Server is running on top of Unix System Services (USS), even within a CICS environment (which includes a standard USS). Therefore, the *pthread* library is used for thread mapping of threads created by an application, while the main thread of an application is mapped to a CICS T8 TCB (refer chapter 2.4.2).

Java's Memory Model

> "The memory model for a multithreaded system species how memory actions (e.g., reads and writes) in a program will appear to execute to the programmer, and specically, which value each read of a memory location may return." [MPAM99, ch. 1, p. 1]

Hence, the groundwork for all synchronization actions is the memory model implemented in Java since version 5.0 that according to [HP07, ch. 1, p. 2] "has been designed with two goals in mind: *(i)* as many compiler optimisations as possible should be allowed, and *(ii)* the average programmer should not have to understand all the intricacies of the model". The memory model is based on two concepts, **the happens-before order** and **causality**, both of which are explained briefly in the following. For a more detailed explanation and formal definition of the memory model used by Java a reference is made to [GJSB05, ch. 17.3], [HP07] and to [MPAM99].

Thread 1	Thread 2
1: $r_2 = A$	3: $r_1 = B$
2: $B = 1$	4: $A = 2$

(a) Before reordering

Thread 1	Thread 2
$B = 1$	$r_1 = B$
$r_2 = A$	$A = 2$

(b) After reordering

Table 2.1: Statement reordering example where the assignment $r2 = 2$ and $r_1 = 1$ is a possible result. Source: [GJSB05, ch. 17.3]

The happens-before order. The general problem that appears when using threads are so called **data races** that are found in "programs that are not correctly synchronised" [HP07, ch. 2, p. 3], implying "counterintuitive results" [GJSB05, ch. 17.3]. As stated in [GJSB05, ch. 17.3], "compilers are allowed to reorder the instructions in either thread" for optimization purposes. Hence, the actions shown in table 2.1 can result in the (unexpected) variable assignment $r_2 = 2$ and $r_1 = 1$ after reordering.
In order to solve this problem, Java implements the happens-before concept that enables defining an order in which actions, reads/writes or locks/unlocks, are executed. As described in [HP07, ch. 2, p. 4], the happens-before order represents a "transitive closer of the union of the **po** [the program order] and the **sw** [the synchronized-with] orders", where the **po** defines the order of executions within a thread and the **sw** order defines the order of executions of different threads accessing the same resource [HP07, ch. 2, p. 4], [MPAM99, ch. 2.1, p. 3]. In a simplified way, a transitive closure defines the reachability of objects in binary relations. This can be explained within the context of thread concurrency as the propagation of synchronization: if one resource access is synchronized, all subsequent accesses of the same resource are synchronized as well [MPAM99, ch. 2, p. 3]. An simple example may clarify matters: consider a set of binary relations $R = \{(v, w), (x, y)\}$. In this case the transitive closure T contains the relation (v, y) (hence: $T = \{(v, y)\}$). Now consider that the elements v and w represent actions within a thread T_1 on the variable foo, while x and y represent actions within thread T_2 on the same variable. Then each relation itself represents the **po** (program order) and the order between the two threads T_1 and T_2 the **sw** (synchronized-with) order. Hence, the relation (v, y) represents the happens-before order, implying that action v is synchronized with action y on variable foo. A formal definition of transitive closers is provided by [Cuy07, ch. 1.5, p. 10].
Note that for using the happens-before concept, one needs to make explicit use of synchronization techniques offered by Java.

Causality. As described in [GJSB05, ch. 17.4.8], the happens-before order is "necessary [...] but not sufficient" since it allows "out of thin air" variable assignment arising from "speculative reasoning" [HP07, ch. 2, p. 5]. Considering the example shown in table 2.2, that "is correctly synchronised, because in every sequentially consistent execution none of the guarded writes is executed" [HP07, ch. 2, p. 5] and that aims the assignment $r_1 == r_2 == 0$ and $y == x == 42$. Now "imagine that a compiler speculates that one of the writes could happen, if afterwards can justify it to happen it could optimise the program to make it

Initial: $x == y == 0$	
Thread 1	**Thread 2**
$r_1 = x$	$r_2 = y$
if $(r_1 \neq 0)$:	if $(r_2 \neq 0)$:
$y = 42$	$x = 42$

Table 2.2: Out of the air variable assignment, where $r_1 == r_2 == 42$ is a possible result even with applied happens-before order. Adapted from [GJSB05, ch. 17.4.8].

happen always" [HP07, ch. 2, p. 5]. Hence, the speculation, that is actually used for optimization purposes, can lead to the unexpected variable assignment $r_1 == r_2 == 42$. This problem is called **causality**, due to the fact that the problem implies "no "first cause" for the actions" [GJSB05, ch. 17.4.8].
The solution of causality includes what is called "**causality requirements**" [GJSB05, ch. 17.4.8] defining a procedure where for each action a justified execution should be found [HP07, ch. 2, p. 5] in order for the action to be accepted for the actual execution. Hence, in a simplified way one can describe the procedure as a precautionary measure for justifying certain executions.
Due to its degree of complexity, a formal explanation of these requirements is out of the scope of this introductory section. It is therefore referred to [HP07, ch. 2.1] for a detailed explanation of the causality requirements.
Note that the solution for causality is implemented within the memory model and is used implicitly, implying that causality requirements carried out by the JVM without the need of additional instructions.

Problems with the Java Memory Model. The Java Memory Model specification [GJSB05, ch. 17.4] has been proven to allow incorrect compiler reordering. Aspinall and Sevcik define what they refer to as eight "ugly executions" in [AS07] resulting from statement reordering that imply "surprising behavior". Most of these problems however appear only in unsynchronized program statements. The only issue arising from a synchronized statement appears in context of "roach motel semantics". A reordering action where an unsychronized statement is moved to a synchronized block. Aspinall and Sevcik proved in [AS07, ch. 5], that this action can result in a different outcome of the synchronized block compared to its original version. Although this represents a major issue, the justification of the execution is of theoretical nature and has yet not been proved practically. Moreover Sevcik states in [Sev08, ch. 5.5, p. 115] that "Suns implementation of Java might be in fact correct [...] and it is only the JMM [the Java Memory Model] specification that needs fixing". Therefore, it is advised to use Java's synchronization techniques for all concurrent access. An overview of the most basic techniques are mentioned in the following section.

Practical Thread Synchronization in Java

Prior to outline actual synchronization methods it is important to mention a definition of thread-safety - a term used intensively in context with CICS applications. Within a CICS environment thread-safety is defined

> "as a collection of application programs that employ an agreed-upon form of serialized access to shared application resources. A program written to threadsafe standards, then, is a program that implements the agreed-upon serialization techniques." [RAB+10, ch. 1.2.6, p. 6]

A similar definition of thread-safety is provided for Java applications. According to [GPB+06, ch. 2]

> "a class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code."

Hence, thread-safety does not address specific synchronization methods of applications but their proper execution instead. This leads to the fact, that an application may not need to implement synchronization at all in order to be considered thread-safe. In most applications however, one needs to make use of specific synchronization methods. Java provides several options for synchronization, most of which are based on the memory model. The very basic ones are mentioned in the following.

Note that although thread-safety seems similar to application isolation, both topics differ due to the fact that thread-safety is specifically concerned with the correct synchronization of resource access within applications and application isolation is concerned with the actual restriction of undesired application communication and interaction.

Monitors and Locks.

> "The Java programming language provides multiple mechanisms for communicating between threads. The most basic of these methods is synchronization, which is implemented using monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor. A thread t may lock a particular monitor multiple times; each unlock reverses the effect of one lock operation." [GJSB05, ch. 17.1]

The simplest way of creating monitors and their corresponding locks can be achieved via the `synchronized` keyword, that can be applied to methods as well as to blocks. For method synchronization `synchronized` is used as a modifier shown in listing 2.1.
For synchronizing a particular set of instructions or resources without locking a complete method, `synchronized` is used as a block statement shown in listing 2.2.

As previously mentioned "each object in Java is associated with a monitor" [GJSB05, ch. 17.1]. A thread therefore locks a complete object while accessing a particular synchronized block or method. Roetters states in [Roe01], that

```
sychronized(this){
// code
}
```

Listing 2.1: Usage of synchronized as a modifier

```
public void sychronized myMethod(){
// code
}
```

Listing 2.2: Usage of synchronized as a block statement

"if an object has multiple resources, it's unnecessary to lock all threads out of the whole object in order to have one thread use only a subset of the thread's resources. Because every object has a lock, we can use dummy objects as simple locks", which he refers to as "fine-grain locks" shown in listing 2.3.

```
class myClass{

        Object xlock=new Object();
        Object ylock=new Object();

        public void do(){

                synchronized(xlock){
                // code
                }
                synchronized(ylock){
                // code
                }
        }
}
```

Listing 2.3: Usage of fine-grain locks. Adapted from [Roe01].

Apart from the above mentioned ways of creating monitored sections within an application, Oaks and Wong describe two other options. **Explicit locking** by using the `Lock` interface provided since Java version 5 [OW99, ch. 3.4] and **nested locks** used in special cases to avoid deadlocks for instance [OW99, ch. 3.7].

The volantile Modifier. According to [OW99, ch. 3.2] in Java "threads are allowed to hold the values of variables in local memory (e.g., in a machine register)". This leads to the fact that changes carried out by a thread to a particular variable may not be visible to other threads accessing the same variable. Although synchronizing the variable using the `synchronized` keyword can solve this issue, "the simple task of acquiring and releasing a lock adds more work for the processor and slows execution" [Hyd99, ch. 7]. Therefore, using monitors extensively will result in performance issues of applications. In order to enable

synchronization with respect to performance one can make use of the `volantile` modifier.

> "The volatile keyword is used as a modifier on member variables to force individual threads to reread the variable's value from shared memory every time the variable is accessed. In addition, individual threads are forced to write changes back to shared memory as soon as they occur. This way, two different threads always see the same value for a member variable at any particular time." [Hyd99, ch. 7]

Note that `volantile` variables should be used with care. As described in [GPB+06, ch. 3.4.1], `volantile` variables can guarantee visibility but not atomicity. Hence, `volantile` variables cannot be used for synchronizing simple increments for instance, because an increment represents a "read-modify-write operation" [GPB+06, ch. 2.2].

Advanced synchronization methods. In contrast to the simple synchronization methods mentioned above, Oaks an Wong describe in [OW99, ch. 6.2] some advanced methods, such as locks with counters and barriers. The locks with counters, also referred to as semaphores, enable to lock a resources by multiple threads at the same time [OW99, ch. 6.2.1]. Barriers, on the other hand, that represent a central point of synchronization for all threads of a particular resource [OW99, ch. 6.3.2]. Moreover, Oaks and Wong describe in a detailed fashion some interesting approaches for the prevention of deadlocks in [OW99, ch. 6.3]. Since these advanced techniques are not subject to the practical investigations carried out in chapter 3, a detailed explanation is not provided here.

2.1.3 Java's Platform Security

Some basic security concepts implemented in Java since Version 2 are discussed below.
Security at application level generally aims to prevent malicious functioning of applications, covering a huge number of security concepts. Apparently, important concepts need to be implemented already within the programming language in order to create what is referred to as "well-behaving" applications. Due to Java's portability characteristic, it strictly implements many of these. Java's built-in security therefore includes several features, some of which are implemented within the language itself, leading to the fact, that developers are often not aware of using them. We address these "hidden" security features as implicit security. Other features are optional and have to be used intentionally, implying that developers have to take notice of these features and often invest effort into their configuration. We refer to these features as "explicit" security. The following sections aim to give an overview of the major implicit and explicit security features of the Java programming language and its runtime environment. Since most features are connected to each other, their interaction is shown in figure 2.3.

Language security

Language security features usually "force" developers to follow certain rules defined by the language. The following paragraphs mention some major properties implemented for security purposes. There are other language security properties, such as array bound checks explained in [Oak98, ch. 2], not mentioned below. A complete list is provided in [GJSB05].

Access level modifiers. Protection of objects, methods and variables from each other is provided by what is referred to as *access level modifiers*. "Within a Java program, every entity–that is, every object reference and every primitive data element–has an access level associated with it "[Oak98, ch. 2]. The several modifiers implemented in Java are: `public`, `package`, `private` and `protected`. Using these modifiers one can restrict the access, also called the visibility, of particular objects, methods or variables to certain objects based on their physical location. Classes that have been declared as `private`, for instance, are not accessible to other classes, while `package` classes are visible for all classes within the same package. Note that this mechanism represents a major security feature and aid in object isolation, although the modifiers cannot be used for full isolation of applications. The reason is obvious e.g.: although declaring all classes in an application as `protected` will lead to fully isolated objects, it will also lead to a permanent interruption in inter object communication.

Type safety. Java implements strict restrictions for object conversion. As described in [Oak98, ch. 2] "Java does not allow arbitrary casting between objects; an object can only be cast to one of its superclasses or its subclasses". This implies that objects are not able to masquerade themselves as other objects, for violating access restrictions for instance.

Storage protection. "One of the Java compilers primary lines of defense is its memory allocation and reference model "[GM96, ch. 6.1]. Gosling et. al. [GM96, ch. 6.1] state that memory layout is administered by the JVM enabling what they refer to as "very late binding"; a safety measure ensuring that "programmers cant infer the physical memory layout of a class by looking at its declaration". Another major security advantage mentioned by [GM96, ch. 6.1] are the absence of pointers known from C++, that enable universal access of memory storage within the heap and can be used to exploit malicious behavior.
Next to the methods described above, Java's garbage collector also contributes a major part to memory storage security by ensuring type safety due to the removal of unused object references [Mue05, ch. 3, p. 11].

Bytecode Verification

Based on the assumption that every code can be malicious, the bytecode of non-Java API classes is tested by the *Bytecode Verifier*. "The tests range from simple verication that the format of a code fragment is correct, to passing each code fragment through a simple theorem prover to establish that it plays by the rules:

- it doesn't forge pointers,
- it doesn't violate access restrictions,
- it accesses objects as what they are (for example, InputStream objects are always used as InputStreams and never as anything else) "[GM96, ch. 6.3].

McGraw and Felden [MF99, ch. 2.6] state, that after the verification process, a class guarantees that the following attributes are included:

- the class file has the correct format
- stacks will not be overflowed or underflowed
- byte code instructions all have parameters of the correct type
- no illegal data conversions (casts) occur
- private, public, protected, and default accesses are legal
- all register accesses and stores are valid.

Bytecode verification therefore is a major implicit security feature, that enables the identification of malicious software behavior *prior* to its execution. The bytecode itself however, is not tested for detailed access control due to the access restriction verifications carried out by the theorem prover aim to identify access with respect to access level modifiers [Ler01] instead of objects or resources. Note that this particular functionality is provided by the Access Controller.

Class loading and resulting namespaces

A very important security attribute of Java is provided as a side effect by the class loading mechanism.

According to [Tra01]

> "a Java program, unlike one written in C or C++, isn't a single executable file, but instead is composed of many individual class files, each of which corresponds to a single Java class. Additionally, these class files are not loaded into memory all at once, but rather are loaded on demand, as needed by the program. The ClassLoader is the part of the JVM that loads classes into memory."

McGraw states in [MF99, ch. 2.7] that "there are two basic varieties of class loaders: Primordial Class Loaders and Class Loader objects. There is only one Primordial Class Loader, which is an essential part of each Java VM. It cannot be overridden. The Primordial Class Loader is involved in bootstrapping the Java environment." As mentioned in [Mue05, ch. 3, p. 14], classes in Java are loaded either by using the `new` operator or by referencing static fields and methods such as the system's default print stream `System.out`. In each case however, the JVM uses an instance of the class `java.lang.ClassLoader`, that implements a class resolution algorithm [Gon03, ch. 5.6] for finding particular classes. In detail, "class loading proceeds according to the following general algorithm:

- Determine whether the class has been loaded before. If so, return the previously loaded class.
- Consult the Primordial Class Loader to attempt to load the class from the CLASSPATH. This prevents external classes from spoofing trusted Java classes.
- See whether the Class Loader is allowed to create the class being loaded. The Security Manager makes this decision. If not, throw a security exception.
- Read the class file into an array of bytes. The way this happens differs according to particular class loaders. Some class loaders may load classes from a local database. Others may load classes across the network.
- Construct a Class object and its methods from the class file.
- Resolve classes immediately referenced by the class before it is used. These classes include classes used by static initializers of the class and any classes that the class extends.
- Check the class file with the Verifier. " [MF99, ch. 2.7]

In order to differentiate classes and objects, each class loader instance defines its own namespace, that represents a group of objects accessible by each other. Hence, the class loader security side effect is represented by these namespaces due to their property of "hiding" their objects from other namespaces. Therefore, one can explicitly use different instances of the `ClassLoader` for isolation of objects active within the same runtime environment. Note that, "there is nothing to stop namespaces from overlapping" [MF99, ch. 2.7], which can arise from reference sharing, for instance. A problem that will be discussed more detailed in chapter 3.4.2.

Access Controller

A more suitable approach to access control, compared to access level modifiers, is provided by the *stack inspection* mechanism, implemented in the `java.security.AccessController` class [MF99, ch. 3.6]. As described in [MF99, ch. 3.6], the JVM examines the stack during the runtime and, based on the generalized algorithm explained in [WF98], verifies if particular calls are legit. For this aim, the `AccessController` class provides a hand full of methods for permission checking and temporary access granting [Oak98, ch. 5.5]. The verifications are based on predefined permissions, also called privileges, that can be granted within a policy file for multiple classes, also referred to as a *protection domain*. According to [Gon03, ch. 3] there are 16 different permissions supported by the Access Controller.

Security Manager

Similar to the Access Controller, the Security Manager represents a dynamic mechanism for controlling the access of certain resources. It defines a more general approach, since it implements various customizable methods enabling

what is referred to as fine-grained access control [Gon03, ch. 1.2]. "To understand the relationship between SecurityManager and AccessController, it is sufficient to note that SecurityManager represents the concept of a central point of access control, while AccessController implements a particular access control algorithm" [Gon03, ch. 6.2]. This implies that permission checks by default are deferred to the Access Controller and are not carried out by the Security Manager itself [Oak98, ch. 6]. Another important fact is "that the core Java API never calls the access controller unless a security manager is in place" [Oak98, ch. 5]. The Access Controller and the Security Manager therefore do not supplement, but complement each another.
According to [MF99, ch. 2.8], "the Security Manager has the following duties:

- Prevent installation of new class loaders [...] .
- Protect threads and thread groups from each other. [...]
- Control the execution of other application programs.
- Control the ability to shut down the VM.
- Control access to other application processes.
- Control access to system resources such as print queues, clipboards, event queues, system properties, and windows.
- Control file system operations such as read, write, and delete. Access to local files is strictly controlled.
- Control network socket operations such as connect and accept.
- Control access to Java packages (or groups of classes), including access to security enforcement classes."

Each duty represents a number of methods, that can be customized following the desired behavior. By default these methods implement simple permission checks. The permissions are defined within a *policy* file, that represents a white list, implying that by default no permissions are granted. Therefore, in order to allow access to particular methods of system classes one needs to grant a permission explicitly within the policy file once a Security Manager is installed. To grant the access to the `stop` method of the `Thread` class for instance one needs to add the following lines to the policy file

```
grant {
   java.lang.RuntimePermission "stopThread";
}
```

Altering the Security Manager will be discussed more detailed in chapter 4.2.1. Note that the Security Manager is an optional feature and hence, has to be used explicitly.

2.2 OSGi

Since JVM Servers include an implementation of the OSGi framework, its basic concepts and advantages are discussed briefly within this section.

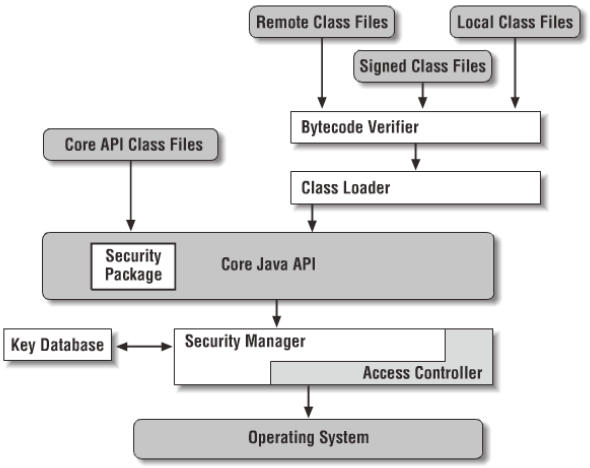


Figure 2.3: Java security features interaction. Source: [Oak98, ch. 1.3]

The OSGi (Open Services Gateway initiative) Alliance, is a consortium of major technology companies aiming to "create open specifications that enable the modular assembly of software built with Java technology" [OSG11a]. The need for these standardized specifications arises from the increasing complexity of software products that require high costs for development and administrational tasks. As stated in [OSG11a] the major problem is that "today, software development largely consists of adapting existing functionality to perform in a new environment". As pointed out in [OSG11a], combining Java's portability characteristic with "standardized primitives that allow applications to be constructed from small, reusable and collaborative components" is a significant approach towards solving this problem. OSGi specifications cover a number of services and an application framework enabling modular application development.

The OSGi framework. The basic idea behind the framework covers the encapsulation of programs, functions or software components into modules, called bundles, that are controlled by the framework. A bundle is a commonly known JAR file that, as outlined in [OSG11b, ch. 3, p. 25], contains "the resources necessary to provide some functionality", "a manifest file describing the contents of the JAR file" and an "optional documentation". The manifest file should not be mistaken for a deployment descriptor used in Java EE applications due to the fact that it carries in "information about itself" [OSG11b, ch. 3.2.1], while a deployment descriptor is "used to communicate the needs of application components to the Deployer" [CS09, ch. 2.12.4, p 24.]. The bundles can be installed, updated, started or stopped without restarting the JVM. Apart from this major advantage, the modules are separated from each other and are only able to share resources using predefined interfaces. The following section provides a more detailed explanation.
Note that within the scope of this paper bundles are interpreted as actual Java applications (refer chapter 3.2).

OSGi framework implementations. The framework specifications have been implemented into several commercial and open source frameworks such as Concierge[1], Equinox[2], Klopferfish[3] and Apache Felix[4] (refer [OSG11a] for more information). Most of these open source implementations have been successfully implemented in production environments. The Equinox framework for instance, is implemented in the Eclipse Integrated Development Environment[5] as well as in CICS JVM Servers (refer chapter 2.4).

2.2.1 OGSi Framework Architecture Overview

The framework is divided into several layers (refer figure 2.4), which are briefly explained in the following. Note that the execution environment is represented by the Java Virtual Machine and is therefore not mentioned below.

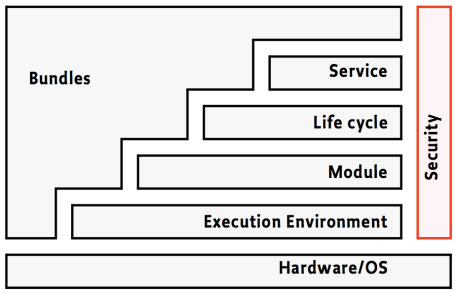


Figure 2.4: OSGi framework layered architecture. Source [OSG11b, ch. 1, p. 2]

The module layer provides the general modularization functionality, that enables to keep modules separated. Tavares and Valente [TV08, ch. 2, p. 2] state that "each bundle is assigned to a different class loader, thus creating a particular address space for resources and classes packaged in bundles". A property, that will be investigated within the scope of chapter 3.

The life-cycle layer enables the feature of continuously using the JVM without the need for restarts for software updates. It therefore enables advanced versioning of software components by providing methods for life-cycle control such as install/uninstall and start/stop. The OSGi framework specification defines six different bundle states described in [OSG11b, ch. 4, p. 81] and shown in figure 2.5. Note that the lazy activation shown in figure 2.5 is a policy indicating "that the bundle, once started, must not be activated until it receives the first request to load a class" [OSG11b].

The service layer offers an efficient and secure way of information exchange between bundles via services. According to [OSG11a], "the reason we needed the service model is because Java shows how hard it is to write collaborative model

[1] http://concierge.sourceforge.net/
[2] http://equinoxosgi.org/
[3] http://www.knopflerfish.org/
[4] http://felix.apache.org/site/index.html
[5] http://www.eclipse.org/

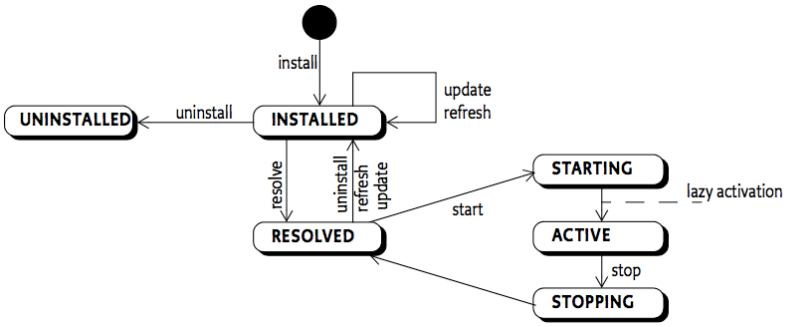


Figure 2.5: OSGi framework life-cycle layer bundle states. Source [OSG11b, ch. 4, p. 82]

with only class sharing". As described in [RAR07, ch. 3.2, p. 4] "in the OSGi model, any Java class can be published as a service to be used by other bundles in the system". The service layer includes a *service registry* that monitors services. This implies that one or more bundles can register their service in the registry while other bundles can look up for offered services within the registry and use them for their specific purpose.

The security layer "is an optional layer" and "is based on the Java 2 security architecture" [OSG11b, ch. 2, p. 11]. It includes permission, digital signing and certificate related functionalities. Using these functionalities one can allow a bundle to manage others, or restrict the usage of certain interfaces to particular bundles.

2.3 Previous Java integration in CICS

Since the introduction of the Java programming language, intense efforts have been invested into integrate Java within CICS. The following sections aim to give an overview of CICS and the development of Java integration in CICS as well as technologies related to the integration.

2.3.1 Introduction to CICS

The Customer Control Information System (CICS) is a proprietary transaction processing system introduced by IBM in 1969 [Gar09] for z/OS systems. Evolved from the emerging need for "systems that could process data in real time from terminals connected to the computers" [IBM04b], which are referred to as "online systems", during the 1970s', CICS became one of the most used transaction processing system worldwide, handling more than 30 billion transactions per day [IBM04a]. As described in [Hor00, ch. 1, p. 1], "hardly a day goes by when something that you do has not involved a CICS application somewhere in the world - whether it is a trip to the supermarket, taking money from your bank account [...] or personnel records - CICS is involved". The reasons for its popularity are among others its robustness and its "superior perfomance

characteristics" [Spr10, p. 28].
Although CICS is nowadays available for most operating systems, including Linux and Microsoft Windows, it is still primarily used on z/OS. Therefore, the following paragraphs have a clear focus on introducing major CICS characteristics for z/OS systems.

Storage protection keys in CICS. Similar to TSO (Time Sharing Option) and USS (Unix System Services), CICS runs as a subsystem of z/OS on top of the kernel. Therefore, CICS has an own virtual address space, which is partially mapped to the central storage, also called the memory, and the auxiliary storage, such as external hard drives. In order to prevent unauthorized access to the storage blocks, z/OS implements a unique mechanism that includes the assignment of each storage block to a *protection key* representing the kernel, a subsystem or any user. According to [IBM10b, ch. 1, p. 12],

> "when a request is made to modify the contents of a central storage location, the key associated with the request is compared to the storage protect key. If the keys match or the program is executing in key 0, the request is satisfied. If the key associated with the request does not match the storage key, the system rejects the request and issues a program exception interruption."

Therefore, prior to each access of a particular storage block, the kernel's storage manager verifies if the access is legit. This feature leads to highly isolated storage blocks implying prevention of buffer overflow attacks.
The protection key 0 is assigned to the kernel with universal access to all storage blocks, keys $1 - 7$ are assigned to z/OS related subsystems and keys $8 - 15$ are assigned to user access.
Note that the protection keys are not bound to specific users and do not represent user ids but the access type instead. Therefore, as described in [IBM10b, ch. 1, p. 12] "most users - those whose programs run in a virtual region - can use the same storage protect key" while "these users are called V [V = virtual users] and are assigned a key of 8".

CICS manages its own virtual address space by implementing a middleware, called the CICS Nucleus, that can be described as a mini kernel including task, storage, terminal, file and program management components (refer figure 2.6). As a result, users of CICS can either be interpreted as virtual users and assigned to protection key 8 or as regular users and assigned to protection key 9. Using key 9 by a particular program within CICS will restrict the access of that program to CICS' storage blocks to *read only* [IBM11b, ch. 45, p. 575], and can therefore be interpreted as a feature.

Transactions. As described in [IBM11d, ch. 3, p. 26], a CICS transaction is known for both, the commonly used meaning of a transaction, defining a unit of recovery or a unit of work (UOW) with ACID characteristics as well as "all other transactions of the same type" (refer figure 2.7) combined to a group of components. The group includes at least one program, representing the business logic, a mapset, representing the output interface for applications using 3270 protocol and a unique transaction id. This implies that all operations carried

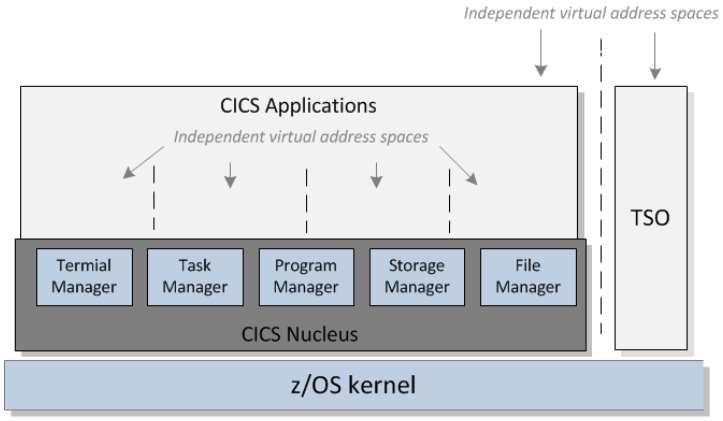


Figure 2.6: Simplified CICS architecture. Modified after [Spr08].

out from each program defined within a particular CICS group strictly follow all ACID characteristics. Thus, CICS is "acting as an "application server" to user applications" [IBM04a] and one might "find it helpful to think of CICS as an operating system within your own operating system" [IBM91, ch. 1] offering various services for programs.
Transaction processing in CICS is initiated by calling the transaction id of a par-

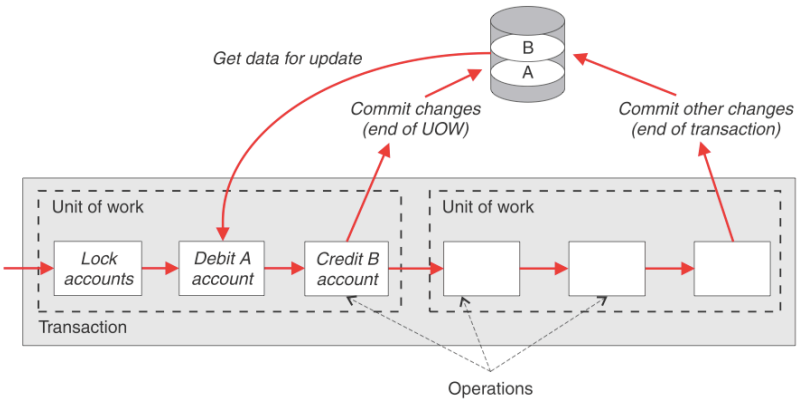


Figure 2.7: CICS transaction including several units of work. Source: [IBM05, ch. 5, p. 46]

ticular transaction. Thereupon CICS takes over control and starts a task, also called a "dispatchable unit of work" [IBM10b, ch. 1, p. 23], which is represented by a task control block (TCB) and responsible for the transaction processing (refer figure 2.8). The TCB has a function somewhat similar to the Process Control Block (PCB) in Unix and other operating system kernels sharing the same life cycle as shown in figure 2.9. It is used to enable multiprocessing and is maintained by the CICS task manager.

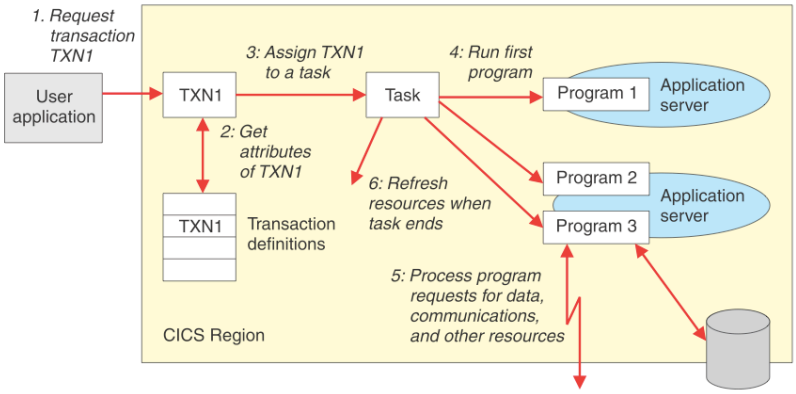


Figure 2.8: CICS transaction execution process. Source: [IBM05, ch. 5, p. 46]

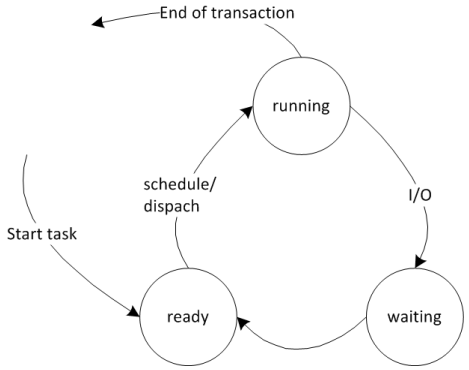


Figure 2.9: CICS transaction task life cycle. Modified after [Spr08]

Application development for CICS. As previously indicated, programs defined within a transaction are under full control by CICS, which enables the possibility of accessing a broad variety of services provided by CICS. Examples of such services are

- unit of work services representing ACID properties of transactions
- data management services for operating system specific data sets
- communications services such as remote function calls.

According to [Bat08, p. 14] "this allows developers to focus on solving business problems rather than implementing system functions" leading to the fact that application development can be performed faster and carried out programs are more reliable.
In general, CICS programs do not need to follow specific patterns due to accessing CICS services is achieved using the CICS API, supported for the programming languages C/C++, COBOL, PL/1 and Assembler, by calling the 'EXEC CICS ...' statement. These statements are translated by a pre-compiler, the CICS translator, into language specific instructions for CICS service calls. Java

programs, however, do not need to be pre-compiled for accessing CICS services, due to the JCICS library provided by CICS, which implements common Java library-like functions.

2.3.2 CICS JVM support

General Java support for CICS, including JCICS, was introduced in 1998 with the release of the CICS Transaction Server version 1.3 [IBM11e]. Within this release however, Java programs were not run within a Java Virtual Machine but were compiled into C code, that itself was compiled into S/390 machine code using the High Performance Java (HPJ) product [RBC+09, ch. 2, p. 18]. With the implementation of real JVM support in CICS version 2.1, HPJ became unnecessary for running Java applications within CICS.

Java Virtual Machines in CICS. Since the introduction of real JVM support in CICS, JVMs run within the Language Environment (LE). This is a common runtime environment for several programming languages and can be described as a collection of resources, libraries and operating system specific services and interfaces [IBM10a, ch. 1, p. 3]. One of the reasons for running the JVMs within the LE, is the feature of accessing services not provided by the CICS-API, such as operating system services (similar to USS). Another reason for using the LE are the enclaves provided by the LE, which enable isolation of programs running within the same address space. This enables the feature of running multiple JVMs within the same CICS region. Unlike other programs running in CICS, JVMs imply a specific workload management. As outlined in [IBM11d, ch. 1, p. 10], the JVM runs within "z/OS as a UNIX process" and "therefore uses MVS Language Environment services rather than CICS Language Environment services". This Unix process includes the LE enclave. From the LE point of view the JVM and the programs using the JVM are interpreted as a thread within the enclave. The heap allocation therefore is carried out by the LE. The storage however, remains under the control of CICS, and the entire enclave is therefore interpreted as a task (refer figure 2.10). This implies that all external operations carried out from a Java program by using CICS services are thread-safe (refer chapter 2.1.1) .

As mentioned in [RBC+09, ch. 2, p. 20], CICS differs between three different JVM operations modes categorized by their usage. These modes are briefly introduced in the following while their graphical differentiation is shown in figure 2.11.

Single Use. In order to ensure full isolation between programs, which might change the state of the JVM, by updating the time zone for instance, CICS implements a JVM mode that handles only one program during the JVM lifetime, meaning that after the program execution the JVM is destroyed. Although this approach results in full isolation of programs, it obviously also implies performance issues caused by the processing overhead of JVM creation and destruction.

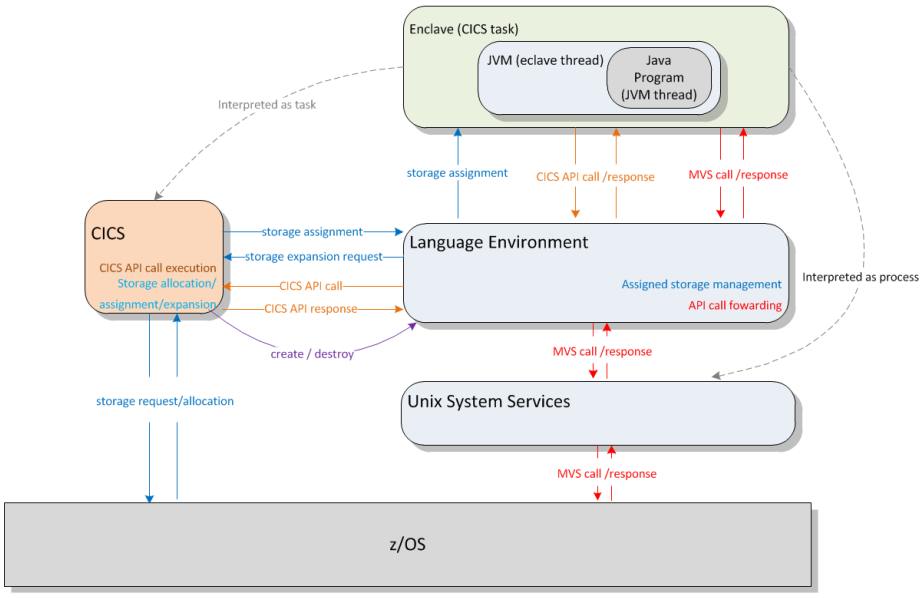


Figure 2.10: Simplified architecture of JVMs in CICS

Continuous. For a significant performance increase, one can make use of the continuous JVMs, that are reused sequentially by multiple transactions. They are however only suitable for programs that do not change the state of a JVM.

Resettable. With CICS release 2.1 a new function for continuous JVMs was introduced that enabled to reset the state of a JVM into its original state prior to certain program executions. This operation mode is also called the persistent reusable JVM. According to [IBM01, ch. 5, p. 75] however, there are certain JVMs that are not resettable; one with an updated static variable for instance. In that case, CICS destroys the unresettable JVM and creates a new one. Doubtless, the reset functionality has been a major approach towards the isolation of Java programs. Due to the fact that the reset functionality and the creation of new JVMs resulting from unresettable states requires computational resources as well, the performance increase compared to single use JVMs was moderate. Moreover, according to [RBC+09, ch. 1, p. 14], resettable JVMs claim more CPU costs per transaction and a more complex set and tune up procedure than continuous JVMs. Hence, the resettable JVM mode has been discarded and is not supported since CICS version 3.2. Another reason for the discard, was the fact that the resettable JVM's reset functionality is not part of the official JVM specification (refer [LY99]). Therefore, it was considered as a proprietary JVM implementation.

Note that all JVM operation modes mentioned above differ from each other, although they share a common property: each JVM can handle only one Java program at any time. Therefore, usually several JVMs are running within separate enclaves for Java program execution - also referred to as the *JVM pool*, that

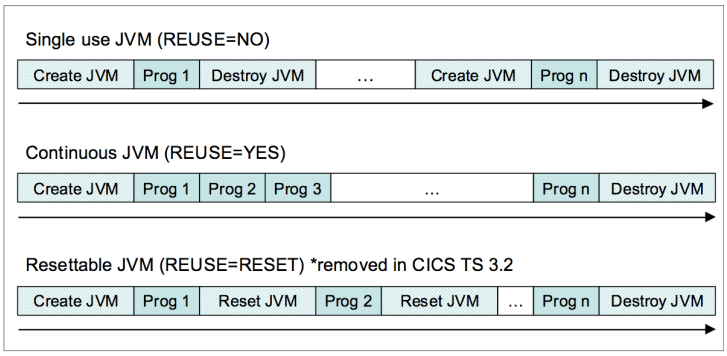


Figure 2.11: JVM operation modes in CICS. Source [RBC+09, ch. 2, p. 20]

is managed by CICS. This property obviously agrees with the characteristic of many common application servers and results in performance issues. Moreover, as stated in [Arn11], due to not only the programs but also the JVM consuming storage space too, only about 20 JVMs fit into a CICS region. In order to solve these issues, the so-called JVM Servers were introduced in CICS version 4.2 (refer chapter 2.4).

2.3.3 CICS Open Transaction Environment

As previously mentioned each program within CICS is interpreted as a task with a life-cycle similar to a thread in Unix. Prior to CICS version 1.3 this task was represented by a *Quasi Reentrant (QR)* TCB, which could be suspended by the task manager if and only if the task was accessing the CICS API. This property implies high isolation of programs accessing CICS services but also the fact that no concurrent task processing was possible because only one TCB could be active at any time within each CICS region, with no respect to the amount of computational resources available. It is because of that reason that Open Transaction Environment (OTE) was introduced in CICS 1.3. According to [RAB+10, ch. 1, p. 3], OTE aims to increase throughput, improve performance and enable the use of non-CICS APIs. OTE includes several different TCB modes, called *open TCBs*, which can coexist and prevent TCB blocking. This mainly arises from the fact that most APIs accessed by programs within CICS were rewritten for thread-safe execution.
Since CICS version 2.1 and the introduction of the JVM specific TCB modes *J8*, CICS key, and *J9*, user key, JVMs benefit from OTE too, as these modes enable the option of running multiple JVMs within the same CICS region. Note that the QR TCB is still available for the use of non thread-safe APIs and as stated in [Hac06, p. 4] due to the "heavy CPU exploiting nature of Java applications", Java TCBs have been prioritized lower than the QR TCB.
In CICS version 4.2 however, a new TCB for Java related tasks has been introduced, the *T8* TCB, which is used only by JVM Servers.

2.4 JVM Server

2.4.1 Overview

A new fully reusable continuous JVM mode was introduced in CICS version 4.2 where multiple programs are using the same JVM called a *JVM Server* (refer figure 2.12). The pooled JVMs however, are still supported including, both single use and continuos modes, but "will be removed in a future release of CICS" [IBM11d, ch. 1, p. 2]. A major benefit of JVM Servers is their implementation of the open source Equinox OSGi framework including all features of OSGi such as advanced versioning, modularization and service based communication.
The OSGi framework is not part of the JVM. In a simplified way it represents an application that itself serves as an application container. Thus, there is no need of extending the JVM specification. Unlike as the resettable JVM (refer chapter 2.3.2), the JVM Server is not considered as a proprietary JVM implementation. Note that despite the renewals, a JVM Server implies the same environment as pooled JVMs including an enclave, LE and USS as shown in figure 2.10.

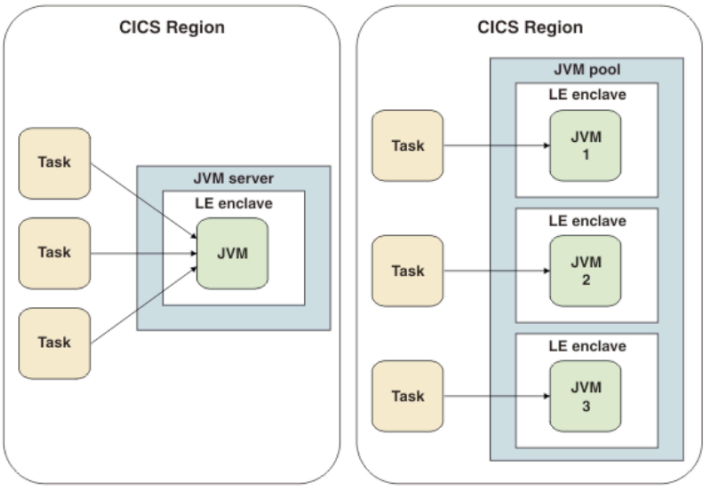


Figure 2.12: Comparison of the JVM Server and the pooled JVM environment. Source [BCC+11, p. 3]

2.4.2 JVM Server and OTE

In JVM Servers each application is assigned to the *T8* TCB, a new TCB type exclusively used by JVM Servers. It is running concurrently as a thread with other programs active within the JVM [IBM11d]. Moreover, as outlined in [IBM11d, ch.3, p. 29], "not only is the JVM shared by all CICS tasks, which might be running multiple applications concurrently, all static data and static classes are also shared". This arises from the fact that all threads within a JVM belong to the same process and therefore are using the same address space (refer chapter 2.1.2).
CICS also allows to run multiple JVM Servers within the same CICS region; an important feature since as mentioned in [BCC+11, p. 2] "each JVM Server

can be configured with a different set of runtime components or even different sets of middleware components“. According to [IBM11d, ch. 1, p. 5] one JVM Server can run up to 256 threads, while the maximum amount of T8 threads is limited to 1024. This implies that “the JVM Server gives CICS the ability to handle many more Java tasks in one region than ever before” [BCC+11, p. 2].

2.4.3 Threads

JVM Servers imply a quite unique thread management compared to UNIX or Microsoft Windows operating systems. As described in [IBM11d, ch. 3, p. 29] “CICS tasks run in parallel as threads in the same JVM Server process”. Considering that the JVM Server runs within a Language Environment on top of Unix System Services (refer figure 2.10), JVMs consequently use the pthread library with the many-to-many model for thread execution. Since Java applications within CICS are represented by CICS tasks, the pthreads accessing CICS services are mapped to a CICS T8 TCB. As CICS-API calls are full transaction safe, the major problems in terms of safe concurrent access reside within the JVM Server environment, that includes several pthreads sharing the same address space. Hence, according to [IBM11d, ch. 1, p. 1], Java applications should be thread-safe in order to be installed on a JVM Server. A detailed discussion to thread-safety in JVM Server environments is pointed out in chapter 3.
It is important to mention that when creating user threads within Java applications running in a JVM Server, one has to consider a major fact that needs additional care: according to [IBM11d, ch. 3, p. 29] “these threads cannot access CICS services [...] [and] any attempt to access CICS services from an application-spawned thread results in a Java bm.exception”. Moreover, one needs to “ensure that they [the application-spawned threads] do not run beyond the lifetime of the CICS task that runs the application” [IBM11d, ch. 3, p. 29]. Note that except of user created threads all bundles active within a the JVM Server use the OSGi framework thread.

2.4.4 OSGi in JVM Servers

Although JVM Server implement the Equinox OSGi framework, they imply unique characteristics compared to the stand-alone Equinox OSGi framework implementation. At first, OSGi bundles need to be included into a CICS bundle, that represents the actual resource for CICS. Each CICS bundle can include one or more OSGi bundles. This enables a higher level of bundle grouping and leads in some cases to an advantage in terms of maintenance and administration of bundles. At second, CICS bundles can include the `CICS-MainClass:` statement in the manifest file referencing the main class of the bundle. Therefore, a CICS JVM Server application does not need to implement an activator class that according to [OSG11b, ch. 4.2, p. 88] is used to start and stop the bundle. Note that as stated in [Bre11] these Activator classes “do not run on a T8 TCB” but on an application-spawned thread, that implies the previously mentioned application-spawned thread properties (refer chapter 2.4.3). The third important unique characteristic of JVM Server is the bundle administration. Unlike in stand-alone OSGi frameworks that include an OSGi console for the administrational tasks such as installment and disablement of bundles, JVM Servers are exclusively administered by the CICS Explorer (refer to [RBB+10]),

that is available as a plug-in for the Eclipse SDK[6]. There is however, an option of starting the console described in [Bre11]. It includes adding the parameter `osgi.console=<port>` into the JVM Server profile and connecting via Telnet to the specified port. Although this option represents an interesting alternative to the CICS Explorer, according to [Bre11] it should only be used for "program diagnosis", since administration of bundles from the console "leads to [their] inconsistent state in CICS".

2.4.5 Subsystem Interaction

As previously indicated, the entire application development and application administration for JVM Servers is carried out by using the CICS Explorer. It offers a user interface for USS access via FTP for the upload of bundles as well as a CICS management interface for application control. The interaction of the Local Development Environment, USS and CICS among with the responsibilities of each system and the application flow is shown in figure 2.13. It includes the CICS bundle within USS to be used as the resource for the actual bundle installed on a predefined JVM Server within CICS. The CICS program includes a reference to the main class of the bundle and a reference to the JVM Server while the CICS specific transaction, that can be started using a 3270 terminal emulator for instance, refers to the CICS program.

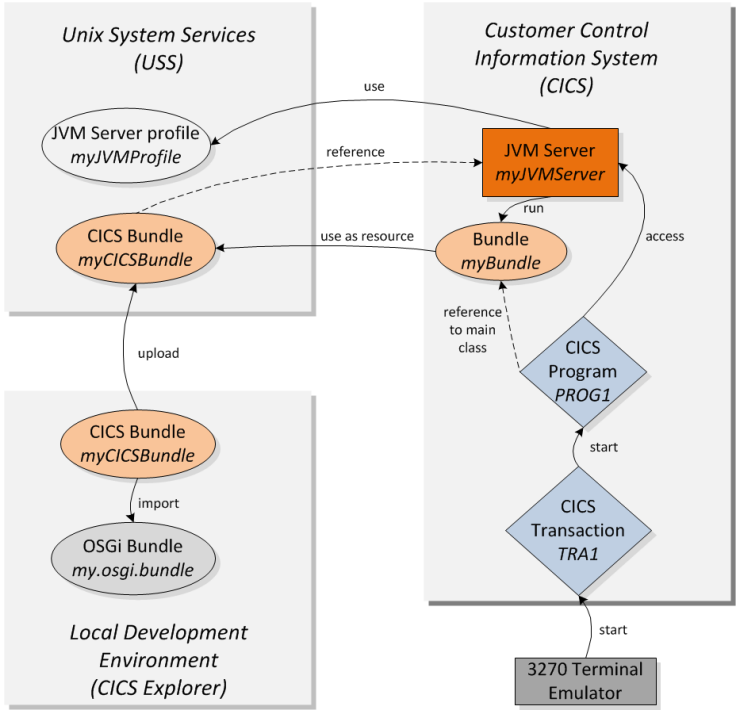


Figure 2.13: Subsystem interaction

[6]http://www-01.ibm.com/software/htp/cics/explorer/

Chapter 3

Application Isolation Properties of JVM Servers

3.1 Overview and Related Work

The identification of isolation issues in multiple application JVMs and the development of solutions for these issues has been subject to several researches. Balfanz and Gong [BG97] for instance outlined several issues with multiple application JVMs and proposed several extensions to the JVM specification for isolation purposes. Another significant approach towards application isolation was found to be the Barcelona Project[1], that included several approaches towards application isolation described in [Cza00] and [CD01]. Although most problems that have been identified in such researches were resolved within the resettable JVM mode (refer chapter 2.3.2), this mode has been replaced by the JVM Servers, where multiple applications are using the same JVM in parallel. Therefore, known application isolation issues were applied to a JVM Server in order to verify if these still play a major role.

Most isolation issues have been adapted from [Mue05], that outlines several important issues related to isolation deficits of the JVM and from [PF09], that outlines several vulnerabilities of OSGi frameworks.

Note that within the scope of this paper application isolation issues are defined as undesired influence to applications carried out within the same JVM. Therefore, other vulnerabilities of Java or the OSGi framework, such as outlined in [PF09], for instance, were not considered.

3.2 Testing Procedure

In order to verify if common isolation issues are applicable to JVM Servers practical tests have been carried out. Since most issues have been adapted from [Mue05], that describes general Java issues, and from [PF09], that has focus on OSGi specific issues, these publications serve as a groundwork for the tests. The

[1] http://labs.oracle.com/projects/barcelona/

Software	Version
z/OS	1.12
CICS	4.2
CICS Explorer	1.1
Eclipse	3.7.1
Equinox	3.6.1
Java (64 Bit)	6.0.1
Java Health Center	2.0.0

Table 3.1: Software versions used for the verifications.

actual issues applied to JVM Servers and their results are documented within the following sections. The results are represented in form of

- System Display and Search Facility (SDSF) log files,
- CICS output messages,
- Java specific error messages such as exceptions and
- outputs such as charts and messages provided by the Java Health Center[2], that represents a diagnostic tool for JVM Servers.

All software and the corresponding versions used for the verifications are shown in table 3.1.
Note that the source code used to reproduce open issues in chapter 3.4 is provided in appendix A. Also note that for the sake of simplicity and due to the reason that an OSGi bundle can include an additional `CICS-MainClass:` statement in the manifest file (refer chapter 2.4), each OSGI bundle is considered as a Java application in the following.

3.3 Closed Issues

3.3.1 C_1: Multitasking with Single Class Loaders

The issue outlined by Mueller in [Mue05, ch. 3.2.5] in context of using single class loaders, where an application is able to update static variables of other applications, is resolved in JVM Servers. This is due to the fact, that each bundle is using its own class loader, that represents one of the techniques proposed in [BG97] for the solution of isolation issues. Therefore, bundles are not able to access each other in a JVM Server. Since system classes however, "are not subject to per-application replication" [Cza00], application isolation using different class loaders is not complete [Mue05, ch. 3.2.7, p. 16] (refer I_1, chapter 3.4.1). Moreover, isolation can simply be omitted when namespaces overlap (refer I_2, chapter 3.4.2).

[2] http://www-01.ibm.com/support/docview.wss?uid=swg21413628

3.3.2 C_2: Concurrent Access to Shared Resources

In a stand alone JVM environment, without CICS, file and database access shared by multiple applications needs synchronization. Since CICS offers services for these tasks (refer [RBC+09, ch. 6]) and since CICS services are thread-safe, additional synchronization within application code becomes obsolete when strictly using these services. This however, does not apply to inter application communication if applications share objects as described in I_2, chapter 3.4.2, for instance.

3.4 Open Issues

3.4.1 I_1: System Classes

Static Fields of System Classes

In a OGSi framework each bundle has an own class loader, leading to the fact that each bundle has an own namespace. This property results in high isolation due to classes from different bundles are not visible to each other. As described in [Mue05, ch. 3.2.7, p. 16], although using different class loaders for isolation purposes certainly represents a good approach, isolation remains incomplete. Czajkowski [Cza00, ch. 1, p. 1], states that "the place where the isolation breaks [when using different class loaders] is the interaction of applications through static fields and static synchronized methods of system classes (they are not subject to per-application replication)". Hence, by default each application running within a JVM can change static fields and execute synchronized methods of system classed with no restriction, which could effect other applications running within the same JVM.
This fact actually represents one of the major reasons for the introduction of the resettable JVM mode in CICS described in chapter 2.3.2. Since this mode is replaced by the JVM Server, this section aims to outline the issue that is related to changes of the state of the JVM within JVM Servers.

Within one practical test, it was verified that the problem of static field and synchronized method access of system classes is applicable to JVM Servers. The code used to reproduce the issue is shown in appendix A.1.1, that basically shows the `Access` class of bundle B accessing the `file.separator` system property which has been changed previously by the `Change` class of bundle A. The output produced by both applications is shown in listing 3.1. It indicates that the change of the system property carried out by the `Access` class instance is visible to the `Change` class instance.

Severe Issues with Methods of System Classes

The fact that all bundles are able to access methods of system classes implies the fact that an application is able to shutdown the JVM by using the `exit(int status)` method of the `java.lang.System` class [PF09, ch. 3.2]. In JVM Servers however, this call leads to a more crucial issue: the shut down of the entire CICS region implying that *all* applications running within the particular CICS region will be unexpectedly closed. Using the code provided in

```
Starting bundle A
Current file separator: /
Changing file separator to \
Current file separator: \

Starting bundle B
Current file separator: \
```

Listing 3.1: Output produced by code from appendix A.1.1 indicating that changes to static fields of system classes are globally visible.

appendix A.1.2 this issue was reproduced. The output message of CICS displayed on a 3270 terminal emulator prior to the shutdown is shown in listing 3.2. It indicates that the shutdown was carried out immediately - without additional need of any user interaction. It is therefore stated in [IBM11d, ch. 3, p. 27] that the `System.exit` method should not be used since "this method causes both the JVM Server and CICS to shut down, affecting the state and availability of your applications". This results from the fact that CICS cancels the region on purpose. Since the JVM Server shutdown using the `exit(int status)` method of the `java.lang.System` class is carried out of the scope of CICS and since it is uncertain what transactions are affected by the unexpected JVM Server shutdown, CICS cancels the region in order to to assure that a rollback can be performed. Therefore, this particular action does not represent a bug.
There is however, another option for achieving the same result and beyond de-

```
DFHTM1703 CICS1 CICS is being terminated by userid IBMUSER in
    transaction STO at netname SC0TCp26.
```

Listing 3.2: CICS message prior to an unexpected shutdown of the CICS region CICS1 caused by the transaction STO.

scribed in [PF09, ch. 3.2.5]: the `java.lang.Runtime` class, that enables the execution of native operating system calls. Using the `halt()` method, for instance, will lead to the same result as above (sourcecode shown in appendix A.1.3). Moreover, using the `exec(String "command")` method, one can execute operating system calls and access the file system for applying changes to certain resources out of the scope of any synchronization such as synchronized file services provided by the JCICS library (refer [IBM11d, ch. 3, p. 60]). The source code of a successfully applied example is shown in appendix A.1.4, where the bundle deletes a file within the working directory of the USS file system. This leads to the fact, that it is generally possible to execute the Unix specific command `rm -rf *`, that leads to the deletion of all files within the working directory, i.e. all logs and file resources created by applications active within the JVM Server. Note that this does not imply a security issue at operating system level due to the reason, that the JVM Server should have read/write/execute rights for his own working directory.

Another major issue related to isolation issues in context with system classes was found in the `java.lang.Thread` class (refer [Mue05, ch. 3.5]). Since all

applications are represented by threads within a JVM Server (refer chapter 2.4) and since all threads can be accessed using the `getAllStackTraces` method of the `Thread` class, one can stop, suspend or interrupt other applications that are active within the same JVM Server using the corresponding methods of the `Thread` class (refer [Ora11b]). Appendix A.1.5 shows the source code used to reproduce the issue. Its execution led to a termination of all active transactions with the `AKC3` abend code that according to [IBM11a, p. 131] indicates that "the task has been purged, probably due to operator action".

3.4.2 I_2: Overlapping Namespaces

Although using different class loaders leads to isolated bundles, one has to be aware of overlapping namespaces. These omit the security side effect of multiple class loaders (refer chapter 2.1.3). In the case of OSGi, overlapping namespaces appear when importing/exporting packages of bundles.

Class Loading and Static Fields

In some cases it is necessary to access objects in between bundles. For this aim a bundle can export a package which another bundle imports. The resulting relationship between the bundles is called a *wire* (refer [OSG11b, ch. 3.5.1, p. 39]). In that case the importer bundle and the exporter bundle namespaces will overlap. An example modified after [Lip08] may clarify matters: consider a class `XClass` of bundle X and a class `YClass` of bundle Y, both of which are importing `AClass` class from bundle A and are accessing a static field of `AClass`. Since `AClass` "is loaded only once [...] ([by] its defining class loader)" [Lip08], bundle Y's instance of `AClass` will see changes to static fields carried out by bundle X's instance of `AClass` (refer figure 3.1). This results from the resolution algorithm, that includes the search for classes loaded within the imported bundles via delegation [OSG11b, ch. 3.5, p. 28].
The mentioned example is similar to the scenario of using single class loaders described in [Mue05, ch. 3.2.5] and has been applied to a JVM Server. Appendix A.2.1 shows the source code used to reproduce this example, where bundle X's change to the static string variable `myName` of `AClass` is visible to bundle Y (refer to listing 3.3 for the textual output produced by the application).

Creating wires however, enables another potentially dangerous feature: sharing of objects. The fact that a wire enables the visibility of certain packages, leads to the fact that an exporter bundle can return a reference of a particular object to an importer bundle. In that case both bundles are able to access the same object. Mueller refers to this problem in [Mue05, ch. 3.3, p. 18] as "deficits in the area of inter process communication". It implies the absence of a mechanism to revoke a reference and administrational issues resulting from uncontrolled reference sharing [Mue05, ch. 3.3, p. 18]. Note that since each Java application represents a thread-safe CICS task, one does not need to take care of synchronization at this point. This particular problem however appears when sharing objects and accessing these by application-spawned threads as described in the following.

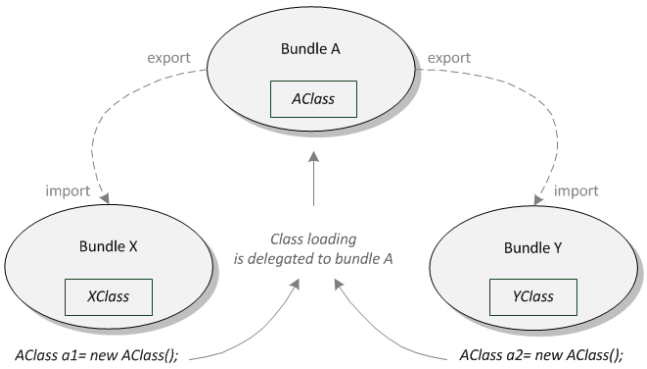


Figure 3.1: OSGi classloading is delegated to the exporter class leading to overlapping namespaces. Modified after [Lip08].

```
## instance of XClass ##
(XClass) class loader used to load AClass:
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader@87e63e0
(XClass) current value of myName: undefined
(XClass) changing value of myName
(XClass) current value of myName: Smith

## instance of YClass ##
(YClass) class loader used to load AClass:
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader@87e63e0
(YClass) current value of myName: Smith
(YClass) changing value of myName
(YClass) current value of myName: Doe
```

Listing 3.3: Output message indicating that changes carried out from bundles x and y to bundle a's static variable $myName$ are visible to each other due to the fact that both bundles are using the same class loader to load $AClass$.

Threaded access of shared Objects

When accessing shared objects using threads, synchronization problems become isolation issues. Consider an example with three bundles: *ABundle*, including `AClass`, *XBundle* including `XClass`, and *YBundle* including `YClass`. In this example an object of `AClass` creates an instance of `ZClass` and shares this instance with instances from `XClass` as well as `YClass`, while `XClass` and `YClass` represent threads, that access and increment a variable from the instance of `ZClass` (refer figure 3.2). Now if `ZClass` does not implement synchronization, a data race appears. The source code applied to JVM Servers implementing the example to exploit a data race is shown in appendix A.2.2. The textual output of the application written into the standard output file is shown in listing 3.4. It implies a data race because *Y object* did not see the update of the counter to the value 7 carried out by *X object* and hence, *Y object* resets the counter from 7 to 3. Using synchronized methods or synchronized blocks however, implies another major problem in context of hanging threads as described in the following.

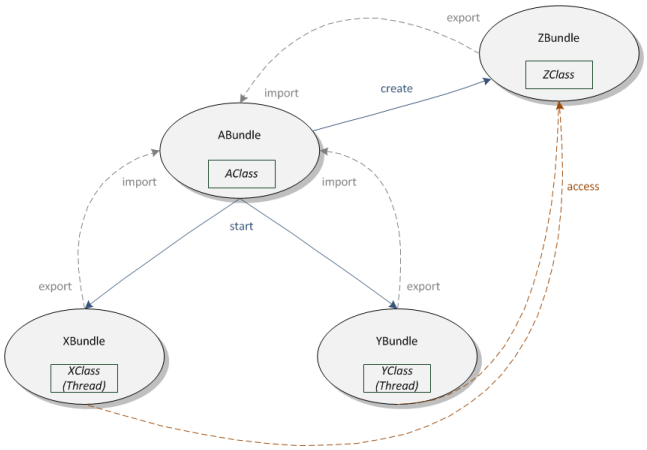


Figure 3.2: Schematic representation of threads from *XClass* and *YClass* accessing the shared instance from *ZClass*.

```
(A object) Starting data race:
(Y object) value of counter before update: 0
(Y object) updating value of counter to: 1
(Y object) updated value of counter: 1
...
(Y object) value of counter before update: 7
(X object) value of counter before update: 2
(X object) updating value of counter to: 3
(X object) updated value of counter: 3
...
```

Listing 3.4: Output message of an unsynchronized threaded access of a shared object indicating a data race.

Hanging Threads in synchronized Methods

Problems with synchronized methods and synchronized blocks appear in context of hanging threads. As described in chapter 2.1.2, synchronized methods and synchronized blocks are locked by a thread before the access, while only one thread can acquire a lock at the same time. Other threads will wait until the lock is released before entering the synchronized method or the synchronized block. Now the major issue is a hanging thread that owns a lock. Since the hanging thread can never release the acquired lock, a deadlock appears that may lead to a hanging application. A hanging thread can arise from suspending an active thread, for instance. Although methods that result in a hanging thread are officially deprecated (refer to [Ora11b]), their execution is not prohibited. In order to show that this issue is not resolved in JVM Servers, an example of a suspended thread similar to the example from [Mue05, ch. 4.4] has been applied to a JVM Server. The basic scenario of this test is identical to the previous example (refer figure 3.2), where a shared object of `ZClass` is accessed by two threads, `XClass` and `YClass`, both of which are located in different bundles. In this example however, `ZClass` implements synchronization and suspends the first thread (`XClass`) that acquires a lock (refer listing 3.5 for source code of

the modified `ZClass`). The result of an execution is a deadlock due to the fact that the `YClass` thread remained within **blocked** state until the JVM Server has been restarted - no further processing was carried out by the transaction. Refer figure 3.3, for the graphical interpretation provided by the Java Health Center Thread view.
Note that hanging threads can lead to the same problem when accessing methods of system classes because these are synchronized as well (refer [Mue05, ch. 4.4]). Therefore, the access and update of the file separator system property shown in I_1 (refer chapter 3.4.1) is potentially dangerous if it is carried out by a thread.

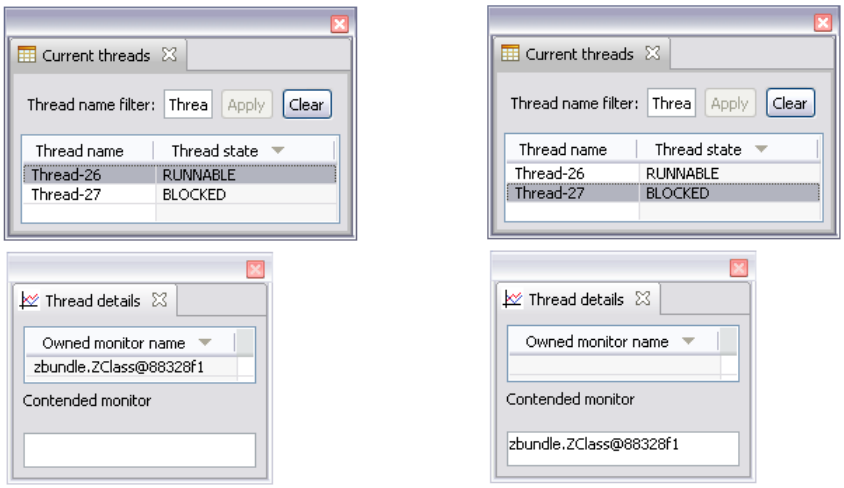
```
package zbundle;

public class ZClass {

        private int counter=0;

        public synchronized int getCounter() {
                Thread.currentThread().suspend();
                return counter;
        }

        public synchronized void setCounter(int x) {
                this.counter = x;
        }
}
```

Listing 3.5: Modified ZClass causing a deadlock due to suspending a thread within a synchronized method.

3.4.3 I_3: The Java Native Interface

As described in [Mue05, ch. 4, p. 32], in some cases it is desirable to use platform specific code for certain tasks. This is often the case if particular tasks cannot be implemented within a Java application. For this purpose Java offers the Java Native Interface (JNI) that according to [Ora11a] "is a standard programming interface for writing Java native methods and embedding the JavaTM virtual machine into native applications". Using JNI one can outsource specific functions into a program written in C for instance, that leads to advanced inter application communication. According to [She99, ch. 1.2, p. 5], "the JNI is a powerful feature that allows you to take advantage of the Java platform, but still utilize code written in other languages ". This feature however, comes not without a price. As described by Liang in [She99, ch. 1.3, p. 6] "while the Java programming language is type-safe and secure, native languages such as C or C++ are not. As a result, you must use extra care when writing applications using the JNI". Moreover, one has to be aware of a major fact. As stated in [She99, ch. 1.1, p. 4] "native applications and native libraries are typically dependent on a particular host environment. A C application built for one operating system, for example, typically does not work on other operating systems."

The following tests aim to show that the JNI omits the **storage protection**



(a) Thread-26 (XClass) holding monitor of ZClass

(b) Thread-27 (YClass) waiting for monitor held by Thread-26 (XClass)

Figure 3.3: Java Health Center Thread Panel view indicating that the XClass thread holds a monitor of ZClass required by the YClass thread. Since the XClass thread is suspended, a deadlock occurred.

and **access level modifier** Java security concepts mentioned in chapter 2.1.3.

Updating private fields out of a C program via JNI. Liang describes in [She99, ch. 4.1, p. 41] the possibility of accessing and updating private instance fields of a Java object from a C program via JNI along with a practical example. This example has been adapted into a JVM Server.

Bundle *Alpha* imports a package from bundle *Beta* that contains the `BetaClass`. This class defines a private string variable `myName` and no methods for the manipulation of this variable. Now, due to the access level modifier security (refer chapter 2.1.3), bundle *Alpha* is not able to modify `myName`. This security feature however can be omitted by using the JNI since the C Program does not follow the security rules defined in Java. Figure 3.4 shows a graphical interpretation of the scenario while the source code of this example is provided in appendix A.3.2. The output generated by bundle *Alpha* shows (refer listing 3.6), that the modification of the private field `myName` of bundle *Beta*, contained in `BetaClass`, was successful.

Although updating private fields can be interpreted as a feature, in terms of application isolation it represents a major issue.
Note that the C program *Backdoor.c* (refer listing A.13 of appendix A.3.2) does not include any specific code that designates the access to a private field. Therefore, a private field modification might be carried out unindented.

The functionality of updating private fields is also provided by the Reflection API. As described in [Ora12] "reflection is commonly used by programs which

require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine." This implies the fact that, unlike to the JNI, the main objective of the Reflection API is the modification of classes and therefore, it is not considered as a cause of potential isolation issues within the scope of this paper.

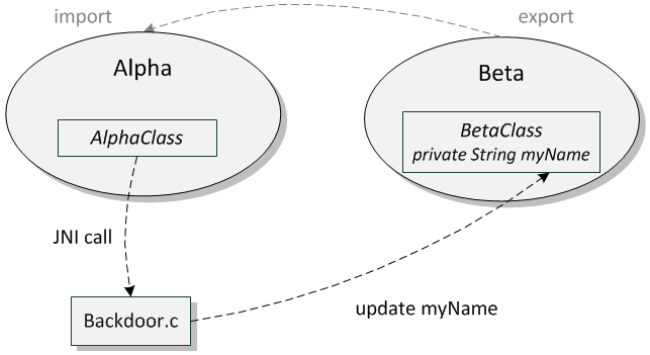


Figure 3.4: Modification of private fields enabled trough the use of the JNI.

```
DELT transaction started
Creating new Beta object
Original private string value of Beta object: BetaObj
Updating private string value of Beta object out of C program via
    the JNI
Updated private field value of Beta object: foobar
```

Listing 3.6: Output message of AlphaClass indicating that the modification to private string value of BetaClass was successful.

Impact of a segmentation fault within the C program. Mueller practically investigated in [Mue05, ch. 4.7] the impact of a segmentation fault caused by a C program, that has been called via JNI out of a stand alone JVM. His result indicates that the JVM unexpectedly shuts down leaving all applications in an undefined state. This scenario has been reproduced in the JVM Server environment using the code provided in appendix A.3.1. The result is an issue as crucial as in a stand alone JVM. Listing 3.7 shows the relevant SDSF log indicating that the JVM Server has been restarted by CICS after the execution of the segmentation fault, while the restart was carried out immediately; with no respect to active transactions. Note that the SDSF log in listing 3.7 does not provide an abend code, leading to the fact, that it would have been difficult to determine the cause of the problem if this error would have appeared in a productive environment.

A similar issue also appears if the C program *Backdoor.c* (refer listing A.13 of appendix A.3.2) from the previous example regarding the private field update does not include the

```
#pragma convert("UTF-8")
```

statement, that tells the C compiler to compile the code in UTF-8 format. In that case the JVM will not be able to interpret the native library and will start an exhaustive search. CICS will interpret this as an infinite loop and therefore will terminate the transaction with an `AKEC` abend code (refer chapter 3.4.4 for explanation) and moreover, restart the JVM Server.

```
JVMSERVER DFH$JVMS is being disabled by CICS because it is in an
    inconsistent state.
IBMUSER JVMSERVER DFH$JVMS is being disabled due to a PHASEOUT
    request.
IBMUSER JVMSERVER DFH$JVMS is disabled.
CICS is enabling JVMSERVER DFH$JVMS after successfully disabling
    the resource.
IBMUSER An attempt to attach to JVMSERVER DFH$JVMS has failed
    because the transaction abended.
Transaction FSF abend ???? in program DFHTFP term CP06. Updates to
    local recoverable resources will be backed out.
START2 JVMSERVER DFH$JVMS is processing any queued OSGi bundles.
START2 JVMSERVER DFH$JVMS is now enabled and is ready for use.
```

Listing 3.7: SDSF log indicating a restart after a segmentation fault caused by a JNI call. No abend code is provided by CICS.

3.4.4 I_4: Resource Exhaustion

In JVM Servers all Java applications share the same processing and memory resources. These applications are not tied to specific restrictions in terms of resource usage. Each application is therefore allowed to use as many resources as needed. Hence, applications that are using resources extensively could have an effect on other applications running within the same JVM. In the worst case these applications can cause a denial of service, that represents a crucial isolation issue.
Missing resource management is a common problem and therefore has been subject to several researches (refer [BD01], [CvE98] and [Yak02]). Since the OSGi framework does not implement a resource manager, all known problems are applicable to a JVM with an implemented OSGi framework. Therefore, Parrend and Frénot [PF09, ch. 3] identified several Java specific vulnerabilities in an OSGi environment, that result in resource exhaustion and that according to [GTM+09, ch. 2, p. 5] arise from the "lack of resource accounting". In detail these vulnerabilities are:

- Memory Exhaustion
- Stand Alone Infinite Loop
- Exponential Object Creation
- Recursive Thread Creation
- Hanging Threads

Note that hanging threads have been discussed in I_2 (refer chapter 3.4.2) and therefore are not mentioned below. Also note that memory exhaustion, recursive

thread and exponential object creation result in the same issue: exceedance of available memory. Therefore, only the recursive thread creation scenario as an example for exceedance of available memory modified after [PF09, ch. 3.2.5, p. 484] has been applied to a JVM Server. In addition, another crucial problem known as memory leaks and an OSGi specific issue similar to exponential object creation outlined in [PF09, ch. 3.2.8, p. 486], defined as "Numerous Service Registration", have been verified within this section as well.

Infinite Loops

Problems with infinite loops in Java have been discussed in [PF09, ch. 3.2.6, p. 484] and in [Mue05, ch. 4.6.2]. The basic issue with loops is that an application cannot be terminated as long as it is still processing the loop; even if the exit condition is faulty and will therefore never occur. Parrend and Frénot [PF09, ch. 3.2.6, p. 484] state that an infinite loop consumes "much of the available CPU". Therefore, this certainly represents an issue in a shared environment. In a JVM Server however, as described in the following, a different issue with infinite loops appears.

CICS includes a feature to detect infinite loops. Once an infinite loop is identified, CICS terminates the transaction with the abend code **AKEC**, that according to [IBM11a, ch. 2, p. 133] describes that "the kernel (KE) domain has detected runaway". Now, this termination implies, similar to a segmentation fault (refer I_3, chapter 3.4.3), an immediate JVM Server restart; with no respect to active transactions. Therefore, the execution of the code shown in appendix A.4.1 will lead to the message shown in listing 3.8 and to a JVM Server restart. Hence, issues resulting from infinite loops are considered as more severe in a JVM Server compared to a stand alone JVM.
Note that, the detection of infinite loops, is not applicable to activator classes, due to the fact, that these classes do not run under the control of CICS. Therefore, infinite loops in Activator classes do not lead to a JVM Server restart (refer I_5, chapter 3.4.5).

Although the detection of infinite loops was found to be accurate, one has to

```
Transaction INFL failed with abend AKEC. Updates to local
recoverable resources backed out.
```

Listing 3.8: SDSF output indicating that the transaction containing the infinite loop has been terminated.

be aware of the fact, that if an infinite loop contains calls to CICS services, the detection will be omitted. This was identified by adding a `println` statement within the infinite loop of the bundle code shown in appendix A.4.1, that displayed the current iteration count on the 3270 terminal emulator. In this case the transaction was not terminated even after millions of iterations. Moreover, the Java Health Center Method Profile view, that monitors the CPU usage per method, reported that the infinite loop consumed a high amount of available CPU resources (refer listing 3.9).

```
The method InfiniteLoop.main() is consuming approximately 22% of
    the CPU cycles. It may be a good candidate for optimization.
```

Listing 3.9: Java Health Center Method Profile view diagnostic message indicating that the infinite loop consumes 24% of the available CPU power.

Recursive Thread Creation

One of the most simple ways of exhausting memory is enabled by recursion. This requires an object or a thread to create multiple instances of itself recursively. The source code of the recursive thread creation example modified after [PF09, ch. 3.2.5, p. 484] is shown in appendix A.4.2, where each thread creates 15 instances of itself recursively, increments a static counter and prints the counter to the standard output. The execution of the example in a JVM Server lead to a `java.lang.StackOverwflowError` exception, indicating that "an application recurses too deeply" [Ora11b]. Although the transaction carrying out the recursive thread creation was terminated, according to the standard output more than 5900 threads have been created before the exception was thrown. As shown in the Java Health Center Garbage Collection view in figure 3.5, this high amount of created threads had an impact on the heap storage, that indicates an increase in heap usage and pause time, representing the time needed for garbage collection. Moreover, the Java Health Center reported the increase of the heap usage as shown in listing 3.10 after the execution of the recursive thread example.

```
Heap usage seems to be growing over time. It increased by 12%
in the last third of the log compared to the middle of the log
    .
The number of collections also increased by 2.838% in response
to the increased pressure on the heap.
```

Listing 3.10: Java Health Center diagnostic message indicating that the heap usage is increasing.

Memory Leaks

> "Because Java uses automatic garbage collection, many think that Java programs are free from the possibility of memory leaks. Unfortunately, this is not necessarily the case. Although automatic garbage collection solves the main cause of memory leaks, you can still have memory leaks in a Java program" [Nyl99].

As explained by Dingle in [Din04, ch. 2], garbage collection in Java is based on the verification if objects are alive. An alive object basically implies other objects referencing to it. "When an object has no more references, the object is a candidate for garbage collection" [GM96, ch. 2.1.6, p. 24]. According to [Din04, ch. 2, p. 10] "herein lays the problem: a live object can contain, possibly unwittingly, a reference to a dead object". Therefore, Java's garbage collection

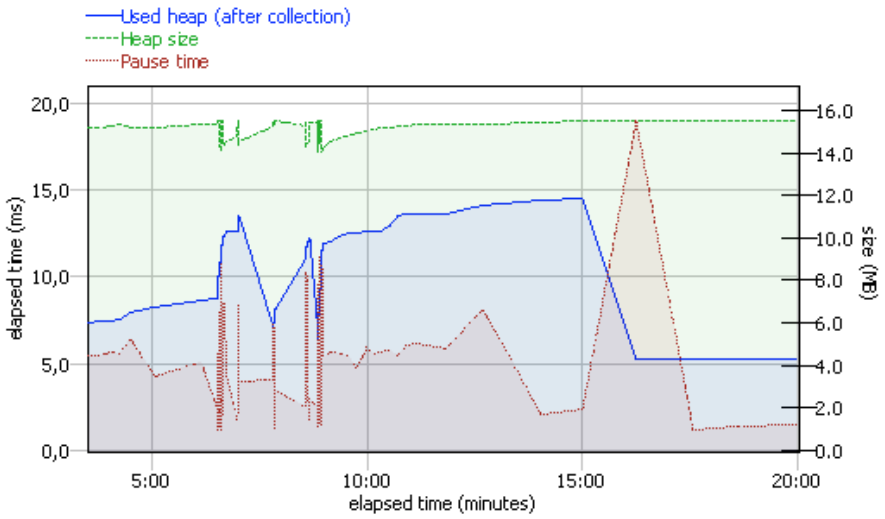


Figure 3.5: Java Health Center Garbage Collection View chart indicating an increase in used heap space and paused times due to recursive thread creation.

does not guarantee a program to be free of memory leaks. The effects of a memory leak is tremendous: an application that includes a memory leak can consume most of the available storage, which will not be garbage collected even after the application has finished execution. This implies an even more severe issue than recursive thread creation.

As mentioned in [Kop11a] and [Kop11b], there are many sources for a memory leak such as static objects or class loading. A simple code example for a memory leak adapted from [Fri02, ch. 8] is shown in listing 3.11. The basic aim of the code is to create 100.000 instances of the `MemoryLeak` class, each one containing a character array of 100.000 bytes and each one to be encapsulated within a `List` object. Each `List` object includes a reference to the next `List` object and the previous `List` object and therefore represents a classic Java *list*. Due to garbage collection, it is actually not possible to keep all objects alive in such a list because there are no external references pointing to it. Since, however, the `MemoryLeak` class includes the static variable `top`, that references the latest `List` object, each single object is considered to be alive and will not be removed by the garbage collector. Moreover, since the static variable `top` is not tied to a specific `MemoryLeak` instance, all created objects will be considered to be alive beyond the life time of the initial `MemoryLeak` instance.
In order to avoid a `java.lang.OutOfMemoryError` exception in the practical verification regarding memory leaks in JVM Servers, the source code shown in listing 3.11 has been altered to create not more than four `MemoryLeak` objects, each one holding a char array of the size of one megabyte (refer appendix A.4.4). The result of an execution of the source code is shown as a Java Health Center Garbage Collection chart in figure 3.6. It indicates that the memory is not garbage collected and moreover, that the pause time has increased significantly. In addition, the Java Health Center reported, that the issue might result from

```
// List.java
class List
{
   MemoryLeak mem;
   List next;
}

class MemoryLeak
{
   static List top;

   char [] memory = new char [100000];

   public static void main (String [] args)
   {
      for (int i = 0; i < 100000; i++)
      {
            List temp = new List ();
            temp.mem = new MemoryLeak ();
            temp.next = top;
            top = temp;
      }
   }
}
```

Listing 3.11: geoff]Example of a Memory Leak in Java. Source: [Fri02, ch. 8]

a memory leak as shown in listing 3.12.

Note that executing the transactions, that carries out the memory leak source

```
Heap usage seems to be growing over time. It increased by 14%
...
The number of collections also increased by 2.006% in response
to the increased pressure on the heap
...
If you don't know of a reason why the memory requirements of
your application should be growing, your application may be
leaking memory.
```

Listing 3.12: Java Health Center diagnostic message indicating that an application using the heap might leak memory.

code multiple times will lead to a `java.lang.OutOfMemoryError` and to a JVM Server restart. This however, is not obvious at first due to the reason that CICS will terminate the transaction with the message shown in listing 3.13, that does not provide an abend code. In the standard error output however a message will be displayed that indicates the OutOfMemoryError as the source of the problem (refer listing 3.14).

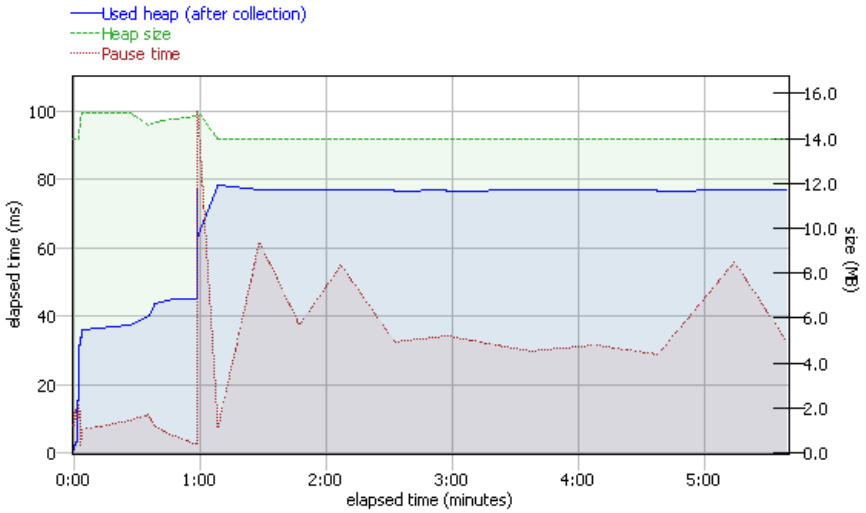


Figure 3.6: Java Health Center Garbage Collection view chart indicating an increase in used heap space and paused times due to recursive thread creation.

```
CICS1 Transaction LEAK failed with abend ??. Updates to
  local recoverable resources backed out.
```

Listing 3.13: CICS output message indicating that the memory leak transaction has been terminated. No abend code is provided.

Infinite Service Registration

As described in [PF09, ch. 3.2.8, p. 486], the "registration of a high number of (possibly identical) services through a loop [...] [makes] the whole platform freeze in the Concierge implementation". This has been proven to be applicable to the Equinox OSGi implementation in JVM Servers as well. The execution of the source code shown in appendix A.4.3, that aims to register identical services within an infinite loop, leads to a hanging JVM Server and therefore, to a denial of service until the heap storage exceeds the maximum amount and a `java.lang.OutOfMemoryError` exception is thrown or until the infinite loop will be terminated by CICS with an **AKEC** abend code. The Java Health Center Garbage Collection view chart shown in figure 3.7, indicates the increase in consumed memory cause by the enormous amount of service registrations.

```
... Processing dump event "systhrow", detail "java/lang/
    OutOfMemoryError" ...
```

Listing 3.14: Standard error output indicating that a java.lang.OutOfMemoryError exception is the cause of the JVM Server restart.

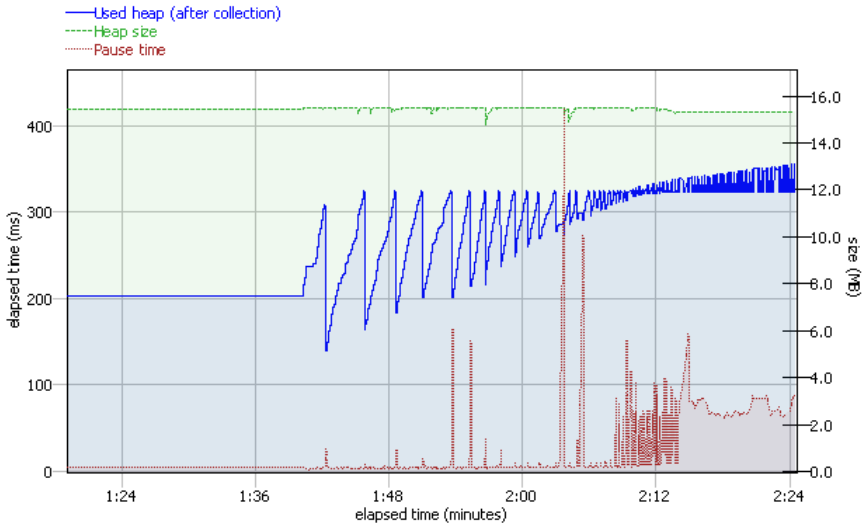


Figure 3.7: Java Health Center Garbage Collection view chart indicating an increase in used heap space and paused times due to infinite service registration.

3.4.5 I_5: Issues with Activator Classes

Parrend and Frénot describe in [PF09, ch. 3.2.3] two issues that can appear within the bundle activator and that can effect the entire OSGi framework: an infinite loop and a hanging thread. According to [PF09, ch. 3.2.3, p. 483] an infinite loop "freezes the process that has launched the starting of the bundle [the OSGi thread], and consumes most of the available CPU" and a "hanging thread in the Bundle Activator makes the management utility freeze".

As indicated in chapter 2.4, Activator classes are not mapped to a CICS TCB and therefore run as an application-spawned thread [Bre11]. This leads to the fact, that one cannot use CICS services within Activator classes and moreover, that infinite loops within an Activator are not detected by CICS.

As stated in [IBM11d, ch. 3, p. 29] "CICS has a timeout that specifies how long to wait for these classes to complete before continuing to start or stop the JVM Server". This timeout however, is not applied for the start/stop procedure of bundles. Due to this reason infinite loops and hanging threads in activator classes represent a problem in JVM Servers as well. Moreover, since the CICS Explorer carries out an installation/deinstallation of a bundle together with its activation/deactivation, the installation of a bundle will already lead to the problem with the management utility. With the code shown in appendix A.5 an infinite loop (refer to appendix A.5.1) and a hanging thread (refer to appendix A.5.2) in the activator class were applied to a JVM Server. Unlike as outlined in [PF09, ch. 3.2.3, p. 483], the installation of both bundles led to the same problem: the freeze of the management utility. Consequently, it was not possible to communicate with the CICS region using the CICS Management Interface as long as the activation of the bundle was processed. Although this problem had

no effect on other bundles, due to the reason that all bundle were active and all transactions were accessible through the 3270 terminal emulator, in both cases a CICS region restart was the only solution for regaining control. Breitbach [Bre11] even "strongly discourages" using Activator classes in JVM Servers because they "do not run on a T8 TCB" but on an application-spawned thread (refer chapter 2.4). Therefore, it is advised that, except of special cases, such as for security bundles responsible for defining permissions, Activator classes should not be used in JVM Servers.

Note that if an infinite service registration (refer I_4, chapter 3.4.4) is carried out within an Activator class, resource exhaustion will appear in combination with a hanging management utility.

3.4.6 I_6: Illegal Control

OSGi Bundle Context

The OSGi framework offers several features of managing bundles. This is enabled by the `BundleContext` object of the OSGi framework API. According to [OSG11b, ch. 4.3, p. 99]

> "the relationship between the Framework and its installed bundles is realized by the use of BundleContext objects. A BundleContext object represents the execution context of a single bundle within the OSGi Service Platform, and acts as a proxy to the underlying Framework. A BundleContext object is created by the Framework when a bundle is started."

Moreover, the OSGi Specification states in [OSG11b, ch. 4.3, p. 99], that the BundleContext can be used among others for "installing new bundles" and for "interrogating other bundles installed in the OSGi environment". Therefore, using the bundle context one can access various information of bundles such as their install location and moreover, one can start, stop, install and uninstall bundles using the corresponding methods provided by the `BundleContext` class (refer [OSG11b, ch. 9.1.7]).

Although useful, this feature implies an isolation issue since by default there are no restrictions for illegal control. Parrend and Frénot [PF09, ch. 3.2.7, p 485] refer to this as the "Pirate Bundle Manager" vulnerability, that is defined as "a bundle that manages others without being requested to do so".

In general the bundle context is passed to each bundle within the `start` and `stop` methods of the `Activator` class. Therefore, the most simple way for carrying out illegal control is by applying the particular code within the `Activator`. A practical example for illegal control using the bundle context is shown in appendix A.6.1, where a bundle *Alpha* aims to stop and uninstall the bundle *Beta*. Since Activator classes can lead to other issues (refer I_5, chapter 3.4.5), the source code shown in appendix A.6.1 implements an option of accessing the BundleContext without an `Activator` class, that is enabled by the `FrameworkUtil` class of the OSGi framework API (refer [OSG11b, ch. 3.9.9, p. 60]).

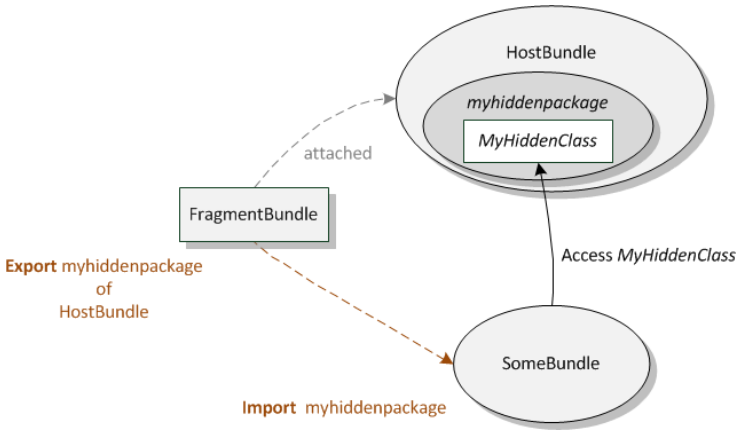


Figure 3.8: Unintended package export enabled through bundle fragment.

The execution of the code in appendix A.6.1 leads as expected to the uninstallation of the bundle *Beta*. Since this action was not carried out by the CICS Explorer, the state change of the bundle was not visible within the CICS Explorer Management Interface. This implies, that bundle control via the bundle context is out of the scope of the CICS Explorer

OSGi Bundle Fragments

An issue related to illegal control is enabled by the fragment functionality provided in OSGi frameworks. According to the OSGi specification [OSG11b, ch. 3.14, p. 69] "fragments are bundles that can be attached to one or more host bundles by the Framework". They basically represent a method for replacing particular functionalities of bundles without replacing the entire bundle. "A key use case for fragments is providing translation files for different locales. This allows the translation files to be treated and shipped independently from the main application bundle" [OSG11b, ch. 3.14, p. 69]. The OSGi specification [OSG11b, ch. 3.14, p. 70] also states that "when a fragment bundle is attached to a host bundle, it logically becomes part of it. All classes and resources within the fragment bundle must be loaded using the class loader (or Bunde object) of its host bundle". This implies that a fragment can override functions of the host and since the host bundle is not aware of any fragments attached to it, this feature can be used for illegal control. In this context Parrend and Frénot [PF09, ch. 3.2.8, p. 486] describe the following scenario: "A fragment bundle exports a package from its host that this latter does not intend to make visible. Other bundles can then execute classes in this package". Figure 3.8 shows a graphical interpretation of this issue, where *FragmentBundle* exports the package *myHiddenPackage* of *HostBundle*, that was not intended to be exported by *HostBundle*. Appendix A.6.2 shows an implementation along with the manifest files of each bundle of this issue. The execution of the application implies, as expected, that *SomeBundle* can access `MyHiddenClass` with no restrictions.

Chapter 4

Approaches to Solving Open Issues

This chapter provides a list of approaches to solutions of open the isolation issues outlined in chapter 3. Since most issues can be resolved by using built-in solutions offered in OSGi and by using tools offered for JVM Servers, proprietary solutions that require the extension of the JVM or the OSGi framework are not discussed in detail.

4.1 Static Program Analysis

> "Static analysis concerns techniques for obtaining information about the possible states that a program passes through during execution, without actually running the program on specic inputs. Instead, static-analysis techniques explore a programs behavior for all possible inputs and all possible states that the program can reach."[RSW04]

An approach of solving particular issues of the Java programming language was found to be automatic code analysis using specific tools. Since issues as infinite loops are common mistakes, most static program analysis utilities are capable of detecting these in most cases.
This section introduces two utilities, one for the identification of accesses to static fields and one infinite loops within program code.

4.1.1 Identifying the Access to Static Fields

Since the state of the JVM is represented by static fields, one needs to make sure that an application active within a JVM Server always resets modifications to these static fields. IBM offers a utility, the CICS JVM Application Isolation Utility, that helps to identify if an application is accessing static fields or synchronized static methods of system classes. According to [IBM11c, ch. 28, p. 214] "the CICS JVM Application Isolation Utility is a code analyzer tool, which inspects Java bytecodes in Java Archive (JAR) files and class files". The utility

is a Java application that is executed out of a USS terminal session. The outcome of its application to the source code from appendix A.1.1, where bundle A is changing the file separator system property is shown in listing 4.1. It indicates that the utility has identified the execution of the methods `getProperty` and `setProperty`.

Since the identification by the application isolation utility regarding the ac-

```
CicsIsoUtil: CICS JVM Application Isolation Utility
Copyright (C) IBM Corp. 2007

Reading jar file: A_1.0.0.jar

  Class: a.Change

    Method: public static void main(com.ibm.cics.server.
        CommAreaHolder)
      Static methods invoked by this method:
        com.ibm.cics.server.Task getTask()
        (defined in class: com.ibm.cics.server.Task)
        java.lang.String getProperty(java.lang.String)
        (defined in class: java.lang.System)
        java.lang.String setProperty(java.lang.String, java.lang.
            String)
        (defined in class: java.lang.System)

  Number of methods inspected       : 2
  Total static writes for this class: 0

Number of classes inspected: 2
End of jar file: A_1.0.0.jar

Number of jar files inspected    : 1
Number of class files inspected  : 0
```

Listing 4.1: Output of the CICS Application Isolation Utility applied to the source code of bundle A, that aims to change a static field of a system class (refer to appendix A.1.1). The output indicates that the Change.class calls the getProperty and the setProperty methods of the java.lang.System class.

cess to static fields is not limited to system classes, the utility can be applied verify static field access of non system classes as well. Listing 4.2, shows the output of the application isolation utility applied to the source code provided in appendix A.2.1, where `XClass` located in bundle X updates a static field from `AClass` located in bundle A. It indicates that the update to the static field in `AClass` has been identified by the application isolation utility. Therefore, applying the utility to all bundles prior to their installation into a JVM Server is mandatory in order to prevent issues resulting from static fields.

4.1.2 Identifying Infinite Loops

As mentioned in I_4 (refer chapter 3.4.4), although CICS is able to identify infinite loops within transactions, the result of an identification leads always to a JVM Server restart. Therefore, it is more appropriate to carry out particular

```
CicsIsoUtil: CICS JVM Application Isolation Utility

Copyright (C) IBM Corp. 2007

Reading jar file: overlap_XBundle_3.0.0.jar

  Class: X.XClass

    Method: public static void main(com.ibm.cics.server.
        CommAreaHolder)
      Static fields written in this method:
        java.lang.String myName
        (defined in class: A.AClass)
      Static methods invoked by this method:
        com.ibm.cics.server.Task getTask()
        (defined in class: com.ibm.cics.server.Task)

  Number of methods inspected        : 2
  Total static writes for this class: 1

Number of classes inspected: 2
End of jar file: overlap_XBundle_3.0.0.jar

Number of jar files inspected    : 1
Number of class files inspected  : 0
```

Listing 4.2: Output of the CICS Application Isolation Utility applied to the source code of bundle X, that aims to change a static field of a non system class (refer appendix A.2.1). The output indicates that the XClass is updating a static field from AClass.

identifications of infinite loops prior to running an application within a JVM Server. An interesting approach for the detection of infinite loops and beyond is represented by the FindBugs[1] tool (refer [APM+07]), that includes "nealry 300 different bug patterns" [APM+07, ch. 2, p. 1] for the detection of several common problems. Moreover, since the utility is provided as a Eclipse Plug-In, it can be integrated into the CICS Explorer environment.
In order to show that infinite loops can be identified using static code analysis, the FindBugs utility has been applied to the bundle containing the infinite loop (refer appendix A.4.1) discussed in I_4 (refer chapter 3.4.4). The output provided in listing 4.3 clearly indicates that the infinite loop has been detected by the utility.
Note that although static program analysis is reliable in most cases, there is no guarantee that actual bugs will be detected due to the fact that certain bugs might vary and therefore could require a new bug pattern for their identification.

4.2 Java and OSGi Security

As discussed in chapter 2.1.3, the Java Security Manager, that includes several permissions, represents a fine-grained security concept. Therefore the installation of the Security Manager and its customization for the purpose of resolving

[1] http://findbugs.sourceforge.net/

```
Bug: There is an apparent infinite loop in infloop.InfiniteLoop.
    main(CommAreaHolder)

This loop doesn't seem to have a way to terminate (other than by
    perhaps throwing an exception).

Confidence: High, Rank: Scary (8)
Pattern: IL_INFINITE_LOOP
Type: IL, Category: CORRECTNESS (Correctness)
```

Listing 4.3: FindBug output of application to the infinite loop bundle shown in appendix A.4.1.The output indicates that FindBug has identified the infinite loop.

isolation issues is described in the following. Moreover, the OSGi security layer, which is a more powerful approach compared to the standard Java Security Manager, will be discussed in detail within this section.

4.2.1 Customizing the Java Security Manager

As indicated in chapter 2.1.3, the Security Manager represents a Class that includes a number of customizable methods. By default these methods inlcude simple permission checks that are implemented within the Access Controller (refer chapter 2.1.3). Since permissions are defined within a *policy* file, that represents a white list, by default no permissions are granted. Therefore, stopping threads (refer I_1, chapter 3.4.1) is not allowed by default once a Security Manager is active. This however does not apply to the `System.exit` and the `Runtime.halt` methods. As stated in [Ora11b], "the "exitVM.*" permission is automatically granted to all code loaded from the application class path, thus enabling applications to terminate themselves". Since the JVM Server is a multi-application VM, this fact represents a problem. Hence, in order to revoke the granted permission of exiting the VM, one needs to customize the existing Security Manager by overwriting the `checkExit` method of the `SecurityManager` class. In addition one needs to create an instance of the customized Security Manager and install it by executing the method `System.setSecurityManager`. Since the Security Manager is installed globally, activating the customized Security Manager be implemented within a separate bundle.

An example of a customized Security Manager is shown in listing 4.4. Note that since `MySecurityManager` extends from the `SecurityManager` class, all unimplemented methods are adapted from the default Security Manager.
With the customized Security Manager shown in listing 4.4 active within the JVM Server, the transactions accessing the source code shown in appendix A.1.2 and appendix A.1.3, that both aim to shutdown the VM, will be aborted with the abend code **AJ05**. This indicates according to [IBM11a, p. 127] that "an unhandled exception has been caught by the Java environment" - in this case the security exception stating ****Not allowed to stop the VM**** (refer listing 4.4).

Due to the fact that suspending threads can be prevented by the Security Man-

ager, the problem of stopping applications discussed in I_1 (refer chapter 3.4.1) is also reduced if a Security Manager is active. Since hanging threads do not only result from the use of the deprecated `suspend` method but also from other issues such as unexpected shutdowns of (sub)systems or faulty thread implementations as shown in [Blo05, ch. 9, Puzzles 77 and 85], the issues resulting from hanging threads can not be resolved completely with built-in solutions. However, due to the fact that the Java Health Center provides a Thread tiew, it can be used to identify hanging threads (refer chapter 4.3).
The problems arising from wires are more difficult to prevent since wires actually represent a feature, that enables inter-bundle communication. A built-in solution for restricting wires is represented by the security layer of the OSGi framework. According to [OSG11b, ch. 2.2, p. 11] "the Framework security model is based on the Java 2 specification". Moreover, it includes a set of OSGi specific permissions such as `org.osgi.framework.PackagePermission` that enables to control the export and import functionality of bundles.

```
public class MySecurityManager extends SecurityManager{

        public void checkExit(int status){
                throw new SecurityException("***Not allowed to stop
                     the VM***");
        }
}
```

Listing 4.4: Customized Security Manager that throws an exception once an application attempts to stop the VM.

4.2.2 Using OSGi Security

A more suitable option for security than customizing the Java Security Manager is represented by the OSGi security layer. It basically includes its own Security Manager where executing `System.exit()` is prohibited by default. Moreover, it extends the Security Manager by several OSGi specific permissions as

- **AdaptPermission**, that defines "a bundle's authority to adapt an object to a type" [OSG11b, ch. 9.1.1, p. 163].
- **CapabilityPermission**, that defines "a bundle's authority to provide or require a capability" [OSG11b, ch. 9.1.13, p. 195], while capabilities are defined as "attribute sets in a specific name space" [OSG11b, ch. 3.3, p. 32].
- **ServicePermission**, that defines "a bundles authority to register or get a service" [OSG11b, ch. 9.2.26, p. 228].
- **BundlePermission**, that defines "a bundle's authority to require or provide a bundle or to receive or attach fragments" [OSG11b, ch. 9.1.11, p. 193], while requiring a bundle implies that "the framework must take all exported packages from a required bundle, including any packages" [OSG11b, ch. 3.13.1, p. 66].

- **PackagePermission**, that defines "a bundle's authority to import or export a package" [OSG11b, ch. 9.1.21, p. 222].
- **AdminPermission**, that defines "a bundle's authority to perform specific privileged administrative operations on or to get sensitive information about a bundle" [OSG11b, ch. 9.1.3, p. 159].

With these fine-grained permissions most issues related to overlapping namespaces (refer I_2, chapter 3.4.2) and illegal control (refer I_6, chapter 3.4.6) can be resolved. Therefore, using OSGi security is mandatory when aiming to develop isolated applications.
OSGi offers two different services for the management of permissions:

1. The **Permission Admin** service and
2. The **Conditional Permission Admin** service.

Since as outlined in [OSG11b, ch. 51, p. 291] "the Permission Admin has been superseded by the Conditional Permission Admin", the Permission Admin Service is not explained in the following.
In order to activate the OSGi security layer in a JVM Server one needs to activate the Java Security Manager as well. Therefore, as described in [IBM11d, ch. 5, p. 88] activating the OSGi security layer in JVM Servers is achieved by adding the lines

```
org.osgi.framework.security=osgi
-Djava.security.policy=/u/policies/all.policy
```

into the JVM profile. The second line is used to grant all permissions to the JVM. Hence, as mentioned in [IBM11d, ch. 5, p. 88], the *all.policy* file includes only the following content

```
grant {
      permission java.security.AllPermission;
};
```

Local OSGi Security

A simple solution to activate the OSGi security layer is enabled by *permissions.perm* files located in the bundle, that include all permissions needed for the particular bundle, and as described in [HPM10, ch. 14.6], is located within the *OSGI-INF* folder of the bundle. In order to activate the OSGi security layer for particular bundles one needs to create an *OSGI-INF* folder that includes a *permissions.perm* file containing all permissions granted to the bundle. Hence, for a bundle to be able to import packages, for instance, the *permissions.perm* file should contain the following line

```
(org.osgi.framework.PackagePermission "*" "import")
```

while the wildcart represents that all packages can be imported by the bundle. Due to the simple activation and due to the fine-grained permissions, local OSGi security represents a powerful approach towards application isolation. Since, however, the security layer has no effect on bundles that do not include a *permissions.perm* file, they are granted all permissions, local security might not be a suitable solution in specific cases for isolating applications. Therefore, a global approach of applying security is explained in the following.

Global OSGi Security

A more sophisticated approach towards application isolation and security in OSGi bundles compared to local security is represented by the *Conditional Permissions Admin Service*. According to [OSG11b, ch. 50.1.3, p. 257]

> "a Conditional Permission Admin service maintains a system wide ordered table of ConditionalPermissionInfo objects. This table is called the policy table. The policy table holds an encoded form of conditions, permissions, and their allow/deny access type. A manager can enumerate, delete, and add new policies to this table via a ConditionalPermissionsUpdate object."

One of the most important features of the Conditional Permission Admin is that it "allows you to grant permissions based on arbitrary conditions. A condition acts as a Boolean guard that determines whether a permission group is applicable" [HPM10, ch. 14.4]. This feature enables a very fine-grained permission management based on particular constraints. Moreover, the Conditional Permission Admin includes an access parameter that enables to define if a permission is granted or denied (refer [OSG11b, ch. 50.2.5, p. 261]). This enables to define black lists of permissions, which is not possible within the *policy* file used by the Security Manager and the *permissions.perm* file used for local OSGi security.
By default, the Conditional Permission Admin implements two different conditions:

- the Bundle Signer Condition (refer [OSG11b, ch. 50.9.1]) and
- the Bundle Location Condition (refer [OSG11b, ch. 50.9.2].

There is, however, an option of defining customized conditions as explained in [OSG11b, ch. 50.8]. It implies to define a condition as a set of checks with a boolean return value within a class that implements the `Condition` interface provided by the OSGi framework.
As described in [HPM10, ch. 14.4], there are three steps for using the Conditional Permission Admin within an OSGi framework:

1. Acquire a a reference to the Conditional Permission Admin Service
2. Grant all permissions to the current bundle
3. Specify global permissions.

An example for these three steps is shown in listing 4.5.

```
/* 1. Step: Get the Conditional Permissions Admin service */

ServiceReference cpaRreference = context.getServiceReference(
    ConditionalPermissionAdmin.class.getName());
ConditionalPermissionAdmin conditionalPermissionAdmin = (
    ConditionalPermissionAdmin) context.getService(cpaRreference);


/* 2. Step: As security agent attempt to grant all permissions to
    the current bundle */
```

```
ConditionInfo conditionInfo = new ConditionInfo(
    BundleLocationCondition.class.getName(), new String[] { context
    .getBundle().getLocation() } ); PermissionInfo permissionInfo =
     new PermissionInfo(AllPermission.class.getName(), "", "");

ConditionalPermissionInfo allPermissions =
    conditionalPermissionAdmin.newConditionalPermissionInfo("agent
    ", new ConditionInfo[] { conditionInfo }, new PermissionInfo[]
    { permissionInfo }, ConditionalPermissionInfo.ALLOW);


/* 3. Step: Specify global permissions */

String encodedPermission=new String("allow { [org.osgi.service.
    condpermadmin.BundleLocationCondition \"file:/usr/lpp/cicsts/
    cicsts42/lib/*\"] (java.security.AllPermission) } \"CICS\"");

ConditionalPermissionInfo allowCICS = conditionalPermissionAdmin.
    newConditionalPermissionInfo(encodedPermission);

/* Update the permissions table */
ConditionalPermissionUpdate permissionUpdate =
    conditionalPermissionAdmin.newConditionalPermissionUpdate();
List infos = permissionUpdate.getConditionalPermissionInfos();
infos.clear();
infos.add(allPermissions);
infos.add(allowCICS);
permissionUpdate.commit();
```

Listing 4.5: Example of using the Conditional Permission Admin. Modified after the CICS Explorer Security Example.

The basic procedure for adding new permissions using the Conditional Permission Admin is as follows

1. Create a *ConditionInfo* object, that describes the condition
2. Create a *PermissionInfo* object, that describes the permission
3. Create a *ConditionalPermissionInfo* object, that as described in [OSG11b, ch. 50.14.5.5, p. 283] receives the following parameters
   (a) a string describing the name that represents "a unique key to identify the entry" [HPM10, ch. 14.4],
   (b) the *ConditionInfo* object,
   (c) the *PermissionInfo* object and
   (d) a string naming the access parameter, also referred to as "access decision" in [OSG11b, ch. 50.14.5.5, p. 283], that as outlined in [HPM10, ch. 14.4] can either be *ConditionalPermissionInfo.ALLOW* or *ConditionalPermissionInfo.DENY*.
4. Get a *ConditionalPermissionUpdate* object
5. Create a list and save all permissions active of the policy table into the list by using the `getConditionalPermissionInfos()` method of the *ConditionalPermissionUpdate* object
6. Clear all entries within the list (optional step)

7. Add the created *ConditionalPermissionInfo* object into the list

8. Execute the `commit()` method of the *ConditionalPermissionUpdate* object to save permissions into the policy table

Note that first two steps can be omitted when adding encoded conditional permissions, that as described in [OSG11b, ch. 50.14.6.7, p. 286] have the following format

```
access {conditions permissions} name
```

An encoded conditional permission to grant all permissions to CICS specific bundles, that are located in the directory `/usr/lpp/cicsts/cicsts42/lib/` is shown in listing 4.6.

```
String encodedPermission=new String("allow { [org.osgi.service.
    condpermadmin.BundleLocationCondition \"file:/usr/lpp/cicsts/
    cicsts42/lib/*\"] (java.security.AllPermission) } \"CICS\"");
```

Listing 4.6: Encoded conditional permission to allow the import of bundles.

Note that this specific permission is mandatory for the activation of the OSGi security in JVM Servers due to the fact, that without all permissions granted to the CICS bundles, no management of OSGi bundles would be possible.

The code shown in listing 4.5 can be added into an Activator class and the corresponding bundle can be uploaded and installed as a "normal" CICS bundle within the JVM Server. Once installed, all actions carried out by bundles active within the same JVM Server will be verified based on the policy table prior to their execution.
A more suitable approach of installing a bundle that includes global security permissions, referred to as the *security bundle* in the following, is described in [IBM11d, ch. 5, p. 88]. It implies to install the security bundle as a "middleware bundle" that according to [IBM11d, ch. 6, p. 126], is "installed in the OSGi framework during the initialization of the JVM Server". This implies, that the security bundle does not need to be included within a CICS bundle and moreover, that it is not managed by the CICS Explorer Management Interface. Therefore, the explicit OSGi specific installation of middleware bundles using the CICS Explorer Management Interface is not necessary.
As described in [IBM11d, ch. 5 , p.88], including a middleware bundle into a JVM Server is simply carried out by adding the line

```
OSGI_BUNDLES=<path to bundle>
```

into the JVM profile.

In order to show that the security bundle restricts the actions that can be carried out, the security bundle containing the code shown in listing 4.5 has been installed into a JVM Server. In addition, the simple bundle shown in listing 4.7, that basically prints a *Hello CICS* statement on the 3270 terminal, referred to as the *printer bundle* in the following, has been installed as well. Due to the fact, that the policy table created in listing 4.5 did not include a permission for allowing to import packages and due to the fact that the *printer bundle* requires

the `com.ibm.cics.server` package to access the 3270 interface, the installation of the printer bundle failed. As shown in listing 4.8 the reason was a *Missing Permission* that can be granted by adding the code shown in listing 4.9 to the security bundle.
Note that besides all the advantages enabled by the OSGi security layer, one specific drawback needs to be considered. As discussed in [HS05], using the Java Security Manager might lead to a performance degradation of the JVM. Although no specific researches related to performance characteristics of the OSGi security layer are known, due to the fact that it includes a Security Manager, similar effects are expected.

```
package printer;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class Printer {

        public static void main(CommAreaHolder CAH) {
                Task t = Task.getTask();

                t.out.println(" Hello CICS");
        }
}
```

Listing 4.7: Hello World application. Modified after the CICS Explorer HelloWorld Example.

```
IBMUSER The CICS resource life-cycle manager has started to
create the BUNDLE resource cicspx.
IBMUSER An exception has been thrown by the route method of
class com/ibm/cicsrouter/Router running in
JVMSERVER DFH$JVMX. Exception 'The bundle "printer_4.0.0 [16]"
could not be resolved. Reason: Missing Permission:
(org.osgi.framework.PackagePermission com.ibm.cics.server
    import),
Missing Constraint: Import-Package: com.ibm.cics.server;
version="0.0.0"'.
An attempt to enable an OSGi bundle in JVM Server DFH$JVMX has
failed. OSGi bundle symbolic name printer, version 4.0.0,
reason code EXCEPTION_FROM_JVMSERVER.
CWWU The CICS resource life-cycle manager has created the
BUNDLE resource cicspx and the BUNDLE is in the disabled state
    .
CWWU BUNDLE cicspx has been installed as disabled because one
or more of its associated resources failed to install.
```

Listing 4.8: SDSF log describing a failed installation of the printer bundle due to the fact that the installed security permissions did not allow to import bundles.

```
String x=new String("allow { (org.osgi.framework.PackagePermission
    \"*\" \"import\") }\"IMPORT ALL\"");
ConditionalPermissionInfo globalImportPermission =
    conditionalPermissionAdmin.newConditionalPermissionInfo(x);
//...
infos.add(globalImportPermission);
//...
```

Listing 4.9: Encoded conditional permission to allow the import of bundles.

4.3 Monitoring

Most causes of resource exhaustion such as over use of memory and computational resources, can be detected with applied monitoring. Therefore, it is mandatory to use diagnostic tools provided for the JVM Server. The Java Health Center for instance, that is enabled by adding the option

`-Xhealthcenter:port=<port>`

into the JVM profile, includes several panels such as

- The Class view, that shows an overview of loaded classes.
- The Garbage Collection view, that graphically gives an overview of the heap and based on diagnostic messages such as shown in listing 3.12, indicates possible memory leaks.
- The Thread view, that list all active threads as well as detailed information such as owned or contended locks.
- The Method Profile view, that shows the computational resources used by each method.

Moreover, monitoring is carried out during runtime of the JVM Server "with a very small impact on the application's performance" [IBM]. Therefore, using the particular information provided by each panel, one is able to identify resource exhaustion issues, such as outlined in I_4 (refer chapter 3.4.4), prior to a denial of service. In addition, hanging threads can also be identified as shown in figure 3.3 (refer I_2, chapter 3.4.2). Note that the Java Health Center does not enable to carry out specific administrational operations such as the termination of tasks or deinstallation of bundles. These actions should be performed directly in CICS using a 3270 terminal emulator or the CICS Explorer Management Interface.
The Java Health Center is a local software that is connected to a JVM Server. Therefore, monitoring requires a particular amount of resources, especially for information exchange between the remote and the local system. Since some resource exhaustion issues result in massive denial of service, where no resources for the Java Health Center communication might be left, real time monitoring is not always the most suitable solution for the identification of causes for resource exhaustion. In this case offline monitoring tools, that enable to inspect the heap or the entire JVM coredump after the JVM Server was shutdown, should be considered. The tools found to be most suitable for memory analysis of the JVM

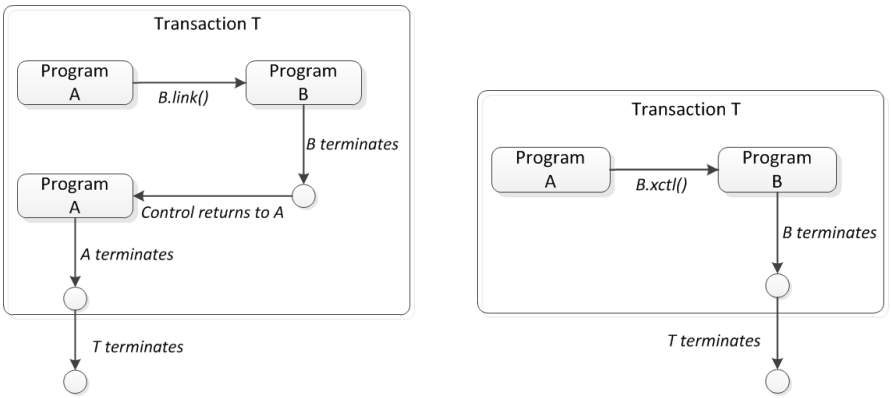
Server are the *Memory Analyzer*[2], that enables among others the identification of memory leaks and the *Garbage Collection and Memory Visualizer*[3], that inspects among others the *verbose garbage collection* logs enabled by adding `-verbose:gc` within the JVM profile of the JVM Server [IBM11d, ch. 7, p. 160]. For detailed dump analysis the *Dump Analyzer*[4] represents a suitable offline analysis tool.

4.4 CICS Services for Program Control

As outlined in issue I_3, chapter 3.4.3, the use of the JNI can lead to severe problems, that can effect other applications active within the same JVM Server, while the identification of possible causes responsible for errors is not straightforward due to the missing abend code for segmentation faults for instance (refer chapter 3.4.3). Moreover, since the C program called out of a Java application is not mapped to a CICS TCB but runs as a USS thread, all executions carried out by the C program are not synchronized with other CICS applications. Therefore, it is not advised to use the JNI in a JVM Server environment.
In order to execute C programs out of a Java application running within a JVM Server however, one can make use of the CICS services for program control provided within the JCICS library. These allow to call CICS programs defined within the same CICS region out of a program; regardless the programming language used for their creation. Therefore, one can call a C program out of a Java application running within a JVM Server and omit the use of the JNI. Since all CICS programs are mapped to a CICS TCB, the called program remains under the control of CICS, that implies the advantages mentioned in chapter 2.3 such as storage protection and synchronization. Moreover, the identification of errors is less difficult since abend codes are provided for most errors that might appear during program execution (refer [IBM11a]).
CICS provides two services for program control: *link*, and *xctl*. As explained in [RBB+11, ch. 9.3.2, p. 289], using *link* implies that the calling program remains control while using *xctl* leads to handing over control to the called program. A graphical interpretation of differences between *link* and *xctl* is shown in figure 4.1. For a detailed instruction including code examples regarding the JCICS services *link* and *xctl*, it is referred to [Hus11].
Apart from program control, CICS enables options for the exchange of information between programs such as the *COMMAREA*. According to [RBB+11, ch. 9.22.1] "a COMMAREA is a facility that transfers information between two programs within a transaction or transfers information between two transactions from the same terminal." Therefore, by using program control services with the *COMMAREA* as the preferred option of inter application communication, one can omit the creation of wires and therefore, resolve most problems arising from overlapping namespaces (refer I_2, chapter 3.4.2). Note that this communication technique is somewhat similar to communication between applications running within different JVMs, where the Java Remote Method Invocation (RMI) is used for information exchange. In addition to the COMMAREA, CICS offers several data management services that are useful for global information storage.

[2] http://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/
[3] http://www.ibm.com/developerworks/java/jdk/tools/gcmv/
[4] http://www.ibm.com/developerworks/java/jdk/tools/dumpanalyzer/

In context of these, CICS "implements two proprietary file structures, and provides commands to manipulate them" [IBM11d, ch. 3, p. 27]: the **temporary storage** (TS) and the **transient data** (TD), both of which are "making data readily available to multiple transactions" [IBM11d, ch. 3, p. 27]. While the TS "queues can reside in main memory, or can be written to a storage device" [IBM11d, ch. 3, p. 27], TD "queues are always written to a data set" [IBM11d, ch. 3, p. 27].



(a) Program control flow with *link* indicating that control returns to program A after program B terminates.

(b) Program control flow with *xctl* indicating that control is not returned to program A after program B terminates.

Figure 4.1: Differentiation of program control flow. Modified after [Hus11, ch. 3.2.1, p. 20].

4.5 Extensions of the OSGi Framework

Several JVM and OSGi implementations have been developed in order to resolve isolation issues of Java and the OSGi framework. Mueller for instance outlined in [Mue05] more than ten proprietary solutions for enabling application isolation in JVMs. Most of the mentioned solutions however, are either discontinued, as the Echidna library and the Persistent Reusable Virtual Machine (resettable JVM mode, refer [IBM01] and chapter 2.3.2) or outdated as the JKernel[5] and the Application Isolation API [Jav06] implemented in the Multitasking Virtual Machine (refer [CD01]). Therefore, this section aims to outline recent (proprietary) solutions for application isolation in JVMs.

4.5.1 I-JVM

The I-JVM, introduced in [GTM+09], represents a modification of the JVM to enable isolation, resource accounting and OSGi bundle termination. According to [GTM+09, ch. 3, p. 6], "each bundle is executed within a separate isolate". Moreover, communication between isolates is achieved by "inter-isolate method calls" [GTM+09, ch. 3, p. 6]. Therefore, according to [GD10, ch. 3, p. 4]

[5] http://www.cs.cornell.edu/slk/JKernel/Default.html

the I-JVM implements concepts from the Java Application Isolation API (refer [Jav06]). In addition, the I-JVM implements a resource accounting monitor that keeps track of all resources used by single isolates or bundles and a functionality for safe bundle termination. Geoffray et. al. [GTM+09] proved that the I-JVM solves problems arising from static fields of system classes (refer I_1, chapter 3.4.1) and synchronized methods (I_2, chapter 3.4.2). Moreover it resolves issues of resource exhaustion (refer I_4, chapter 3.4.4).

4.5.2 Hardened OSGi Implementation

Parrend and Frenot recommend in [PF09], that has served as a major reference regarding OSGi vulnerabilities in chapter 3, several extensions of the OSGi framework in order to resolve the vulnerabilities that cannot be resolved by using the security layer. The issues related to application isolation, that have also been discussed in chapter 3 are

- "*Bundle Start Process*: Launch the Bundle Activator in a separate thread" [PF09, ch. 4.1, p. 491], that resolves issues with activator classes (refer I_5, chapter 3.4.5) and
- "*OSGi Service Registration*: set a platform property that explicitly limits the number of registered services (default could be 50)" [PF09, ch. 4.1, p. 491], that resolves the issue of infinite service registration (refer I_4, chapter 3.4.4).

4.5.3 Sandboxed OSGi

For the aim of "the execution of third party code not enough tested or not known in advance, as well as other potentially dangerous code" [GD10, ch. 3], Gama and Didier introduce in [GD09] the *Sandboxed OSGi*, that basically implements the Application Isolation API ([Jav06]) of the Multitasking Virtual Machine (MVM, [CD01]). Therefore, each application in the *Sandboxed OSGi* encapsulated within an *isolate* entity that as described in [Mue05, ch. 6.1] has an own heap and communicates with other isolates through specific methods provided by the Application Isolation API or by RMI.
Using this OSGi framework many issues outlined in chapter 3 are resolved due to the facts that resources are not shared and no overlapping namespaces can occur since communication is enabled only through predefined methods.
Note that this implementation follows the Java specification since only the OSGi framework has been altered and since the MVM was officially introduced by SUN within the scope of the Barcelona Project[6].

4.5.4 OSGi RFC-0138 Multiple Frameworks In One JVM

An early draft of the OSGi framework specification version 4.3 [OSG10] defines two approaches for isolating applications. The approach implies using "nested" or "embedded" frameworks while "without the creation of special mechanisms to share resources, each embedded framework provides an isolated scope" [OSG10, ch. 3.1, p.10]. These embedded frameworks represent what is referred to as

[6] http://labs.oracle.com/projects/barcelona/

"composite bundles", that encapsulate "constituent bundles", which are isolated from normal bundles by default. "Through a sharing policy, the composite is in control of what capabilities from constituent bundles are exposed (exported) out of the composite" [OSG10, ch. 5, p. 15].
Although this draft represents the most promising concept towards full application isolation in OSGi platforms, it has not been included into the final OSGi framework specification release 4.3.

4.5.5 Applicability to CICS

As mentioned in [GTM+09, ch. 3, p. 6] the I-JVM is based on a proprietary JVM implementation, the LadyVM (refer [GTCF08]), that is not available for the System z architecture. The same problem applies to the Sandboxed OSGi, that is based on the MVM (refer [CD01]), which has not been developed to run on the z/OS operating system.
The only suitable proprietary solutions therefore are represented by the Hardened OSGi Implementation and by the *OSGi RFC-0138 Multiple Frameworks In One JVM*. Since however, the *RFC-0138* is an an standardized approach carried out by the OSGi Alliance, it is found to be more suitable isolation approach for being applied to a CICS environment.

Chapter 5

Summary and Conclusion

5.1 Key Result

Within the scope our investigations a detailed analysis of application isolation in an enterprise critical Java Transaction Server environment has been provided. It was shown that most known isolation issues remain yet to be resolved. Moreover, it was shown that with the introduction of the OGSi framework new issues came up. However, it was also shown that most issues can be resolved with built-in functionalities provided by the Transaction Server environment used. A summary of these solutions is discussed in the following.

5.2 Summary of Solutions

Using the CICS environment several Java related isolation issues are resolved (refer chapter 3.3). Due to the fact that JVM Servers include the OSGi framework, which implies namespace isolation by using different class loaders for each bundle, applications active within a JVM Server are isolated by default. However, it was shown that isolation remains incomplete and can be easily circumvented by wiring, leading to overlapping namespaces. Moreover, due to the CICS environment and several additional functionalities provided by the OSGi framework, new isolation problems such as illegal control (refer I_6, chapter 3.4.6) appear. However, as shown in chapter 4, most issues can be resolved with built-in functionalities, such as the Java Security Manager, or with tools provided for JVM Servers such as the Java Health Center.

Access to static fields discussed in I_1 (refer chapter 3.4.1) and I_2 (refer chapter 3.4.2), can be identified by the IBM CICS JVM Application Isolation Utility (refer chapter 4.1.1). In addition, the utility also identifies the access to methods of system classes, such as the `java.lang.Runtime` class. As shown in I_1 (refer chapter 3.4.1), the execution of particular methods from these classes can lead to severe problems. Since, however, their execution can be prevented using a standard or customized Security Manager or the OSGi security layer (refer chapter 4.2), isolation issues mentioned in I_1 (refer chapter 3.4.1) can be considered as resolved with applied static program analysis and active Java security features.

Overlapping namespaces (refer I_2, chapter 3.4.2) represent an issue more difficult to solve. Although one can prevent wiring with the OSGi specific Bundle Permission, in some cases it might be necessary to enable inter bundle communication. Once enabled by a particular permission, problems mentioned in chapter 3.4.2, as access to static fields, data races in unsynchronized blocks and hanging threads are likely to appear. Therefore, CICS services for program control are proposed as a secure alternative to wiring in terms of inter application communication (refer chapter 4.4). In addition, since issues resulting from the use of the JNI are severe (refer I_3, chapter 3.4.3) and their causes are difficult to identify, it is proposed to use CICS services for program control instead of the JNI (refer chapter 4.4).
Due to the fact that the identification of issues resulting in resource exhaustion (refer I_4, chapter 3.4.4) is not straight forward, the only solution for prevention of denial of service issues was found to be monitoring (refer chapter 4.3). As shown in chapter 4.3, using tools provided for the JVM Server enables to identify upcoming resource shortages in real time.
As shown in I_6 (refer chapter 3.4.6), the OSGi framework enables illegal control, that represents a major isolation issue. Due to the fact that the OSGi security layer includes specific permissions to control this feature, all problems related to illegal control are considered as resolved with active OSGi security.

No solutions were found for hanging threads (refer I_2, chapter 3.4.2) and infinite loops (refer I_4, chapter 3.4.4) that also imply issues in Activator classes (refer I_5, chapter 3.4.5). Hanging threads within application code however, can be detected with applied monitoring and affected bundles can be restarted, while infinite loops can be identified with static program analysis as shown in chapter 4.1.2. Therefore, only the hanging management interface arising from hanging threads within Activator classes is considered as a severe issue.

5.3 Outlook

In general, the integration of Java in CICS in terms of application isolation was found to be a satisfying approach; especially due to the provided tools for the JVM Server and due to the OSGi specific security layer. Since, however, the OSGi security layer is expected to imply performance degradation similar to the standard Security Manager (refer [HS05]), its activation may not be suitable in certain enterprise critical applications. Therefore, extensions of the OSGi framework such as the *RFC-0138* (refer chapter 4.5.4) might be interesting candidates in terms of isolating applications for future JVM Server releases.

Appendix A

Source Code

A.1 I_1: System Classes

A.1.1 Changes to Static Fields of System Classes

```
// Bundle A

package a;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class Change
{
    public static void main(CommAreaHolder CAH)
    {
        Task t = Task.getTask();
        if ( t == null )
            System.err.println("Change class: Can't get Task")
                ;
        else

        t.out.println("Starting bundle A");
                t.out.println("Current file separator: "+
                    System.getProperty("file.separator"));
                t.out.println("Changing file separator to \\")
                    ;
                System.setProperty("file.separator", "\\");
                t.out.println("Current file separator: "+
                    System.getProperty("file.separator"));

    }
}


// Bundle B

package b;
```

```
import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class Access
{
    public static void main(CommAreaHolder CAH)
    {
        Task t = Task.getTask();
        if ( t == null )
            System.err.println("Access class: Can't get Task")
                ;
        else

        t.out.println("Starting bundle B");
                t.out.println("Current file separator: "+
                    System.getProperty("file.separator"));

    }
}
```

Listing A.1: Changes to the a static field of a system class are globally visible.

A.1.2 CICS Region Shutdown Using System.exit

```
package stop;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class StopJVM
{
    public static void main(CommAreaHolder CAH)
    {

        Task t = Task.getTask();
        if ( t == null )
            System.err.println("StopJVM class: Can't get Task");
        else

        t.out.println("Attempting to stop JVM");

        System.exit(0);

        t.out.println("Attempt not succeeded");

    }
}
```

Listing A.2: Execution of java.lang.System.exit(0) from a bundle.

A.1.3 CICS Region Shutdown Using Runtime.halt

```
package halt;
```

```
import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class Shutdown {

        /**
         * @param args
         */
        public static void main(CommAreaHolder CAH) {

                Task t = Task.getTask();

                if (t == null ){
                System.err.println("Shutown class: Can't get Task")
                    ;
                        }
                else{
                        t.out.println(" transaction started");
                        t.out.println("attempting to stop JVM");
                        Runtime.getRuntime().halt(0);
                        t.out.println("attempt not succeeded");
                    }




        }

}
```

Listing A.3: Execution of java.lang.Runtime.getRuntime.halt(0) from a bundle.

A.1.4 Execution of OS Commands Using Runtime.exec

```
package delete;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class DeleteAFile{

        public static void main(CommAreaHolder CAH){

        BufferedReader consoleOutput;
        Runtime currentRuntime = Runtime.getRuntime();
        String line;

        Process runtimeProcess;
        Task t = Task.getTask();

        if ( t == null ){
        System.err.println("DeleteAFile class: Can't get Task");
                }

        else{
            try {
```

```
                t.out.println(" transaction started.");

                String[] whoami={"/bin/sh","-c","whoami"};
                runtimeProcess = currentRuntime.exec(whoami);
                consoleOutput = new BufferedReader(
                                new InputStreamReader(
                                    runtimeProcess.
                                    getInputStream()));
                t.out.println("logged in as: "+consoleOutput.
                    readLine());


                String[] ls ={"/bin/sh","-c","ls /u/prak101/
                    myprivatedir"};
                runtimeProcess = currentRuntime.exec(ls);
                consoleOutput = new BufferedReader(
                                new InputStreamReader(
                                    runtimeProcess.
                                    getInputStream()));
                t.out.println("files in directory before deleting:
                    ");
                while ((line = consoleOutput.readLine()) != null) {
                        t.out.println(line);
                        }

                t.out.println("attempting to remove \"myprivatefile
                    \"");
                String[] remove ={"/bin/sh","-c","rm /u/prak101/
                    myprivatedir/myprivatefile"};
                runtimeProcess = currentRuntime.exec(remove);

                runtimeProcess = currentRuntime.exec(ls);
                consoleOutput = new BufferedReader(
                                new InputStreamReader(
                                    runtimeProcess.
                                    getInputStream()));

                t.out.println("files in directory after deleting:
                    ");
                while ((line = consoleOutput.readLine()) != null) {
                        t.out.println(line);
                    }
                }
            catch (IOException e) {
                e.printStackTrace();
                }

                }
        }
}
```

Listing A.4: Execution of OS commands from a bundle.

A.1.5 Stopping Active Transactions Using the Thread Class

```
package stopframework;

import java.util.Map;
import java.util.Set;
```

```
import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class Stop {

    public static void main(CommAreaHolder CAH)
    {
        Task t = Task.getTask();

        t.out.println(" Transaction started");

        Map<Thread, StackTraceElement[]> threadMap=Thread.
            getAllStackTraces();
        Set<Thread> threadSet=threadMap.keySet();

        t.out.println("Stopping all active threads...");

        for(Thread currentThread:threadSet){

                if(currentThread.getName().contains("STHR")!=true){
                    //do not stop current task
                                t.out.println("Suspending "+
                                    currentThread.getName());
                                currentThread.stop();
                }
        }
        t.out.println("Done");

    }
}
```

Listing A.5: Get and stop all transactions active within a JVM server using the java.lang.Thread class.

A.2 I_2: Overlapping Namespaces

A.2.1 Class loading in wired bundles

```
//Bundle A
package a;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class AClass {

        public static String myName="undefined";


        public static void main(CommAreaHolder CAH){
                Task t = Task.getTask();

                if (t == null ){
                System.err.println("AClass: Can't get Task");
                        }
                else{
                        t.out.println("Bundle a is active!");
                }
```

```
        }
}


//Bundle X
package x;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

import a.AClass;

public class XClass {

        public static void main(CommAreaHolder CAH){

                AClass a=new AClass();
                String setName="Smith";

                Task t = Task.getTask();

                if (t == null ){
                System.err.println("CLass: Can't get Task");
                }
                else{
                        t.out.println(" ## instance of XClass ##");
                        t.out.println("(XClass) class loader used
                            to load AClass: ");
                        t.out.println(a.getClass().getClassLoader()
                            );
                        t.out.println("(XClass) current value of
                            myName: "+a.myName);
                        t.out.println("(XClass) changing value of
                            myName");
                        a.myName=setName;
                        t.out.println("(XClass) current value of
                            myName: "+a.myName);
                }
        }

}


//Bundle Y
package y;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

import a.AClass;

public class YClass {

        public static void main(CommAreaHolder CAH){

                AClass a=new AClass();
                String setName="Doe";

                Task t = Task.getTask();

                if (t == null ){
                System.err.println("YClass: Can't get Task");
```

```
                }
                else{
                        t.out.println(" ## instance of YClass ##");
                        t.out.println("(YClass) class loader used
                            to load AClass:");
                        t.out.println(a.getClass().getClassLoader()
                            );

                        t.out.println("(YClass) current value of
                            myName: "+a.myName);
                        t.out.println("(YClass) changing value of
                            myName");
                        a.myName=setName;
                        t.out.println("(YClass) current value of
                            myName: "+a.myName);
                }
        }

}
```

Listing A.6: Class loading is delegated to exporter bundle.

A.2.2 Threaded Access of a Shared Object (Data Race)

```
// AClass (in ABundle)
package abundle;

import xbundle.XClass;
import ybundle.YClass;
import zbundle.ZClass;

public class AClass {

        public static void main(String[] args){

                ZClass z=new ZClass();
                XClass x=new XClass(z);
                YClass y=new YClass(z);

                System.out.println("(A object) Starting data race:
                    ");

                x.start();
                y.start();

        }

}


// ZClass (in ZBundle)
package zbundle;

public class ZClass {

        private int counter=0;

        public int getCounter() {
                return counter;
        }
```

```
        public void setCounter(int x) {
                this.counter = x;
        }

}



// XClass (in XBundle)
package xbundle;

import zbundle.ZClass;

public class XClass extends Thread{

        private ZClass z;

        public XClass(ZClass ztemp){
                z=ztemp;
        }

        public void run(){
                int x=z.getCounter();

                while(x<=40){
                        x++;
                        System.out.println("(X object) value of
                            counter before update: "+z.getCounter()
                            );
                        System.out.println("(X object) updating
                            value of counter to: "+x);
                        z.setCounter(x);
                        System.out.println("(X object) updated
                            value of counter: "+z.getCounter());
                }
        }
}


// YClass (in YBundle)
package ybundle;

import zbundle.ZClass;

public class YClass extends Thread{

        private ZClass z;

        public YClass(ZClass ztemp){
                z=ztemp;
        }

        public void run(){
                int x=z.getCounter();

                while(x<=40){
                        x++;
                        System.out.println("(Y object) value of
                            counter before update: "+z.getCounter()
                            );
```

```
                        System.out.println("(Y object) updating
                            value of counter to: "+x);
                        z.setCounter(x);
                        System.out.println("(Y object) updated
                            value of counter: "+z.getCounter());
                }
        }
}
```

Listing A.7: Data race arising from threaded access of an object shared between several classes that are located in different bundles.

A.3 I_3: JNI

A.3.1 Segmentation Fault Caused via JNI

```
package seg;

public class Fault {

        private native void exploit();

        public static void main(String[] args) {

                System.out.println("Fault class attempting to
                    exploit segmentation fault");
                 new Fault().exploit();
        }

        static{
                System.loadLibrary("Fault");
        }


}
```

Listing A.8: Source code of OSGi bundle (Java) calling a C program with JNI. Modified after [She99].

```
#include <jni.h>
#include <stdio.h>
#include "seg_Fault.h"

JNIEXPORT void JNICALL
Java_seg_Fault_exploit(JNIEnv *env, jobject obj)
{
        int * myAddress=0;
        *myAddress=4;
}
```

Listing A.9: Source code of C program.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class seg_Fault */
```

```
#ifndef _Included_seg_Fault
#define _Included_seg_Fault
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     seg_Fault
 * Method:    exploit
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_seg_Fault_exploit
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Listing A.10: Source code of header file generated by javah (C header file generator).

A.3.2 Modification of private Fields via JNI

```
package alphapackage;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

import betapackage.BetaClass;

public class AlphaClass {

        private native void updateField(BetaClass myobj, String
            newName);

        public static void main(CommAreaHolder CAH) {
                 Task t = Task.getTask();

                t.out.println(" transaction started");


                AlphaClass alphaObject= new AlphaClass();

                t.out.println("Creating new Beta object");
                BetaClass betaObject=new BetaClass();

                t.out.println("Original private string value of
                    Beta object: "+betaObject.getName());

                String name="foobar";
                t.out.println("Updating private field of Beta
                    object out of C program via the JNI");
                alphaObject.updateField(betaObject, name);
                t.out.println("Updated private string value of Beta
                     object: "+betaObject.getName());
        }

        static {
                System.loadLibrary("Backdoor");
        }
```

```
}
```

Listing A.11: Source code of OSGi bundle (Alpha) calling a C program via the JNI to modify a private field (myName) from bundle Beta.

```
package betapackage;

public class BetaClass {

        private String myName="BetaObj";

        public String getName(){
                return myName;
        }

}
```

Listing A.12: Source code of OSGi bundle (Beta) defining the private string myName.

```
#include <jni.h>
#include <stdio.h>
#include "alpha_AlphaClass.h"

JNIEXPORT void JNICALL
Java_alpha_Backdoor_updateField(JNIEnv *env, jobject obj, jobject
    myobj, jstring jstr)
{
        static jfieldID fid_s = NULL; /* cached field ID for s */
        jclass cls = (*env)->GetObjectClass(env, myobj);
        if (fid_s == NULL) {
                fid_s = (*env)->GetFieldID(env, cls, "s","Ljava/
                    lang/String;");
                if (fid_s == NULL) {
                        return; /* exception already thrown */
                }
        }

        (*env)->SetObjectField(env, myobj, fid_s, jstr);
}
```

Listing A.13: Source code of C program that updates a private field (myName) of bundle Beta. Modified after [She99].

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class alphapackage_AlphaClass */

#ifndef _Included_alphapackage_AlphaClass
#define _Included_alphapackage_AlphaClass
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     alphapackage_AlphaClass
 * Method:    updateField
 * Signature: (Lbetapackage/BetaClass;Ljava/lang/String;)V
 */
```

```
JNIEXPORT void JNICALL Java_alphapackage_AlphaClass_updateField
  (JNIEnv *, jobject, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Listing A.14: Source code of header file generated by javah (C header file generator).

A.4 I_4: Resource Exhaustion

A.4.1 Infinite Loop

```
package infloop;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class InfiniteLoop {

        public static void main(CommAreaHolder CAH) {

                Task t=Task.getTask();
                t.out.println(" Infinite loop transaction started")
                    ;

                boolean neverfalse=true;
                while(neverfalse){
                        // do nothing
                }
        }
}
```

Listing A.15: Bundle code containing an infinite loop.

A.4.2 Recursive Thread Creation

```
package manythreads;

public class MyThread extends Thread{

        static int id;

        public void run(){

                for(int i=0;i<35;i++){
                        id=id+1;
                        System.out.println("Thread started. Id: "+
                            id);
                        new MyThread().run();
                }

        }
}
```

```
package manythreads;


public class Excess{

        public static void main(String[] args) {

                Thread myThread=new MyThread();
                myThread.run();
        }
}
```

Listing A.16: Recursive thread creation. Modified after [PF09].

A.4.3 Infinite Service Registration

```
package blueprint;

public interface Service {
        String saySomething();
}




package serviceimplementation;

import blueprint.Service;

public class MyService implements Service{

        public String saySomething() {
                return "Hello from MyService";
        }
}




package serviceimplementation;

import org.osgi.framework.BundleContext;
import org.osgi.framework.FrameworkUtil;
import blueprint.Service;
import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class Registration {


                public static void main(CommAreaHolder CAH){

                        Task t = Task.getTask();

                        BundleContext myBundleContext =
                            FrameworkUtil.getBundle(Registration.
                            class).getBundleContext();
```

```
                        Service my=new MyService();

                        t.out.println("Registering services");

                        while(true){

                                myBundleContext.registerService(
                                    Service.class.getName(), my,
                                                null);
                        }
                }
}
```

Listing A.17: Registration of identical services through an infinite loop.

A.4.4 Memory Leak

```
package memleak;

public class List
{
        MemoryLeak mem;
        List next;
}



package memleak;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;


public class MemoryLeak
{
   static List top;

   //each char array has 1MB
   char [] memory = new char [1048576];

   public static void main (CommAreaHolder CAH)
   {
          Task t = Task.getTask();

      // do not create more than 4 instances of MemoryLeak
      for (int i = 0; i < 4; i++)
      {
          t.out.println("Creating List instance no. "+i);
           List temp = new List ();
           temp.mem = new MemoryLeak ();
           temp.next = top;
           top = temp;
      }
   }
}
```

Listing A.18: Memory leak storing four megabytes of data permanently within the heap. Modified after [Fri02].

A.5 I_5: Issues with Activator Classes

A.5.1 Infinite Loop in the Activator

```
package zinfloopactiv;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

        private static BundleContext context;

        static BundleContext getContext() {
                return context;
        }

        public void start(BundleContext bundleContext) throws
            Exception {
                Activator.context = bundleContext;

                boolean neverfalse=true;
                while(neverfalse){
                        //do nothing
                }
        }

        public void stop(BundleContext bundleContext) throws
            Exception {
                Activator.context = null;
        }
}
```

Listing A.19: Source code of Activator with infinite loop.

A.5.2 Hanging Thread in the Activator

```
package zhangthreadactiv;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

        private static BundleContext context;

        static BundleContext getContext() {
                return context;
        }
        public void start(BundleContext bundleContext) throws
            Exception {
                Activator.context = bundleContext;
                Thread myThread=new MyThread();
                myThread.run();
        }

        public void stop(BundleContext bundleContext) throws
            Exception {
```

```
                Activator.context = null;
        }

}
```

Listing A.20: Source code Activator starting the hanging thread MyThread.

```
package zhangthreadactiv;

public class MyThread extends Thread{

        public void run(){

                System.out.println("Hanging thread started");
                Thread.currentThread().suspend();
                System.out.println("Hanging thread ended");
        }
}
```

Listing A.21: Source code of hanging thread started by the Activator.

A.6 I_6: Illegal Control

A.6.1 OSGi Bundle Context

```
//Alpha bundle
package alpha;
import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;
import org.osgi.framework.BundleException;
import org.osgi.framework.FrameworkUtil;
import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;

public class AlphaClass {

        public static void main(CommAreaHolder CAH){

                Task t = Task.getTask();

                BundleContext myBundleContext = FrameworkUtil.
                    getBundle(AlphaClass.class).getBundleContext();


                Bundle[] list=myBundleContext.getBundles();

                t.out.println(" Transaction started");

                t.out.println("List of all active bundles:");

                for(int i=0; i<list.length; i++){
                        t.out.println(list[i].getSymbolicName());
                        if(list[i].getSymbolicName().equals("Beta")
                            ){
                                try {
                                        list[i].stop();
                                        list[i].uninstall();
                                } catch (BundleException e) {
```

```
                                        t.out.println("Stopping␣
                                            bundle␣not␣successful")
                                            ;
                                }
                                t.out.println("done");
                        }
                }
                t.out.println("List␣of␣all␣active␣bundles:");

                for(int i=0; i<list.length; i++){
                        t.out.println(list[i].getSymbolicName());
                }

                t.out.println("Transaction␣ended");

        }
}


//Beta bundle
package beta;

import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;


public class BetaClass {

        public static void main(CommAreaHolder CAH){

                Task t = Task.getTask();
                t.out.println("Hello␣World␣from␣Beta␣bundle");
        }
}
```

Listing A.22: Source code of false control using the bundle context. Alpha bundle stops and uninstalls Beta bundle.

A.6.2 OGSi Bundle Fragments

```
/* SomeBundle MANIFEST.MF:

Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: SomeBundle
Bundle-SymbolicName: SomeBundle
Bundle-Version: 1.0.0
Import-Package: com.ibm.cics.server;version="1.0.0",
myhiddenpackage,
org.osgi.framework;version="1.3.0"
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
CICS-MainClass: somebundle.SomeClass;alias=some
*/

package somebundle;

import myhiddenpackage.MyHiddenClass;
import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.Task;
```

```
public class SomeClass {

        public static void main(CommAreaHolder CAH){

                Task t = Task.getTask();

                t.out.println("␣Transaction␣started");
                t.out.println("Attempting␣to␣access␣MyHiddenClass␣
                    of␣HostBundle");

                MyHiddenClass my=new MyHiddenClass();
                my.sayHello();

                t.out.println("Done.");

        }
}

/* FragmentBundle MANIFEST.MF:

Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: FragmentBundle
Bundle-SymbolicName: FragmentBundle
Bundle-Version: 1.0.0
Fragment-Host: HostBundle;bundle-version="1.0.0"
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Export-Package: myhiddenpackage
*/

/* HostBundle MANIFEST.MF:

Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: HostBundle
Bundle-SymbolicName: HostBundle
Bundle-Version: 1.0.0
Import-Package: com.ibm.cics.server;version="1.0.0",
org.osgi.framework;version="1.3.0"
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
*/

package myhiddenpackage;

import com.ibm.cics.server.Task;

public class MyHiddenClass {

        public void sayHello(){
                Task t = Task.getTask();
                t.out.println("␣Hello␣from␣MyHiddenClass");
        }
}
```

Listing A.23: Unintended package export of myhiddenpackage enabled through FragmentBundle.

Appendix B

Compact Disc (CD) Contents

A compact disk is attached to the hard copy of this document. Its contents are listed in the following.

- The directory **Thesis** contains this document in PDF format while the Tex files used for its creation are located in **Thesis**/**Tex**.
- The directory **References** contains all freely available online references. The names of the files correspond to the abbreviations used in the bibliography of this document.
- The directory **Tutorial** contains a tutorial in PDF format with detailed instructions and screenshots for practical software development using Eclipse and the CICS Explorer SDK, while the directory **Tutorial**/**Tex** contains the Tex files used for the creation of the document.
- The directory **Code** contains all Java code examples.

List of Abbrevations

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

CICS Customer Control Information System

JEE Java Enterprise Edition

JNI Java Native Interface

JRE Java Runtime Environment

JVM Java Virtual Machine

SDK Software Development Kit

LE Language Environment

MVM Multi-Tasking Virtual Machine

OSGi Open Standard Gateway Initiative

OTE Open Transaction Environment

RMI Remote Method Invocation

PRVM Persistent Reusable Virtual Machine

SDSF System Display and Search Facility

TCB Task Control Block

TS Temporary Storage

TD Transient Data

USS Unix System Services

Bibliography

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating Static Analysis Defect Warnings On Production Software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 1–8, New York, NY, USA, 2007. ACM. Available online at `http://findbugs.cs.umd.edu/papers/FindBugsExperiences07.pdf`; Accessed on 22. March 2012.

[Arn11] Isabel Arnold. CICS TS V4.2 News Tour 2011: CICS 4.2 Java. Presentation slides, June 2011. Available online at `ftp://ftp.software.ibm.com/software/emea/de/events/cics/CICSTSv42_Java.pdf`; Accessed on 2. December 2011.

[AS07] David Aspinall and Jaroslav Sevcik. Java Memory Model Examples: Good, Bad and Ugly. In *VAMP 2007*, Sep 2007. Available online at `http://groups.inf.ed.ac.uk/request/jmmexamples.pdf`; Accessed on 10. February 2012.

[Bat08] Andrew Bates. Why to choose CICS Transaction Server for new IT projects, 2008. Available online at `http://www-01.ibm.com/support/docview.wss?uid=swg27013865&aid=1`; Accessed on 23. November 2011.

[BCC+11] Frasier Bohm, Ben Cooper, Ben Cox, Elisabetta Flamini, Ivan Hsrgreaves, and Matthew Webster. Running Java workloads with CICS JVM server and OSGi, August 2011. Available online at `ftp://ftp.software.ibm.com/software//htp/cics/pdf/CICS_TS_V4.2_Java_paper_final.pdf`; Accessed on 2. December 2011.

[BD01] Mirza Beg and Mike Dahlin. A Memory Accounting Interface for The Java Programming Language. Technical report, University of Texas at Austin, 2001. Available online at `http://www.cs.uwaterloo.ca/~mbeg/papers/tr-01-40.pdf` Accessed on 29. February 2012.

[BG97] Dirk Balfanz and Li Gong. Experience with Secure Multi-Processing in Java. Technical report, Department of Computer Science, Princeton University,, September 1997. Available online at `http://sip.cs.princeton.edu/pub/icdcs.pdf`; Accessed on 13. January 2012.

[Blo05] Joshua Bloch. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, Upper Saddle River, NJ, 2005. Available online at `http://proquest.safaribooksonline.com/032133678X`; Made available through: Safari Books Online, LLC. Accessed on 19. March 2012.

[Bre11] Philipp Breitbach. IBM European WebSphere Technical Conference: ICS TS 4.2 JVM Server Application Best Practises. Presentation slides, October 2011.

[CD01] Grzegorz Czajkowski and Laurent Daynès. Multitasking without Compromise: A Virtual Machine Evolution. In *In ACM OOPSLA01*, 2001. Available online at `http://labs.oracle.com/projects/barcelona/papers/oopsla01.pdf`; Accessed on 14. January 2012.

[CS09] Roberto Chinnici and Bill Shannon. Java Platform, Enterprise Edition (Java EE) Specification, v6, October 2009. Available online at `http://download.oracle.com/otndocs/jcp/javaee-6.0-fr-oth-JSpec/`; Accessed on 6. March 2012.

[Cuy07] Hans Cuypers. Discrete Mathematics. Lecture Notes, October 2007. Available online at `http://www.win.tue.nl/~hansc/dw/notes.pdf`; Accessed on 3. January 2012.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 21–35, New York, NY, USA, 1998. ACM. Available online at `http://www.cs.cornell.edu/slk/papers/oopsla98.ps` Accessed on 29. February 2012.

[Cza00] Grzegorz Czajkowski. Application isolation in the Java Virtual Machine. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 354–366, New York, NY, USA, 2000. ACM. Available online at `http://labs.oracle.com/projects/barcelona/papers/oopsla00.pdf`; Accessed on 4. March 2012.

[Din04] Adair Dingle. Reclaiming Garbage and Education: Java Memory Leaks. *J. Comput. Small Coll.*, 20:8–16, December 2004.

[Eck11] David J. Eck. *Introduction to Programming Using Java*. Lulu Enterprises, Inc, Raleigh, North Carolina, USA, 6th ed edition, June 2011. Available online at `http://math.hws.edu/javanotes/`; Accessed on 28. December 2011.

[Fri02] Geoff Friesen. *Java 2 by Example*. Que, Indianapolis, IN, [2nd] edition, 2002. Available online at `http://proquest.safaribooksonline.com/0789725932`; Made available through: Safari Books Online, LLC. Accessed on 5. February 2012.

[Gar09] Nick Garrod. The Evolution of CICS. *IBMSystems Magazine*, September 2009. Available online at `http:`

//www.ibmsystemsmag.com/mainframe/administrator/cics/Modern-Information-Management/The-Evolution-of-CICS/; Accessed on 14. November 2011.

[GD09] Kiev Gama and Didier Donsez. Towards Dynamic Component Isolation in a Service Oriented Platform. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, CBSE '09, pages 104–120, Berlin, Heidelberg, 2009. Springer-Verlag. Available online at http://membres-liglab.imag.fr/donsez/pub/publi/cbse2009-gamadonsez.pdf; Accessed on 9. March 2012.

[GD10] Kiev Gama and Didier Donsez. A survey on approaches for addressing dependability attributes in the OSGi service platform. *SIGSOFT Softw. Eng. Notes*, 35(3):1–8, May 2010.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, May 2005. Available online at http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf; Accessed on 8. December 2011.

[GM96] James Gosling and Henry McGilton. The Java Language Environment: A white paper, May 1996. Available online at http://www.oracle.com/technetwork/java/index-136113.html; Accessed on 8. December 2011.

[Gon03] Li Gong. Java 2 Platform Security Architecture, 2003. Available online at http://docs.oracle.com/javase/1.4.2/docs/guide/security/spec/security-spec.doc.html; Accessed on 15. December 2011.

[GPB+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, Upper Saddle River, NJ, c2006. Available online at http://proquest.safaribooksonline.com/0321349601; Made available through: Safari Books Online, LLC. Accessed on 3. January 2012.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[GTCF08] N. Geoffray, G. Thomas, C. Clément, and B. Folliot. A Lazy Developer Approach: Building a JVM with Third Party Software. In *International Conference on Principles and Practice of Programming In Java (PPPJ 2008)* , Modena, Italy, September 2008. Available online at http://pagesperso-systeme.lip6.fr/Nicolas.Geoffray/files/pppj-08.pdf; Accessed on 26. March 2012.

[GTM+09] Nicolas Geoffray, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot, and Bertil Folliot. I-JVM: a Java Virtual Machine for component isolation in OSGi. In *DSN*, pages 544–553, 2009.

[Hac06] Steven R. Hackenberg. CICS open transaction environment and other TCB performance considerations. In *Int. CMG Conference*, pages 967–974, 2006. Available online at

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.3952&rep=rep1&type=pdf; Accessed on 30. November 2011.

[HKS04] Paul Herrmann, Udo Kebschull, and Wilhelm G. Spruth. *Einführung in z/OS und OS/390: Web-Services und Internet-Anwendungen für Mainframes*. Oldenbourg, Munich, 2., korr. aufl. edition, 2004.

[Hor00] John Horswill. *Designing and Programming CICS Applications*. O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472, July 2000.

[HP07] Marieke Huisman and Gustavo Petri. The Java Memory Model: a Formal Explanation. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, pages 81–96, 2007. Available online at http://www-sop.inria.fr/everest/personnel/Gustavo.Petri/publis/jmm-vamp07.pdf; Accessed on 2. January 2012.

[HPM10] R. Hall, K. Pauls, and S. McCulloch. *Osgi in Action: Creating Modular Applications in Java*. Manning Pubs Co Series. Manning Publications, April 2010. Available online at http://proquest.tech.safaribooksonline.de/9781933988917/; Made available through: Safari Books Online, LLC. Accessed on 17. March 2012.

[HS05] Almut Herzog and Nahid Shahmehri. Performance of the Java security manager. *Computers & Security*, 24(3):192 – 207, 2005. Available online at http://rewerse.eu/publications/download/REWERSE-RP-2005-141.pdf; Accessed on 26. March 2012.

[Hus11] Stefan Huster. Software Integration mit Java und XML unter CICS. Masterthesis, January 2011. Available online at http://www-ti.informatik.uni-tuebingen.de/~spruth/DiplArb/Huster.pdf; Accessed on 8. March 2012.

[Hyd99] Paul Hyde. *Java Thread Programming*. Sams Pub., Indianapolis, Ind., c1999. Available online at http://proquest.safaribooksonline.com/0672315858; Made available through: Safari Books Online, LLC. Accessed on 3. January 2012.

[IBM] IBM Corp. IBM Monitoring and Diagnostic Tools for Java - Health Center Version 2.0. Website. Available online at http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/; Accessed on 26. March 2012.

[IBM91] IBM Corp. *CICS Programming Primer*, 1991. Available online at http://www.informatik.uni-leipzig.de/cs/Literature/Textbooks/CicsCobPrimer.pdf; Accessed on 17. November 2011.

[IBM01] IBM Corp. New IBM Technology featuring Persistent Reusable Java Virtual Machines, October 2001. Available online at http://www-03.ibm.com/systems/resources/servers_eserver_zseries_software_java_pdf_prjvm13.pdf; Accessed on 29. November 2011.

[IBM04a] IBM Corp. CICS - An Introduction, 2004. Presentation slides. Available online at ftp://public.dhe.ibm.com/software/htp/cics/PDF/cics_introduction.pdf Accessed on 14. November 2011.

[IBM04b] IBM Corp. CICS: 35 years. Website, 2004. Available online at http://www-01.ibm.com/software/htp/cics/35/60s/; Accessed on 14. November 2011.

[IBM05] IBM Corp. *TXSeries for Multiplatforms: Concepts and Planning*, 2005. Available online at http://publib.boulder.ibm.com/infocenter/txformp/v5r1/topic/com.ibm.txseries510.doc/atshak02.pdf; Accessed on 16. November 2011.

[IBM10a] IBM Corp. *Language Environment Programming Guide*, 2010. Available online at http://publibz.boulder.ibm.com/epubs/pdf/ceea21b0.pdf; Accessed on 28. November 2011.

[IBM10b] IBM Corp. *z/OS concepts*, 2010. Available online at http://publib.boulder.ibm.com/infocenter/zos/basics/topic/com.ibm.zos.zconcepts/zconcepts_book.pdf; Accessed on 18. November 2011.

[IBM11a] IBM Corp. *CICS Messages and Codes Vol. 2*, 2011. Available online at http://publib.boulder.ibm.com/infocenter/cicsts/v4r2/topic/com.ibm.cics.ts.messages.doc/dfhg4v2_pdf.pdf; Accessed on 13. February 2012.

[IBM11b] IBM Corp. CICS Transaction Server for z/OS Version 4 Release 2: Application Programming Guide, 2011. Available online at http://publib.boulder.ibm.com/infocenter/cicsts/v4r2/topic/com.ibm.cics.ts.applicationprogramming.doc/dfhp3_pdf.pdf; Accessed on 30. November 2011.

[IBM11c] IBM Corp. CICS Transaction Server for z/OS Version 4 Release 2: Upgrading from CICS TS Version 3.1, 2011. Available online at http://publib.boulder.ibm.com/infocenter/cicsts/v4r2/topic/com.ibm.cics.ts.migration.doc/dfheg_pdf.pdf; Accessed on 27. February 2012.

[IBM11d] IBM Corp. *Java Applications in CICS (Version 4 Release 2)*, 2011. Available online at http://publib.boulder.ibm.com/infocenter/cicsts/v4r2/topic/com.ibm.cics.ts.java.doc/dfhpj_pdf.pdf; Accessed on 28. November 2011.

[IBM11e] IBM Corp. Techreport: History of CICS. Website, 2011. Available online at https://www-304.ibm.com/support/docview.wss?uid=swg21025234; Accessed on 24. November 2011.

[Jav06] Java Community Process. Sr-000121 application isolation api specification. Website, February 2006. Available online at http://jcp.org/aboutJava/communityprocess/final/jsr121/index.html; Accessed on 31. January 2012.

[Kop11a] Michael Kopp. The Top Java Memory Problems - Part 1. *about:performance*, April 2011. Available online at `http://blog.dynatrace.com/2011/04/20/the-top-java-memory-problems-part-1/`; Accessed on 14. February 2012.

[Kop11b] Michael Kopp. The Top Java Memory Problems - Part 2. *about:performance*, December 2011. Available online at `http://blog.dynatrace.com/2011/12/15/the-top-java-memory-problems-part-2/`; Accessed on 14. February 2012.

[Ler01] Xavier Leroy. Java Bytecode Verification: An Overview, 2001. Available online at `http://gallium.inria.fr/~xleroy/publi/survey-bytecode-verification.ps.gz`; Accessed on 20. December 2011.

[Lip08] Martin Lippert. Classloading and Type Visibility in OSGi. Presentation slides, 2008. Available online at `http://www.martinlippert.org/events/WJAX2008-ClassloadingTypeVisibilityOSGi.pdf`; Accessed on 2. February 2012.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd ed edition, 1999. Available online at `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`; Accessed on 21. December 2011.

[MF99] Gary McGraw and Ed Felden. *Securing Java*. John Wiley & Sons, Inc., January 1999. Available online at `http://www.securingjava.com/`; Accessed on 15. December 2011.

[MPAM99] Jeremy Manson, William Pugh, Sarita V. Adve, and Jeremy Manson. The Java Memory Model. In *In ACM Java Grande Conference*, 1999. Available online at `http://www.cs.uoregon.edu/Classes/06W/cis607atom/readings/manson-pugh-adve-popl05.pdf`; Accessed on 2. January 2012.

[Mue05] Jens Mueller. Anwendungs- und Transaktionsisolation unter Java, May 2005. Available online at `http://www-ti.informatik.uni-tuebingen.de/~spruth/DiplArb/jmueller.pdf`; Accessed on 8. December 2011.

[Nyl99] Joel Nylund. Memory Leaks in Java Programs. *Application Development Trends*, December 1999. Available online at `http://adtmag.com/articles/1999/12/24/memory-leaks-in-java-programs.aspx`; Accessed on 14. February 2012.

[Oak98] Scott Oaks. *Java Security*. O'Reilly & Associates, Inc, 101 Morris Street, Sebastopol, CA 95472, May 1998. Available online at `http://docstore.mik.ua/orelly/java-ent/security/`; Accessed on 14. December 2011.

[Ora11a] Oracle. Java Native Interface (Technote). Website, 2011. Available online at http://docs.oracle.com/javase/6/docs/technotes/guides/jni/; Accessed on 1. February 2012.

[Ora11b] Oracle. Java Platform, Standard Edition 6 API Specification. Website, 2011. Available online at http://docs.oracle.com/javase/6/docs/api/; Accessed on 2. February 2012.

[Ora12] Oracle. Trail: The Reflection API. Tutorial, 2012. Available online at http://docs.oracle.com/javase/tutorial/reflect/; Accessed on 26. February 2012.

[OSG10] OSGi Alliance. Version 4.3 Core Early Draft 1, April 2010. Available online at http://www.osgi.org/download/osgi-core-4.3-early-draft1.pdf; Accessed on 26. March 2012.

[OSG11a] OSGi Alliance. Website, 2011. Available online at http://www.osgi.org/; Accessed on 1. December 2011.

[OSG11b] OSGi Alliance. OSGi Service Platform Core Specification Release 4 Version 4.3, April 2011. Available online at http://www.osgi.org/download/r4v41/r4.core.pdf; Accessed on 30. November 2011.

[OW99] Scott Oaks and Henry Wong. *Java Threads*. Java series (OReilly & Associates). OReilly & Associates, Sebastopol, CA, 2nd ed edition, 1999. Available online at http://proquest.safaribooksonline.com/1565924185; Made available through: Safari Books Online, LLC. Accessed on 29. December 2011.

[PF09] Pierre Parrend and Stéphane Frénot. Security benchmarks of OSGi platforms: towards Hardened OSGi. *Softw. Pract. Exper.*, 39:471–499, April 2009.

[RAB+10] Chris Rayns, Edward Addison, Diana Blair, George Bogner, David Carey, Tony Fitzgerald, Scott McClure, Christen Plum, John Tilling, and Andy Wright. *Threadsafe Considerations for CICS*. IBM International Technical Support Organization (ITSO), November 2010. Available online at http://www.redbooks.ibm.com/redbooks/pdfs/sg246351.pdf; Accessed on 28. November 2011.

[RAR07] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: Distributed Applications Through Software Modularization. In *In Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference*, 2007. Available online at http://www.inf.ethz.ch/personal/troscoe/pubs/middleware07-rosgi.pdf; Accessed on 2. December 2011.

[RBB+10] Chris Rayns, George Bogner, Nicholas Bingell, Gordon Keehn, Lisa Fellows, Tommy Joergensen, and Erhard Woerner. *IBM CICS Explorer*. IBM International Technical Support Organization (ITSO), December 2010. Available online at http://www.redbooks.ibm.com/redbooks/pdfs/sg247778.pdf; Accessed on 1. February 2012.

[RBB+11] Chris Rayns, Sarah Bertram, George Bogner, Chris Carlin, Andre Clark, Amy Ferrell, Gordon Keehn, Peter Klein, Ronald Lee, and Erhard Woerner. *CICS Transaction Server from Start to Finish*. IBM International Technical Support Organization (ITSO), December 2011. Available online at `http://www.redbooks.ibm.com/redbooks/pdfs/sg247952.pdf`; Accessed on 8. March 2012.

[RBC+09] Chris Rayns, George Burgess, Scott Clee, Tom Grieve, John Taylor, Yun Peng Ge, Guo Qiang Li, Qian Zhang, and Derek Wen. *Java Application Development for CICS*. IBM International Technical Support Organization (ITSO), February 2009. Available online at `http://www.redbooks.ibm.com/redbooks/pdfs/sg245275.pdf`; Accessed on 24. November 2011.

[Roe01] Alex Roetters. Writing multithreaded Java applications. Online Article, February 2001. Available online at `http://www.ibm.com/developerworks/java/library/j-thread/index.html`; Accessed on 3. January 2012.

[RSW04] Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Static Program Analysis via 3-Valued Logic. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 401–404. Springer Berlin / Heidelberg, 2004. Available online at `http://groups.csail.mit.edu/cag/crg/papers/reps04shape.pdf`; Accessed on 22. March 2012.

[San04] Bo Sandén. Coping with java threads. *Computer*, 37(4):20–27, April 2004. Available online at `http://www.cse.chalmers.se/edu/course/ZZ_courses_2011/EDA222/Documents/Misc/Risky_Java.pdf`; Accessed on 27. March 2012.

[Sev08] Jaroslav Sevcik. *Program Transformations in Weak Memory Models*. PhD thesis, School of Informatics, University of Edinburgh, 2008. Available online at `http://www.cl.cam.ac.uk/~js861/thesis.pdf`; Accessed on 10. February 2012.

[She99] Liang Sheng. *The Java Native Interface: Programmer's Guide and Specification*. Java series. Addison-Wesley, Reading, MA, 1999. Available online at `http://java.sun.com/docs/books/jni/download/jni.pdf`; Accessed on 1. February 2012.

[Sil05] Abraham Silberschatz. Operating System Concepts, 2005. Available online at `http://proquest.safaribooksonline.com/9780471694663`; Made available through: Safari Books Online, LLC. Accessed on 30. December 2011.

[Spr08] Wilhelm G. Spruth. Enterprise Computing lecture slides, 2008. Available online at `http://www-ti.informatik.uni-tuebingen.de/~spruth/vorlesung_cs/CSSUM09.pdf`; Accessed on 17. November 2011.

[Spr10] Wilhelm G. Spruth. System z and z/OS unique Characteristics. Technical report, Wilhelm Schickard Institute for Computer Science, April

2010. Available online at http://tobias-lib.uni-tuebingen.de/volltexte/2010/4710/pdf/report_spruth_2010.pdf; Accessed on 17. November 2011.

[Tan03] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, Munich, 2nd ed edition, 2003.

[Tra01] Greg Travis. Understanding the Java ClassLoader. Tutorial, April 2001. Available online at http://www.ibm.com/developerworks/java/tutorials/j-classloader/j-classloader-pdf.pdf; Accessed on 6. March 2012.

[TV08] Andre L.C. Tavares and Marco Tulio Valente. A Gentle Introduction to OSGi. *SIGSOFT Softw. Eng. Notes*, 33:8:1–8:5, August 2008. Available online at http://homepages.dcc.ufmg.br/~mtov/pub/2008_sen.pdf; Accessed on 6. December 2011.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *In Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63, 1998. Available online at http://sip.cs.princeton.edu/pub/oakland98.pdf; Accessed on 15. December 2011.

[Yak02] Vladimir Omar Calderon Yaksic. J-RAF – The Java Resource Accounting Facility. Masterthesis, June 2002. Available online at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.7.742&rep=rep1&type=pdf Accessed on 29. February 2012.