D4.1 – API Specification

Version 1.0
6 November 2014
Final

EC Distribution

acias and University of York

Every effort has been made to ensure that all statements and information contained herein are accurate, however the DreamCloud Project Partners accept no liability for any error or omission in the same.

© 2014 Copyright in this document remains vested in the DreamCloud Project Partners.
## Project Partner Contact Information

<table>
<thead>
<tr>
<th>Company</th>
<th>Contact Person(s)</th>
<th>Address 1</th>
<th>Address 2</th>
<th>City</th>
<th>Country</th>
<th>Phone</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>aicas</strong></td>
<td>Fridtjof Siebert</td>
<td>Haid-und-Neue Strasse 18</td>
<td>71701 Schwieberdingen</td>
<td>Schwieberdingen</td>
<td>Germany</td>
<td>+49 711 811 24517</td>
<td><a href="mailto:jochen.haerdtlein@de.bosch.com">jochen.haerdtlein@de.bosch.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>76131 Karlsruhe</td>
<td>Robert-Bosch-Strasse 2</td>
<td>Karlsruhe</td>
<td>Germany</td>
<td>+49 721 66396823</td>
<td><a href="mailto:siebert@aicas.com">siebert@aicas.com</a></td>
</tr>
<tr>
<td><strong>Bosch</strong></td>
<td>Jochen Härdtlein</td>
<td>Haid-und-Neue Strasse 18</td>
<td>71701 Schwieberdingen</td>
<td>Schwieberdingen</td>
<td>Germany</td>
<td>+49 711 811 24517</td>
<td><a href="mailto:jochen.haerdtlein@de.bosch.com">jochen.haerdtlein@de.bosch.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>76131 Karlsruhe</td>
<td>Robert-Bosch-Strasse 2</td>
<td>Karlsruhe</td>
<td>Germany</td>
<td>+49 721 66396823</td>
<td><a href="mailto:siebert@aicas.com">siebert@aicas.com</a></td>
</tr>
<tr>
<td><strong>CNRS</strong></td>
<td>Gilles Sassatelli</td>
<td>Rue Ada 161</td>
<td>20 Leighton Avenue</td>
<td>Montpellier</td>
<td>France</td>
<td>+33 4 674 18690</td>
<td><a href="mailto:sassatelli@lirmm.fr">sassatelli@lirmm.fr</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>34392 Montpellier</td>
<td>Pinner Middlesex HA5 3BW</td>
<td>Montpellier</td>
<td>France</td>
<td>+33 4 674 18690</td>
<td><a href="mailto:sassatelli@lirmm.fr">sassatelli@lirmm.fr</a></td>
</tr>
<tr>
<td><strong>Rheon Media</strong></td>
<td>Raj Patel</td>
<td>Rue Ada 161</td>
<td>20 Leighton Avenue</td>
<td>Montpellier</td>
<td>France</td>
<td>+44 7547 162920</td>
<td><a href="mailto:raj@rheonmedia.com">raj@rheonmedia.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>34392 Montpellier</td>
<td>Pinner Middlesex HA5 3BW</td>
<td>London</td>
<td>United Kingdom</td>
<td>+44 7547 162920</td>
<td><a href="mailto:raj@rheonmedia.com">raj@rheonmedia.com</a></td>
</tr>
<tr>
<td><strong>The Open Group</strong></td>
<td>Scott Hansen</td>
<td>Avenue du Parc de Woluwe 56</td>
<td>1160 Brussels</td>
<td>Brussels</td>
<td>Belgium</td>
<td>+32 2 675 1136</td>
<td><a href="mailto:s.hansen@opengroup.org">s.hansen@opengroup.org</a></td>
</tr>
<tr>
<td><strong>University of Stuttgart</strong></td>
<td>Bastian Koller</td>
<td>Nobelstrasse 19</td>
<td>70569 Stuttgart</td>
<td>Stuttgart</td>
<td>Germany</td>
<td>+49 711 68565891</td>
<td><a href="mailto:koller@hlrs.de">koller@hlrs.de</a></td>
</tr>
<tr>
<td><strong>University of York</strong></td>
<td>Leandro Indrusiak</td>
<td>Deramore Lane</td>
<td>York YO10 5GH</td>
<td>York</td>
<td>United Kingdom</td>
<td>+44 1904 325 570</td>
<td><a href="mailto:leandro.indrusiak@cs.york.ac.uk">leandro.indrusiak@cs.york.ac.uk</a></td>
</tr>
</tbody>
</table>
Contents

1 Executive Summary 1

2 Background 2

3 Requirements 4

4 Low-Level Resource Control 7

4.1 Realtime Java and OS .................................................. 7

4.1.1 Base Java environment ........................................... 7

4.1.2 Base OS environment ............................................. 7

4.2 Controlling CPU frequency .......................................... 7

4.3 Performance Optimization and Compilation ......................... 8

4.4 Java APIs to control Threads ........................................ 8

4.4.1 ThreadGroupController ......................................... 8

4.4.2 AllocationHandler .................................................. 11

5 OSGi 13

5.1 OSGi concepts .......................................................... 13

5.2 OSGi architecture ....................................................... 13

5.3 OSGi implementations ............................................... 14

5.4 Life Cycle layer ......................................................... 15

5.5 migration ................................................................. 15

5.5.1 error handling ....................................................... 16

6 High-Level APIs 17

6.1 Bundle resource monitoring .......................................... 17

6.1.1 CPU Budgets ......................................................... 17

6.1.2 Cost enforcement ................................................... 17

6.1.3 Admission control protocol ...................................... 18

6.1.4 Memory budgets .................................................... 18

6.2 Extensions to OSGi life cycle management ......................... 19
### VM Level APIs

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Dynamic Adding and Removing of CPUs and Frequency Changes</td>
<td>20</td>
</tr>
<tr>
<td>7.2</td>
<td>Freeze and Resume of Tasks</td>
<td>25</td>
</tr>
<tr>
<td>7.3</td>
<td>Freeze and Resume of whole VM</td>
<td>28</td>
</tr>
<tr>
<td>7.4</td>
<td>Enhancing freeze and resume</td>
<td>29</td>
</tr>
</tbody>
</table>
List of Figures

1. Tasks running on the DreamCloud Framework ........................................ 2
2. The OSGi framework layers .................................................................. 14
## Document Control

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Document outline</td>
<td>1 October 2014</td>
</tr>
<tr>
<td>0.9</td>
<td>Version project partner review</td>
<td>31 October 2014</td>
</tr>
<tr>
<td>1.0</td>
<td>Final review and QA</td>
<td>6 November 2014</td>
</tr>
</tbody>
</table>
1 Executive Summary

This document constitutes deliverable *D4.1 – API Specification* of WP4 of the DreamCloud project. This deliverable describes the APIs that can be used to extend the existing functionality of Java platform, the underlying OS and the component framework in order to achieve the goals of the project. The goals include the control over tasks running on the DreamCloud platform and for the greatest flexibility different levels of control will be provided: from low-level APIs at the OS level to APIs at the level of OSGi-based component framework up to APIs that control the entire Java VM.
2 Background

For the purposes of the implementation DreamCloud tasks are treated as abstract entities scheduled on a networked system.

Within the system software APIs specified in this document, however, we need to decide what basic concepts are used to implement tasks.

One of the execution environments used in DreamCloud is a Java VM. We must define how to map DreamCloud tasks onto Java concepts a little weird about mapping tasks to concepts, but ok and to determine the characteristics of these tasks and how they can be separated from one another. Figure 1 provides a high-level overview of a Java VM running several tasks within DreamCloud. These tasks are separated from one another, ensuring that each task exclusively locks the required resources and those can not be utilized by other tasks.

![Figure 1: Tasks running on the DreamCloud Framework](image)

Java provides an extensive set of features that can serve as containers for tasks, some of these features are native to the language or the environment, some are established component frameworks used in the Java world. Roughly sorted by the level of granularity, the following entities could be used to contain tasks within DreamCloud:

- Java methods – these are the basic building blocks of Java applications
- Java classes or objects – these group several methods and provide a clear interface
- `java.util.concurrentRunnableFuture<V>` from the Java concurrency APIs provides a container for an activity that may run in parallel and produce a result value.
- `java.lang.Thread` represents a thread of execution in Java
- `javax.realtime.RealtimeThread` represent threads of execution in realtime Java with additional properties such as priorities with realtime semantics.
- `javax.realtime.AsyncEventHandler` represents activities in realtime Java that are not strictly associated with a thread.
• Java Xlets provides a framework for components with a simple and clear life cycle model using transitions init, start, pause, stop, and destroy.
• JamaicaCAR App is a specific Xlet widely used in app platforms within automotive head units. These components are controlled by a ThreadGroup together with restricted CPU budget, heap memory budget and permission.
• OSGi provides a component model that is applied in a wide variety of applications including home automation, IDEs (Eclipse) or the Internet-of-Things.
• java.lang.ThreadGroup provides a mechanism to group several threads
• javax.realtime.ProcessingGroup is a realtime Java mechanism to control the scheduling and resource usage of a set of schedulable objects
• Java applications could be treated as tasks within DreamCloud.

After careful analysis of requirements (see next section), it was decided not to rely on one single mechanism for DreamCloud, but to provide APIs at different levels for tasks of different granularities. An important role is allocated to OSGi bundles that provide a mature component model DreamCloud can use as a basis (see section). However this model has to be extended and new mechanisms need to be provided for the migration of components (section).

An aspect that is orthogonal to the component model is the extent to which the DreamCloud scheduling and task migration may remain transparent to the developer of the components. Different levels of support (support for what?) are discussed in section that describes the migration support provided by the Java VM.
3 Requirements

In this section, we review the requirements defined in deliverable D1.2 – Dynamic Resource Allocation Requirements [14] and map those that affect the definitions of APIs in the system software infrastructure to the specific sections of this document.

- **Req 3.3.1 Objectives of dynamic resource management should be configurable**
  
  *This requirement has no direct effect on the system software APIs.*
  
  Specific objectives, however, will indirectly result in requirements on the system software infrastructure. A good candidate is energy efficiency, that requires APIs for dynamically adding and removing CPUs detailed in section 7.1. Meeting realtime deadlines requires APIs for cost enforcement (sections 4.4.1, 6.1.2), admission control (section 6.1.3), and guaranteeing memory budgets require APIs to enforces tasks to stay within their budgets (sections 4.4.2 and 6.1.4).

- **Req 3.3.2 Specified hard real-time constraints shall not be violated**
  
  *This requirement prescribes for the system software to provide realtime guarantees.*
  
  This does not affect any new APIs defined in this document, but it requires the underlying Java implementation and operating system to provide the realtime guarantees as detailed in section 4.1.

- **Req 3.3.3 Dynamic resource allocation shall be used to provide different levels of performance guarantees**
  
  *This requirement demands the availability of APIs for management of system level resources.*
  
  The system infrastructure must provide APIs for the resources managed by the Java VM and the OS, in particular CPUs (section 4.1 and 7.1) and memory (sections 4.4.2 and 6.1.4).

- **Req 3.3.4 The average latency of jobs shall be minimised**
  
  *This requirement translates into performance requirements for the system infrastructure.*
  
  In particular, it would not be sufficient to execute Java tasks by a Java bytecode interpreter since this would probably result in unacceptably poor runtime performance compared to non-DreamCloud. As a result, the system software will require mechanisms such as compilers to enhance the performance (section 4.3).

- **Req 3.3.5 The total energy dissipation of jobs shall be minimised**
  
  *This requirement requests the APIs for controlling resources that cause energy dissipation such as CPU and memory usage.*
  
  This is related to the resource allocation requirement 3.3.3 since controlling the usage of CPUs (section 4.1 and 7.1) and memory (sections 4.4.2 and 6.1.4) has a direct effect on the energy consumption. In particular, disabling unused CPUs (section 7.1) or reducing the clock frequency (section 4.2) gives some control over the energy dissipation.

- **Req 3.3.6 Communication overhead parameters shall be predictable**
  
  *This requirement calls for general realtime guarantees in the system software infrastructure.*
  
  From the Java VM and OS perspective, this requirement is covered by requirement 3.3.2, realtime guarantees (section 4.1).

- **Req 3.3.7 Dynamic resource allocation overhead shall be predictable and bounded**
  
  *This requirement affects all APIs defined within the system infrastructure.*
  
  The time required for all APIs provided by the system software infrastructure and used for dynamic resource allocation, must provide predictable and bounded overhead. (not clear what
"required" refers to, please clarify) This overhead should be very small, in the order of µ-
seconds, for monitoring and controlling memory and CPU usage (sections 4.1 and 6.1.4), in
the order of milli-seconds for enabling or disabling CPUs or changing their frequencies (sec-
tions 4.2 and 7.1), and in the order of 10-100s of milli-seconds for the migration of larger tasks
between computational nodes (section 7.2).

• **Req 3.3.8 The dynamic resource allocation mechanisms shall cope with dynamic work-
load**
  
  *This requirement applies to the resource allocation mechanism, not to the system software
  infrastructure APIs.*
  
  The APIs defined in this deliverable are not affected by this.

• **Req 3.3.9 The dynamic resource allocation mechanisms shall not limit hardware scaling**
  
  *This requirement applies to the resource allocation mechanism, not to the system software
  infrastructure APIs.*
  
  The APIs defined in this deliverable are not affected by this.

• **Req 3.3.10 The dynamic resource allocation mechanisms shall cope with limited informa-
tion about the state of the overall system**
  
  *This requirement applies to the resource allocation mechanism, not to the system software
  infrastructure APIs.*
  
  The APIs defined in this deliverable are not affected by this.

• **Req 3.3.11 The dynamic resource allocation mechanisms shall respect mapping con-
straints that restrict the allowed computational unit**
  
  *This requirement applies to the resource allocation mechanism, not to the system software
  infrastructure APIs.*
  
  The APIs defined in this deliverable are not affected by this.

• **Req 3.3.12 The dynamic resource allocation mechanisms shall consider cost, runtime and
power efficiency for different type of resources available to a multi-typed job**
  
  *This requirement applies to the resource allocation mechanism, not to the system software
  infrastructure APIs.*
  
  The system software infrastructure, however, can help to increase the number of resources that
can be used to perform a given job: The use of different Java virtual machines on different HW
may allow the same job to run on very different systems using, e.g., ARM and x86/64 CPUs or
even accelerators such as FPGAs (section 4.1.1).

Table 1 gives a compact overview of the requirements that are addressed by the APIs defined in the
sections of this document.
## Requirement

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Section(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req 3.3.1 Objectives of dynamic resource management should be configurable</td>
<td>7.1 4.4.1 6.1.2</td>
</tr>
<tr>
<td></td>
<td>6.1.3 4.4.2 6.1.4</td>
</tr>
<tr>
<td>Req 3.3.2 Specified hard real-time constraints shall not be violated</td>
<td>4.1</td>
</tr>
<tr>
<td>Req 3.3.3 Dynamic resource allocation shall be used to provide different levels of performance guarantees</td>
<td>4.1 7.1 4.4.2 6.1.4</td>
</tr>
<tr>
<td>Req 3.3.4 The average latency of jobs shall be minimised</td>
<td>4.3</td>
</tr>
<tr>
<td>Req 3.3.5 The total energy dissipation of jobs shall be minimised</td>
<td>4.1 7.1 4.4.2 6.1.4</td>
</tr>
<tr>
<td></td>
<td>7.1 4.2</td>
</tr>
<tr>
<td>Req 3.3.6 Communication overhead parameters shall be predictable</td>
<td>4.1</td>
</tr>
<tr>
<td>Req 3.3.7 Dynamic resource allocation overhead shall be predictable and bounded</td>
<td>4.1 6.1.4 4.2 7.1</td>
</tr>
<tr>
<td></td>
<td>7.2</td>
</tr>
<tr>
<td>Req 3.3.8 The dynamic resource allocation mechanisms shall cope with dynamic workload</td>
<td>–</td>
</tr>
<tr>
<td>Req 3.3.9 The dynamic resource allocation mechanisms shall not limit hardware scaling</td>
<td>–</td>
</tr>
<tr>
<td>Req 3.3.10 The dynamic resource allocation mechanisms shall cope with limited information about the state of the overall system</td>
<td>–</td>
</tr>
<tr>
<td>Req 3.3.11 The dynamic resource allocation mechanisms shall respect mapping constraints that restrict the allowed computational unit</td>
<td>–</td>
</tr>
<tr>
<td>Req 3.3.12 The dynamic resource allocation mechanisms shall consider cost, runtime and power efficiency for different type of resources available to a multi-typed job</td>
<td>4.1.1</td>
</tr>
</tbody>
</table>

Table 1: Requirements addressed by System Software APIs in this document.
4 Low-Level Resource Control

This section defines a set of low-level APIs for resource control that can be used by higher level frameworks or applications. Particular focus is given to the OSGi framework that will be described in following sections 5 and 6.

This section lists Java VM requirements necessary to provide realtime features and high performance that can be fulfilled by the existing realtime Java implementation provided by Jamaica, as well as extensions that allow additional control of Threads, CPU usage and memory allocation.

4.1 Realtime Java and OS

4.1.1 Base Java environment

A fundamental prerequisite for the system software infrastructure is to provide realtime guarantees. For the Java implementation, this means ability to support the Realtime Specification for Java (RTSJ) as specified in the JSR 282 [5] and implement concurrent and parallel realtime garbage collection [15]. Realtime APIs also include fine-grained control over scheduling decisions such as CPU affinities provided by the RTSJ through class `javax.realtime.Affinity` and APIs related to this class.

Usage of a Java VM makes it possible to run the same job on different hardware, like ARM or x86/64 CPUs with different numbers of cores. One might even go further and use the Java VM on top of hardware accelerators, for example FPGA-based ones, as demonstrated by the JUNIPER projects [9].

4.1.2 Base OS environment

At the OS level, support for realtime priorities is provided, for instance via the POSIX scheduling policy `SCHED_FIFO` [13]. Control over CPU affinities can be obtained via the Linux-specific `sched_setaffinity` system call.

4.2 Controlling CPU frequency

The energy dissipation of today’s hardware depends mostly on the frequencies of the underlying CPUs. In order to match the requirement `Req 3.3.5 The total energy dissipation of jobs shall be minimised` from D1.2 ([14]), we might need certain means to control these frequencies. This is typically done at the OS level. On Linux CPU frequency can be controlled for each individual CPU using the sysfs mechanism [10].

Although Linux command `cpufreq-set` [6] provides a more convenient way to control the frequency, it essentially performs direct writes to the same sysfs cpufreq interface. (here I’m a little lost: is there any conclusion from the whole subsection?)
4.3 Performance Optimization and Compilation

The use of a Java VM within DreamCloud brings up the question on how the performance requirements (Req 3.3.4 The average latency of jobs shall be minimised from D1.2 cited1.2) can be achieved. The Java VM specification [11] defines a virtual machine that can be implemented either as a simple interpreter, or the one that uses different compilation techniques for better performance. The compilation techniques usually rely on either just-in-time compilation, as provided by OpenJDK [7], or on static ahead-of-time compilation as provided by JamaicaVM [1].

In the context of DreamCloud, a simple interpreter implementation provides the fastest and the most flexible results, in particular considering the dynamic migration features described in section 7. The just-in-time compilation approach results in very unpredictable timing and can therefore not be applied within DreamCloud. The ahead-of-time compilation, however, provides high performance and predictable timing, but is harder to apply to dynamic systems such as OSGi. It also typically does not allow migration between different hardware architectures.

Within DreamCloud, Jamaica ahead-of-time compiler will therefore be extended to allow the compilation of single bundles. A new tool – the bundler – will provide the ability to equip a component (OSGi bundle, xlet, etc.) provided as a Java jar file with pre-compiled machine code for one or several target platforms.

4.4 Java APIs to control Threads

Two classes are to be added to provide additional control over the resource usage by threads or groups of threads that belong to one component (OSGi bundle, Xlet, etc.). A limited control is already provided via standard Java class java.lang.ThreadGroup [2] and by RTSJ [5, 3] classes javax.realtime.ProcessingGroupParameters and javax.realtime.MemoryParameters.

4.4.1 ThreadGroupController

Class ThreadGroupController permits low-level control over the CPU usage and the number of threads running within one ThreadGroup. An instance of ThreadGroupController should be made accessible from an instance of ThreadGroup via a call to the static function get. This approach enables low-level control over the threads in that group.

```java
/**
 * This class permits execution time control over the members of a
 * ThreadGroup.
 */
public class ThreadGroupController {

    /**
     * Class that permits to define a wrapper method around a thread's
     * run method.
     */
```
public static abstract class RunWrapper
{
    /**
     * wrapper method that has to call r.run().
     * @param r the runnable.
     */
    public abstract void callRun(Runnable r);
}

/**
 * get obtains the ThreadGroupController for a given ThreadGroup.
 * If no controller was installed yet, this method will create one.
 * @param g the ThreadGroup
 * @return the controller, never null.
 * @throws SecurityException If a security manager has been installed and a
 * call to its <code>checkAccess</code> method with the controlled
 * <code>ThreadGroup</code> as its argument results in throwing a
 * <code>SecurityException</code>.
 */
public static ThreadGroupController get(ThreadGroup g);

/**
 * Install a wrapper to be called around the run() method for all
 * newly started threads in this thread group.
 * @param r the run wrapper, must not be null
 * @throws SecurityException If a security manager has been installed and a
 * call to its <code>checkAccess</code> method with the controlled
 * <code>ThreadGroup</code> as its argument results in throwing a
 * <code>SecurityException</code>.
 */
public void setRunWrapper(RunWrapper r);

/**
 * setMaxActiveCount sets the maximum number of active threads
 * allowed for this ThreadGroup.
 * If the number of active threads is >= max, no new threads will be
 * permitted for this ThreadGroup. Threads created in the past are
 * unaffected if max is set lower later.
 */
**public void** setMaxActiveCount**(int max)**;

/**
 * @param max the maximum number of active threads.
 * @param SecurityException If a security manager has been installed and a
call to its <code>checkAccess</code> method with the controlled
ThreadGroup as its argument results in throwing a
SecurityException.
*/

/**
 * @return the number of cycles.
 * @param SecurityException If a security manager has been installed and a
call to its <code>checkAccess</code> method with the controlled
ThreadGroup as its argument results in throwing a
SecurityException.
*/

/**
 * setMaxPriority sets the maximum priority of this thread group and
of all its children.<p>
* Unlike ThreadGroup.setMaxPriority, this function also reduces the
priority of any threads in this group that may already have a
higher priority. However, it does not change the priority of
running threads of any children of this thread group.
* @param pri the new maximum priority. A priority value of "0" can
be used to force all threads to Java priority "1" with Jamaica's
Scheduler.microAdjustPriority() of "−1" such that these threads
will never run if there is a higher priority thread, even in the
non-strict priority range.
* @throws SecurityException If a security manager has been installed and a
call to its <code>checkAccess</code> method with the controlled
ThreadGroup as its argument results in throwing a
SecurityException.
*/

/**
 * For all threads in this thread group that are blocked in an I/O
function, try to interrupt this I/O function. This only works
for I/O operations that are interruptible. Currently,
interruptible I/O operations include
*/
4.4.2 AllocationHandler

Class AllocationHandler permits the installation of a handler called on every allocation performed by a given thread. This handler can independently decide whether to disallow allocation or to trace it, e.g., via instances of PhantomReference, etc.

```java
/**
 * AllocationHandler provides a means to attach a handler to a thread
 * that will be invoked every time the thread performs an allocation.
 */
public abstract class AllocationHandler {
    /*------------------------ methods ------------------------*/
    /**
     * set the allocation handler for a given thread
     *
     * @param t the thread
     * @param h the allocation handler, must redefine method handle(Object)
     * @throws SecurityException If a security manager has been installed and it
denies [@link RuntimePermission]<tt>"AllocationHandler.set"</tt>
* installation of an allocation handler.
*/
    public static synchronized void setAllocationHandler(Thread t,
                                                AllocationHandler h);
```
/**
 * Remove the given memory handler.
 * @param t the thread the handler was assigned to
 * @param h the allocation handler
 * @throws SecurityException If a security manager has been installed and it
denies [@link RuntimePermission]<tt>("AllocationHandler.remove")</tt>
removal of an allocation handler.
*/
public static synchronized void removeAllocationHandler(Thread t,
AllocationHandler h);

/**
 * Routine that will be called when a thread allocated an object.
 * @param o the allocated object
 * @return the allocated object o or null if the allocation should
 * fail.
 */
public abstract Object handle(Object o);

/**
 * Routine that will be called before a thread attempts to allocate an object.
 * This routine may disallow an allocation by returning false.
 * On the allocation of a multi-dimensional array, e.g., new int[3][4], this
 * method will be called on the allocation of every single-dimensional array
 * separately, e.g. once with <tt>allow(int[][], class, 3)</tt> and three times
 * <tt>allow(int[], class, 4)</tt>.
 * The default implementation always returns true.
 * @param class the class of the object to be allocated.
 * @param length the length of the array that is to be allocated in case class
 * is an array, −1 otherwise.
 * @return true iff allocation should be allowed, false if it should be
 * declined
 */
public boolean allow(Class clazz,
        int length);
5 OSGi

In this section the OSGi framework is introduced as the system of executable entities to build upon. The desired real-time and resource monitoring features are implemented as extensions to the basic OSGi framework. The basic terminology and concepts such as bundles, the bundle life cycle and resource management are explained and then extended. Furthermore, an API that enables migration of bundles is proposed.

5.1 OSGi concepts

The OSGi framework [8] gives the proposed answer to the question of what the Java representation of an executable entity ("runnable") looks like: An executable entity is represented by an OSGi module, a so called bundle. OSGi is a specification for a dynamic module system for Java.

The key features of an OSGi implementation are:

- **Modularity** OSGi embodies a component-based model where modules are dynamically installed and loaded. A bundle hides its internals from other bundles and communicates only via well-defined services.
- **Dependency management** Each module clearly specifies its dependencies in its `manifest.mf` file. Dependency management of bundles, including semantic versioning, is provided.
- **Distribution** A bundle explicitly registers the `services` that it provides or requires in order to run.
- **Robustness** Each bundle may be stopped or uninstalled from the framework at any moment. Since each bundle may provide a service, bundles may cooperate with each other. If a bundle providing a service is stopped, uninstalled or fails, the OSGi framework provides a mechanism to safely handle the bundle removal by deactivating the bundle listeners. Once this action is performed, all the bundles that were using the services provided by the removed bundle are automatically notified by the framework.

Other notable features are:

- **Security** The OSGi security layer is built on top of the Java security layer and extends it by providing a fine-grained model which can be executed at all OSGi layers. The bundle developer may specify the requested security details for one or more layers and implement a security policy for those. For example, it is possible to apply a security policy to the bundles by applying a JAR signature and thus limiting the number of authorized users.
- **Evolution** As mentioned previously, a bundle can make its dependencies available through the directives Import-Package and Export-Package. OSGi also supports versioning as part of dependency management. OSGi allows for multiple versions of a Java package to be present at runtime simultaneously.

5.2 OSGi architecture

In order to meet the goals of the DreamCloud project, a real-time version of the OSGi framework is proposed. In order to introduce the required extensions that add real-time capabilities to the framework, it is necessary to describe the underlying concepts of OSGi and real-time systems in general.
The functionality of the OSGi framework is divided in six layers, as shown in Figure 2.

![Figure 2: The OSGi framework layers](image)

A brief introduction to the OSGi layers as follows:

- **Bundles**: the OSGi components provided by the developers.
- **Services**: connect bundles in a dynamic way by offering a publish-find-bind model.
- **Life Cycle**: provides the API to install, start, stop, update and uninstall bundles.
- **Modules**: defines how a bundle can import and export other bundles’ packages.
- **Security**: handles the security aspects, it can be applied to all OSGi layers.
- **Execution Environment**: defines what methods and classes are available in a specific platform.

5.3 OSGi implementations

It is not possible to extend the OSGi architecture with the proposed APIs without implementing these for a particular OSGi implementation. Hence a choice needs to be made.

There are lots of different OSGi implementations available:

- Apache Felix
- Concierge
- Knopflerfish
- Equinox (Eclipse).

Out of these *Concierge* was selected, as it has the lowest footprint and can easily be extended with the required features such as bundle migration or enforcement of real-time requirements.
5.4 Life Cycle layer

The Life Cycle layer adds bundles that can be dynamically installed, started, stopped, updated and uninstalled. Bundles rely on the OSGi implementation for class loading but need to implement the `BundleActivator` interface so that they can be managed at run time.

<table>
<thead>
<tr>
<th>Bundle State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTALLED</td>
<td>The bundle has been successfully installed.</td>
</tr>
<tr>
<td>RESOLVED</td>
<td>All Java classes that the bundle needs are available. This state indicates that the bundle is either ready to be started or has stopped.</td>
</tr>
<tr>
<td>STARTING</td>
<td>The bundle is being started, the BundleActivator.start method will be called, and this method has not yet returned. When the bundle has an activation policy, the bundle will remain in the STARTING state until the bundle is activated according to its activation policy.</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>The bundle has been successfully activated and is running; its Bundle Activator start method has been called and returned.</td>
</tr>
<tr>
<td>STOPPING</td>
<td>The bundle is being stopped. The BundleActivator.stop method has been called but the stop method has not yet returned.</td>
</tr>
<tr>
<td>UNINSTALLED</td>
<td>The bundle has been uninstalled. It cannot move into another state.</td>
</tr>
</tbody>
</table>

5.5 migration

It is natural to extend the life cycle layer with two states PAUSED and RESUMED so that bundle migration is supported.

On the API side however, there are lots of possibilities to implement the bundle migration. We propose an extension to the BundleActivator API:

```java
/* Existing OSGi interface: */
interface BundleActivator {
    void start(BundleContext context) throws Exception;
    void stop(BundleContext context) throws Exception;
}

/***
 * Proposed extension
 /**
 interface MigratableBundleActivator extends BundleActivator {
    Serializable pause(BundleContext context) throws Exception;
    void resume(BundleContext context, Serializable state) throws Exception;
}
```
It is important to note that this proposal is completely independent of the VM extensions discussed in section 7.2.

The `pause` method is used to serialize any state required by the bundle into an object. This serialized object then is transferred to the other node that will continue to run the bundle. The bundle then will be started with the help of the `resume` method that receives the serialized object.

If a bundle migration occurs, the methods `start` and `stop` are not invoked, so the chain is `start -> pause -> resume -> stop` where `start` and `pause` happen on the node that is migrated from and `resume` and `stop` occurs on the node that the bundle is migrated to.

### 5.5.1 error handling

If `pause` fails by throwing some exception, the bundle shall not be paused or stopped in any way. The idea is that a critical service is currently running and cannot be paused. Instead the exception can be logged or a message can be shown to the user that the request to pause the bundle failed.

The user may then chose whether to pause the bundle later or `stop` the bundle instead.

If `resume` fails by throwing some exception, it indicates a fatal error and the bundle state cannot be resumed. However, since the state has been serialized, it is safe to retry the `resume` operation on the same or on a different node.
6 High-Level APIs

This section describes how the OSGi architecture is extended with the required real-time and monitoring features.

6.1 Bundle resource monitoring

The Bundle Resource Monitoring mechanism ensures that resources of each bundle can be monitored separately. In order to implement this feature, a thread group is allocated to each bundle and all the threads spawned in that bundle must belong to that bundle thread group. Typical OSGi framework implementations, such as Concierge, do not implement this feature since it is not part of the OSGi specification. If this mechanism is not provided by the OSGi Framework implementation then upon a start of a bundle containing threads and/or thread groups, all the nested threads and thread groups will be spawned as child processes of "main" Java ThreadGroup. This thread group is the default one created by the Java runtime system upon a start up of the application.

Such behavior is problematic since there is a need to distinguish between the threads spawned in different bundles. The work of Miettinen et al. [12] on OSGi resources monitoring proposed an approach to deal with the issue. The author states that in order to achieve a bundle-level separation, it is necessary is to identify threads and thread groups that are associated with each bundle. In this way, the OSGi application execution is separated in different bundle-associated threads. Once this separation is achieved, it becomes easy to monitor all the threads associated with a bundle. This mechanism is necessary to implement many features implemented in this contribution such as Memory Limits Enforcement and Forcefully Thread Termination.

6.1.1 CPU Budgets

CPU budgets are per bundle / thread group. A CPU budget is defined by the following attributes:

- **Period**: The fixed phases in which the running time is subdivided.
- **Cost**: The CPU budget of the group, namely its processing time available per period.
- **Start**: Time at which the first period begins.
- **Deadline**: The latest permissible completion time measured from the start of the current period.

All these attributes are measured in milliseconds.

6.1.2 Cost enforcement

The cost enforcement is a mechanism that enables the application to recover from an overrun whenever cost monitoring fires an event,. There are several approaches to ensure that the application recovers from an overrun, but they may require the application to be designed to be interruptable or to poll for an interruption. For the purposes of this project, the recovery function is designed in such
a way that it does not require any interference on the user’s side. The logic of the recovery function is to reduce the priorities of the overrunning threads to a lower value, so that other threads can make progress. Therefore, the cost enforcement implementation relies on the thread priority level: it does not stop threads, but simply lowers or raises their priorities.

On a cost overrun event, the cost overrun handler is executed lowering the priorities of every thread belonging to the thread group of the bundle.

A periodic timer fires cost replenishment events. These are handled by the replenishment handler that then raises the priorities of these threads again.

The entire process that describes the cost monitoring and enforcement functionality is called budget replenishment mechanism.

### 6.1.3 Admission control protocol

Admission control protocol (ACP) acts as a filter; whenever a start of a bundle is requested, an acceptance test is performed in order to establish if the activation of the bundle is feasible or not.

The main purpose of the protocol is to control the load of the CPU in terms of the number of bundles that are simultaneously started in the OSGi framework. It may be unclear why providing an admission control protocol for the OSGi Framework since the RTSJ already provides admission control for Schedulable objects, namely RealtimeThreads and AsyncEventHandlers.

It is technically possible to assign certain parameters to Schedulable objects in order to specify their temporal requirements in terms of costs, start times, periods and deadlines. However, the RTSJ specification only states that this information can be used to prevent the admission of a schedulable object if the resulting system would not be feasible from a scheduling perspective. However this approach does not provide any detail about how to implement it. Since the OSGi Framework with real-time enhancements runs on JamaicaVM and the default feasibility analysis done by this real-time virtual machine cannot be applied to OSGi bundles, we do not to rely on the VM scheduling feasibility analysis. For this reason, a separate admission control protocol, designed specifically for OSGI bundles, needs to be implemented.

Since only the starting and updating phases of the life cycle can increase the load of a system, the ACP needs to be implemented just for these parameters:

- **Starting:** The start of a bundle that fails the acceptance test of ACP is denied. The acceptance test makes use of the values cost (the computational time) and the replenishment period that are provided with the bundle. If its activation is denied, the bundle has to wait until some other bundle in the start state is stopped and its resources are freed.
- **Updating:** Since updates are modelled as start+stop the action taken for "starting" suffices.

### 6.1.4 Memory budgets

There are two attributes that can be used to configure the enforcement of memory limits:

- **MIN_FREE_ALARM** minimum free available memory before raising an alarm to the user. If this memory value is reached, the platform generates a warning to the user.
MIN_FREE_OUT_OF_MEMORY represents the minimum free available memory before an action is taken. The bundle is stopped to avoid an OutOfMemoryError. If this value is set to 0 then no action is taken.

The RTSJ functionality called Asynchronous Transfer of Control is used to stop every thread in the bundle as soon as MIN_FREE_OUT_OF_MEMORY is reached.

6.2 Extensions to OSGi life cycle management

To implement the mentioned features, the standard OSGi life cycle needs to be extended. In particular, the following list describes the actions that are taken for each bundle’s life cycle state.

Starting
- First the bundle needs to be assigned a budget: The budget configuration file is parsed, assigns its parameters to the bundle and starts the replenishment periodic timer.
- Then the admission control needs to check the feasibility of the bundle.
- For starting the bundle the resource monitoring assigns a ThreadGroup to the bundle.

Stopping
- The forceful thread termination calls the fire() method to notify threads and real-time threads to be safely interrupted.
- The replenishment periodic timer for the bundle is disabled.
- The cost overrun handler for the bundle is disabled.

Active
- During the runtime of a bundle cost monitoring and enforcement need to run continuously.
- Likewise every single allocation could mean that the bundle uses more memory than allowed by its limit and so needs to be tracked by the memory limits enforcement mechanism.

Updating
- The admission control needs to check the feasibility of the bundle.
7 VM Level APIs

Unlike the low-level APIs described in section 4, the APIs specified at the VM level are meant to provide functions of external control for the VM. Nevertheless, these APIs are provided as Java APIs. These APIs are to be used by specific frameworks that communicate with the external scheduler in order to perform the required actions on the VM.

These actions in particular are: disabling and enabling of CPUs as well as (section 7.1) freezing and resuming of tasks (section 7.2).

7.1 Dynamic Adding and Removing of CPUs and Frequency Changes

The dynamic addition and removal of CPUs poses a significant challenge for the realtime scheduler of the Java virtual machine. It has to be ensured that communication between threads running on different CPUs would not suffer from CPUs being disabled. That means a thread that performs an *flip* operation to start a new garbage collection cycle performs a busy loop to wait for all other CPUs to finish all heap modifications such as write-barriers executed on reference assignments. Disabling of a CPU must result in any thread being blocked until this CPU is re-enabled.

Therefore, the APIs require notification before hardware CPUs are to be stopped and, vice versa, hardware CPUs must be restarted before the Java VM can be allowed to use these CPUs again.

Low-level APIs from the underlying operating system enable dynamic modification of the CPU frequencies (see section 4.2). To abstract from the underlying OS, it will help to have corresponding VM level APIs in Java. Therefore, the VM level APIs should include APIs to modify the CPU frequencies.

The following listing provides the API specification and JavaDoc documentation of the CPU control APIs proposed by DreamCloud.

```java
import java.util.BitSet;
import java.util.Set;

/**
 * CpuControl provides means to enable or disable CPUs for the use by the Java application.
 *
 * @return the set of available CPUs stored in a newly allocated instance of BitSet.
 */
public class CpuControl {

    /*----------------------------- methods -----------------------------*/

    /**
     * obtain the set of processors that are available for the Java VM to execute Java threads, independent of whether they are enabled or disabled.
     *
     * @return the set of available CPUs stored in a newly allocated instance of BitSet.
     */
```

```
public static BitSet allCpus();

/**
 * obtain the set of processors that are available for the Java VM to execute
 * Java threads, independent of whether they are enabled or disabled.
 * @param dest a bitset to receive the set of available CPUs. May
 * be null to ask for allocation of a new bitset.
 * @return the set of available CPUs. In case dest!=null, result will be equal
 * to dest, otherwise it will be a newly allocated instance of BitSet.
 */
public static BitSet allCpus(BitSet dest);

/**
 * obtain the set of processors that are currently enabled for the Java VM to
 * execute Java threads.
 * @param dest a bitset to receive the set of enabled CPUs. May
 * be null to ask for allocation of a new bitset.
 * @return the set of enabled CPUs stored in a newly allocated instance of
 * BitSet.
 */
public static BitSet enabledCpus();

/**
 * obtain the set of processors that are currently enabled for the Java VM to
 * execute Java threads.
 * @param dest a bitset to receive the set of enabled CPUs. May
 * be null to ask for allocation of a new bitset.
 * @return the set of enabled CPUs. In case dest!=null, result will be equal
 * to dest, otherwise it will be a newly allocated instance of BitSet.
 */
public static BitSet enabledCpus(BitSet dest);

/**
 * obtain the set of processors that are currently disabled for the Java VM to
 * execute Java threads.
 * @return the set of disabled CPUs stored in a newly allocated instance of
 * BitSet.
 */
public static BitSet disabledCpus();

/**
 * obtain the set of processors that are currently disabled for the Java VM to
 * execute Java threads.
 *
* @param dest a bitset to receive the set of disabled CPUs. May be null to ask for allocation of a new bitset.
* @return the set of disabled CPUs. In case dest!=null, result will be equal to dest, otherwise it will be a newly allocated instance of BitSet.
*/
public static BitSet disabledCPUs(BitSet dest);

/**
 * disable the CPUs that are specified by the given BitSet. Any Java threads currently running on any of these CPUs will be preempted and put into READY state or moved to another available CPU if their affinity set permits and other CPUs are available. The Jamaica scheduler will unschedule all tasks on any of these CPUs before this call returns.
 * After this call, <tt>enabledCpus()</tt> will equal old <tt>enabledCpus().andNot(cpus)</tt>.
 * Any threads with an [@link javax.realtime.Affinity] that is disjoint with the new <tt>enabledCpus()</tt> will no longer be able to run.
 * After this call has returned, the hardware CPUs corresponding to <tt>cpus</tt> may be stopped. Stopping the hardware CPUs earlier may otherwise result in deadlocks or livelocks due to spin loops used for inter-thread communication.
 * @param cpus the set of CPUs that should be disabled for use by the JamaicaVM scheduler.
 * @throws IllegalArgumentException if <tt>!cpus.andNot(allCpus()).isEmpty()</tt> (cpus contains a CPU that is not in <tt>allCpus()</tt>, i.e., the caller attempts to disable a non-existing CPU) or if <tt>enabledCpus().andNot(cpus).isEmpty()</tt> (cpus is a superset of enabled CPUs, i.e., the caller attempts to disable all CPUs).
 * @throws SecurityException If a security manager has been installed and it denies [@link RuntimePermission]<tt>"CpuControl.disableCpus"</tt> disabling of Cpus.
*/
public static void disableCpus(BitSet cpus);

/**
 * enable the CPUs that are specified by the given BitSet.
 * After this call, <tt>enabledCpus()</tt> will equal old <tt>enabledCpus().or(cpus)</tt>.
 * Any threads with an [@link javax.realtime.Affinity] that is no longer disjoint with the new <tt>enabledCpus()</tt> will again be able to run.
 * Before this method is called, the hardware CPUs corresponding to
enableCpus(BitSet cpus);

availableFrequencies(int cpuId);
currentFrequency(int cpuId, int freq);
setFrequency(int cpuId, int freq);
/**
 * Request the current minimum frequency of the given CPU. For a CPU whose
 * frequency is changed dynamically by the OS, this defines the lower bound of
 * the permitted range of frequency values.
 * @param cpuId the id of a CPU.
 * @return the minimum frequency in kHz.
 */
public static int minFrequency(int cpuId);

/**
 * Request the current maximum frequency of the given CPU. For a CPU whose
 * frequency is changed dynamically by the OS, this defines the upper bound of
 * the permitted range of frequency values.
 * @param cpuId the id of a CPU.
 * @return the minimum frequency in kHz.
 */
public static int maxFrequency(int cpuId);

/**
 * For a CPU whose frequency is changed dynamically by the OS, set the minimum
 * frequency of the given CPU to the given value
 * @param cpuId the id of a CPU.
 * @param the desired minimum frequency in kHz.
 * @throws SecurityException If a security manager has been installed and it
denies [@link RuntimePermission]"CpuControl.disableCpus"</tt> disabling of Cpus.
 */
public static void setMinFrequency(int cpuId, int freq);

/**
 * For a CPU whose frequency is changed dynamically by the OS, set the maximum
 * frequency of the given CPU to the given value
 * @param cpuId the id of a CPU.
 * @param the desired maximum frequency in kHz.
 * @throws SecurityException If a security manager has been installed and it
denies [@link RuntimePermission]"CpuControl.disableCpus"</tt> disabling of Cpus.
 */
public static void setMaxFrequency(int cpuId, int freq);
7.2 Freeze and Resume of Tasks

Live migration of tasks from one Java VM to another Java VM running on a different machine is a difficult task, in particular if the machines involved may use different hardware architecture or different operating systems. The Java VM approach allows for an abstraction that nevertheless may enable such a migration between very different underlying systems.

Migration support is much simpler if the threads that are to be migrated are in a defined state that enables freezing of these threads and resuming without having to reconstruct the exact call context and stack frames on the target systems. We therefore intend to provide APIs for different levels of freeze-and-resume support: Basic support that requires migrated threads to be stopped at a freeze point, and full support that enables migration of threads at an arbitrary point during execution.

```java
/**
 * Migration provides VM-level support for migration of tasks from one Java VM to another Java VM.
 */
public class Migration {

    /**
     * Freeze a Task that is identified by a set of Threads running in one ThreadGroup, a set of classes loaded by a given class loader and memory allocation through a given AllocationHandler.
     *
     * First, the code will wait for all threads to execute a {@code freezePoint}. The threads will be stopped at these points. Then, all the memory allocated through the given allocation handler, all the classes loaded through the given loader and all the threads in the given group will be serialized into an stream of bytes that can be {@code link resume}d on another virtual machine.
     *
     * @param group a ThreadGroup that specifies all the threads of this task that should be frozen.
     *
     * @param load the ClassLoader that identifies all the classes that belong to this task.
     *
     * @param handler the allocation handler that identifies all the objects that belong to the task.
     *
     * @return a stream of bytes containing all the information on the frozen task.
     *
     * @throws SecurityException If a security manager has been installed and it denies {@code link RuntimePermission}="Migration.freeze"
     */
```
public static InputStream freeze(ThreadGroup group,
    ClassLoader loader,
    AllocationHandler handler);

/**
 * Resume a Task that was serialized by a previous call to \[\texttt{freeze}\].
 * All the objects, classes and threads of the frozen task will be recreated
 * in the current VM within the framework of the provided ThreadGroup,
 * ClassLoader and allocation handler. The tasks will then resume at the
 * freeze points they were stopped when freeze() was called.
 * @param data the stream of data produced by \[\texttt{freeze}\].
 * @param group the ThreadGroup the threads will be added to.
 * @param load the ClassLoader that will be used to load the classes of the
 * task.
 * @param handler the allocation handler that will be used to allocate the
 * memory of the task.
 * @throws SecurityException If a security manager has been installed and it
denies \[\texttt{RuntimePermission}"(Migration.resume")\] resuming of threads.
*/
public static void resume(InputStream data,
    ThreadGroup group,
    ClassLoader loader,
    AllocationHandler handler);

/**
 * Permit freezing of the current thread for migration. This is a no-op in
 * case the thread is not migrated. However, in case of migration, this call
 * will not return, but will execute \[\texttt{run()}\] when it is resumed. The
 * thread will terminate once \[\texttt{run()}\] returns.
 * At the point this function is called, the current thread must not have
 * entered any Java monitor.
 * This should be used in the \[\texttt{run()}\] methods of all threads that
 * should be part of a task that permits migration. An example use is as
 * follows:
 *<code>
 * class MyThread extends Thread
 * {
 *     public void run()
 *     {
 *         // Migrate
 *         //...</code>
```java
*     init();
*     execute();
* }
*
* public void execute()
* {
*     while (true)
*     {
*         // permit migration, on resume, continue with execute():
*         freezePoint() -> execute();
*         Action a = getNextAction();
*         a.perform();
*     }
* }
* </code>
*
* @param r the code to execute after a resume.
* @throws IllegalStateException if the current thread holds any Java monitor.
*/
public static void freezePoint(Runnable r);

/**<*
   * Wait for notification and permit freezing of the current thread for
   * migration. This is equal to o.wait() in case the thread is not migrated.
   * However, in case of migration, this call will not return, but will execute
   * <tt>r.run()</tt> when it is resumed. The thread will terminate once
   * <tt>r.run()</tt> returns.
   *
   * At the point this function is called, the current thread must not have
   * entered any Java monitor.
   *
   * This should be used in the <tt>run()</tt> methods of all threads that
   * should be part of a task that permits migration. An example use is as
   * follows:
   *
   * <code>
   * class MyThread extends Thread
   * {
   *     List<Action> actionList;
   *     
   *     public void run()
   *     {
   *         init();
   *         execute();
   *     }
   *     
   *     public void execute()
   *     {
```
D4.1 – API Specification

```java
while (true)
{
    Action action;
    synchronized (actionList)
    {
      while (actionList.isEmpty())
      {
        // permit migration, on resume, continue with execute():
        waitingFreezePoint(actionList, () -> execute());
      }
      action = actionList.removeHead();
    }

    action.perform();
}
</code>

* @param o the object that we are waiting for notification.
* @param r the code to execute after a resume.
* @throws IllegalMonitorStateException if the current thread does not own the
*     monitor associated with o.
* @throws IllegalStateException if the current thread holds any Java monitor
*     except o.
*/
public static void waitingFreezePoint(Object o, Runnable r);
```

7.3 Freeze and Resume of whole VM

Alternatively to migrating single tasks identified by a ThreadGroup, a ClassLoader, and an
AllocationHandler, the Java VM may provide means to migrate the whole state of the VM
to another system. In this case, Java APIs are not sufficient since the current VM instance on the
original system will not survive the freezing and the whole VM state will be reconstructed on the
target system.

Instead, a Java API may dump the VM state to a file, while the VM may permit a command line
option to resume execution loading a state from a file. The freeze API would look as follows:

```java
/**
 * Freeze all threads running on this VM and store the state in a new file
 * with the given name.
 *
 * First, the code will wait for all threads to execute a @freezePoint. The
 * threads will be stopped at these points. Then, all the memory allocated in
```
The VM, all the classes loaded and all the threads will be serialized into an stream of bytes and saved into a file with the given name such that it can be resumed on another virtual machine.

This call never returns. When resumed, the given Runnable will be executed by the resumed version of the current thread on the target system.

@throws SecurityException If a security manager has been installed and it denies [link RuntimePermission]("Migration.freezeVM") freezing of the VM.

```java
public static void freeze(String fileName, Runnable r);
```

To resume using the stored state, a VM command line option would permit reloading the state from a given file as follows:

```
> jamaicavm –classpath <classpath> –resume <filename>
```

After a resume, all threads will continue execution after the freeze point they were stopped in. The thread that called freeze will continue execution in the Runnable instance provided as the second argument to freeze.

### 7.4 Enhancing freeze and resume

In the general case, freezing and resuming of threads that are not stopped at a freezePoint will make this approach much more flexible and usable. Threads that are not running are typically in one of the following states:

- blocked in Java code waiting for notification, i.e. in method Object.wait().
- blocked in Java code waiting for notification or a timeout, i.e., in methods such as Object.wait(long) or Thread.sleep().
- blocked waiting to enter a Java monitor owned by another thread
- blocked in a system library native call, e.g., Socket.accept()
- blocked in an application specific native call

If the virtual machine’s freeze and resume mechanism would be extended to transfer the execution stack of threads as well, these cases could be handled as well. The difficulties that will arise here are as follows:

- Code waiting for notification can be handled straightforward in a way very similar to code executing a waitingFreezePoint.
- Code blocked with a timeout is similar, but we have to map the time on the source system to a time on the target.
• Migration of threads blocked waiting to enter a monitor would mean that the internal state of Java monitors would need to be transferred. Since this state typically is resolved soon (when the monitor becomes available again), it might be sufficient to wait for the thread to no longer be in this state before freezing it.

• Code that is blocked in a system library native call requires special handling to resume the state on the target depending on the kind of library call. The resume mechanism might try to restore the exact state on the target, or, alternatively, return with an error from the native call if this might make sense.

• Code that is blocked in a application specific native call cannot be supported by a general VM migration approach so should be forbidden.

The approach to be taken by DreamCloud is to first gain experience with migration of different tasks and then to decide what additional support would be most useful.
References


