Red Hat Data Grid 7.2

Performance Tuning Guide

For Use with JBoss Data Grid 7.2

Last Updated: 2019-07-18

Red Hat Data Grid 7.2 Performance Tuning Guide

For Use with JBoss Data Grid 7.2

Legal Notice

Abstract

This guide presents information about the common tunables of Red Hat JBoss Data Grid 7.2.

Table of Contents

CHAPTER 1. INTRODUCTION

1.1. RED HAT JBOSS DATA GRID

Red Hat JBoss Data Grid is a distributed in-memory data grid, which provides the following capabilities:

- Schemaless key-value store – JBoss Data Grid is a NoSQL database that provides the flexibility to store different objects without a fixed data model.
- Grid-based data storage – JBoss Data Grid is designed to easily replicate data across multiple nodes.
- Elastic scaling – Adding and removing nodes is simple and non-disruptive.
- Multiple access protocols – It is easy to access the data grid using REST, Memcached, Hot Rod, or simple map-like API.

1.2. SUPPORTED CONFIGURATIONS

The set of supported features, configurations, and integrations for Red Hat JBoss Data Grid (current and past versions) are available at the Supported Configurations page at https://access.redhat.com/articles/115883.

1.3. COMPONENTS AND VERSIONS

Red Hat JBoss Data Grid includes many components for Library and Remote Client-Server modes. A comprehensive (and up to date) list of components included in each of these usage modes and their versions is available in the *Red Hat JBoss Data Grid Component Details* page at https://access.redhat.com/articles/488833

1.4. ABOUT PERFORMANCE TUNING IN RED HAT JBOSS DATA GRID

The Red Hat JBoss Data Grid *Performance Tuning Guide* provides information about optimizing and configuring specific elements within the product in an attempt to improve the JBoss Data Grid implementation performance.

Due to each business case being different it is not possible to provide a "one size fits all" approach to tuning. Instead, this guide attempts to present various elements that have proven to be effective in increasing performance, along with potential values to begin testing for a user's specific case. It is imperative that after each individual change performance be measured once again to isolate any improvements or negative effects; this approach allows a methodical approach to establishing a baseline of parameters.

When testing it is strongly recommended to use a workload that closely mirrors the expected production load. Using any other workload may result in performance differences between the testing and production environments.

CHAPTER 2. JAVA VIRTUAL MACHINE SETTINGS

2.1. JAVA VIRTUAL MACHINE SETTINGS

Tuning a Java Virtual Machine (JVM) is a complex task because of the number of configuration options and changes with each new release.

The recommended approach for performance tuning Java Virtual Machines is to use as simple a configuration as possible and retain only the tuning that is beneficial, rather than all tweaks. A collection of tested configurations for various heap sizes are provided after the parameters are discussed.

Heap Size

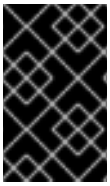
The JVM's heap size determines how much memory is allowed for the application to consume, and is controlled by the following parameters:

- **-Xms** - Defines the minimum heap size allowed.
- **-Xmx** - Defines the maximum heap size allowed.
- **-XX:NewRatio** - Define the ratio between young and old generations. Should not be used if **-Xmn** is enabled.
- **-Xmn** - Defines the minimum and maximum value for the young generation.

In the majority of instances **Xms** and **Xmx** should be identical to prevent dynamic resizing of the heap, which will result in longer garbage collection periods.

Garbage Collection

The choice of which garbage collection algorithm to use will largely be determined by whether throughput is valued over minimizing the amount of the time the JVM is fully paused. As JBoss Data Grid applications are often clustered it is recommended to choose a low pause collector, such as Garbage First (G1) collector, to prevent network timeouts.



IMPORTANT

The garbage collector, and options for each collector, will vary wildly based on the type of the workload. For this reason it is strongly recommended to test a variety of policies and tunings to determine the best configuration for each environment.

The following sections, beginning in Garbage Collectors discuss common tunables for three different garbage collection policies.

Large Pages

Large, or Huge, Pages are contiguous pages of memory that are much larger than what is typically defined at the OS level. By utilizing large pages the JVM will have access to memory that is much more efficiently referenced, and memory that may not be swapped out, resulting in a more consistent behavior from the JVM. Large pages are discussed in further detail at About Page Memory.

- **-XX:+UseLargePages** - Instructs the JVM to allocate memory in Large Pages. These pages must be configured at the OS level for this parameter to function successfully.

Server Configuration

This parameter relates to JIT (Just-In-Time) compilation, which requires extended loading times during startup, but provides extensive compilation and optimization benefits after the startup process completes.

- **-server** - Enables server mode for the JVM.

2.2. GARBAGE COLLECTORS

2.2.1. Parallel Collector

The parallel collector maximizes throughput at the cost of all collections pausing the JVM. It behaves as the traditional, serial, collector, but uses multiple garbage collection threads running in parallel to collect unreferenced objects across both the old and young generations.

For additional information regarding the Parallel Collector refer to the JVM vendor's documentation.

Commonly Configured Options

- **-XX:+UseParallelGC** - This option enables the parallel collector on the JVM. For JVMs with **-server** specified this is the default collector.
- **-XX:+UseParallelOldGC** - This option enables the parallel collector on the old generation, and should be used when the parallel collector is enabled.
- **-XX:MaxGCPauseMillis=<N>** - This option is a suggestion regarding the maximum amount of time, in milliseconds, for the JVM to be paused due to a garbage collection. This value is not a guarantee that collections will result in the specified time or less; however, the collector will attempt to keep pauses shorter than the specified value by making adjustments to the throughput of the application.
- **-XX:ParallelGCThreads=<N>** - By default the parallel collector uses a number of threads based on the number of cores in the system. For systems with 8 or fewer cores N is set to the value of cores, but for larger systems N is defined as **3 + ((ncpus * 5) / 8)**. This option is particularly important when multiple JVMs are running on the same system, as each JVM will calculate its own garbage collection thread pool independently of any other JVMs. In these instances it is recommended to define the number of garbage collection threads per JVM, with the total number of threads across all JVMs not exceeding the number of cores on the system.

2.2.2. Concurrent Mark Sweep (CMS) Collector

The Concurrent Mark Sweep (CMS) collector performs collections in a series of phases, and only two of these pause the JVM. This behavior makes the CMS collector useful for applications that want to minimize pause time, allowing the collector to run simultaneously with the application.

For additional information regarding the CMS Collector refer to the JVM vendor's documentation.

Commonly Configured Options

- **-XX:+UseConcMarkSweepGC** - Enables the CMS collector.
- **-XX:+UseParNewGC** - Uses a parallel version of the young generation collector with the CMS collector, minimizing pause times by allowing multiple collection threads functioning in parallel.
- **-XX:+CMSClassUnloadingEnabled** - Enables the CMS collector to sweep PermGen and remove unload classes that are no longer used.

- **-XX:+CMSScavengeBeforeRemark** - Performs a young collection before the CMS **remark** phase, allowing the collector to minimize the pause time of the **remark** phase by having already pruned the young generation.

- **-XX:MaxGCPauseMillis=<N>** - This option is a suggestion regarding the maximum amount of time, in milliseconds, for the JVM to be paused due to a garbage collection. This value is not a guarantee that collections will result in the specified time or less; however, the collector will attempt to keep pauses shorter than the specified value by making adjustments to the throughput of the application.

- **-XX:CMSInitiatingOccupancyFraction=<N>** - This option controls when the CMS collector begins a concurrent collection of the tenured generation. By default, the tenured generation will begin to be collected when it is approximately 92% occupied; however, it may be useful to lower this threshold so that a collection begins earlier, reducing the chance of an **OutOfMemoryError**.

- **-XX:+UseCMSInitiatingOccupancyOnly** - Enabling this option disables the CMS auto-tuning. If the load on the server fluctuates, such as drastically lower load overnight, it may be useful to disable this auto-tuning so that the JVM does not become adapted to lower load periods and delay collections during busier times.

2.2.3. Garbage First (G1) Collector

The Garbage First (G1) collector is another low-pause collector that divides the memory into a number of equally sized regions, each of which may be collected independently of another region. This allows the G1 collector to collect regions with the most unreachable objects, allowing more active regions to be used by the application without interference from the garbage collector.

For additional information regarding the G1 Collector refer to the JVM vendor's documentation.

Commonly Configured Options

- **-XX:+UseG1GC** - Enables the G1 collector.

- **-XX:+UseStringDeduplication** - Enables the String deduplication feature, which will search for duplicate strings, update references so that only one active reference to the string exists, and then collect the duplicate. This feature reduces the memory footprint of the application.

- **-XX:InitiatingHeapOccupancyPercent=<N>** - Sets the percentage of the entire heap at which the G1 collector will begin a marking cycle. Defaults to **45** percent of the total heap.

- **-XX:MaxGCPauseMillis=<N>** - This option is a suggestion regarding the maximum amount of time, in milliseconds, for the JVM to be paused due to a garbage collection. This value is not a guarantee that collections will result in the specified time or less; however, the collector will attempt to keep pauses shorter than the specified value by making adjustments to the throughput of the application.

NOTE

Due to the way the G1 collector defines regions it is not recommended to explicitly define a young generation, as this will override the pause time goal.

2.3. MEMORY REQUIREMENTS

Minimum Requirements

The default minimum amount of memory required to run JBoss Data Grid varies based on the configuration in use.

For Library mode the server should have a minimum of 1 GB of RAM for a single JBoss Data Grid instance, and is configured using the JVM arguments on the command line.

For Client-Server mode the requirements are determined based on the configuration file in use:

- *standalone.conf* - The server should have a minimum of 2 GB of RAM for a single JBoss Data Grid instance, as the default heap may grow up to 1.3 GB, and Metaspace may occupy up to 256 MB of memory.
- *domain.conf* - The server should have a minimum of 2.5 GB of RAM for a single JBoss Data Grid managed domain consisting of two JBoss Data Grid server instances, as the heap may grow up to 512 GB for the domain controller, the heap for each server instance may grow up to 256 MB, and the Metaspace may occupy up to 256 MB of memory for the domain controller and each server instance.

Recommended Memory Requirements

There is no official memory recommendation for JBoss Data Grid, as the memory requirements will vary depending on the application and workload in use; however, there are guidelines for memory usage as outlined below:

- The amount of data in the heap should not exceed **50%** of the total memory when only basic operations, such as **put**, **get**, **remove**, are used.
- When using analytics, such as Queries, Streams, Distributed Executors, only **33%** of the heap should be occupied. The extra memory allows the system to deal with memory allocations from these analytic operations with minimal impact.

It is strongly recommended to test each application, measuring throughput and collection times to determine if they are acceptable for the application in question.

Physical Memory Requirements

Each JVM process has a memory footprint that adheres to the following formula:

```
JvmProcessMemory = JvmHeap + Metaspace + (ThreadStackSize * Number of Threads) + Jvm-native-c++-heap
```

Adjusting these values are discussed in Java Virtual Machine Settings.

The **Jvm-native-c++-heap** will vary based on the native threads and if any native libraries are used; however, for a default installation it is safe to assume this will use no more than **256** MB of memory.

2.4. JVM EXAMPLE CONFIGURATIONS

The following configurations have been tested internally, and are provided as a baseline for customization. These configurations show various heap sizes, which allows users to find one appropriate for their environment to begin testing:

8GB JVM using CMS

```
-server
-Xms8192m
```

```
-Xmx8192m
-XX:+UseLargePages
-XX:NewRatio=3
-XX:+UseConcMarkSweepGC
-XX:+UseParNewGC
-XX:+DisableExplicitGC
```

32GB JVM using G1

```
-server
-Xmx32G
-Xms32G
-XX:+UseLargePages
-XX:+UseG1GC
-XX:InitiatingHeapOccupancyPercent=70
-XX:MaxGCPauseMillis=3000
-XX:+DisableExplicitGC
```

64GB JVM using G1

```
-server
-Xmx64G
-Xms64G
-XX:+UseLargePages
-XX:+UseG1GC
-XX:InitiatingHeapOccupancyPercent=70
-XX:MaxGCPauseMillis=3000
-XX:+DisableExplicitGC
```

CHAPTER 3. CONFIGURE PAGE MEMORY

3.1. ABOUT PAGE MEMORY

A memory page is a fixed size, continuous block of memory and is used when transferring data from one storage medium to another, and to allocate memory. In some architectures, larger sized pages are available for improved memory allocation. These pages are known as large (or huge) pages.

The default memory page size in most operating systems is 4 kilobytes (kb). For a 32-bit operating system the maximum amount of memory is 4 GB, which equates to 1,048,576 memory pages. A 64-bit operating system can address 18 Exabytes of memory (in theory), resulting in a very large number of memory pages. The overhead of managing such a large number of memory pages is significant, regardless of the operating system.

Large memory pages are pages of memory which are significantly larger than 4 kb (usually 2 Mb). In some instances it is configurable from 2 MB to 2 GB, depending on the CPU architecture.

Large memory pages are locked in memory, and cannot be swapped to disk like normal memory pages. The advantage for this is if the heap is using large page memory it can not be paged or swapped to disk so it is always readily available. For Linux, the disadvantage is that applications must attach to it using the correct flag for the **shmget()** system call. Additionally, the proper security permissions are required for the **memlock()** system call. For any application that does not have the ability to use large page memory, the server behaves as if the large page memory does not exist, which can be a problem.

For additional information on page size refer to the Red Hat Enterprise Linux Configuring Hugetlb Huge Pages.

3.2. CONFIGURE PAGE MEMORY

Page memory configuration to optimize Red Hat JBoss Data Grid's performance must be implement at the operating system level and at the JVM level. The provided instructions are for the Red Hat Enterprise Linux operating system. Use both the operating system level and JVM level instructions for optimal performance.

Configure Page Memory for Red Hat Enterprise Linux

1. Set the Shared Memory Segment Size
   As root, set the maximum size of a shared memory segment in bytes; below we define this to be 32 GB:

   ```
   # echo "kernel.shmmax = 34359738368" >> /etc/sysctl.conf
   ```

2. Set the Huge Pages
   The number of huge pages is set to the total amount of memory the JVM will consume (heap, meta space, thread stacks, native code) divided by the **Hugepagesize**. In Red Hat Enterprise Linux systems **Hugepagesize** is set to 2048 KB.

   a. The number of huge pages required can be determined by the following formula:

   ```
   Heap + Meta space + Native JVM Memory + (Number of Threads * Thread Stack Size)
   ```

   b. Assuming a JVM with a 32 GB Heap, 2 GB of Meta space, a 512 MB native footprint, and 500 threads, each with a default size of 1 MB per thread, we have the following equation.

```
32*(1024*1024*1024) + 2*(1024*1024*1024) + 512*(1024*1024) + (500 * 1024*1024)
```

c. The resulting value can now be converted to hugepages. Since there are 2048 KB in a single hugepage we perform the following:

```
37568380928 / (2*1024*1024)
```

d. As root, set the number of huge pages determined from the previous steps to be allocated to the operating system:

```
# echo "vm.nr_hugepages = 17914" >> /etc/sysctl.conf
```

3. Assigned Shared Memory Permissions
As root, set the ID of the user group that is allowed to create shared memory segments using the *hugetlb_shm_group* file. This value should match the group id of the user running the JVM:

```
# echo "vm.hugetlb_shm_group = 500" >> /etc/sysctl.conf
```

4. Update the Resource Limits
To allow a user to lock the required amount of memory, update the resource limits in the */etc/security/limits.conf* file by adding the following:

```
jboss     soft   memlock     unlimited
jboss     hard   memlock     unlimited
```

This change allows the user **jboss** to lock the system's available memory.

5. Configure Authentication using PAM
Linux's PAM handles authentication for applications and services. Ensure that the configured system resource limits apply when using **su** and **sudo** as follows:

- Configure PAM for su
Add the following line to the */etc/pam.d/su* file:

```
session    required   pam_limits.so
```

- Configure PAM for sudo
Add the following line to the */etc/pam.d/sudo* file:

```
session    required   pam_limits.so
```

6. Reboot the system for the changes to take effect. Since Huge Pages allocate a contiguous block of memory these must be allocated at system boot; attempts to claim these dynamically while the system is running may result in system hangs if the memory is unable to be reclaimed.

Procedure: Configure Page Memory for the JVM

1. Set the Heap Size
Use the **-Xms** and **-Xmx** parameters to set the minumum and maximum heap sizes for your JVM, as discussed in Java Virtual Machine Settings .

2. Enable Large Pages

Enabled large pages for the JVM by adding the following parameter, as discussed in Java Virtual Machine Settings:

```
-XX:+UseLargePages
```

CHAPTER 4. NETWORKING CONFIGURATION

4.1. NETWORK SETTINGS

As JBoss Data Grid is distributed data grid, adjusting the underlying network settings may improve performance. This chapter seeks to address the most common tunables used, and provide recommendations on their configuration.

4.2. ADJUSTING SEND/RECEIVE WINDOW SETTINGS

In many environments packet loss may be caused by the buffer space not being large enough to receive all of the transmissions, resulting in packet loss and costly retransmissions. As with all tunables, it is important to test these settings with a workload that mirrors what is expected to determine an appropriate value for your environment.

The kernel buffers may be increased by following the below steps:

1. Adjust Send and Receive Window Sizes
   The window sizes are set per socket, which affects both TCP and UDP. These may be adjusted by setting the size of the send and receive windows in */etc/sysctl.conf* file as root:

   a. Add the following line to set the send window size to a value of **640** KB:

   ```
   net.core.wmem_max=655360
   ```

   b. Add the following line to set the receive window size to a value of **25** MB:

   ```
   net.core.rmem_max=26214400
   ```

2. Increase TCP Socket Sizes The TCP send and receive socket sizes are also controlled by a second set of tunables, which may be defined no larger than the system settings set previously.

   a. Increase the TCP send socket size by adjusting the **net.ipv4.tcp_wmem** tuple. This tuple consists of three values, representing the **minimum**, **default**, and **maximum** values for the send buffer. To set it to the same size as the send socket above we would add the following line to */etc/sysctl.conf*:

   ```
   net.ipv4.tcp_wmem = 4096  16384  655360
   ```

   b. Increase the TCP receive socket by adjusting the **net.ipv4.tcp_rmem** tuple. This tuple consists of three values, representing the **minimum**, **default**, and **maximum** values for the receive buffer. To set it to the same size as the receive socket above we would add the following line to */etc/sysctl.conf*:

   ```
   net.ipv4.tcp_rmem = 4096  87380  26214400
   ```

3. Apply Changes Immediately
   Optionally, to load the new values into a running kernel (without a reboot), enter the following command as root:

   ```
   # sysctl -p
   ```

   If the user reboots after the second step, the final step is unnecessary.

4.3. FLOW CONTROL

JGroups utilizes flow control for TCP connections to prevent fast senders from overflowing slower receivers. This process prevents packet loss by controlling the network transmissions, ensuring that the targets do not receive more information than they can handle.

Some network cards and switches also perform flow control automatically, resulting in a performance decrease due to duplicating flow control for TCP connections.

NOTE

The following content will vary based on the network topology in the site. As with all performance adjustments, each time a change is made benchmark tests should be executed to determine any performance improvements or degradations.

4.3.1. TCP Connections

If the network card or switch performs flow control it is recommended to disable flow control at the ethernet level, allowing JGroups to prevent packet overflows. Any of the following will disable flow control:

- **Option 1**: For managed switches, flow control may be disabled at the switch level, typically through a web or ssh interface. Full instructions on performing this task will vary depending on the switch, and will be found in the switch manufacturer's documentation.
- **Option 2**: In RHEL it is possible to disable this at the NIC level. This may be disabled by using the following command:

```
/sbin/ethtool -A $NIC tx off rx off
```

4.3.2. UDP Connections

JGroups does not perform flow control for UDP connections, and due to this it is recommended to have flow control enabled.

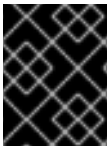
Flow control may be enabled using one of the following methods:

- **Option 1**: For managed switches, flow control may be enabled at the switch level, typically through a web or ssh interface. Full instructions on performing this task will vary depending on the switch, and will be found in the switch manufacturer's documentation.
- **Option 2**: In RHEL it is possible to enable this at the NIC level. This may be enabled by using the following command:

```
/sbin/ethtool -A $NIC tx on rx on
```

4.4. JUMBO FRAMES

By default the maximum transmission unit (MTU) is 1500 bytes. Jumbo frames should be enabled when the MTU is larger than the default, or when smaller messages are aggregated to be larger than 1500 bytes. By enabling jumbo frames, more data is sent per ethernet frame. The MTU may be increased to a value up to 9000 bytes.



IMPORTANT

For jumbo frames to be effective every intermediate network device between the sender and receiver must support the defined MTU size.

To enable jumbo frames add the following line to the configuration script of the network interface, such as */etc/sysconfig/network-scripts/ifcfg-eth0*:

```
MTU=9000
```

4.5. TRANSMIT QUEUE LENGTH

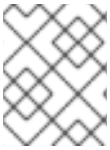
The transmit queue determines how many frames are allowed to reside in the kernel transmission queue, with each device having its own queue. This value should be increased when a large number of writes will be expected over a short period of time, resulting in a potential overflow of the transmission queue.

To determine if overruns have occurred the following command may be executed against the device. If the value for **overruns** is greater than 0 then the transmission queue length should be increased:

```
ip -s link show $NIC
```

This value may be set per device by using the following command:

```
ip link set $NIC txqueuelen 5000
```



NOTE

This value does not persist across system restarts, and as such it is recommended to include the command in a startup script, such as by adding it to */etc/rc.local*.

4.6. NETWORK BONDING

Multiple interfaces may be bound together to create a single, bonded, channel. Bonding interfaces in this manner allows two or more network interfaces to function as one, simultaneously increasing the bandwidth and providing redundancy in the event that one interface should fail. It is strongly recommended to bond network interfaces should more than one exist on a given node.

Full instructions on bonding are available in the Networking Guide, available in Red Hat Enterprise Linux's Product Documentation.

CHAPTER 5. RETURN VALUES

5.1. ABOUT RETURN VALUES

Values returned by cache operations are referred to as return values. In Red Hat JBoss Data Grid, these return values remain reliable irrespective of which cache mode is employed and whether synchronous or asynchronous communication is used.

5.2. DISABLING RETURN VALUES

Library Mode

By default, in Library Mode, JBoss Data Grid returns previous values in the cache for **put()** and **remove()** API operations.
To conserve resources, you can disable these return values where you do not require them.

Use the **IGNORE_RETURN_VALUES** flag from the **org.infinispan.context.Flag** class as follows:

1. Set the **IGNORE_RETURN_VALUES** flag. This flag signals that the operation's return value is ignored. For example:

```
cache.getAdvancedCache().withFlags(Flag.IGNORE_RETURN_VALUES)
```

2. Set the **SKIP_CACHE_LOAD** flag. This flag does not load entries from any configured CacheStores. For example:

```
cache.getAdvancedCache().withFlags(Flag.SKIP_CACHE_LOAD)
```

For more information, see Flag.IGNORE_RETURN_VALUES

Remote Client/Server Mode

By default, when remotely accessing caches in Server Mode, JBoss Data Grid disables return values so that **put()** and **remove()** API operations do not return previous values in the cache.
You can set the **FORCE_RETURN_VALUE** in **org.infinispan.client.hotrod.Flag** to return previous values for **put()** and **remove()** API operations. However, doing this adds performance costs through serialization and network requirements.

For more information, see Flag.FORCE_RETURN_VALUE

CHAPTER 6. MARSHALLING

6.1. MARSHALLING

Marshalling is the process of converting Java objects into a format that is transferable over the wire. Unmarshalling is the reversal of this process where data read from a wire format is converted into Java objects.

Red Hat JBoss Data Grid uses marshalling and unmarshalling to:

- transform data for relay to other JBoss Data Grid nodes within the cluster.
- transform data to be stored in underlying cache stores.

6.2. ABOUT THE JBOSS MARSHALLING FRAMEWORK

Red Hat JBoss Data Grid uses the JBoss Marshalling Framework to marshall and unmarshall Java **POJO**s. Using the JBoss Marshalling Framework offers a significant performance benefit, and is therefore used instead of Java Serialization. Additionally, the JBoss Marshalling Framework can efficiently marshall Java **POJO**s, including Java classes.

The Java Marshalling Framework uses high performance **java.io.ObjectOutput** and **java.io.ObjectInput** implementations compared to the standard **java.io.ObjectOutputStream** and **java.io.ObjectInputStream**.

6.3. CUSTOMIZING MARSHALLING

Instead of using the default Marshaller, which may be slow with payloads that are unnecessarily large, objects may implement **java.io.Externalizable** so that a custom method of marshalling/unmarshalling classes is performed. With this approach the target class may be created in a variety of ways (direct instantiation, factory methods, reflection, etc.) and the developer has complete control over using the provided stream.

Implementing a Custom Externalizer

To configure a class for custom marshalling an implementation of **org.infinispan.marshall.AdvancedExternalizer** must be provided. Typically this is performed in a static inner class, as seen in the below externalizer for a **Book** class:

```
import org.infinispan.marshall.AdvancedExternalizer;

public class Book {

  final String name;
  final String author;

  public Book(String name, String author) {
    this.name = name;
    this.author = author;
  }

  public static class BookExternalizer implements AdvancedExternalizer<Book> {
    @Override
    public void writeObject(ObjectOutput output, Book book)
```

```
            throws IOException {
        output.writeObject(book.name);
        output.writeObject(book.author);
      }

      @Override
      public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
        return new Person((String) input.readObject(), (String) input.readObject());
      }

      @Override
      public Set<Class<? extends Book>> getTypeClasses() {
        return Util.<Class<? extends Book>>asSet(Book.class);
      }

      @Override
      public Integer getId() {
        return 2345;
      }
    }
}
```

Once the **writeObject()** and **readObject()** methods have been implemented the Externalizer may be linked up with the classes they externalize; this is accomplished with the **getTypeClasses()** method seen in the above example.

In addition, a positive identifier must be defined as seen in the **getId()** method above. This value is used to identify the Externalizer at runtime. A list of values used by JBoss Data Grid, which should be avoided in custom Externalizer implementations, may be found at JBoss Data Grid Externalizer IDs.

Registering Custom Marshallers

Custom Marshallers may be registered with JBoss Data Grid programmatically or declaratively, as seen in the following examples:

Declaratively Register a Custom Marshaller

```
<cache-container>
  <serialization>
    <advanced-externalizer class="Book$BookExternalizer"/>
  </serialization>
</cache-container>
```

Programmatically Register a Custom Marshaller

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
    .addAdvancedExternalizer(new Book.BookExternalizer());
```

6.4. JBOSS DATA GRID EXTERNALIZER IDS

The following values are used as Externalizer IDs inside the Infinispan based modules or frameworks, and should be avoided while implementing custom marshallers.

Table 6.1. JBoss Data Grid Externalizer IDs

Module Name	ID Range
Infinispan Tree Module	1000-1099
Infinispan Server Modules	1100-1199
Hibernate Infinispan Second Level Cache	1200-1299
Infinispan Lucene Directory	1300-1399
Hibernate OGM	1400-1499
Hibernate Search	1500-1599
Infinispan Query Module	1600-1699
Infinispan Remote Query Module	1700-1799
Infinispan Scripting Module	1800-1849
Infinispan Server Event Logger Module	1850-1899
Infinispan Remote Store	1900-1999

CHAPTER 7. JMX

7.1. ABOUT JAVA MANAGEMENT EXTENSIONS (JMX)

Java Management Extension (JMX) is a Java based technology that provides tools to manage and monitor applications, devices, system objects, and service oriented networks. Each of these objects is managed, and monitored by **MBeans**.

JMX is the de facto standard for middleware management and administration. As a result, **JMX** is used in Red Hat JBoss Data Grid to expose management and statistical information.

7.2. USING JMX WITH RED HAT JBOSS DATA GRID

Management in Red Hat JBoss Data Grid instances aims to expose as much relevant statistical information as possible. This information allows administrators to view the state of each instance. While a single installation can comprise of tens or hundreds of such instances, it is essential to expose and present the statistical information for each of them in a clear and concise manner.

In JBoss Data Grid, JMX is used in conjunction with JBoss Operations Network (JON) to expose this information and present it in an orderly and relevant manner to the administrator.

7.3. ENABLING JMX WITH RED HAT JBOSS DATA GRID

By default JMX is enabled locally on each JBoss Data Grid server, and no further configuration is necessary to connect via JConsole, VisualVM, or other JMX clients that are launched from the same system.

To enable remote connections it is necessary to define a port for the JMX remote agent to listen on. When using OpenJDK this behavior is defined with the **com.sun.management.jmxremote.port** parameter. In addition, it is recommended to secure the remote connection when this is used in a production environment.

Enable JMX for Remote Connections using the OpenJDK

This example assumes that a SSL keystore, entitled **keystore** has already been created, and will configure a standalone instance to accept incoming connections on port 3333 while using the created keystore.

```
## Default configuration, 1.3GB heap
JAVA_OPTS="-server -Xms1303m -Xmx1303m -XX:MetaspaceSize=96m -
XX:MaxMetaspaceSize=256m -Djava.net.preferIPv4Stack=true"

## Add the JMX configuration
JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote.port=3333"
JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote.ssl=true"
JAVA_OPTS="$JAVA_OPTS -Djavax.net.ssl.keystore=keystore"
JAVA_OPTS="$JAVA_OPTS -Djavax.net.ssl.keystorePassword=password"
```

As JMX behavior is configured through JVM arguments, refer to the JDK vendor's documentation for a full list of parameters and configuration examples.

CHAPTER 8. HOT ROD SERVER

8.1. ABOUT HOT ROD

Hot Rod is a binary TCP client-server protocol used in Red Hat JBoss Data Grid. It was created to overcome deficiencies in other client/server protocols, such as Memcached.

Hot Rod will failover on a server cluster that undergoes a topology change. Hot Rod achieves this by providing regular updates to clients about the cluster topology.

Hot Rod enables clients to do smart routing of requests in partitioned or distributed Red Hat JBoss Data Grid server clusters. To do this, Hot Rod allows clients to determine the partition that houses a key and then communicate directly with the server that has the key. This functionality relies on Hot Rod updating the cluster topology with clients, and that the clients use the same consistent hash algorithm as the servers.

Red Hat JBoss Data Grid contains a server module that implements the Hot Rod protocol. The Hot Rod protocol facilitates faster client and server interactions in comparison to other text-based protocols and allows clients to make decisions about load balancing, failover and data location operations.

8.2. WORKER THREADS IN THE HOT ROD SERVER

8.2.1. About Worker Threads

Worker threads, unlike system threads, are threads activated by a client's request and do not interact with the user. As the server is asynchronous client requests will continue to be received after this limit is hit; instead, this number represents the number of active threads performing simultaneous operations, typically writes.

In Red Hat JBoss Data Grid, worker threads are used as part of the configurations for the REST, Memcached and Hot Rod interfaces.

8.2.2. Change Number of Worker Threads

In Red Hat JBoss Data Grid, the default number of worker threads for all connectors is **160**, and may be changed . The number of worker threads may be specified as an attribute on each interface, as seen in the following example:

```
<hotrod-connector socket-binding="hotrod" cache-container="local" worker-threads="200">
    <!-- Additional configuration here -->
</hotrod-connector>
```