OpenRDS – Open Requisition Distribution System

User Guide

Document version: 1.0 – Last change: 2006/10/04 16:46

1 GETTING STARTED....2

1.1 What is OpenRDS?....2
1.2 Requirements....2
1.3 Running a “hello world”....2
1.4 Running a sample parallelism performance test....3
1.5 Understanding how OpenRDS works....5
1.6 Using OpenRDS with an existing application....6
1.7 The “Cluster Node” tool....10
1.7.1 Installation....10
1.7.2 Initialization scripts....12
1.7.3 Configuration....12
1.7.4 Web update....13

2 OPENRDS IN ACTION....14

2.1 Working with clock and memory factors....14
2.1.1 Node factors....14
2.1.2 Requisition factors....14
2.2 Dealing with common issues....17
2.2.1 Class loading....17
2.2.2 Static initialization....18
2.2.3 Singletons....18
2.2.4 Threads....18
2.2.5 Data access....19
2.2.6 Host operating system....19
2.2.7 JNI....19
2.2.8 Inner classes....19
2.3 Dynamic class download....20
2.3.1 Understanding how it works....20
2.3.2 Advantages and disadvantages....21
2.4 Working with databases....21
2.4.1 Overview....22
2.4.2 Connecting to a database....22

3 GETTING AND USING THE SOURCE-CODE....23

3.1 Downloading the full source....23
3.2 Compiling OpenRDS....23
3.3 Compiling JNI code....24
3.4 Running the tests....24

4 ABOUT....25

4.1 How did the project start?....25
4.2 Who maintains OpenRDS project?....25
4.3 How can I contribute?....25

1 GETTING STARTED

This chapter explains the basic concepts of OpenRDS and leads you through examples that will surely make you get anxious to start using this framework.

1.1 What is OpenRDS?

OpenRDS is a Java framework that helps you to build distributed systems and clusters with the lowest possible effort using requisition-based communication. The name stands for "Open Requisition Distribution System", which greatly expresses what this framework does.

The main purpose of this project is to be extremely simple to use yet extensible enough to cover more complex scenarios.

Using OpenRDS you can take advantage of idle processing capacity of your network to increase the throughput of your main application(s) or reduce the time to process a requisition using parallelism techniques.

1.2 Requirements

OpenRDS works on virtually any operating system with J2SE 1.4 or higher, but has been currently tested only on Windows 98, 2000, XP and on some KNOPPIX Linux distributions with kernel versions 2.4 and 2.6.

1.3 Running a "hello world"

The first step of learning how to use any new library is always a "hello world", so this is probably the best way to demonstrate OpenRDS usage too.

Firstly, download the latest OpenRDS release and extract the zip contents to a folder. Since we are going to demonstrate the "hello world" running in 3 separated processes, you will need to open **three** console windows (or shell terminals) and change to the directory where OpenRDS was extracted to, by typing the command "**`cd c:\path_to_openrds`**" (Windows) or "**`cd /path_to_openrds`**" (Linux) - don't forget to do that on the three terminals.

After that, type the following command on **terminal #1**:

```
java -cp OpenRDS.jar net.sf.openrds.examples.GenericMainNode
```

(Don’t forget that if your java executable is not on the PATH, you will need to include the full path to it).

You should see the following message: “`Main node active`”, indicating that the main node was successfully started.

Congratulations, you have started your first “Main Node”. Don’t worry about understanding what a “Main Node” is, this will be explained on the topic [Understanding how OpenRDS works]. Note that you can press “CTRL+C” at any time to abruptly close the java process and stop the application, but you can also press the key ‘q’ and press [enter] to close the application normally.

Now type the following command on **terminal #2**:

```
java -cp OpenRDS.jar net.sf.openrds.examples.GenericProcessNode
```

See that the message “`Process node active`” is displayed and also take a look on the terminal #1; you should see a message like this: “`Node registered: (node id)`”, indicating that the main node was successfully contacted by this process node.

Now type the following command on **terminal #3**:

```
java -cp OpenRDS.jar net.sf.openrds.examples.HelloWorld
```

You should see the following message: “`Hello from process node`”. This means that your system has successfully processed a requisition, and this requisition passed among the three different processes. The terminal #2 should also display the message: “`Hello world from client`”.

This finishes the hello world example, but I guess this is not enough exciting for you. You surely want to see the system performing something smarter than a distributed “hello world”, so keep these 3 terminals open and follow to the next topic: “Running a sample parallelism performance test”, which will probably be much more interesting and will give you a better idea of the overall functionality.

1.4 Running a sample parallelism performance test

This topic describes how to test the performance improvement using OpenRDS on multiple machines by distributing a sample requisition over all nodes in the network. This requisition computes prime numbers between 1 and 60000, but note that it has been intentionally coded using the dumbest possible algorithm in order to be very heavy.

Before we start the test, follow all steps described on the last topic [Running a “hello world”] (if you haven’t already), and keep all the terminals open. Also, be sure that you have only one IP configured and only one network interface. If you have more than one IP and/or network interface, you will need to set a java system property called “**`openrds.base.ip`**” with the IP address of your local network. To do that, you must add the following argument after each “java” command:

“**`-Dopenrds.base.ip=192.168.0.1`**” (where 192.168.0.1 is the desired IP)

`E.g.:` **`java -Dopenrds.base.ip=192.168.0.1 -cp OpenRDS.jar ...`**

Okay, after running the “hello world” you should have 3 terminals, where the #1 is running a “Main Node”, the #2 is running a “Process Node” and the #3 is idle.

To run the sample requisition, type the following command on **terminal #3**:

```
java -cp OpenRDS.jar net.sf.openrds.examples.PrimeNumber
```

You should see the message “`Starting test!`” that indicates the test is running. Some “dots” should now be printed on terminal #2 (the sample requisition print those dots so that you can be sure the test is running). After some time you will see 2 messages like this on terminal #3:

```
Found 6057 prime numbers...
Total time: 43328 milliseconds.
```

The second line indicates how much time the requisition took to be processed (about 43 seconds on my machine), write down that number, so that you can compare with the next results.

Okay, we have processed a more complex requisition in one machine, but that is not so exciting… We want to see real parallelism in action, so go to another computer and open a terminal on that computer, changing to OpenRDS directory as previously described and type the following command:

```
java -cp OpenRDS.jar net.sf.openrds.examples.GenericProcessNode
192.168.0.1
```

Note that this a single command and it contains the IP 192.168.0.1 at the end. Change this to the IP of the other machine (where the main node is running).

You should see the following message: “`Process node active`”, which means that you now have 2 process nodes. To be sure that the node successfully connected to the other machine, look on the terminal #1 for another “`Node registered: (node` `id)`" message. That message is displayed every time a process node connects to the system.

Now go back to terminal #3 (which is idle), and type the following command again:

```
java -cp OpenRDS.jar net.sf.openrds.examples.PrimeNumber
```

Just like the last time, you will see two messages like this after some time:

```
Found 6057 prime numbers...
Total time: 18516 milliseconds.
```

Compare the "Total time" with the last result, it should be considerably lower, which means that the system successfully processed the same requisition in 2 different machines at the same time, what gives a great performance improvement.

Start more process nodes on more machines and check the performance improvement. You can now try different things, such as starting a process node while the test requisition is already being processed (this node will immediately start to help processing it) or closing a process node while the test is running (the system will automatically redistribute the requisitions and this wont affect the final result). Just note that the test will FAIL if there are no process nodes available, since the main node itself does not process a requisition.

I hope that these examples were enough to give you at least a superficial overview of the framework potential.

1.5 Understanding how OpenRDS works

The system architecture consists basically in 2 different major modules: the "Main Node" and the "Process Node".

The main node is the "master" of the system; it is responsible for controlling all process nodes, distributing the requisitions to them and controlling load-balancing. Since all requisitions must be sent to the main node, it is a good idea to start it on the same JVM of your application (this way you ensure that it is always running), but this is not mandatory for the system to work (as seen in the "hello world" example, the main node was on a completely separated process).

A process node is just a slave responsible for processing requisitions sent by the main node and returning the result, it doesn't do much more than that, but the more process nodes you start on your network, more processing power and better results you will get.

Since the main node cannot process a requisition, client applications often start both the main node and a process node on their application's JVM, what ensures that at least one node will be always available.

OpenRDS uses the Java RMI for communication, but that is completely transparent to the client system. If your application already uses RMI you can still use OpenRDS, with the sole constraint that the main node and the process nodes must be not started in the same virtual machine of your application (just like we did in the "hello world" example).

After starting the nodes, the client application just have to create requisitions and pass them to the main node to be processed. The requisitions are somewhat similar to the well known "`Runnable`" java interface, where you must implement a method that contains the code to be executed.

There are two distinct types of requisitions: "Indivisible Requisitions" and "Divisible Requisitions". The first one is the simplest one, which does not use parallelism techniques. This type of requisition is used to greatly improve your application's throughput, allowing more requisitions to be processed at the same time.

The second type of requisition, the "Divisible Requisition" is used when your code can be divided into many "Sub-requisitions" to be processed at the same time in parallel. This type of requisition is used to decrease the time necessary to process a single request and also to increase the application's throughput. This type of requisition was used to create the example on the topic [Running a sample parallelism performance test].

1.6 Using OpenRDS with an existing application

Now that you know the basic concepts of OpenRDS, you must be asking yourself how to use it on your application, and I think that the best way to learn that is thought practical examples.

Let's start with code necessary to start a main node and a process node from your client application. You just have to import "**`net.sf.openrds.*`**" and add the following code on your application startup:

```java
try {
    NodeFactory.getInstance().startMainNode();
    NodeFactory.getInstance().startProcessNode("localhost", true);
} catch (Exception e) {
    System.out.println("Could not start OpenRDS nodes.");
    e.printStackTrace();
    System.exit(1);
}
```

As you can see, it's very easy to startup OpenRDS nodes. This code starts a main node and then starts a process node that will connect to the local host. The second parameter of the method "`startProcessNode`" tells the factory to test the connection with the main node during initialization.

Now, to illustrate a real problem, let's imagine that your application runs on a server and is responsible to convert documents from one format to another. But this task is very heavy and you want to improve it. Then you will have a method like this somewhere in your code:

```java
public void convertDocument(File source, File destination) {
    // Reads source pages
    Page pages[] = readPages(source);
    // Creates the array to store converted pages
    ConvertedPage conv[] = new ConvertedPage[pages.size];
    // For each page, convert it and store on that array
    for (int i = 0; i < pages.length; i++) {
        conv[i] = convertPage(pages[i]);
    }
    // Writes the converted pages to disk
    writePages(conv, destination);
}
```

Since the conversion of the pages is very heavy, you want to send this task to OpenRDS nodes, so that you will be able to convert many more documents at the same time. In order to do that, you just have to change your code to the following:

```java
public void convertDocument(File source, File destination) {
    // Reads source pages
    Page pages[] = readPages(source);
    // Creates a requisition that converts those pages
    Requisition r = new ConvertReq(pages);
    // Retrieves a reference to the main node
    IMainNode n = RegistryHandler.getInstance().getMainNode();
    // Send the requisition to be processed and retrieves the result
    ConvertedPage conv[] = (ConvertedPage[]) n.processRequisition(r);
    // Writes the converted pages to disk
    writePages(conv, destination);
}
private static class ConvertReq extends IndivisibleRequisition {
    private Page pages[];
    private ConvertReq(Page pages[]) {
        this.pages = pages;
    }
    public Object process() throws ProcessingException {
        // Creates the array to store converted pages
        ConvertedPage conv[] = new ConvertedPage[pages.size];
        // For each page document, convert it and store on array
        for (int i = 0; i < pages.length; i++) {
            conv[i] = convertPage(pages[i]);
        }
        return conv;
    }
}
```

We just had to wrap the code that calls "**`convertPage()`**" in an "**`IndivisibleRequisition`**" and send it to the main node to handle it. It's as simple as that. The only requirement is that both the "`Page`" and the "`ConvertedPage`" objects must implement the "`Serializable`" interface, since it will be transferred over the network.

Well, this small modification gives you a great improvement when your system receives many requests to convert documents at the same time, but this won't help you to increase the conversion speed of a single document. Your clients would be much happier if the system could convert documents faster when it is not being heavily used.

Since we convert the document page-by-page, we just have to create one requisition for each page and then put all the converted pages together. Well, this is not very hard to implement in your code… but it is even easier to implement using a "**`DivisibleRequisition`**", so take a look:

```java
public void convertDocument(File source, File destination) {
    // Reads source pages
    Page pages[] = readPages(source);
    // Creates a requisition that converts those pages
    Requisition r = new ConvertMultReq(pages);
    // Retrieves a reference to the main node
    IMainNode n = RegistryHandler.getInstance().getMainNode();
    // Send the requisition to be processed and retrieves the result
    ConvertedPage conv[] = (ConvertedPage[]) n.processRequisition(r);
    // Writes the converted pages to disk
    writePages(conv, destination);
}
private static class ConvertMultReq extends DivisibleRequisition {
    private Page pages[];
    private ConvertReq(Page pages[]) {
        this.pages = pages;
    }
    public SubRequisition[] getSubRequisitions(int availableNodes) {
        // Creates one requisition for each page
        ConvertReq reqs[] = new ConvertReq[pages.size];
        for (int i = 0; i < reqs.length; i++) {
            reqs[i] = new ConvertReq(pages[i]);
        }
        return reqs; // Return the requisitions to OpenRDS
    }
    public Object getResponse(Object[] subResults) {
        return subResults; // Just return the results in order
    }
}
private static class ConvertReq extends SubRequisition {
    private Page page;
    private ConvertReq(Page page) {
        this.page = page; // This will handle only one page
    }
    public Object process() throws ProcessingException {
        return convertPage(page); // Converts that single page
    }
}
```

The “`DivisibleRequisition`” has a method called “**`getSubRequisitions`**”, which is called by OpenRDS to retrieve an array of sub-requisitions which will be processed in parallel. When all requisitions have finished processing, the method “**`getResponse`**” is called to put all the results together. The array of objects received is the concatenation of all sub-requisitions’ results in the same order that the sub-requisitions where returned by the method “`getSubRequisitions`”. Since our expected final result is the exact concatenation of all converted pages we don’t need to do anything, so we just return the received array of sub-results.

As you can see, the code got a little bigger but it is still very simple considering the great performance improvement that you will deliver to your clients, and they will surely be much happier because they will convert documents much faster than before. Just start a good number of process nodes over your network and you will surely get amazing results.

Of course that this is just a simple example, so the performance can be improved even more by using more advanced options of OpenRDS (such as requisition factors) and some other implementation techniques. Requisition factors are explained on the topic [Working with clock and memory factors].

1.7 The "Cluster Node" tool

The first topics have taught you how to start a process node using the "Generic Process Node" or starting it inside your application's JVM. But installing and managing process nodes can become time-consuming if you don't have at least some automation to help you on that task

Thinking always on simplicity, OpenRDS comes with a small application called "Cluster Node", which has been designed to help on the installation, execution and administration of process nodes.

This tool is just like a generic process node, but it includes other functionalities, such as an installation wizard, web update capability and an administration console where you can run commands such as "`restart`", "`exit`", "`config`" and more.

1.7.1 Installation

To install a fresh new cluster node, extract OpenRDS package to a directory and run either the "`clusterNode.bat`" (on Windows) or "`clusterNode.sh`" (on Linux). Since this is the first time that you are executing the cluster node it will start with the "Installation wizard", making some questions to you. The first message that will be displayed is:

```
Please type the IP or host-name of the Main Node:
```

Here you need to type either the IP address or hostname of the machine that will be running the main node. If you are going to run it on the same machine, just type "`localhost`" (without quotes). After that, the following message will be displayed:

```
Do you want to use the default registry port? (y/n):
```

Just type "y" or "yes" for this question, unless you have changed the default registry port on the main node (which is only needed if you have a problem with the default port). The next question is pretty much the same:

```
Do you want to use the default HTTP port? (y/n):
```

Just type “y” or “yes” again, unless you have a problem using the default port on the main node. Now the system should display:

```
Do you want to define a base-ip for OpenRDS?
(This is only needed if you have more than one IP.) (y/n):
```

If you have (or plan to have) more than one IP, you should probably set a “`base-ip`”, because this will be the address used by OpenRDS to communicate over the network. If you type “yes”, the system will display something like that:

```
The system has detected the following IPs:
127.0.0.1
192.168.211.1
192.168.112.1
192.168.0.205
Please enter the desired base ip to use:
```

As you can see, the system listed all detected addresses and requested you to choose one. **ATTENTION:** You don’t need to enter the full desired IP address, just the left-most part that will not cause ambiguity over your addresses. This is VERY important if you use dynamic IP (DHCP).

For the addresses in this example (“127.0.0.1”, “192.168.211.1”, “192.168.112.1” and “192.168.0.205”), I could set the “`base-ip`” to “**`192.168.0.`**” or “**`192.168.0`**” in order to use the IP “192.168.0.205”. If on the next boot this IP changes to “192.168.0.220” the node will still work.

If you have an IP address that conflicts with the left-most part of another IP address (such as “192.168.0.1” and “192.168.0.10”), and you need to use the first one, just set the `base-ip` to match exactly the desired IP address. Note that this is an extremely rare situation.

After you choose the desired address (or if you typed “no” for the last question), the system will display the startup messages, which will look like this:

```
Using network address '192.168.0.205'.
# OpenRDS 1.1-development (Built on September 26 2006 20:41:36)
# During the execution of the node, type any of the following
# commands and press [ENTER] to execute it.
```

```
#
# exit    - Stops the process node execution.
# quit    - Same as 'exit'.
# restart - Restarts the process node, unloading any loaded code.
# update  - Checks for online updates (requires internet).
# config  - Change node's configuration (will cause a restart).
# clear   - Clears the console, by printing 40 '\n' chars.
# help    - Displays this message again.
# h       - Same as 'help'.

Node has been successfully started - Trying to connect to main
node.
```

These messages will always be displayed when you start a cluster node. As you can see, the node lists all available commands that you can execute, and I think the messages are self-explanatory, so I will not detail them here.

1.7.2 Initialization scripts

As you have noted, the cluster node tool is initialized by a script file called "`clusterNode.xxx`" (where xxx changes for each platform).

On the beginning of each script, there is a section with some variables that you can modify as needed. This includes: "Java location", "Additional java arguments", "Additional JAR libraries", etc.

Take a look at the script file, since you may find out something that you would like to modify. One important thing to modify if you don't want to use "Dynamic class download" is the "Additional JAR libraries", which is defined by the variable "**`LIBS`**". You should list all your application's JAR files on this variable.

1.7.3 Configuration

When you install a cluster node, a file called "`clusternode.properties`" will be generated and will store all configuration data. You can delete this file and restart the node if you want to reconfigure it, but you can also type the command "**`config`**" while the node is running, so that the node will do that for you.

Configuration is not different from the installation procedure, so I think you won't have problems with that.

1.7.4 Web update

When a new OpenRDS version is released, you can download the package and manually copy the updated JAR file to your cluster nodes. But you may find it easier to run the "Web update" feature. To do that, just type the command "`update`" on the cluster node console and the system will display the following message:

```
Are you sure you want to check for updates on the web? (y/n):
```

Just type "yes" and after some time the system will display something like:

```
There is a new version of OpenRDS available: 1.1-beta
Do you want to update this node? (Will restart application) (y/n):
```

Or

```
The version is up to date!
```

If the system detected a new version, just type "yes" and it will download and install the latest version (restarting the node); otherwise it means that you already have the latest version.

2 OPENRDS IN ACTION

This chapter works on several important aspects of OpenRDS that will be surely useful for developers using the framework. This is a must-read for anyone starting to use OpenRDS or wanting to understand it better before taking the decision of using it.

2.1 Working with clock and memory factors

All the examples previously seen in this guide demonstrates how to use OpenRDS to distribute requisitions, but how are those requisitions balanced over the nodes? Well, to explain this you need firstly to understand the usage of clock and memory factors, which are applicable for nodes and requisitions.

2.1.1 Node factors

All process nodes MUST define a processing factor (also called “clock factor”) and a memory factor. These factors determine the processing power and the memory allocation capacity of that node. Main nodes do not define those factors.

By default, OpenRDS will set the clock factor to the same value of the processor clock in MHz (E.g.: 2400 for a 2.4GHz processor) and the memory factor to the amount of physical memory of the machine in MB. But these factors can be overridden by setting two java system-properties: “**`openrds.clock.amount`**” and “**`openrds.memory.amount`**” (which sets the clock and memory factors respectively).

These two factors are used by OpenRDS to determine the best node to process a requisition. I won’t describe in details the algorithm used to balance the requisitions, but you will better understand how to use those factors and why they are necessary after reading the next topic, “Requisition factors”.

2.1.2 Requisition factors

Any requisition can define a processing factor and/or a memory factor, which are used by OpenRDS to determine how “heavy” that requisition is in terms of processor time usage and memory consumption. To determine the processing and/or memory factor(s) of a requisition, override the following method(s) of an indivisible requisition:

```
public int getProcessingFactor();
public int getMemoryFactor();
```

The first one defines the processing factor (or clock factor) of the requisition, and the second one defines its memory factor.

OpenRDS will keep track of requisitions being processed on all nodes. If the sum of clock factors of all requisitions being processed on a node matches or exceeds the clock factor of that same node, it will be marked as "processing unavailable" and will not receive any other requisition that defines a clock factor. The node will remain in that state until it finishes processing a requisition that define a clock factor (and will then be "processing available" again).

The same rule is applied to memory factors; it will be marked as "memory unavailable" and will not receive requisitions that define the memory factor. Please note that if a node is marked as "memory unavailable" it can still receive requisitions that do not define any factor or define just the clock factor (the same applies to "processing unavailable").

This characteristic of OpenRDS makes it possible for the system to distinguish "heavy" requisitions from "light" requisitions as well as different characteristics of memory consumption and processor time usage. You can, for example, model your nodes to be able to process two light requisitions at the same time, but not two heavy ones.

To illustrate these rules, here follows an example: Consider you have a node called "A", with clock factor equals to 1000 and a memory factor equals to 512, so:

`NODE A – CF: 1000, MF: 512` (CF stands for Clock factor and MF stands for memory factor)

Then this node receives a requisition that defines just a memory factor of 400:

```
START REQ#1 – CF: 0, MF: 400
```

(Starts to process requisition #1, with CF=0 and MF=400).

While this requisition was still being processed, the node receives another requisition, which defines just a memory factor of 200:

```
START REQ#2 – CF: 0, MF: 200 ** Unavailable: memory **
```

(Starts to process requisition #2 and marks memory as unavailable)

Since the sum of memory factors of all requisitions have exceeded the node's memory factor, it has been marked as "memory unavailable" (400+200=600) and (600 >= 512).

Now, the node receives another requisition:

```
START REQ#3 – CF: 400, MF: 0
```

(This is ok, since this requisition does not define a memory factor, just a processing factor).

And this node now finally finishes processing the first requisition:

```
END REQ#1 ** Available: memory **
```

Since the node has finished a requisition that defines a memory factor, it is now marked as “memory available” and will be able to receive requisitions that define this factor again.

Here follows a sample activity for a node, which will be illustrated just like in the last example:

```
(NODE B – CF: 1000, MF: 512)
START REQ#1 – CF: 0, MF: 400
START REQ#2 – CF: 0, MF: 200 ** Unavailable: memory **
END REQ#1 ** Available: memory **
END REQ#2 (node idle)
START REQ#3 – CF: 500, MF: 0
START REQ#4 – CF: 499, MF: 0
START REQ#5 – CF: 1, MF: 0 ** Unavailable: processing **
START REQ#6 – CF: 0, MF: 400
END REQ#5 ** Available: processing **
START REQ#7 – CF: 0, MF: 112 ** Unavailable: memory **
END REQ#4
END REQ#3
START REQ#5 – CF: 1500, MF: 0 ** Unavailable: processing **
```

I hope that this small example could give you a good idea of how factors are used to determine if a node can process a requisition or not. Also, as you can now realize by yourself, if you want to limit a single requisition to be processed at a time on all nodes you can set any of the requisition’s factor to a very high number, such as 99999. And if you want, for any reason, to limit the number of requisitions on a single node, just set a very low factor on that node (such as 1), and define that same factor on you requisitions to any desired value (greater than zero).

2.2 Dealing with common issues

This topic works the overall idea of some basic but necessary concepts that a developer will need before starting to construct real applications using OpenRDS.

2.2.1 Class loading

One of the most important things that a developer must understand is how to deal with the Java class loading mechanism when using OpenRDS. If you understand it well, it will surely help you to avoid a large number of common mistakes.

The java runtime environment always loads a class when it is used or referenced for the first time on that JVM. Every class that a requisition references directly or indirectly will follow the same rule, with the very difference that you can never predict on which JVM that requisition will be running. As a result of that, a requisition cannot assume that something has been previously loaded and/or executed on that JVM in order for it to work properly.

If you executed some initialization code that created some objects or any other sort of data, those objects will not be available on the process node's JVM. If you really depend on any sort of initialization you should create a static flag to tell your requisitions if the initialization has already been performed or not on the current JVM. Every requisition should then check this flag and execute the initialization if necessary and update the flag on the end.

The following code is a small example of how to perform initialization on requisitions:

```java
public class Req extends IndivisibleRequisition {
    public Object process() throws ProcessingException {
        Starter.assertInitialized(); // Asserts initialization
        // .. requistion code ..
        return result;
    }
}
public class Starter {
    private static boolean initialized = false;
    public static void assertInitialized() {
        if (!initialized) {
            // .. initialization code ..
            initialized = true;
        }
    }
}
```

2.2.2 Static initialization

Another thing that complements what we have learned about class loading is the classes' static initialization.

Static initialization is executed by the JRE when any class is loaded for the first time and includes anything that is directly assigned to a static variable in the field declaration and code contained in "**`static { }`**" blocks, as in the example:

```
public class MyClass {
    private static int someValue = determineValue();
    private static Object someObj = new SomeObject();
    static {
        someMethod();
    }
}
```

On this example, the methods "**`determineValue()`**", "**`someMethod()`**", and the constructor "**`SomeObject()`**" will all be executed when the class "**`MyClass`**" is loaded for the first time for any reason.

An important thing to notice is that if the main node is running on a separated process of your application, some static initialization may be executed on the main node while it is loading a requisition to send it to a process node.

2.2.3 Singletons

The "singleton" design pattern is used on many java systems to make a single object reference globally accessible all over the JVM. If your requisition is going to access any singleton object, you should take the same precautions as described on the [Class loading] topic.

2.2.4 Threads

If you need, for any reason, to start a thread from your requisition, make sure that this thread is terminated when the requisition finishes its job, otherwise you may have serious memory leaks when running your nodes for a long time.

You may also choose to start a thread for any purpose and keep it running on the process node's JVM when the requisition completes, but you should then be sure to don't start that same thread again when another requisition comes to be processed in that process node. Again, you can use static flags and other techniques to do that.

2.2.5 Data access

Any data that the requisition will need must come serialized with the requisition itself since you will not have access to any other kind of data. As this data is transferred over the network, make sure that you are referencing only the necessary amount of data in the requisition or you may have performance problems because of the time necessary to transfer the data.

If you need to access a large amount data, see the topic [Working with databases].

2.2.6 Host operating system

If you have process nodes running on multiple operating systems, just be sure to don't call any OS-specific API. This is not a big problem on Java applications, but this topic serves only to remind you about this issue. A good option is to check the current OS on the requisition code before calling this kind of APIs.

2.2.7 JNI

To use JNI with OpenRDS, you just need to be sure that your JNI library is available on the process node's library path. The "cluster node tool" always put the current working directory in the java library path, so all libraries can be put on the same directory of `OpenRDS.jar`.

Also, make sure that the "**`System.loadLibrary()`**" method has been executed on the current process node before calling any JNI, as we have previously seen about class loading on previous sub-topics.

2.2.8 Inner classes

Remember that all non-static inner classes implicitly holds a reference to the object that instantiated it (this includes anonymous inner classes). So be very careful when passing this kind of object to a requisition.

Also, if you implement a requisition as an inner class, be sure that the requisition class is declared as "static", otherwise every time you process a requisition, the object that instantiated that requisition will also be transferred over the network if it implements "`Serializable`", what can give you a big performance bottleneck.

The following example demonstrates what you can and what you can't do:

```java
public class Example {
    public void wrongCode() {
        // This is wrong... Anonymous inner class
        Requisition req = new IndivisibleRequisition() {
            public Object process() throws ProcessingException {
                // .. do something ..
                return null;
            }
        };
        IMainNode n = RegistryHandler.getInstance().getMainNode();
        n.processRequisition(req);
    }
    public void method() {
        Requisition wrong = new WrongReq();
        Requisition right = new RightReq();
        IMainNode n = RegistryHandler.getInstance().getMainNode();
        n.processRequisition(wrong);
        n.processRequisition(right);
    }
    // This is wrong... it is not static
    private class WrongReq extends IndivisibleRequisition {
        public Object process() throws ProcessingException {
            // .. do something ..
            return null;
        }
    }
    // This is right
    private static class RightReq extends IndivisibleRequisition {
        public Object process() throws ProcessingException {
            // .. do something ..
            return null;
        }
    }
}
```

2.3 Dynamic class download

One of the best features of OpenRDS is the transparent dynamic class download. This feature allows you to transparently transfer java executable code over the network to the process nodes.

With dynamic class download, your process nodes don’t need to have the code for your requisitions (your JAR file for example); what makes it easier to manage the system.

2.3.1 Understanding how it works

Every time that a node needs to load any class but it could not be found in the local class-path, OpenRDS will try to transparently download that class from the main node and load it into the node's JVM. This includes any class directly or indirectly referenced by a requisition received by the node.

Once a class is downloaded, it will remain in that node's memory and it will not be downloaded anymore while that node keeps running. If you restart the node, the JVM will not contain downloaded classes anymore and the node will be start to download those classes again the next time it needs them.

2.3.2 Advantages and disadvantages

The main advantage of using dynamic class download is to make it easier to manage versions. If you update your software version (not OpenRDS version), you just have to restart the process nodes and they will automatically load the new classes, removing the necessity of updating each process node individually.

Using dynamic class download also decreases the necessary time to install a new process node in your network, since you just have to turn it on and it will start to work perfectly with your requisitions.

In the other hand, using dynamic class download will reduce the performance for the first requisition that a process node executes, since it will have to download all the necessary classes, which can take some precious time depending on the amount of classes necessary and your network speed.

You should analyze your particular situation to decide if you need or not to use dynamic class downloads. If you choose not to use it, the process node will need to have your application's JAR file in the class-path, so that it will not download any classes from the main node. If you are using the "cluster node tool", see the sub-topic [Initialization scripts] for information on how to do that.

Also, if you want to completely disable remote class downloads, just set the http port to `-1` when starting your main node and process nodes. If you set the http port to -1 just on the main node, it will not respond to download requests, but the process nodes will still attempt to download a class when it is not found on the its class-path, what can result on a performance bottleneck.

2.4 Working with databases

This topic explains how to use OpenRDS with databases and gives some examples of how to do that.

2.4.1 Overview

There are many situations where sending data with the requisition is not suitable for an application. A solution for this problem could be to get the necessary data from inside a requisition, by using a database connection.

Database connections may also be useful for other situations, such as when your requisition needs to manipulate any data which is not possible for the requestor to predict. An example of that is when you need to manipulate two groups of data and the second group will be chosen based on the result of the first manipulation.

OpenRDS fully supports database connections, even using dynamic class download, since the JDBC driver classes will be downloaded as any other normal system class to the process node's JVM.

2.4.2 Connecting to a database

When creating database connections, we need to take the same precautions described in the sub-topic [Class loading], so just use one of the described techniques to create a database connection normally. Here follows a simple code example of how to create a requisition that reads data from a database:

```java
public class DbReq extends IndivisibleRequisition {
    public Object process() throws ProcessingException {
        try {
            Connection con = Connector.getDbConnection();
            /* .. do something with the connection .. */
            con.close();
            return null;
        } catch (SQLException e) {
            throw new ProcessingException("Database error.", e);
        }
    }
}
public class Connector {
    private static boolean driverLoaded = false;
    public static Connection getDbConnection() throws SQLException {
        if (!driverLoaded) {
            try { // Loads JDBC driver (HSQLDB on this example)
                Class.forName("org.hsqldb.jdbcDriver");
            } catch (ClassNotFoundException e) {
                throw new SQLException("Could not load JDBC driver");
            }
            driverLoaded = true;
        }
        return DriverManager.getConnection("url", "user", "pwd");
    }
}
```

3 GETTING AND USING THE SOURCE-CODE

This chapter explains how to download and use the full source-code of OpenRDS.

3.1 Downloading the full source

When you download a release package of OpenRDS, it comes with a pre-packaged version of the source-code. But the source included on this package is only the main java source of the library to help on debugging and some other things, such as integrating the source-code on an IDE that supports this feature.

To download the full source-code structure, you will need a CVS client. See http://www.nongnu.org/cvs/ for more information about CVS.

Now connect to the following CVS repository:

```
:pserver:anonymous@openrds.cvs.sourceforge.net:/cvsroot/openrds
```

(This is a read-only anonymous login, which does not require a password).

Now, just checkout the module "**openrds**" to some local directory, and you are ready to use the full source-code.

If you have any problem downloading the source, or need more information, see the page http://sourceforge.net/cvs/?group_id=133866.

OpenRDS is developed using the Eclipse Platform (http://eclipse.org), so it can be checked out directly as an eclipse project if desired.

3.2 Compiling OpenRDS

If you have "**ant**" (http://ant.apache.org), just execute the "**build.xml**" file located on the "**/build**" folder of the module and it will generate two jar files in the "**/build/jar**" directory of the module, one with debug information and one optimized.

If you don't have ant, open a console, change to the folder "**/src**" of the downloaded module, then type the following commands:

```
javac -d ../bin net/sf/openrds/*.java
javac -d ../bin net/sf/openrds/examples/*.java
javac -d ../bin net/sf/openrds/tools/*.java
cd ..
cd bin
rmic -classpath ./ net.sf.openrds.Node
rmic -classpath ./ net.sf.openrds.MainNode
rmic -classpath ./ net.sf.openrds.ProcessNode
rmic -classpath ./ net.sf.openrds.LocalRegistryHandler
rmic -classpath ./ net.sf.openrds.RemoteRegistryHandler
```

After running these commands, the full source will have been compiled to the folder "**`/bin`**" of the module. Now just package it using the "`jar`" application and you are ready to use it.

3.3 Compiling JNI code

At the current version, OpenRDS just uses JNI on the Windows operation system. You can find the JNI code under the folder "**`/jni`**" of the module.

To compile the source, you just need to open the "**`.dsw`**" workspace file in the Microsoft Visual Studio 6 or greater and run the "build" command.

3.4 Running the tests

If you have "**`ant`**" (http://ant.apache.org), just execute the "**`junit`**" target of the "**`build.xml`**" file located on the "**`/build`**" folder, to do that, just type "**`ant junit`**" on that folder. Please note that you need to add the **`junit.jar`** file on ant's class-path.

If you don't have ant, you need to manually compile the source as previously described and then compile the classes contained on "**`/junits/net/sf/openrds/`**" and run the "`junit`" class "**`net.sf.openrds.AllTests`**".

For more information, look at http://www.junit.org/.

4 ABOUT

4.1 How did the project start?

OpenRDS started as my monograph's practical project for achieving the Bachelor of Computer Science grade in the Brazilian university where I graduated, on December of 2005.

When I finished the initial version of the project, it remained inactive for about 6 months, but I then realized that it could be useful for me and even other people looking for an extremely simple way of making distributed systems, so I decided to continue constantly improving the project.

4.2 Who maintains OpenRDS project?

OpenRDS is maintained only by Rodrigo Zechin Rosauro, a Brazilian born on August of 1984, currently working as a Java Developer.

4.3 How can I contribute?

If you like the framework, you can contribute by sending suggestions and bugs reports using the tracker located on http://sourceforge.net/tracker/?group_id=133866.