JBoss and ECperf 1.1

A guide for configuration

Author: Jon Barnett

Document version: 1.2

Table of contents

1 JBoss and ECperf 1.1

JBoss is the most popular open source J2EE server. ECperf is a benchmark and implementation for measuring performance and scalability of J2EE servers. With the current release of both products and not much in the way of documentation, I decided that we needed to be able to at least determine the performance capabilities of JBoss 3.2.x. Judging by the accumulated requests in the JBoss forums it seemed that others were also interested in the subject.

The initial thought on the integration was that the task should not be too laborious as the one page guide provided by the CSIRO at http://www.cmis.csiro.au/adsat/jbossecperf.htm seemed very straight forward. This assembled kit was based on ECperf 1.0 update 2. However, ECperf 1.1 proved more challenging.

We've updated this release to use the ECperf 1.1 modifications from the JBoss SourceForge repository.

1.1 Requirements

For the integration work you will need JBoss 3.2.x and ECPerf 1.1 for JBoss 3.x.x. You can download these respectively from http://www.jboss.org/index.html?module=html&op=userdisplay&id=downloads and http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/jboss/ecperf/. You will actually need to obtain the JBoss ECperf 1.1 source using a CVS tool.

Typically, you would do this from Linux/Unix using:

```
cvs -d:pserver:anonymous@cvs.jboss.sourceforge.net:/cvsroot/jboss login
cvs -z3 -d:pserver:anonymous@cvs.jboss.sourceforge.net:/cvsroot/jboss co ecperf
```

If you wish, you can also get the original ECperf 1.1 software from http://developer.java.sun.com/servlet/SessionServlet?url=/developer/earlyAccess/j2ee/ecperf/download.html. This would need modifications prescribed by the JBoss ECperf 1.1 change notes to be able to be used.

You will also need a Java JDK 1.4.0 or higher and an installed and operational Ant distribution. Depending on your patience, you may make the modifications prescribed here or download the software distribution containing the changes.

JBoss and ECPerf 1.1 should be extracted from their respective packed distributions and they should be installed on the same machine. You should test that your JBoss distribution is operational, as this guide assumes that both Java and JBoss are operational. Further, it is assumed that you have some level of familiarity with JBoss and J2EE.

1.2 Databases

ECperf 1.1 already supplies scripts for the major DBMS and these scripts create the necessary schema. The scripts that exist are for DB2, Oracle and Sybase. However, since we normally use Postgresql for testing purposes I thought we should create scripts and schema SQL for Postgresql. I started from the DB2 scripts since DB2 and Postgresql have similar syntax as they adhere to the SQL92 standard.

The scripts are designed to be run from the same machine as the DBMS. Should the DBMS be installed on a separate machine from where you have installed the ECperf distribution, you will need to transport the schema directory in the distribution to the DBMS machine.

1.2.1 Postgresql modifications

The first thing to do is to create a new repository for the Postgresql scripts. I created a directory called *schema/postgresql* in the ECperf distribution, where the directory *schema* should already exist. Copy the contents of *schema/db2* to *schema/postgresql*.

The schema definitions are contained in the ECperf distribution directory, *schema/sql*. The only specialised schema SQL change for Postgresql arose from the use of *long varchar* in *schema_C.sql*. Postgresql does not support this so there is a single line change from:

```
        r_text          long varchar
```
at line 79, to:
```
        r_text          character varying
```

This changed schema SQL file on recommendation from the ECperf guides, should be placed in a new directory under the creation scripts. In this instance, the directory will be *schema/postgresql/sql*.

blddb.sh

This is the main shell script for creating the database schema required by the ECperf 1.1 testing. I have modified this to be more tolerant when the script is not executed from the same directory as the script.

createdb.sh

This shell script removes the existing database instance and recreates it. The commands to achieve this in DB2 are replaced with the Postgresql *dropdb* and *createdb* commands respectively. The script has also been modified to operate independently of the directory from where the execution was requested.

schema_C.sh, schema_M.sh, schema_O.sh, schema_S.s and schema_U.sh

Respectively, these shell scripts create the customer, manufacturing, order, supplier and utility tables, related to the domains they represent. The tablespace related commands have been removed as these are specific to the DB2 database. The DB2 command line interface operation has been replaced with the Postgresql *psql* command line interface and drops much of the additional DB2 database *connect* and database *disconnect* commands. The scripts have been modified to operate independently of the directory from where the execution was requested. *schema_C.sh* has also been changed to refer to the *schema_C.sql* file located in *schema/postgresql/sql*.

The schema is created by piping the SQL contained in the corresponding schema files contained in *schema/sql* directory or by special cases in the *schema/postgresql/sql* directory. These individual schema creation scripts may be run individually and on their own. You may want to do this if you only want to recreate tables related to a particular area such as customers without recreating the entire database.

1.2.2 Creating the database and schema

The database may be created by running the *blddb.sh* script. You will need to run this as the account that will own the database, and that will also be used by the ECperf 1.1 EJB components to read and write data to that database. This will destroy the existing database instance, create a new instance and build the schema. The command is of the form:

```
blddb.sh <database_instance> <database_directory>
```

The *database_instance* is the name of the database instance that will be recreated. For the purposes of this guide, I will refer to that instance as *ecperf*. The *database_directory* will be the location of temporary data files. The second parameter, from examination does not appear to serve any function and may be a vestigial set of commands from previous development.

The Postgresql scripts retain the functionality for uniformity with existing scripts so you may use the current directory as the *database_directory* since no temporary data files are actually created. Running the main *blddb.sh* script will generate error messages when tables cannot be dropped. This is to be expected as the individual schema creation scripts may also be run separately against an existing database instance to recreate sections of the schema and so need to be able to drop possibly existing tables.

1.3 Building the ECperf middle-tier components

The instructions supplied with ECperf 1.1 are cryptic at best. However, the majority of the JBoss specific work relates to creating middle-tier components that are acceptable by the JBoss application server.

1.3.1 Environmental configuration

The first task is to provide the build environment with the necessary information so that the middle-tier components can be built and used. For JBoss, we will create a *config/jboss.env* file by copying the *config/ri.env* file.
The following variables which are mainly used by Ant should be modified and the examples reflect the JBoss focus where appropriate:

JAVA_HOME – the topmost directory of your Java JDK/SDK installation
e.g. `JAVA_HOME=/java/jdk1.4.1_02`

J2EE_HOME – the topmost directory of target JBoss instance
e.g. `J2EE_HOME=/java/jboss-3.2.0/server/default`

CLASSPATH – the necessary driver support classes
e.g.
```
CLASSPATH=${J2EE_HOME}/lib/jboss-j2ee.jar:${ECPERF_HOME}/jars/ecperf-
client.jar:{ECPERF_HOME}/jars/driver.jar:${ECPERF_HOME}/jars/mfg.jar:${EC
PERF_HOME}/jars/orders.jar:${J2EE_HOME}/../../client/jbossall-client.jar
```

JDBC_CLASSPATH – the location of your JDBC driver for the target database (used for *loaddb*)
e.g. `J2EE_HOME${J2EE_HOME}/lib/pg73jdbc3.jar`

JAVAX_JAR – the jar containing the javax.* libraries used by the JBoss specific build
e.g. `CLASSPATH=${J2EE_HOME}/lib/jboss-j2ee.jar`

ECPERF_PORT – the port from which the ECperf application is served (defined in JBossWeb)
e.g. `ECPERF_PORT=8080`

EMULATOR_PORT – the port from which the emulator application is served (defined in JBossWeb)
e.g. `EMULATOR_PORT=8080`

A definition for the servlet libraries, not included in the original environment file, was added to help with the build process.

SERVLET_JAR – the location of the jar containing servlet libraries used by the JBoss specific build
e.g. `SERVLET_JAR=${J2EE_HOME}/lib/javax.servlet.jar`

Lastly, the JAVA command line options and the exports are re-defined for the introduced variables and the JBoss-specific JNDI interface. Note these should be contained in a single line.

e.g.
```
JAVA="$JAVA_HOME/bin/java
-Djava.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
-Djava.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
-Djava.naming.provider.url=jnp://localhost:1099
-Dorg.omg.CORBA.ORBInitialHost=$ECPERF_HOST"
```

e.g.
```
export JAVA_HOME J2EE_HOME JAVAX_JAR SERVLET_JAR ECPERF_HOST ECPERF_PORT
EMULATOR_HOST EMULATOR_PORT
```

1.4 Deploying components

The next stage involves configuring JBoss for the ECperf components and moving the built ECperf applications to JBoss. This step should not be difficult with an operational, clean install of the JBoss distribution.

1.4.1 Configuring the datasource

The datasource must be configured for the ECperf application and you must use the appropriate datasource template for the database you are using. Normally five definitions required, and they relate to the database schema partitioning or domains discussed earlier. With the JBoss ECperf release, there is only one datasource required. The datasource definition file is placed in the JBoss instance's deployment directory. An example is given for the Postgresql DBMS. I use an ECperf prefix as a naming convention to identify the datasource usage.

Use these examples as a guide to defining your own datasources.

ecperf-all-ds.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ===================================================================== -->
<!--                                                                       -->
<!--  JBoss Server Configuration                                           -->
<!--                                                                       -->
<!-- ===================================================================== -->

<!-- $Id: postgres-ds.xml,v 1.1 2002/07/22 22:57:24 d_jencks Exp $ -->
<!-- ==================================================================== -->
<!--  Datasource config for Postgres                                      -->
<!-- ==================================================================== -->
<datasources>
  <local-tx-datasource>
    <jndi-name>ECPerfDS</jndi-name>
    <connection-url>jdbc:postgresql://localhost:5432/ecperf</connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>x</user-name>
    <password>y</password>
  </local-tx-datasource>
</datasources>
```

1.4.2 Modifying the entity bean operation

By default the ECperf application's build and deploy process creates BMP-type entity beans. The JBoss modifications build CMP-type entity beans. The ECperf application also makes some demands on the operation of these beans that we must accommodate in JBoss.

The JBoss modification to ECperf 1.1 packages the JBoss specific deployment descriptors for the entity beans. This is located in the *src/deploy/jboss* directory of the distribution.

The modifications define a local *jboss.xml* that has the following line, controlling the commit policy:

```
<commit-option>B</commit-option>
```

The latter is a recommendation taken from the ECperf 1.0 JBoss kit.

1.4.3 Creating a deployment Ant script

We must now create the master build and deploy Ant script. Copy the ri.xml in the root directory of the ECperf 1.1 distribution to the same directory and rename it jboss.xml. The recommendations that come with the distribution suggest creating the following targets:

- ecperf-war-deploy
- ecperf-ejb-deploy
- supplier--deploy
- emulator--deploy

However, these aren't necessary for a JBoss deployment and the following script, modified from ri.xml script provides sufficient functionality for the service. The emulator and the ECperf applications are the only necessary targets to be deployed into JBoss.

```
<?xml version="1.0"?>
<!--  ECperf deploy actions -->

<project name="ECPerf-Ref-Server" default="ecperf-deploy">

  <!-- Load the external properties.
       These will override any settings here. -->
  <property file="config/jboss.env"/>

  <property name="appserver" value="jboss"/>
    <property name="compile.classpath"
            value="${SERVLET_JAR}:${JAVAX_JAR}:jars/xerces.jar"/>

  <target name="clean">
    <ant antfile="build.xml"
         target="clean"/></target>

  <target name="emulator-build">
    <ant antfile="build.xml"
         target="emulator-ear"/></target>

  <target name="loaddb">
    <ant antfile="build.xml"
         target="loaddb"/></target>

  <target name="ecperf-build">
    <ant antfile="build.xml"
         target="ecperf-ear"/></target>

  <target name="deploy-all" depends="ecperf-deploy,emulator-deploy" />

<!--  Targets to deploy .EAR files -->

  <target name="ecperf-deploy" depends="ecperf-build">
      <echo message="J2EE_HOME = ${J2EE_HOME} JAVAX_JAR = ${JAVAX_JAR}" />
      <echo message="SERVLET_JAR = ${SERVLET_JAR}" />
      <copy file="./jars/ecperf.ear" todir="${J2EE_HOME}/deploy" />
  </target>

  <target name="emulator-deploy" depends="emulator-build">
      <echo message="J2EE_HOME = ${J2EE_HOME} JAVAX_JAR = ${JAVAX_JAR}" />
      <echo message="SERVLET_JAR = ${SERVLET_JAR}" />
      <copy file="./jars/emulator.ear" todir="${J2EE_HOME}/deploy" />
  </target>

</project>
```

1.4.4 Creating the database loader configurations

For each of the domains in the ECperf database instance, a loader configuration is required so the database has some data loaded into the tables before the ECperf application may be used. These configurations are found in the *config* directory of the ECperf 1.1 distribution.

The files to be modified are:

- corpdb.properties
- mfgdb.properties
- ordsdb.properties
- suppdb.properties

The content of each file is the same, so you should modify each according to the database to be used for the ECperf testing. Since the load is performed from the machine on which have installed ECperf 1.1 and JBoss 3.2.x, you are required to provide the correct JDBC URL to connect to the database. You must also provide a temporary directory where data pipes are created.

An example of one definition file is provided here, giving the connection details for Postgresql.

corpdb.properties

```
# temp directory to write the pipes
# default is /tmp
pipeDir = /tmp

# Uncomment the lines for your specific Database
dbUser = x
dbPassword = y

# For DB2
#dbURL = jdbc:db2:ecperf
#dbDriver = COM.ibm.db2.jdbc.app.DB2Driver
#jdbcVersion = 1

# For Oracle thin driver
#dbURL = jdbc:oracle:thin:@host:1521:ecperf
#dbDriver = oracle.jdbc.driver.OracleDriver
#jdbcVersion = 2

# For Sybase
#dbURL = jdbc:sybase:Tds:host:port/ecperfdb
#dbDriver = com.sybase.jdbc2.jdbc.SybDriver
#jdbcVersion = 2

# For Postgresql
dbURL = jdbc:postgresql://localhost:5432/ecperf
dbDriver = org.postgresql.Driver
jdbcVersion = 2
```

1.4.5 Loading the database

Now that the database has been created and we have created the loader definitions for our database, we will load the database. This is accomplished using the newly created Ant script.

```
ant -f jboss.xml loaddb
```

Check that no error messages are displayed, and if some are thrown, rectify the loader definitions and your database connectivity.

1.4.6 Testing the deployment

This completes the customisation of JBoss for the ECperf application and initialisation of the database. Start JBoss and check there are no problems with the changes you have made earlier to the JBoss configurations. Rectify any problems and restart until JBoss starts cleanly.

Next, deploy the ECperf applications to JBoss.

```
ant -f jboss.xml clean
ant -f jboss.xml deploy-all
```

The ecperf.ear and emulator.ear application packages should have successfully deployed to your running JBoss instance. Should either of these have failed, check the deployment messages and rectify the problems respectively. Usually, the cause will be related to the deployment descriptors or the minor code changes made.

1.4.7 Testing the ECperf application

Once the deployments operate successfully, test the functionality of the application through the web interface. The access URL from the local machine will be:

```
http://<ECPERF_HOST>:<ECPERF_PORT>/ECperf/
```

When you have successfully reached the ECperf application start page, test the areas of customer and manufacturing, as well as accessibility to supply and order. These are the instructions for testing verbatim from the ECperf 1.1 README.

In Orders domain:

- Create a new order with one of the items having a quantity of more than 100 . This should cause a large order to be created in the Manufacturing Domain
- Change the order
- Get order information
- Get information on all orders of a customer
- Cancel an order

In Manufacturing Domain:

- Get a large order and start processing a work order based on it
- Move the resulting work order through its various stages
- Create a new work order
- Cancel the work order

To make sure Delivery and Emulator servlets are functioning, you should be able to verify by connecting to the following URLs:

```
http://<ECPERF_HOST>:<ECPERF_PORT>/Supplier/DeliveryServlet
http://<EMULATOR_HOST>:<EMULATOR_PORT>/Emulator/EmulatorServlet
```

1.4.8 Special note on testing functionality

Remember to initialise the database before using the ECperf application for the first time. The construction of the customer service locks in the list of available items for order and does not seem to update in the near term. Using the web interface will report the following:

"No items in the database !!!"

This will occur even after you load the database. Restarting JBoss will clear this fault.

1.5 Deploying the ECperf load tester

Now that the infrastructure and test applications are operational we can create the load testing harness. This harness is built around an RMI-based agent co-ordination system. The main test system, known as the Driver, communicates with the independent agents to generate load for the full manufacturing lifecycle of the test application, and collects statistics from them. This information is used to calculate the ECperf measure of performance.

The inflexibility of the test harness RMI binding was annoying so I also made modifications. These are not necessary but the default binding to port 1099 clashes with JBoss when they are running on the same machine so I added functionality to allow binding of the test harness' RMI registry to another port. It was also difficult to observe the faults on the test harness when starting it after JBoss which was another motivation to making this change.

The driver source code is contained in a single directory, *src/com/sun/ecperf/driver* of the ECperf 1.1 distribution. The files referred to in the following sections reside in that directory.

1.5.1 Modifying the Large Order Line agent

LargeOLAgent.java is modified to accept a fourth argument that describes the port to which the RMI registry is bound. This requires changing the argument count test conditions and modifying the connection URL to add the port. These changes are contained within the main(String[] argv) method.

Replace:
```
if (argv.length != 3) {
    System.out.println("Usage: LargeOLAgent <propsFile> <agentName> <masterMachine>");
            System.exit(-1);
        }
```

With:
```
if (argv.length < 3) {
    System.out.println("Usage: LargeOLAgent <propsFile> <agentName> <masterMachine>" +
                       "<port>");
            System.exit(-1);
        }
```

Replace:
```
String master = argv[2];
```

With:
```
String master = argv[2];
String port = null;
if (argv.length > 3)
    port = argv[3];
```

Replace:
```
String s1 = "//" + master + "/Controller";
```

With:
```
String s1 = null;
if (port != null)
    if (!port.equals(""))
        s1 = "//" + master + ":" + port + "/" + "Controller";
if (s1 == null)
    s1 = "//" + master + "/" + "Controller";
```

1.5.2 Modifying the Manufacturing agent

MfgAgent.java is modified to accept a fourth argument that describes the port to which the RMI registry is bound. This requires changing the argument count test conditions and modifying the connection URL to add the port. These changes are contained within the main(String[] argv) method.

Replace:
```
if (argv.length != 3) {
    System.out.println("Usage: MfgAgent <propsFile> <agentName> <masterMachine>");
            System.exit(-1);
        }
```

With:
```
if (argv.length < 3) {
    System.out.println("Usage: MfgAgent <propsFile> <agentName> <masterMachine>" +
                       "<port>");
            System.exit(-1);
        }
```

Replace:
```
String master = argv[2];
```

With:
```
String master = argv[2];
String port = null;
if (argv.length > 3)
    port = argv[3];
```

Replace:
```
String s1 = "//" + master + "/Controller";
```

With:
```
String s1 = null;
if (port != null)
    if (!port.equals(""))
        s1 = "//" + master + ":" + port + "/" + "Controller";
if (s1 == null)
    s1 = "//" + master + "/" + "Controller";
```

1.5.3 Modifying the Orders agent

OrdersAgent.java is modified to accept a fourth argument that describes the port to which the RMI registry is bound. This requires changing the argument count test conditions and modifying the connection URL to add the port. These changes are contained within the main(String[] argv) method.

Replace:
```
if (argv.length != 3) {
    System.out.println("Usage: OrdersAgent <propsFile> <agentName> <masterMachine>");
            System.exit(-1);
        }
```

With:
```
if (argv.length < 3) {
    System.out.println("Usage: OrdersAgent <propsFile> <agentName> <masterMachine>" +
                       "<port>");
            System.exit(-1);
        }
```

Replace:
```
String master = argv[2];
```

With:
```
String master = argv[2];
String port = null;
if (argv.length > 3)
    port = argv[3];
```

Replace:
```
String s1 = "//" + master + "/Controller";
```

With:
```
String s1 = null;
if (port != null)
    if (!port.equals(""))
        s1 = "//" + master + ":" + port + "/" + "Controller";
if (s1 == null)
    s1 = "//" + master + "/" + "Controller";
```

1.5.4 Modifying the Controller

Controller Impl.java is modified to accept an argument that describes the port to which the RMI registry is bound. This requires modifying the connection URL to add the port. The necessary changes are contained within the main(String[] argv) method.

Replace:
```
System.setSecurityManager (new RMISecurityManager());
```

With:
```
System.setSecurityManager (new RMISecurityManager());
String port = "";
if (argv.length > 0)
    port = argv[0];
```

Replace:
```
s = "//" + host + "/" + "Controller";
```

With:
```
if (port.equals(""))
    s = "//" + host + "/" + "Controller";
else
    s = "//" + host + ":" + port + "/" + "Controller";
```

1.5.5 Modifying the Driver

Driver.java is modified to accept a second argument that describes the port to which the RMI registry is bound. This requires changing the argument count test conditions and modifying the connection URL to add the port. These changes are contained in various methods within the source.

In method *main(String[] argv)*, replace:
```
if (argv.length < 1 || argv.length > 1) {
    System.err.println("Usage: Driver <properties_file>");
    return;
}
```

With:
```
if (argv.length < 1 || argv.length > 2) {
    System.err.println("Usage: Driver <properties_file> [<port>]");
    return;
}
```

In method *main(String[] argv)*, replace:
```
Driver d = new Driver(propsfile);
```

With:
```
String port = null;
if (argv.length > 1)
    port = argv[1];
Driver d = new Driver(propsfile, port);
```

In method *Driver(String propsFile)*, replace:
```
public Driver(String propsFile) throws Exception {
```

With:
```
public Driver(String propsFile, String port) throws Exception {
```

In method *Driver(String propsFile)*, replace:
```
configureAgents();
```

With:
```
configureAgents(port);
```

In method *configureAgents()*, replace:
```
protected void configureAgents() throws Exception {
```

With:
```
protected void configureAgents(String port) throws Exception {
```

In method *configureAgents()*, replace:
```
getAgentRefs();
```

With:
```
getAgentRefs(port);
```

In method *getAgentRefs()*, replace:
```
protected void getAgentRefs() throws Exception {
```

With:
```
protected void getAgentRefs(String port) throws Exception {
```

In method *getAgentRefs()*, replace:
```
/*****
String port = (props.getProperty("driverPort")).trim();
String s1 = "//" + host + ":" + port + "/" + "Controller";
****/
String s1 = "//" + host + "/" + "Controller";
```

With:
```
String s1 = null;
if (port != null)
    if (!port.equals(""))
        s1 = "//" + host + ":" + port + "/" + "Controller";
if (s1 == null)
    s1 = "//" + host + "/" + "Controller";
```

1.5.6 Modifying the Launcher

The launcher is a Linux/Unix specific co-ordination application for the test harness. This isn't necessary if you are only running the test harness in Windows. However, the changes prescribed here are required not only because of the changes for RMI binding but to also allow the JBoss JNDI interface parameters to be passed to the launcher environment.

The only source file required to be modified is *src/com/sun/ecperf/launcher/Driver.java*.

Replace:
```
String driverHost;
```

With:
```
String driverHost = null;
String port = null;
```

Replace:
```
if (args.length > 0)
    driverHost = args[1];
```

With:
```
if (args.length > 0)
    for (int i = 0; i < args.length; i++)
    {
        match = false;
        if (args[i].trim().length() > 2)
            if (args[i].trim().substring(0, 2).equals("-D"))
            {
                options.add(args[i].trim());
                match = true;
            }
        if (!match)
        {
            if (port == null)
                port = args[i];
            else if (driverHost == null)
                driverHost = args[i];
        }
    }
```

Replace:
```
int cmdLen = 0;
```

With:
```
int cmdLen = options.size();
```

Replace:
```
cmd.add("-Djava.security.policy=" + driverPolicy);
```

With:
```
for (int i = 0; i < options.size(); i++)
    cmd.add((String)options.get(i));
cmd.add("-Djava.security.policy=" + driverPolicy);
```

Replace:
```
cmd.add("-Djava.security.policy=" + driverPolicy);
cmd.add(driverPackage + "ControllerImpl");
```

With:
```
for (int i = 0; i < options.size(); i++)
    cmd.add((String)options.get(i));
cmd.add("-Djava.security.policy=" + driverPolicy);
cmd.add(driverPackage + "ControllerImpl");
cmd.add(port);
```

Replace:
```
// orders agent
```

With:
```
// orders agent
cmd.remove(cmd.size()-1);
```

Replace:
```
cmd.set(cmdLen + 3, "com.sun.ecperf.charts.StreamChart");
```

With:
```
cmd.set(cmdLen + 3, "com.sun.ecperf.charts.StreamChart");
cmd.remove(cmd.size()-1);
```

Replace:
```
Launcher driver = new Launcher(cmd, environment);
```

With:
```
cmd.add(port);
Launcher driver = new Launcher(cmd, environment);
```

1.5.7 Compiling the new Driver

From the root directory of the ECperf 1.1 distribution, execute the following commands to build the driver:

```
ant –Dappserver=jboss clean-driver
ant –Dappserver=jboss driver
```

Observe any errors, rectify them and compile again until all the errors are resolved.

1.5.8 Modifying the Driver batch script

In order for the Driver batch script to use these changes, there are some necessary modifications. We also need to make an accommodation for the JBoss JNDI interface class and include a definition for the JBoss JNDI client libraries. The script is *bin/driver.bat* in the ECperf 1.1 distribution.

Add these definitions after the NAMING_PROVIDER definition but before the CLASSPATH definition. The REGISTRY_PORT is the port to which the RMI registry is to be bound.

```
REM JBoss specific interface definition
set INTERFACES="org.jboss.naming:org.jnp.interfaces"

REM RMI registry port
set REGISTRY_PORT=1098

REM JBoss client library
set CLIENT_JAR=C:\java\jboss-3.2.0\client\jbossall-client.jar
```

The single line CLASSPATH definition contains a few omissions and also needs to include the JBoss JNDI client. The definition should match this.

```
CLASSPATH=%ECPERF_HOME%\jars\ECPerf.jar;%J2EE_HOME%\lib\jboss-j2ee.jar;%ECPERF_HOME%\jars\ecperf-
client.jar;%ECPERF_HOME%\jars\driver.jar;%ECPERF_HOME%\jars\mfg.jar;%ECPERF_HOME%\jars\orders.jar;%CLIENT
_JAR%
```

The code that starts the driver must be replaced with the following:

```
start rmiregistry %REGISTRY_PORT%
Pause
start %JAVA_HOME%\bin\java -Djava.security.policy=%DRIVER_POLICY%
%DRIVER_PACKAGE%.ControllerImpl %REGISTRY_PORT%
Pause

start %JAVA_HOME%\bin\java -Djava.naming.factory.initial=%JNDI_CLASS%
-Djava.naming.factory.url.pkgs=%INTERFACES%
-Djava.naming.provider.url=%NAMING_PROVIDER% -Djava.security.policy=%DRIVER_POLICY%
-Dorg.omg.CORBA.ORBInitialHost=%SUT_MACHINE% %DRIVER_PACKAGE%.MfgAgent
%CONFIG_DIR%/agent.properties M1 %DRIVER_MACHINE% %REGISTRY_PORT%

start %JAVA_HOME%\bin\java -Djava.naming.factory.initial=%JNDI_CLASS%
-Djava.naming.factory.url.pkgs=%INTERFACES%
-Djava.naming.provider.url=%NAMING_PROVIDER% -Djava.security.policy=%DRIVER_POLICY%
-Dorg.omg.CORBA.ORBInitialHost=%SUT_MACHINE% %DRIVER_PACKAGE%.LargeOLAgent
%CONFIG_DIR%/agent.properties L1 %DRIVER_MACHINE% %REGISTRY_PORT%

start %JAVA_HOME%\bin\java -Djava.naming.factory.initial=%JNDI_CLASS%
-Djava.naming.factory.url.pkgs=%INTERFACES%
-Djava.naming.provider.url=%NAMING_PROVIDER% -Djava.security.policy=%DRIVER_POLICY%
-Dorg.omg.CORBA.ORBInitialHost=%SUT_MACHINE%  %DRIVER_PACKAGE%.OrdersAgent
%CONFIG_DIR%/agent.properties O1 %DRIVER_MACHINE% %REGISTRY_PORT%
Pause

%JAVA_HOME%\bin\java -Djava.naming.factory.initial=%JNDI_CLASS%
-Djava.naming.factory.url.pkgs=%INTERFACES%
-Djava.naming.provider.url=%NAMING_PROVIDER% %DRIVER_PACKAGE%.Driver
%CONFIG_DIR%/run.properties %REGISTRY_PORT%
```

Follow the instructions embedded in the batch script, changing definition of NAMING_PROVIDER to the correct destination for the JBoss JNDI server, changing the JNDI_CLASS to refer to the JBoss naming context factory, and changing the definition of REGISTRY_PORT to the port to which you want the RMI registry bound.

1.5.9 Modifying the Driver shell script

In order for the Driver shell script to use these changes, there are some necessary modifications. We also need to make an accommodation for the JBoss JNDI client interface. The script is *bin/driver.sh* in the ECperf 1.1 distribution.

Replace:

```
${JAVA_HOME}/bin/java -classpath ${ECPERF_HOME}/jars/launcher.jar \
    -Djava.compiler=NONE \
    -Decperf.home=${ECPERF_HOME} \
    -Dnode.name=`uname -n` \
    `shellProps` \
    com.sun.ecperf.launcher.Script Driver $*;
```

With:

```
JNDI_CLASS="-Djava.naming.factory.initial=org.jnp.interfaces.NamingContextFactory"
INTERFACES="-Djava.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces"
NAMING_PROVIDER="-Djava.naming.provider.url=jnp://172.16.1.11:1099"
REGISTRY_PORT=1098

${JAVA_HOME}/bin/java -classpath ${ECPERF_HOME}/jars/launcher.jar \
    -Djava.compiler=NONE \
    -Decperf.home=${ECPERF_HOME} \
    -Dnode.name=`uname -n` \
    `shellProps` \
    com.sun.ecperf.launcher.Script Driver ${JNDI_CLASS} ${INTERFACES} \
    ${NAMING_PROVIDER} ${REGISTRY_PORT} $*
```

Change the definition of NAMING_PROVIDER to the correct destination for the JBoss JNDI server and change the definition of REGISTRY_PORT to the port to which you want the RMI registry bound.

1.5.10 Testing the scripts

You can test the scripts now but ensure that the JBoss system is running otherwise the scripts will terminate if the target is not detected. Make any changes to the scripts or alternately to the changes in the driver and launcher code, depending in the source of the errors.

1.6 Running ECperf 1.1

The system under test (SUT) and the test harness are now ready.

The test harness can be installed on a separate machine. The only required directories from the ECperf distribution are the *bin*, *config* and *jars* directories. You will also need the jbossall-client.jar if you are testing against JBoss.

You may rebuild your database again if you wish before you run the test. Just run the *schema/blddb.sh* and then load the database using the Ant *loaddb* target as described earlier.

Restart the JBoss SUT.

Run the applicable script. Refer to the ECperf 1.1 documentation for more information on changing the load and the interpretation of the test results.

More information is given on the files that will govern the testing and the configuration for locating components in the test environment, related to the type of system on which the test harness is run.

1.6.1 Configuring for testing under Windows

Most of the configuration for running the test harness is confined to three files:
- *bin/driver.bat* which controls the run time environment
- *config/run.properties* which controls the test load characteristics and data gathering

The important variables to modify for your situation in *bin/driver.bat* script are:

ECPERF_HOME	The home directory for your ECperf 1.1 distribution
JAVA_HOME	The home directory for your JDK/SDK installation (comment it out if you define it outside the batch file)
J2EE_HOME	The home directory for the J2EE library (normally a helper to locate the *jboss-j2ee.jar* on the test harness machine)
ECPERF_HOST	The host address of the SUT (the JBoss/ECperf machine)
ECPERF_PORT	The port from which the SUT application content is served (8080 for the default JBossweb installation)
EMULATOR_HOST	The host address of the supplier emulator
EMULATOR_PORT	The port from which the supplier emulator content is served (8080 for the default JBossweb installation)
JNDI_CLASS	The JNDI interface context factory (org.jnp.interfaces.NamingContextFactory for JBoss)
INTERFACES	The Interface helper used only in JBoss (*org.jboss.naming:org.jnp.interfaces* for JBoss)
REGISTRY_PORT	The port to which the local RMI registry is bound so the test harness can register and use the test agent services
CLIENT_JAR	The JNDI client libraries necessary for the test harness to communicate with the SUT (in JBoss this refers to the location of *jbossall-client.jar*)
NAMING_PROVIDER	The URL to the naming provider for the SUT (of the form *jnp://localhost:1099* for JBoss)
CLASSPATH	The classpath locating all the necessary libraries for the test harness

The classpath example below shows the necessary libraries for the test harness.

```
CLASSPATH=%ECPERF_HOME%\jars\ECPerf.jar;%J2EE_HOME%\lib\jboss-
j2ee.jar;%ECPERF_HOME%\jars\util.jar;%ECPERF_HOME%\jars\driver.jar;%ECPERF_H
OME%\jars\mfg.jar;%ECPERF_HOME%\jars\orders.jar;%CLIENT_JAR%
```

In particular, note that the original ECperf 1.1 batch script does omit some of the ECperf jars that are actually required. There are also other variables the batch scripts defined that I have found that do not affect any outcomes for the modifications prescribed here. Therefore, I have not described in them in detail for the purposes of clarity.

The *config/run.properties* file controls much of the load conditions and the result collation. This is commented well enough that I do not need to elaborate on the use. However, from a run time perspective, ensure that you define *outDir* to a directory that exists so the results and errors may be generated. The test harness will not run if this directory does not exist.

When this has been configured to your satisfaction, run the batch script to start the testing.

1.6.2 Configuring for testing under Linux/Unix

Most of the configuration for running the test harness is confined to three files:
- *bin/driver.sh* which controls the run time environment
- *config/appserver* which contains the name of the environment file to use
- *config/jboss.env* which contains other definitions used by the run time environment
- *config/run.properties* which controls the test load characteristics and data gathering

The important variables to modify for your situation in *bin/driver.sh* script are:

ECPERF_HOME	The home directory for your ECperf 1.1 test harness and is automatically detected by the script
JAVA_HOME	The home directory for your JDK/SDK installation end is normally defined outside this shell script
J2EE_HOME	The home directory for the J2EE library (normally a helper to locate the *jboss-j2ee.jar* on the test harness machine)
JNDI_CLASS	The JNDI interface context factory (org.jnp.interfaces.NamingContextFactory for JBoss)
INTERFACES	The Interface helper used only in JBoss (*org.jboss.naming:org.jnp.interfaces* for JBoss)
REGISTRY_PORT	The port to which the local RMI registry is bound so the test harness can register and use the test agent services

The *config/appsserver* file only needs to contain a single line entry. For the JBoss test configuration, the entry is required to be the word '*jboss*', which indicates that the shell script should use the *jboss.env* file for defining the remainder of the run time environment.

The important variables to modify for your situation in *config/jboss.env* script are:

J2EE_BASE	The home directory for the J2EE JBoss libraries (normally a helper to locate the *jboss-j2ee.jar* and *jbossall-client.jar* on the test harness machine)
JAVA_HOME	The home directory for your JDK/SDK installation (a backup definition but not normally used)
J2EE_HOME	The home directory for the J2EE library (normally a helper to locate the *jboss-j2ee.jar* on the test harness machine)
CLIENT_JAR	The JNDI client libraries necessary for the test harness to communicate with the SUT (in JBoss this refers to the location of *jbossall-client.jar*)
CLASSPATH	The classpath locating all the necessary libraries for the test harness to operate
ECPERF_HOST	The host address of the SUT (the JBoss/ECperf machine)
ECPERF_PORT	The port from which the SUT application content is served (8080 for the default JBossweb installation)
ECPERF_PREFIX	The prefix used in the ECperf web application (the default '/' used is fine for JBossweb)
EMULATOR_HOST	The host address of the supplier emulator
EMULATOR_PORT	The port from which the supplier emulator content is served (8080 for the default JBossweb installation)
EMULATOR_PREFIX	The prefix used in the supplier emulator web application (the default '/' used is fine for JBossweb)

The classpath example below shows the necessary libraries for the test harness.

```
CLASSPATH=${J2EE_HOME}/lib/jboss-
j2ee.jar:${ECPERF_HOME}/jars/util.jar:{ECPERF_HOME}/jars/driver.jar:${ECPERF
_HOME}/jars/mfg.jar:${ECPERF_HOME}/jars/orders.jar:${CLIENT_JAR}
```

In particular, note that the original ECperf 1.1 shell script does omit some of the ECperf jars that are actually required. There are also other variables the batch scripts define that I have found not to affect any outcomes for the modifications prescribed here. Therefore, I have not described in them in detail for the purposes of clarity.

The *config/run.properties* file controls much of the load conditions and the result collation. This is commented well enough that I do not need to elaborate on the use. However, from a run time perspective, ensure that you define *outDir* to a directory that exists so the results and errors may be generated. The test harness will not run if this directory does not exist.

When this has been configured to your satisfaction, run the shell script to start the testing.

1.7 Applying the code changes

The changes prescribed in this document are provided in a single tarball and includes this document. Simply download your JBoss SourceForge ECperf 1.1 distribution and then unpack the ECperf-JBoss tarball over the top. The necessary files will be overlaid with the changes and new files will be installed in the correct areas. Run clean and make for the ECperf 1.1 applications as well as the driver and you should be set to begin testing and tweaking your ECperf-enabled JBoss installation.