

Universidad Nacional del Centro  
de la Provincia de Buenos Aires  
Facultad de Ciencias Exactas



# Introducción a Grid Computing

## Integrantes:

Pérez Fuentes, Joaquín Alejandro

[joaquinpf@gmail.com](mailto:joaquinpf@gmail.com)

Steimbach, Marcos Maximiliano

[msteimba@alumnos.exa.unicen.edu.ar](mailto:msteimba@alumnos.exa.unicen.edu.ar)

## INTRODUCCION

Se plantea como objetivo la inclusión de la plataforma Condor en el proyecto JGRIM.

La meta consiste en realizar una solución acorde al diseño actual de JGRIM, proveyendo una manera simple y unívoca de invocar ejecutables en un cluster Condor, obtener su respuesta y brindar mecanismos sencillos de configuración al usuario.

## DESARROLLO DE LA SOLUCION

### DISEÑO DE LA SOLUCIÓN

Partiendo de la premisa de obtener una solución simple y que respete los estándares de diseño actuales de JGRIM, se comenzó haciendo un análisis de la plataforma para identificar partes reutilizables del diseño del módulo *parallelization*.

Esto resultó en que el esquema actual (propuesto por IbisServer, IbisClient, IbisMethods, IbisExecutionRequest, IbisResult e IbisInterceptor) podía adaptarse sin mayor problemas a Condor.

En este esquema, el usuario debe generar primero la instanciación particular de IbisMethods (tanto interface como implementación), junto con su correspondiente XML para la ejecución de Spring. El usuario entonces corre uno de los métodos definidos en IbisMethods. Estos son capturados por el interceptor, que genera el cliente pertinente y envía la solicitud de ejecución al servidor, para luego aguardar por la respuesta de este y devolvérsela al usuario.

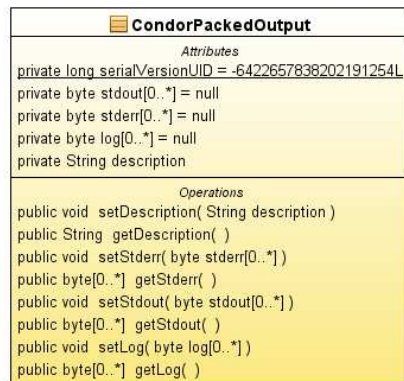
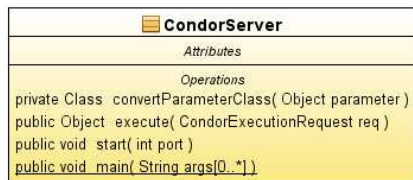
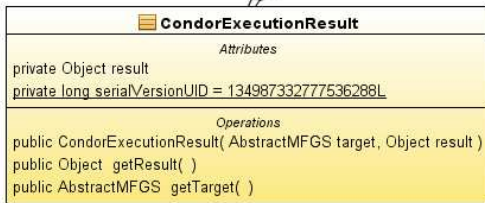
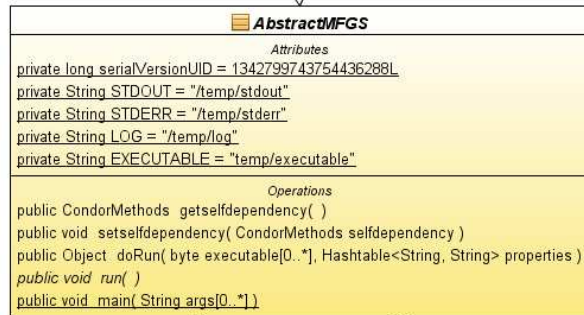
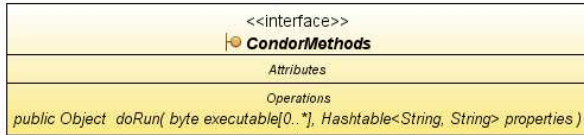
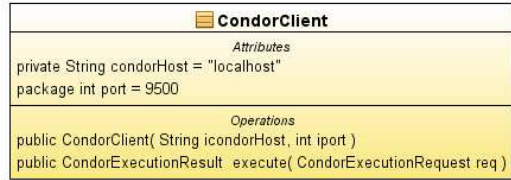
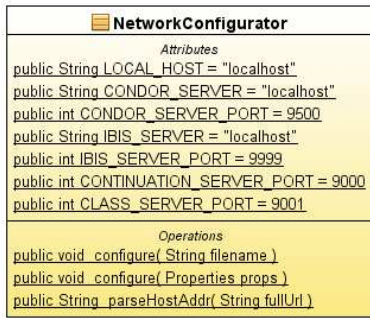
A simple vista las diferencias de Condor con el esquema actual son:

- Necesidad de Condor de serializar y enviar el ejecutable en el pedido de invocación.
- Menor cantidad de puntos de configuración dado que Condor no está atado al lenguaje y simplemente corre “ejecutables” en un universo dado. No existe necesidad de generar la interface “IbisMethods” para cada aplicación en particular ya que siempre se terminará corriendo el mismo método. De la misma manera, ya no es necesario generar un XML (aunque puede requerirse) particular para Spring, este será provisto por el framework. De este modo se le deja al usuario únicamente la carga del ejecutable y sus opciones de invocación.

Una línea de ejecución de la aplicación debería ser:

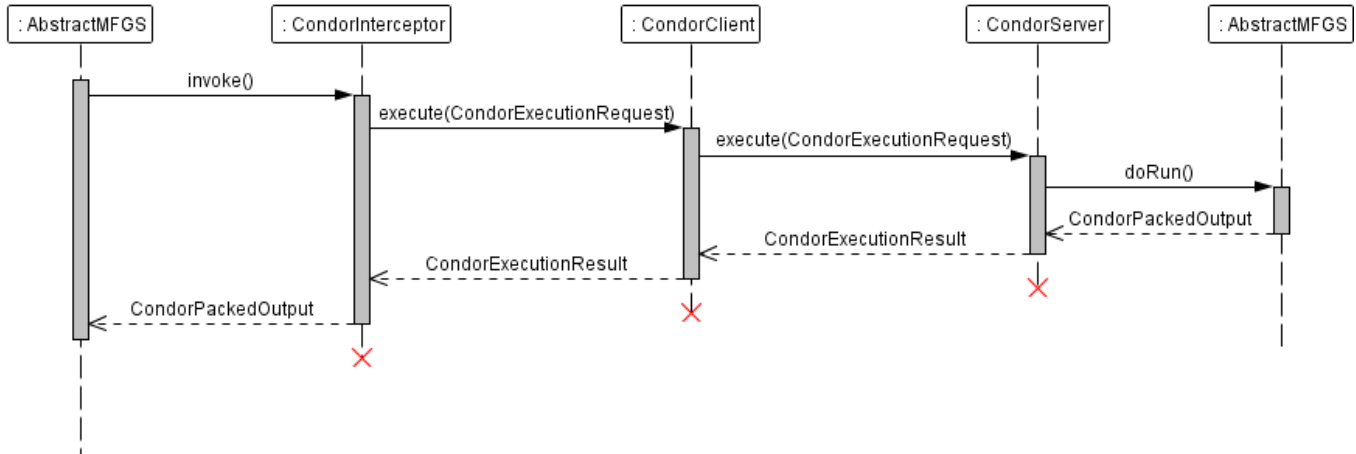
*El usuario carga el ejecutable y opciones de ejecución de Condor en memoria e invoca el método de ejecución de Condor. Este último será capturado por Spring y enviado como solicitud al servidor, que realizará la ejecución real según la configuración de Condor que posea y las opciones enviadas por el usuario. Luego de ejecutarse, se deben obtener los archivos resultantes, serializarlos y empaquetarlos, para entonces devolver normalmente la respuesta al usuario.*

El diseño que soporta todo esto puede verse reflejado en el siguiente diagrama de clases.



Debe notarse que el método “run()” de la clase AbstractMFGS es el único hook que provee la implementación de Condor sobre JGRIM. En este método se deberán cargar el ejecutable (utilizando la clase BinaryManipulator) y las opciones de ejecución de Condor, definidas como pares “clave-valor” válidos para los “submit description” de Condor.

La interacción entre las clases para un curso típico de la aplicación puede apreciarse en el siguiente diagrama de secuencia:




Cabe destacar que el diagrama anterior representa una versión simplificada de la línea de ejecución real, mostrando de manera simple la presencia de Spring en CondorInterceptor, pero no su modo de trabajo real (no es llamado explícitamente).

La descripción específica de cada clase será mostrada a continuación.

## DETALLES DE LAS CLASES Y ARCHIVOS DE CONFIGURACION DE LA SOLUCIÓN

### ABSTRACTMFGS

 <b>AbstractMFGS</b>
<p><i>Attributes</i></p> <pre>private long serialVersionUID = 1342799743754436288L private String STDOUT = "/temp/stdout" private String STDERR = "/temp/stderr" private String LOG = "/temp/log" private String EXECUTABLE = "temp/executable"</pre>
<p><i>Operations</i></p> <pre>public CondorMethods getselfdependency( ) public void setselfdependency( CondorMethods selfdependency ) public Object doRun( byte executable[0..*], Hashtable&lt;String, String&gt; properties ) public void run( ) public void main( String args[0..*])</pre>

La clase AbstractMFGS es el corazón de la ejecución en Condor e implementa el método main que será llamado en toda ejecución Condor. Cabe destacar que este método hace referencia al archivo simple-condor.xml y está configurado para instanciar la clase CondorTest. Para cambiar esto y ejecutar cualquier clase implementada por el usuario, se debe editar la línea siguiente del archivo simple-condor.xml:

```
<bean id="service" class="test.parallelization.condor.CondorTest">
```

**Para configuraciones avanzadas, o utilización de otro XML o varios XML, debe generarse un nuevo método main de la misma manera que el actual, referenciando al XML específico.**

Un ejemplo de esto es:

```
public static void main(String[] args) throws Exception {
    NetworkConfigurator.configure("configuration/network.properties");
    PropertyConfigurator.configure("configuration/log4j.properties");
    ApplicationContext appContext = new
    FileSystemXmlApplicationContext(
        "configuration/simple-condor.xml");
    Runnable service = (Runnable) appContext.getBean("service");
    service.run();
}
```

**Su método “doRun(byte[], Hashtable<String,String>)”** se encarga de implementar el pedido de ejecución a Condor. Esto involucra la creación de un Job Description, objeto que contiene todas las opciones necesarias para la ejecución de Condor, generar el pedido, esperar y devolver el resultado.

Para trabajar de manera genérica los destinos del ejecutable y salidas de Condor se generan de manera opaca al usuario, razón por la cual las opciones “output”, “error”, “log” y “log\_xml” serán ignoradas y generadas por el método como la concatenación de un método predeterminado y el hash del objeto invocador. De esta manera podemos obtener un mejor control de la ejecución y evitar problemas en ejecuciones concurrentes que puedan llegar a sobrescribir los mismos archivos. Se debe tener en cuenta todo esto para luego eliminar los archivos temporales del servidor.

La espera de resultado se realiza de manera no bloqueante verificando el estado de la ejecución cada cierto periodo de tiempo.

Los valores de retorno esperables dentro de la descripción en CondorPackedOutput son:

**"Execution failed with (Exception)"**: Indica el fallo de la ejecución con el nombre de Exception dado.

**"Execution complete"**: Indica que la ejecución fue completada con éxito.

**"Null parameters recieved in AbstractMFGS.doRun(...) (either executable or properties)"**: Indica que no se llegó a ejecutar Condor por errores en los parámetros recibido.

De manera simplificada podemos ver al método como:

```
byte[] toRun = BinaryManipulator.decompressByteArray(executable);
BinaryManipulator.writeByteArray(localExecutable, toRun);

// Condor init
Condor.setDebug(true);
Condor condor = new Condor();

JobDescription jd = new JobDescription();
Enumeration<String> auxEnum = properties.keys();

// Agregar executable, stderr, stdout y logging al job description
jd.addAttribute("executable", localExecutable);
jd.addAttribute("output", localStdout);
jd.addAttribute("error", localStderr);
jd.addAttribute("log_xml", "True");
jd.addAttribute("log", localLog);

// Si el universo no fue enviado como parámetro se utiliza vanilla
if(properties.get("universe") == null){
    jd.addAttribute("universe", "vanilla");
}

// Agregar los parámetros específicos del usuario
while (auxEnum.hasMoreElements()) {
    String key = auxEnum.nextElement();
    if (key != "executable" && key != "output"
        && key != "error" && key != "log"
        && key != "log_xml")
        jd.addAttribute(key, properties.get(key));
}
jd.addQueue();

// Enviar el trabajo y esperar a que condor finalice
Cluster cluster = condor.submit(jd);
Job j = cluster.getJob(0);
while (!j.isCompleted()) {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e1) {
        e1.printStackTrace();
    }
}
String result = "Execution complete";
CondorPackedOutput c = new CondorPackedOutput();
c.setDescription(result);
c.setStderr(BinaryManipulator.readByteArray(STDERR));
c.setStdout(BinaryManipulator.readByteArray(STDOUT));
c.setLog(BinaryManipulator.readByteArray(LOG));
```

```
FileUtils.cleanupDirectory("temp/", hash);  
return c;
```

El método “run()” de la clase AbstractMFGS es el único hook que provee la implementación de Condor sobre JGRIM. En este método se deberán cargar el ejecutable (utilizando la clase BinaryManipulator) y las opciones de ejecución de Condor, definidas como pares “clave-valor” válidos para los “submit description” de Condor.

Valores importantes a tener en cuenta son:

- **universe-standard/vanilla/java:** Define el universo de ejecución de Condor. Si se quiere correr un binario Java (class) se debería definir “java” como el universo. De manera similar, si se desea correr un ejecutable standard de consola de Windows/Dos se puede utilizar el universo “vanilla”. Universe debe pasarse como parámetro, caso contrario se utilizará vanilla.

Para mantener la generación de trabajos sencilla, no se tendrán en cuenta los valores (serán utilizados valores por defecto):

- **output:** Define la ubicación del archivo que contendrá la salida estándar de la ejecución de Condor.
- **error:** Define la ubicación del archivo que contendrá la salida estándar de errores de la ejecución de Condor.
- **log:** Define la ubicación del archivo de log de la ejecución de Condor.
- **log\_xml:** Define si se utilizan logs con formato XML.

Un pequeño ejemplo de la utilización de run() se ve a continuación:

```
//Carga y comprime el ejecutable en memoria  
byte[] b = BinaryManipulator.readByteArray("EJECUTABLE");  
b = BinaryManipulator.compressByteArray(b);  
CondorPackedOutput result;  
//Carga las opciones particulares para setearle a condor  
Hashtable<String, String> h = new Hashtable<String, String>();  
h.put("universe", "vanilla");  
CondorMethods selfDep = getSelfDependency();  
if (selfDep != null) {  
    CondorPackedOutput result = (CondorPackedOutput) selfDep.doRun(b, h)  
}
```

## CONDORMETHODS




La interface CondorMethods define la signatura de los métodos que pueden ejecutarse mediante Spring. En nuestro caso, el único método disponible es doRun(byte[], Hashtable<String,String>) que genera la ejecución Condor del ejecutable pasado como bytearray. Este método está implementado en AbstractMFGS.



---

## CONDORPACKEDOUTPUT

 <b>CondorPackedOutput</b>
<i>Attributes</i> <u>private long serialVersionUID = -6422657838202191254L</u> private byte stdout[0..*] = null private byte stderr[0..*] = null private byte log[0..*] = null private String description
<i>Operations</i> public void setDescription( String description ) public String getDescription( ) public void setStderr( byte stderr[0..*] ) public byte[0..*] getStderr( ) public void setStdout( byte stdout[0..*] ) public byte[0..*] getStdout( ) public void setLog( byte log[0..*] ) public byte[0..*] getLog( )

La clase CondorPackedOutput actua de paquete para los diversos archivos generados en la ejecución de un programa sobre Condor. Estos archivos son:

- Salida de stdout
- Salida de stderr
- Log

Cada uno de estos archivos se serializa con la ayuda de BinaryManipulator y se setea como propiedad a la clase.

Existe ademas un String que contiene la descripción pertinente al resultado de la operación, tanto si fue exitosa como si falló.

---

## CONDOREXECUTIONREQUEST

 <b>CondorExecutionRequest</b>
<i>Attributes</i> <u>private long serialVersionUID = 1342799743777536288L</u> private ServiceCallInfo info
<i>Operations</i> public CondorExecutionRequest( AbstractMFGS itarget, ServiceCallInfo iinfo ) public AbstractMFGS getTarget( ) public ServiceCallInfo getInfo( )

La clase CondorExecutionRequest encapsula todo lo necesario para realizar un pedido de ejecución a un CondorServer. Su funcionamiento es muy similar al de IbisExecutionRequest de JGRIM con la única salvedad



de que la aplicación serializada es en este caso una instancia de la implementación de AbstractMFGS. Además de esto lleva los datos del método a ejecutar, junto con sus parámetros para la invocación en el servidor.

---

## CONDOREXECUTIONRESULT

CondorExecutionResult
<i>Attributes</i>
private Object result <u>private long serialVersionUID = 134987332777536288L</u>
<i>Operations</i>
public CondorExecutionResult( AbstractMFGS target, Object result ) public Object getResult( ) public AbstractMFGS getTarget( )

De manera análoga a CondorExecutionRequest, esta clase realiza el camino inverso siendo la encargada de encapsular una respuesta derivada de un pedido de ejecución en condor.

El objeto resultado contenido en la clase es una instancia de CondorPackedOutput.

---

## CONDORSERVER

CondorServer
<i>Attributes</i>
<i>Operations</i>
private Class convertParameterClass( Object parameter ) public Object execute( CondorExecutionRequest req ) public void start( int port ) <u>public void main( String args[0..*] )</u>

Esta clase tiene por objetivo implementar el lado servidor de una ejecución Condor.


Al arrancar se abre un ServerSocket según la configuración especificada por NetworkConfigurator, para luego esperar hasta recibir un CondorExecutionRequest.

El método **execute(CondorExecutionRequest)** se encarga de, dado el CondorExecutionRequest recibido, desempaquetar las opciones y el ejecutable, para luego invocar el método adecuado, que llega como un ServiceCallInfo incluido en el pedido de ejecución.

Luego de esto debe simplemente esperar por la respuesta del método, empaquetarla en un CondorExecutionResult y devolver el resultado.

---

## CONDORCLIENT

 <b>CondorClient</b>
<i>Attributes</i>
private String condorHost = "localhost"
package int port = 9500
<i>Operations</i>
public CondorClient( String icondorHost, int iport )
public CondorExecutionResult execute( CondorExecutionRequest req )


El propósito de la clase CondorClient es la de manejar el lado cliente de una ejecución Condor. Esto significa que es la encargada de, dado un CondorExecutionRequest, enviar el pedido de ejecución al servidor especificado en NetworkConfigurator y aguardar la respuesta

## CONDORINTERCEPTOR

 <b>CondorInterceptor</b>
<i>Attributes</i>
protected MFGS ownerApp = null
<u>private long serialVersionUID = 134223443754436288L</u>
<i>Operations</i>
public Object invoke( MethodInvocation invoke )
public void setOwnerApp( MFGS ownerApp )
public MFGS getOwnerApp( )
protected void setStateDependencies( Object source, Object target )
protected void setOwnerAgent( Object peer )

Esta clase se encarga de interceptar la ejecución de los métodos definidos en CondorMethods, ejecutados desde la implementación de AbstractMFGS como “selfdependency.method()”. Sus funciones principales son las de generar el pedido de ejecución de Condor (CondorExecutionRequest) e instanciar el CondorClient que será alimentado con dicha solicitud.

## FILEUTILS

 <b>FileUtils</b>
<i>Attributes</i>
<i>Operations</i>
<u>public void cleanUpDirectory( String directory, String hash )</u>


La clase FileUtils provee funciones para el manejo de archivos en disco.

El método “cleanUpDirectory” sera el encargado de limpiar un directorio dado, teniendo en cuenta un hash en el nombre de archivos, luego de finalizada una ejecución de condor. Recordemos que tendremos archivos

guardados en disco como salida de stdout, stderr, log y ejecutable, los cuales luego de la ejecución ya no son necesarios.

---

## BINARYMANIPULATOR


 <b>BinaryManipulator</b>
<i>Attributes</i>
<i>Operations</i>
<u><code>public byte[0..*] readByteArray( String file )</code></u>
<u><code>public void writeByteArray( String newFile, byte bytearray[0..*] )</code></u>
<u><code>public byte[0..*] compressByteArray( byte bytearray[0..*] )</code></u>
<u><code>public byte[0..*] decompressByteArray( byte bytearray[0..*] )</code></u>

La clase BinaryManipulator se encarga de proveer los métodos necesarios para convertir archivos a byte[], comprimirlos, descomprimirlos o escribir un byte[] a un archivo dado del disco.

Esta clase se utiliza para levantar y comprimir en memoria los archivos necesarios para un pedido de ejecución de Condor, o su respuesta.

---

## NETWORKCONFIGURATOR

 <b>NetworkConfigurator</b>
<i>Attributes</i>
<u><code>public String LOCAL_HOST = "localhost"</code></u>
<u><code>public String CONDOR_SERVER = "localhost"</code></u>
<u><code>public int CONDOR_SERVER_PORT = 9500</code></u>
<u><code>public String IBIS_SERVER = "localhost"</code></u>
<u><code>public int IBIS_SERVER_PORT = 9999</code></u>
<u><code>public int CONTINUATION_SERVER_PORT = 9000</code></u>
<u><code>public int CLASS_SERVER_PORT = 9001</code></u>
<i>Operations</i>
<u><code>public void configure( String filename )</code></u>
<u><code>public void configure( Properties props )</code></u>
<u><code>public String parseHostAddr( String fullUrl )</code></u>

La clase NetworkConfigurator maneja la configuración de red para las diferentes plataformas habilitadas y trabaja como punto de acceso único para su utilización.

Hace uso de los diferentes parámetros seteados en el archivo network.properties, como pueden ser la dirección del server Condor, o el puerto a utilizar.

Los parámetros a tener en cuenta de dicho archivo son:

- condor.server: El server al que se debe delegar la ejecución de Condor (ej: localhost, 192.168.1.123).

- condor.server.port: El puerto al que se debe conectar el cliente para delegar (ej: 9500).

## XML DE CONFIGURACION PARA SPRING

El siguiente XML define las propiedades necesarias para Spring. Estas son: el interceptor, los métodos que maneja (por la interface CondorMethods) y la clase que se ejecutará con este. Este último debería ser el único punto que el usuario necesitaría cambiar, como se explicó en la sección de AbstractMFGS.

Un ejemplo completo del XML se muestra a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-2.0.xsd">
    <bean id="service" class="test.parallelization.condor.CondorTest">
        <property name="selfdependency"><ref local="selfdependency"/></property>
    </bean>
    <!-- Parallelism -->
    <bean id="selfdependency" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property
name="proxyInterfaces"><value>core.parallelization.condor.CondorMethods</value></property>
        <property name="interceptorNames">
            <list>
                <value>condorInterceptor</value>
            </list>
        </property>
    </bean>
    <bean class="core.parallelization.condor.CondorInterceptor" id="condorInterceptor">
        <property name="ownerApp"><ref bean="service"/></property>
    </bean>
</beans>
```

## EJEMPLO DE CASO DE PRUEBA

En el paquete “test.parallelization.condor” se presenta un caso de prueba de la aplicación, la clase “CondorTest”. Veamos como se forma esta clase.

En primer lugar, cualquier aplicación que desee utilizar Condor debe extender AbstractMFGS.

```
public class CondorTest extends AbstractMFGS
```

Luego de hacer eso hay dos puntos que se deben tomar en cuenta. El primero y mas sencillo consiste en generar la constante serialVersionUID. Esto puede realizarse de forma automática con Eclipse y es mandatorio para la serialización de la clase.

Un ejemplo de esto es:

```
/** The Constant serialVersionUID. */
private static final long serialVersionUID = 13498730928756288L;
```

Por ultimo, debemos implementar la corrida de Condor. Para realizar esto debemos implementar el método abstracto run() de AbstractMFGS. En este método se deben cargar el ejecutable y las opciones de invocación que deben ser pasadas a Condor. Un ejemplo sencillo puede verse a continuación:

```
@Override
public void run() {
    //Carga del ejecutable en memoria
    byte[] b = BinaryManipulator.readByteArray("x264.exe");
    //Comprime el ejecutable
    b = BinaryManipulator.compressByteArray(b);
    CondorMethods selfDep = getselfdependency();
    CondorPackedOutput result;
    //Setea las propiedades de ejecución para Condor
    Hashtable<String, String> h = new Hashtable<String, String>();
    h.put("universe", "vanilla");
    if (selfDep != null) {
        //Ejecución y captura de resultados
        result = (CondorPackedOutput) selfDep.doRun(b, h);
        System.out.println(result.getDescription());
    }
}
```

En este ejemplo, se carga el ejecutable “x264.exe” y se setea la propiedad “universe” con el valor “vanilla”.

El resultado de la ejecución produce un CondorPackedOutput. En este caso se obtiene su descripción y se la imprime por la salida standard.

Vale destacar que Condor, de estar bien configurado, puede correr class Java con el universo “java”, u obj de C, entre tantas opciones.

En el ejemplo anterior se utilizo el XML por defecto, sin cambios, corriendo el método “main” provisto por AbstractMFGS, tambien sin modificaciones.

## CONCLUSION

Se logró el objetivo de incluir Condor en la plataforma JGRIM, manejado de forma sencilla y sin romper los esquemas ya sentados por JGRIM. El usuario puede, por medio de la implementación de un único método, invocar distintos ejecutables de distinto origen sobre distintas configuraciones particulares de Condor.

Existen puntos que probablemente debieran ser modificados para realizar de manera mas genérica la inclusión de otras plataformas. Un ejemplo claro de esto es la clase NetworkConfigurator, que pese a funcionar correctamente, está muy atada a Ibis y Condor.