Implementación de un Servicio Grid para el Monitoreo de Recursos Computacionales

Martín Chuburu Javier Echaiz Jorge Ardenghi

Laboratorio de Investigación de Sistemas Distribuidos (LISiDi)
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur, Bahía Blanca (8000), Argentina
{mic,je,jrap}@cs.uns.edu.ar

Resumen

La computación grid permite administrar los recursos que se encuentran diseminados en una topología distribuida formada por nodos interconectados mediante redes de área local e Internet, con el fin de asistir a procesos que no disponen de los recursos necesarios para llevar a cabo su tarea en las máquinas locales donde fueron invocados.

Para poder hacer un uso eficiente de estos recursos es necesario tener información sobre el nivel de utilización de los mismos en cada una de las máquinas que conforman el grid, con el objetivo de poder tomar decisiones sobre la migración de los procesos que se están ejecutando en el mismo.

Contar con este tipo de información también permite la búsqueda de comportamientos anómalos como por ejemplo carga excesiva o falla en los servicios críticos. Para ello, la utilización de Servicios Grid constituye una posible herramienta para poder obtener este tipo de información y elaborar, a partir de estos servicios, una jerarquía dentro de la infraestructura grid a medida que ésta vaya creciendo.

Palabras Clave: Monitoreo de Recursos, Globus Toolkit, Servicios Grid.

1. Introducción

Un *Servicio Grid* se define como una interfaz asociada a un recurso Grid. Entonces, en un entorno Grid, un recurso y el estado asociado a éste serán administrados a través del Servicio Grid [1].

Los recursos grid pueden requerir interactuar unos con otros, además es muy probable que estos recursos se encuentren en un entorno tecnológicamente heterogéneo. Por eso es necesario un marco de trabajo que permita abstraer el servicio de mensajes entre Servicios Grid y separarlo de los detalles de implementación del entorno específico. Una *Arquitectura Orientada a Servicios* (SOA) provee tal marco de trabajo.

El *Global Grid Forum* (GGF) ha adoptado *Open Grid Service Architecture* (OGSA) basada en los principios SOA para el modelamiento de los recursos del grid, a través de Servicios Grid. Dichos servicios están construidos en base a la tecnología de Servicios Web.

La diferencia con estos últimos radica en la permanencia de los datos. En un Servicio Web no se mantiene un registro de estado entre llamadas subsecuentes al servicio. Mientras

que con Servicios Grid, a menudo es importante mantener esta información. Por lo tanto, fue necesario implementar un mecanismo complementario de forma de poder utilizar la tecnología de Servicios Web y a la vez mantener información entre un cliente y el servicio, entre una llamada y la siguiente.

Actualmente existen dos estándares disponibles para implementar Servicios Grid que cumplen con los requerimientos OGSA.

- Open Grid Services Interface (OGSI).
- Web Service Resource Framework (WSRF).

Globus Toolkit, a partir de su versión 4 (GT4), comienza a utilizar WSRF debido a que es una solución que, además de cumplir con los requerimientos de los Servicios Grid, se mantiene fiel a los fundamentos de Servicios Web.

El punto principal de discusión es la divergencia entre las especificaciones OGSI y las prácticas populares en la comunidad de Servicios Web. Algunos de los cuestionamientos hechos por esta comunidad están relacionados con la sintaxis XML utilizada por OGSI que causa conflictos con APIs estándares como JAX-RPC, con la extremada orientación a objetos en el modelo de recursos con estado que atenta contra el modelo de Servicio Web (que es "sin estado"), y con la falta de soporte para algunas versiones de WSDL. El principal objetivo en la reestructuración hacia WSRF es mantener las comunidades de Servicios Grid y Servicios Web unidas.

A diferencia de OGSI, WSRF hace una distinción explícita entre "servicio" y "recursos con estado" actuando bajo ese servicio. WSRF define los medios por los cuales un Servicio Web y un recurso con estado se componen. WSRF denomina a esta composición *WS-Resource*.

De acuerdo con WSRF, un recurso con estado tiene los datos que representan al mismo descriptos en un documento XML que es conocido y accedido por uno o más Servicios Web.

Es importante notar que para los clientes, el servicio y los recursos son vistos como una misma cosa a través del archivo WSDL que constituye la interfaz del mismo. Dichos clientes nunca tratarán directamente con instancias de los recursos sino que lo harán implícitamente a través de las interacciones con el servicio que cumple con la especificación WSRF.

La implementación de WSRF implícitamente pasa la información de identificación del recurso cuando ocurre una interacción de mensajes entre un cliente y un *WS-Resource*. El cliente no tiene que incluir explícitamente un identificador de recursos en la solicitud. En su lugar, el identificador requerido está implícitamente asociado a un intercambio de mensajes.

WS-Addressing estandariza la forma de representación de las direcciones de los Servicios Web. Tal representación es conocida como Endpoint Reference (EPR). Un EPR puede contener, además de la dirección de referencia al Servicio Web, otros metadatos asociados con el Servicio Web. Para el caso de un EPR que referencia a un *WS-Resource*, éste puede incluir un elemento que define el recurso con estado a ser utilizado en la ejecución de todos los intercambios de mensajes realizados utilizando este EPR. Este tipo de EPR se denomina *WS-Resource-qualified endpoint reference*.

En la sección 2 se listarán los pasos involucrados en la construcción de un Servicio Grid que sigue el estándar WSRF. A continuación, en la sección 3, se detallará el desarrollo de un modelo basado en Servicios Grid, que fue implementado para monitorear los recursos computacionales de un cluster de máquinas. En la sección 4 se mencionarán cuestiones relativas a la construcción de clientes que soliciten los servicios del modelo implementado. Y por último, en la sección 5 se expondrán las conclusiones del trabajo realizado.

2. Construcción de un Servicio Web WSRF

El WSRF introduce la noción de WS-Resource como base para la construcción de Servicios Grid.

Cuando un WS-Resource es empaquetado como un *Grid ARchive* (GAR) y desplegado en el *container* de GT4, es reconocido por el mismo como un Servicio Web válido que sigue WSRF. Esto es sinónimo de *Servicio Grid*.

Desde el punto de vista del desarrollador, los pasos involucrados para implementar un Servicio Web WSRF para desplegarlo dentro de un *container* de GT4, son los siguientes:

1. **Definir la interfaz del servicio.** Significa preparar el archivo WSDL que define las operaciones del servicio WSRF, y puede incluir definiciones de las propiedades de recursos.

2. **Implementar el servicio.** Se refiere a desarrollar el código fuente para las operaciones del servicio WSRF y propiedades asociadas si las hubiera. En el caso particular del *VisorService*, se eligió el lenguaje de programación *Java* para la implementación del servicio, aunque *Globus* también cuenta con soporte para los lenguajes *C* y *Python*.

3. **Definir los parámetros de despliegue.** Se refiere a preparar un archivo *Web Service Deployment Descriptor* (WSDD) para el servicio WSRF, que define varios aspectos de la configuración del mismo.

4. **Compilar y generar el archivo GAR.** La compilación y creación del archivo GAR involucra la creación de los archivos con los *stubs* apropiados para manejar mensajería SOAP y empaquetar el servicio en un formato requerido por el *container* de GT4.

5. **Desplegar el servicio.** Implica descomprimir el contenido del archivo GAR en una ubicación preestablecida dentro de la estructura de directorios de GT4 de forma que el servicio esté disponible para quien quiera invocarlo.

3. Un Servicio Grid como cliente de otro Servicio Grid

En el resumen de este trabajo se expuso a los Servicios Grid como una herramienta para implementar un control sobre los recursos de un Grid. Con el fin de tener una primera aproximación a esta herramienta, se implementó un modelo basado en Servicios Grid que permita mostrar información de estado de cada uno de los nodos que componen un cluster. Esta prueba fué realizada sobre uno de los cluster pertenecientes al Laboratorio de Investigación en Sistemas Distribuidos (LISiDi) de la Universidad Nacional del Sur (UNS).

Este cluster está formado por nueve computadoras Pentium IV de 3 GHz, con 512 Mb de memoria RAM, discos de 80 Gb y placas de red de 1 Gbps cada una, todas conectadas a un switch. El motivo por el cual se requiere una alta velocidad de comunicación radica en la necesidad de crear la ilusión de una única máquina n veces más potente que una PC común formada por n PCs comunes. Sobre ellas se instaló Globus Toolkit 4 (GT4), que provee el *middleware* necesario para implementar un Grid sobre un cluster de máquinas.

En cuanto al modelo implementado, *VisorService*, se tomó la decisión de dividirlo en dos: un servicio *slave* que se ejecute en cada nodo del cluster y que devuelva, bajo solicitud, la información de estado de la máquina en la que se encuentra; y un servicio *master* que se encuentra alojado solo en la máquina cabecera del cluster y que se encarga de solicitar información de

estado a los servicios slave en cada una de las máquinas con el fin de juntarla en una sola estructura de datos y entregarla a quien solicite el servicio master.

Esto se hizo con el objetivo de ver como dos o más Servicios Grid pueden interactuar entre sí, concientes de que tal vez no es la opción más eficiente debido a que estas solicitudes tienen que atravesar una capa de software correspondiente a GT4 en más de una ocasión. En futuras implementaciones, se reestructurará el servicio de forma que persista el *master* y la información necesaria de todos los nodos sea obtenida mediante una tecnología de más bajo nivel como pueden ser los *sockets* o algún estándar de computación distribuida como MPI.

3.1. Estructura de directorios del servicio

Los archivos que conforman un Servicio Grid, y que han sido mencionados en los pasos de construcción del mismo, deben encontrarse en localizaciones determinadas dentro de una estructura de directorios cuya raíz corresponde al directorio de trabajo del Servicio Grid (el directorio en el que se van guardando los archivos hasta el momento de compilarlos).

Así que como paso "cero" al conjunto de pasos a seguir mencionados en la sección anterior, se establece la jerarquía de directorios necesaria. Dentro de la misma jerarquía pueden estar tanto los archivos pertenecientes al servicio slave como al servicio master.

En este caso, el directorio de trabajo va a ser *VisorService*. Dentro de éste, se va a establecer la siguiente estructura de directorios:

- **schema**: Este directorio debe estar obligatoriamente. Contendrá los archivos de interfaz WSDL necesarios para poder exportar las operaciones de los servicios presentados.
  - *VisorSlaveService*: este subdirectorio de *schema* contendrá el WSDL del servicio slave de VisorService.
  - *VisorMasterService*: de igual manera, contendrá el WSDL correspondiente al servicio master de VisorService.
- **grid/cs/uns/visor**: Estos cuatro directorios se mapearán al nombre de *package* de Java grid.cs.uns.visor. En el directorio *visor*, entonces, se encontrarán las carpetas que contendrán los archivos Java de los servicios, y también la implementación de unos clientes en modo de terminal de texto que serán usados para probar el servicio.
  - *clients*: Esta carpeta, como lo indica su nombre, tendrá los clientes necesarios para probar los servicios. Al igual que la carpeta *schema*, cuenta con los subdirectorios *VisorSlaveService* y *VisorMasterService* para separar los clientes correspondientes a uno y otro servicio, respectivamente.
  - *services*: Como también se puede deducir, este directorio contendrá los archivos de implementación de los servicios.
    - **master** | **slave**: Estos dos subdirectorios tienen una estructura idéntica. Contienen los archivos de despliegue de servicio y un subdirectorio *impl* que contiene los archivos Java que implementan los respectivos servicios.

3.2. Servicio *slave* del VisorService

En esta sección se mostrará cuestiones relativas al cumplimiento de cada uno de los pasos involucrados en el desarrollo de un Servicio Grid. En la subsección 3.3 se mencionarán las

diferencias que se presentan en el desarrollo del servicio master con respecto al del servicio slave.

3.2.1. Definición de la interfaz del servicio

Como primer paso a la construcción del servicio slave, se define la interfaz del servicio: principalmente, qué operaciones va a brindar el servicio sin entrar en cuestiones de implementación y cuáles son los atributos de datos que constituyen su "estado de recurso". Para esto se hace uso de un lenguaje XML especial que puede ser utilizado para especificar qué operaciones ofrece un Servicio Web: el *Web Service Description Language* (WSDL).

Este lenguaje permite abstraerse en esta etapa del lenguaje que se utilizará para implementar las operaciones del servicio.

En el caso particular del *VisorService*, no sería necesario mantener el estado del recurso, debido a que la información que se desea mostrar corresponde a un instante cualquiera y no tiene relación con llamadas anteriores. Sin embargo, dado que es una primera aproximación a la implementación de un Servicio Web, se dividió la tarea de mostrar la información de un nodo en dos operaciones:

- **ejecutar**: que se encarga de recolectar los datos y asignarlos a las propiedades de recursos (atributos que constituyen el estado del recurso).
- **mostrar**: que toma las propiedades de recursos y crea una estructura de datos apropiada para devolver estos datos a quien solicite el servicio.

La estructura de datos que devuelve la operación mostrar (*mostrarResponse*) tiene como atributos los mismos que se utiliza como propiedades de recurso:

- **HostID**: El nombre del nodo (*hostname*) al cual pertenece la información.
- **CpuMHz**: Velocidad del procesador del nodo.
- **CpuUse**: Porcentaje actual de utilización del CPU.
- **MemTotal**: Memoria RAM disponible del nodo.
- **MemPerc**: Porcentaje actual de utilización de memoria sobre la cantidad de memoria RAM disponible (es decir, más de un 100 % indicaría que se está utilizando memoria virtual).

3.2.2. Implementación del servicio

En este punto del desarrollo del servicio, se implementó en Java el código que obtiene los datos de estado del sistema. Java, al ser un lenguaje basado en máquina virtual, no cuenta con operaciones directas que permitan ejecutar comandos del *shell* como podría ser la función `system` del lenguaje C.

Sin embargo, se puede simular a través de las siguientes líneas de código:

```
String cmdString = "..."; // el comando que quiero ejecutar
try
{
    Process p = Runtime.getRuntime().exec(cmdString);
```

```
    BufferedReader pOutput =
    new BufferedReader(
    new InputStreamReader(p.getInputStream()));

    String line;
    StringBuffer tmpCommandOutput = new StringBuffer();

    try
    {
        while ((line = pOutput.readLine()) != null)
            tmpCommandOutput.append(line).append("\n");
        salida = tmpCommandOutput.toString();
    }
    catch (IOException e) {}

    p.waitFor();
    pOutput.close();
}
catch (IOException e) {}
catch (InterruptedException e) {}
```

En esencia, lo que hace este bloque de instrucciones es, en primer lugar, crear un objeto *Process* que representa la ejecución de un comando de *shell* contenido en el string *cmdString*. Luego se crea un buffer a través del cual se va a recibir lo que el comando vuelca en la salida estándar (que visto desde la óptica del programador sería un buffer de entrada porque se quiere leer esta información). Por último, el *try-catch* interno, a través de un bucle, va obteniendo línea por línea la salida del comando desde el buffer y lo va anexando a un *StringBuffer* que, finalmente, se convierte y devuelve como un string (*salida*).

Para ejecutar un comando que provea de la información necesaria, se tuvo que elaborar un *script* con comandos de GNU/Linux que obtuvieran los datos necesarios y filtraran la información de forma que la salida de la ejecución del script contuviera en cada línea un dato a ser asignado directamente a una propiedad de recurso o a ser utilizado para calcular una propiedad de recurso. El *script* debe encontrarse en todas las máquinas que vayan a correr el servicio en el directorio `/usr/bin` de nuestro sistema GNU/Linux.

La implementación del servicio en Java tiene que estar en el subdirectorio grid/cs/uns/visor/services/slave/impl del directorio de trabajo.

3.2.3. Definición de los parámetros de despliegue

Son dos los archivos que deben encontrarse en el subdirectorio grid/cs/uns/visor/slave (el padre del directorio donde se encuentran los archivos Java) para poder desplegar el servicio en el *container* de GT4: `deploy-jndi-config.xml` y `deploy-server.wsdd`.

En el primero, lo que se puede observar es que en la siguiente etiqueta se define el nombre del servicio:

```
<service name="visor/slave/VisorService">
```

es decir, la URI del servicio que va a aparecer al ejecutar el *container* es `http://nombre-dominio-o-IP:8080/wsrf/services/` seguido de lo especificado en la propiedad `name` de la etiqueta `service`. En este caso, sería:

```
http://grid.cs.uns.edu.ar:8080/wsrf/services/visor/slave/VisorService
```

En cuanto al segundo archivo, la siguiente etiqueta produce la asociación entre el servicio y la clase Java que implementa el mismo:

```
<parameter name="className"
value="grid.cs.uns.visor.services.slave.impl.VisorService"/>
```

Otra etiqueta de importancia, es la que define el archivo WSDL que debe utilizarse. Esto es relevante para cuando estamos trabajando con clientes en otras plataformas, ya que es necesario pasarles una URL que devuelva el WSDL con el que podrán crear sus clases o tipos de datos con los que se accederá al servicio.

El WSDL que se utilizará en última instancia no será el que fue editado al principio porque este último contiene órdenes a un preprocesador de WSDL (wsdlpp) que permitirá agregar al WSDL final las líneas necesarias para usar operaciones propias de *Globus* para el establecimiento de referencias al servicio y de la implementación de operaciones comunes a los Servicios Grid desarrollados con GT4 como *GetResourceProperties*.

Es por ello que es necesario indicar el documento que será utilizado para elaborar una respuesta ante la solicitud del archivo WSDL por parte del cliente. Esto se consigue a través de la siguiente etiqueta:

```
<wsdlFile>share/schema/VisorSlaveService/Visor_service.wsdl</wsdlFile>
```

donde `Visor_service.wsdl` es el documento que generará el preprocesador de WSDL (Visor.wsdl es el archivo WSDL que se estableció en el primer paso).

3.2.4. Compilación y generación del archivo GAR

Para simplificar el proceso de compilación se utiliza *globus-build-service*. Esta herramienta, que está incluida como parte del proyecto Globus Service Build Tools (GSBT) [2] de SourceForge, está formada por un archivo *buildfile* de Ant de propósito general y un *script* que dado un conjunto de archivos Java, WSDL, WSDD que respetan una estructura de directorios específica, generará un archivo GAR sin la necesidad de editar manualmente el archivo Ant. Este es el mismo Ant *buildfile* y *script* incluido en el *Globus Toolkit 4 Programmer's Tutorial* [3] en el cual nos basamos para desarrollar este servicio.

Este par de archivos se copian al directorio de trabajo del servicio (*VisorService*) que se denomina como `$BUILD_DIR`. Para que el *buildfile* de Ant de propósito general funcione para el servicio en cuestión es necesario especificar dos parámetros al ejecutar el *script*:

- `SERVICE_DIR` que es el directorio que contiene todos los archivos de implementación y despliegue. En el caso del servicio slave el directorio sería
  grid/cs/uns/visor/services/slave
- `SCHEMA_FILE` que es el archivo WSDL con la descripción de la interfaz del servicio. En este caso, el archivo sería
  schema/VisorSlaveService/Visor.wsdl

Es importante observar que tanto `SCHEMA_FILE` como `SERVICE_DIR` son relativos a `$BUILD_DIR`

El archivo GAR

El archivo GAR es generado en `$BUILD_DIR/GAR_ID.gar`. El `GAR_ID` es generado a partir del parámetro `SERVICE_DIR` reemplazando los separadores de path (/ en UNIX o GNU/Linux) con guiones bajos. Por ejemplo, si el parámetro es:

```
grid/cs/uns/visor/services/slave
```

Entonces, el archivo GAR se generará en:

```
$BUILD_DIR/grid_cs_uns_visor_services_slave.gar
```

El directorio `build`

Todos los archivos intermedios generados por el *script* son ubicados en `$BUILD_DIR/build`. A veces, éste produce resultados inesperados si se realizan muchas compilaciones (el directorio build queda lleno de archivos intermedios de compilaciones previas). Cada vez que se obtenga un error inesperado, la primera cosa que se debería intentar es borrar el directorio build, que asegurará que el build script comience desde cero.

Aunque generalmente no es necesario mirar los contenidos del directorio build, a menudo se quiere verificar si los archivos de *stub* (que permiten la serialización y deserialización de los datos) fueron generados correctamente. Los archivos fuentes de los *stubs* son generados en:

```
$BUILD_DIR/build/stubs-GAR_ID/src
```

Y los *stubs* compilados se ubican en:

```
$BUILD_DIR/build/stubs-GAR_ID/classes
```

3.2.5. El despliegue del servicio

Una vez listo el archivo GAR, se está en condiciones de desplegar el servicio con el comando

```
globus-deploy-gar grid_cs_uns_visor_services_slave.gar
```

Hay tener en cuenta que se necesita especificar el `GAR_ID` (no el nombre de archivo GAR) cuando se hace el *undeploy* (eliminar el servicio de la lista de servicios disponibles) de los archivos GAR usando el comando *globus-undeploy-gar*. Por ejemplo:

```
globus-undeploy-gar grid_cs_uns_visor_services_slave
```

3.3. Servicio *master* del VisorService

Como se había mencionado en la sección 3, el servicio master corre en el nodo cabecera y es el que se va a encargar de solicitar el servicio slave en cada nodo del cluster.

Los pasos para la construcción del mismo son iguales que para el servicio slave, a excepción de que solo contará con una única operación llamada *juntar* que se encarga de solicitar a cada slave que ejecute su servicio y luego recolecta la información que devuelven.

Para lograr la conexión con los servicios slave de cada máquina, el servicio master debe crear una conexión con cada máquina. Para ello es necesario crear una referencia al servicio slave en cada máquina y con esa referencia solicitar las operaciones *ejecutar* y *mostrar*. El código en Java que realiza esta tarea es el siguiente:

```
VisorServiceAddressingLocator locator = new VisorServiceAddressingLocator();
EndpointReferenceType endpoint;

endpoint.setAddress(new Address("http://" + hostID +
":8080/wsrf/services/visor/slave/VisorService"));

// Obtengo el PortType
VisorPortType viz = locator.getVisorPortTypePort(endpoint);

// Llamo a ejecutar
viz.ejecutar(new Ejecutar());
// Ejecutar representa un parametro nulo (necesario para XML Schema)

// Accedo al resultado y lo asigno al recurso correspondiente
MostrarResponse buffer = viz.mostrar(new Mostrar());
// Mostrar tambien representa un parametro nulo para esta operacion.
```

La información devuelta por *juntar* se encuentra en un formato que está definido en el WSDL del servicio slave. Este tipo de objeto devuelto incluye un campo *hostID* que contendrá un *string* para identificar de que máquina proviene (es un nombre de host) o tendrá una cadena `<desconocido>` para indicar que la operación no se concretó debido a que hubo una excepción producida en la conexión (generalmente significa que la máquina está apagada, el servicio no está corriendo o el *container* no está ejecutándose).

Como se mencionaba anteriormente, el formato de este objeto (llamado *MostrarResponse*) se encuentra en el WSDL del slave. Debido a que la información que mantendrá el master es un arreglo con tantas de estas componentes como máquinas, existen dos alternativas para poder usar este formato:

1. Incluir la definición del WSDL del slave en el WSDL del master.
2. Crear un formato exactamente igual (el cual se llamará *status*) y agregar una operación privada que convierta del formato *mostrarResponse* al formato *status*. Dado que esto termina programándose en Java y se puede importar los *stubs* del slave, es posible invocar sus métodos *get* sobre cada una de las propiedades de recurso (coincidentes con los atributos de *mostrarResponse*) para obtener los valores necesarios para utilizar los métodos *set* incluidos en el formato *status*.

La alternativa que se tomó fue la segunda, por comodidad y para no adulterar el archivo WSDL del master con cosas que se podían solucionar desde el lenguaje de programación. Una mejora al servicio sería hacerlo con la primera alternativa, teniendo la ventaja (posiblemente) de que sea más independiente del lenguaje.

4. Implementación de clientes que utilicen el servicio

Una vez que el servicio fué compilado con *globus-build-service*, que fué desplegado en el *container* de GT4 con *globus-deploy-gar* y que la URI del servicio aparece en la lista de servicios al ejecutar *globus-start-container*, está asegurado que el servicio es *sintácticamente* correcto. Ahora se necesita verificar que el servicio realiza la tarea para la cual fué programado. Para

ello es necesario que una aplicación cliente establezca contacto con el servicio y solicite sus operaciones.

Para implementar un cliente de este tipo, se podría hacer un cliente en Java que establezca contacto con el servicio master corriendo en la cabecera del cluster, usando líneas de código similares a las expuestas en la subsección 3.3 cuando se quiso establecer contacto con los servicios slave en cada nodo del cluster.

Sin embargo, el cliente está accediendo desde la misma plataforma de sistema operativo y lenguaje de programación; es decir, el cliente se encuentra en un ambiente homogéneo con el servicio. Incluso está utilizando los *stubs* compilados por el mismo servicio para comunicarse con él. Esta situación no va a ser siempre así, ya que clientes corriendo en otros sistemas operativos, programados en otros lenguajes podrían querer acceder al Servicio Grid (de esto se trata la heterogeneidad, existente en ambientes Grid). Por eso, en la subsección siguiente veremos las cuestiones a considerar para este caso.

4.1. Un cliente creado en otra plataforma

La motivación principal de elaborar el *VisorService* fue la de desarrollar un servicio de información para un sistema de visualización utilizado por el Laboratorio de Investigación en Visualización y Computación Gráfica (VyGLab) que desarrolla sus actividades en el mismo departamento académico que el LISiDi.

El sistema de visualización utiliza la información provista por el *VisorService* para mostrarla de una forma gráfica, intuitiva y fácil de comprender para el ser humano; de manera de poder apreciar de forma rápida y sencilla el balance de carga en un sistema distribuido [4].

Este proyecto en conjunto entre estos dos grupos se viene gestando desde hace tiempo [5, 6] y en *Servicios Grid* se encontró una herramienta para integrar el trabajo de ambos.

La plataforma que utiliza el sistema de visualización utilizado por el VyGLab consta de máquinas con sistema operativo MS Windows XP y Visual Studio .NET como herramienta de programación. Lo cual demuestra la heterogeneidad que se puede presentar al trabajar en un entorno grid. Las clases de *stub* que necesita utilizar el cliente ya no sirven, sino que la plataforma MS Windows/.NET debe crear sus propios *stubs* a partir de la información provista por el archivo WSDL correspondiente al servicio y con ellos realizar las llamadas a los métodos del servicio.

En primer lugar fué necesario cambiar la configuración del *container* de Globus, para que figure *grid.cs.uns.edu.ar* en las URI de los servicios en lugar de la dirección IP correspondiente al nodo cabecera. Para ello, en el siguiente archivo de confugaración

```
$GLOBUS_LOCATION/etc/globus_wsrf_core/server-config.wsdd
```

se debe modificar el valor de `logicalHost` en la siguiente etiqueta:

```
<parameter name="logicalHost" value="grid.cs.uns.edu.ar" />
```

El valor de `logicalHost` puede ser tanto un nombre de host como una dirección de IP. En este caso el nombre *grid.cs.uns.edu.ar* corresponde al host que funciona de cabecera del cluster.

El segundo inconveniente encontrado fué que en el cliente de modo texto sólo se necesitaba la URI del servicio que figuraba en la lista de servicios desplegados en el *container* para poder conectarse al servicio. O sea,

```
http://grid.cs.uns.edu.ar:8080/wsrf/services/visor/master/VisorService
```

Sin embargo, esta URI no era válida para *Visual Studio .NET*, sino que necesitaba la WSDL del servicio. De hecho, al tratar de ingresar esa URI a un *web browser*, emitía el mensaje "File Not Found", a pesar de que el servidor se encontraba escuchando en ese IP y puerto.

El problema es, justamente, que *Visual Studio .NET* espera un descriptor de Servicio Web porque ve al Servicio Grid como tal. Por ende, es necesario agregarle un parámetro a la URI provista de forma que devuelva el archivo WDSL correspondiente. Este parámetro es "wsdl":

```
http://grid.cs.uns.edu.ar:8080/wsrf/services/visor/master/VisorService?wsdl
```

Esta URI se probó sobre un pequeño cliente en modo texto programado en .NET y funcionó de forma satisfactoria, permitiendo crear una referencia a nuestro servicio y realizar subsiguientes llamadas a los métodos del mismo. Sobre esta base, se comenzó el desarrollo, a cargo del VyGLab, de un cliente que utilice como base el sistema de visualización mencionado anteriormente y como herramienta de programación a *Visual Studio .NET*.

5. Conclusiones y trabajos futuros

A través de esta experiencia programando y testeando Servicios Grid, se observa que es una herramienta adecuada (aunque no la más eficiente) para modelar un sistema para el monitoreo de los recursos computacionales en un cluster de computadoras, como prueba a escala de lo que conformaría una infraestructura grid. Alternativamente, se podría utilizar tecnologías pertenecientes al área de *clustering* como pueden ser MOSIX[7], Condor[8], MonALISA[9] que son sistemas que, entre sus funciones, está la de monitorear los recursos en un cluster, y utilizar estos sistemas como proveedores de información para MDS que el sistema de monitoreo provisto por Globus Toolkit 4 y que también está basado en Servicios Grid. Sobre esta alternativa se investigará en el futuro.

Sin embargo, *globus-build-service* constituye una herramienta realmente útil a la hora de generar archivos GAR a partir de un conjunto de archivos Java, WSDL y WSDD que implementan un Servicio Grid sencillo, evitando el trabajo tedioso de aprender a manipular el formato de *buildfile* de Ant y de crear un *builfile* distinto para cada Servicio Grid que se implemente. Sin embargo, su aplicación esta limitada a ejemplos simples y requiere de la colaboración de desarrolladores que puedan extender las funcionalidades de esta herramienta de forma de convertirla en útil para cualquier proyecto que se desee compilar.

Su uso para la compilación del servicio resultó de mucha utilidad y permitió observar los errores que se pueden llegar a presentar ante una mala manipulación de los archivos; además de permitir concentrarse en la implementación del servicio dándole menos relevancia a cuestiones de configuración.

A pesar de ello, existen herramientas que permiten abstraerse aún más de las cuestiones técnicas de los Servicios Grid y que permiten enfocarse en la implementación del servicio, como pueden ser los *plugins* para entornos integrados de desarrollo (IDEs) para programar en Java. Esta opción fue abordada en una primera instancia, sin embargo se abandonó debido a que se presentaron problemas al tratar de compilar y se decidió abordar otra herramienta para programar Servicios Grid.

Sin embargo, las experiencias con *globus-build-service*, al ser una herramienta de más bajo nivel, proveyeron de conocimientos útiles para poder volver a analizar esta alternativa en un futuro, de forma de poder llevar a cabo la tarea de programar un Servicio Grid de forma más automatizada.

Referencias

[1] Miriam Lechner and Martin Chuburu. *Computación Grid, Globus Toolkit y potencia computacional sin límites*, 2006.

[2] *Globus Service Build Tools*. http://gsbt.sourceforge.net/.

[3] Borja Sotomayor. *The Globus Toolkit 4 Programmer's Tutorial*, 2005. http://gdp.globus.org/gt4-tutorial.

[4] Martin Chuburu, Javier Echaiz, and Jorge Ardenghi. Monitoreo de Recursos computacionales en un cluster utilizando Grid Services. *IX Workshop de Investigadores en Ciencias de la Computación, WICC 2007*, pages 612–616, May 2007.

[5] Martín Larrea, Sergio Martig, Silvia Castro, and Javier Echaiz. Visualización del Balance de Carga en un Sistema Distribuido. *11mo Congreso Argentino de Ciencias de la Computación (CACIC 2005)*, pages 1761–1771, October 2005.

[6] Martín Larrea, Sergio Martig, Silvia Castro, and Javier Echaiz. A proposal from the point of view of Information Visualization and Human Computer Interaction for the visualization of distributed system load. *Special Issue on Selected Papers from CACIC 2005, JCS&T*, pages 327–333, December 2005.

[7] Najib A. Kofahi, Saeed Al Zahrani, and Syed Manzoor Hussain. MOSIX evaluation on a linux cluster. *Int. Arab J. Inf. Technol*, 3(1):62–68, 2006.

[8] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[9] Harvey B. Newman, I. C. Legrand, Philippe Galvez, R. Voicu, and C. Cirstoiu. MonALISA: A distributed monitoring service architecture. *CoRR*, cs.DC/0306096, 2003.

[10] Martin Chuburu, Miriam Lechner, Javier Echaiz, and Jorge Ardenghi. Experiencias con Globus Toolkit. *VIII Workshop de Investigadores en Ciencias de la Computación, WICC 2006*, pages 217–220, June 2006.

[11] Bart Jacob, Michael Brown, Kentaro Fukui, and Nihar Trivedi. IBM redbook: Introduction to grid computing, 2005.

[12] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] Ian Foster and Carl Kesselman. *The Grid - Blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers, 1999.

[14] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–??, 2001.