UNIVERSIDAD CARLOS III DE MADRID

Escuela Politécnica Superior - Leganés

INGENIERÍA DE TELECOMUNICACIÓN

Proyecto Fin de Carrera

Herramienta de Simulación Remota en un *Cluster* de Computación Científica.

AUTOR: Adrián Amor Martín.
DIRECTOR: Ignacio Martínez Fernández.
TUTOR: Luis Emilio García Castillo.

Leganés, 2012

Título:	*Herramienta de Simulación Remota en un Cluster de Computación Científica.*
Autor:	Adrián Amor Martín.
Director:	Ignacio Martínez Fernández.
Tutor:	Luis E. García Castillo.

La defensa del presente Proyecto Fin de Carrera se realizó el día 26 de Julio de 2012, siendo calificada por el siguiente tribunal:

Presidente:	Sergio Llorente Romano.
Secretario:	Alejandro García Lampérez.
Vocal:	Pedro J. Muñoz Merino.

Habiendo obtenido la siguiente calificación:

Calificación:

Presidente	**Secretario**	**Vocal**

Suele decirse, "Demos tiempo al tiempo", pero lo que siempre nos olvidamos de preguntar es si quedará tiempo para dar.
José Saramago.

Y esperé tanto tiempo que di con lo inesperado.
Me dejé arrastrar por mareas y corrientes y ahora sé,
sé muy bien, que la noche es sólo para los que sueñan
y entregan su mirada al cielo y a las estrellas.

Quizás, si bajo las persianas, la noche se haga eterna...

Agradecimientos

En primer lugar, quisiera agradecer la elaboración de este proyecto a mi director por toda la ayuda prestada y la gran dedicación que ha empleado: ha conseguido que todo sea mucho más fácil. También quiero dar las gracias a mi tutor, por ofrecerme este proyecto junto con mi director y por la paciencia que ha mostrado conmigo a lo largo de todo el año.

También he de dar las gracias a mis amigos de Toledo, que se han convertido en mi vía de escape de la universidad en todos estos años, y en especial a Samu y a Bachi, por esas tardes de pro, a Carlota, por estar casi siempre ahí, a Álex, por saber escucharme a veces, a Roberto, a Ester, a Sergio, a Carlos, por pesado, y a David, aunque no nos haya metido en sus agradecimientos.

Y a pesar de que no quiera estar aquí, a Mariuca por la elaboración del logo de la aplicación y por convertirse poco a poco en una persona importante.

No puedo olvidarme tampoco de mis compañeros de universidad con los que he podido compartir estos duros años de clases y trabajos, especialmente a Paloma, a Víctor, a David, a Javi, a Rodrigo, a Álex, por ser un gran compañero de prácticas, a Lara, por no dejarse llevar por las primeras impresiones, a Cris, por ayudarme a sobrellevar el día a día de estos últimos años y tener el increíble aguante de soportarme muchas horas seguidas, que no es fácil, y finalmente, a Leti, por ser la única capaz de ponerme siempre una sonrisa aun en los momentos más difíciles y una amiga que jamás pude imaginar.

Por último, me gustaría dar las gracias a mi familia, particularmente a mis abuelos Laura y Ramón por soportar la instalación de un parásito en su casa durante los últimos cinco años y, sobre todo, por aguantar hasta los días de mayor agobio que han sido muchos, y a mis padres por hacer lo mismo, pero durante bastante más tiempo, lo que tiene mucho más mérito. Aunque no os lo diga a menudo, os quiero a todos.

Resumen

Debido a la aparición de problemas cada vez más complejos y pesados computacionalmente hablando, resulta casi indispensable recurrir al uso de *clusters* de altas prestaciones, o HPCC, para reducir sensiblemente el tiempo de ejecución o ser capaces de abordar el inmenso tamaño de éstos. Sin embargo, para la mayoría de usuarios el acceso a estos HPCC es tedioso e, incluso, complicado.

Para superar esta barrera de entrada, en este Proyecto Fin de Carrera se describe una herramienta segura, sencilla y fácil de usar para proporcionar el acceso a los servicios de uno de estos *clusters*. De esta forma, se implementa un protocolo específico de comunicaciones con el cual se ofrecen funcionalidades como la ejecución remota de problemas en un HPCC o la transferencia automática de los ficheros de entrada y salida del trabajo ejecutado.

Asimismo, se ha desarrollado una interfaz gráfica basada en Java para facilitar el control de estas funcionalidades por el usuario desde un equipo de sobremesa y, además, se ha implementado una aplicación en Android que mantiene los mismos servicios que la interfaz de sobremesa en un dispositivo móvil, dotando así de movilidad a la aplicación desarrollada.

Palabras clave: *user-friendly*, seguridad, movilidad, *cloud computing*, SGE, DRMAA, Java, protocolo, ejecución remota, HPCC, supercomputación.

Abstract

As a result of the existence of problems more and more complex and computationally expensive, it is a must to use of high performance computational clusters, or HPCC, in order so as to appreciably reduce the time of execution and be capable of solve such problems. However, for a novice user the access to one of this resources can be difficult and even unpleasant.

This Final Project is developed to solve this barrier to entry. A simple, secure and user-friendly tool to provide the access to the services of one of these clusters is described. Thus, a specific communication protocol is implemented, which provides features as the remote execution of problems in a HPCC or the automatic file transfer of the input and output files of the executed job.

Likewise, a GUI (Graphical User Interface) based on Java has been developed to simplify these features in a desktop computer or laptop. And also, an Android application has been implemented in order to increase the mobility keeping all features from the original GUI.

Keywords: *user-friendly*, security, mobility, *cloud computing*, SGE, DRMAA, Java, protocol, remote execution, HPCC, supercomputing.

Índice general

Índice de figuras

Índice de ficheros

Capítulo 1

Introducción.

Actualmente, pese a los avances realizados en la capacidad de cómputo de los ordenadores de sobremesa, la demanda del tejido industrial de estos recursos computacionales es cada día mayor. Se observa así una creciente necesidad de resolver problemas mayores y más complejos, lo que propicia que se trate de acceder a entornos de supercomputación, donde la capacidad computacional es de varios órdenes de magnitud mayor que en un ordenador de sobremesa.

De esta forma, el acceso a un *cluster* de altas prestaciones computacionales (HPCC, o *High Performance Computing Cluster*) es relativamente común en entornos de ingeniería en los que los problemas a tratar son cada vez de mayor tamaño y requieren una mayor capacidad computacional. El principal inconveniente a la hora de usar estos *clusters* es la elevada barrera de entrada para los usuarios, debido a la complejidad de estos sistemas.

Así, el usuario ha de conocer de forma precisa la localización de los diversos componentes *software* y disponer de nociones sobre el estado de los recursos *hardware* del *cluster*. Y, además, estos *clusters* están diseñados para el acceso de múltiples usuarios por lo que cuentan con una complejidad añadida: para poder simular sus problemas necesitan mandarlos a un sistema de colas complicando enormemente la tarea, ya que además deben conocer cómo son los comandos del gestor de colas implementado.

Estos sistemas de colas son claves dentro de un *cluster* de este tipo, ya que permite el acceso a los recursos computacionales por parte de varios usuarios de forma excluyente. Si esto no fuera así, se puede incurrir en fallos tanto de los trabajos enviados como en los equipos que componen el *cluster* obligando, en la mayoría de los casos, a que un administrador los subsane.

Para generalizar y mejorar las prestaciones de estos sistemas de gestión de colas existen otras aplicaciones, aunque ninguna de ellas se encuentra enfocada a mejorar la experiencia del usuario. Así, estos programas tratan de acercar el gestor de colas al usuario para permitirle mandar tareas más complejas o con un mayor número de parámetros, pero sin facilitar su uso. Existe de esta forma una barrera de entrada para la mayoría de usuarios que es muy difícil de superar, ya que hay que tener conocimientos acerca del funcionamiento de un gestor de colas y de la conexión a un servidor remoto.

Por este motivo, se hace necesaria una herramienta sencilla y segura que simplifique el envío de trabajos, reduzca el número de parámetros y mejore la experiencia de usuario: lo que se propone es que el uso de un HPCC sea algo fácil y rápido aun para usuarios sin conocimientos de computación distribuida, *clusters* y gestores de colas.

En este contexto, es objetivo del presente Proyecto Fin de Carrera desarrollar una herramienta integral que simplifique el acceso a los recursos computacionales, facilitando la interacción del usuario con el *cluster* y, así, permitiendo que se realicen simulaciones de grandes problemas en un *cluster* como si fuese en su propio equipo.

Además esta aplicación, que ha recibido el nombre de *Posidonia*, tiene muchas más funcionalidades como se detallará a lo largo de la presente memoria que la relacionan con el concepto de *cloud computing* (véase [1]), o computación en la nube. En efecto, la herramienta va a permitir no sólo que el acceso al *cluster* sea mucho más sencillo y rápido sin despreciar la seguridad, sino que éste se puede realizar desde cualquier equipo en el que se encuentre instalado *Posidonia* incluidos dispositivos Android, para los que se ha desarrollado una aplicación específica.

Por otro lado, es conveniente observar algunas de las aplicaciones ya desarrolladas que hay en este sentido y observar las diferencias que guarda *Posidonia* con respecto a ellas:

- *Workflow System*, [2]: es una herramienta desarrollada por TIGR, o *The Institute for Genomic Research* y actualmente llamado JCVI (*J. Craig Venter Institute*). La necesidad de esta institución es similar a la de los problemas que se tratan en esta memoria: TIGR tiene procesos computacionalmente muy costosos que, a su vez, se componen de múltiples procesos discretos que pueden ejecutarse secuencialmente o en paralelo.

  Su objetivo es reducir la intervención manual en el envío de estos procesos al gestor de colas mediante documentos XML (*Extensible Markup Language*)

que se generan por el usuario y que contienen los comandos que se deben ejecutar. Hay que destacar que, si bien XML es un lenguaje entendible y más fácil para un usuario no experimentado, aún así hay que tener nociones básicas acerca de él, por lo que se reduce relativamente la complejidad de envío de trabajos.

Se desarrolla también una interfaz gráfica para hacer al usuario más fácil la creación de estos documentos XML, pero su uso no es transparente; es decir, el usuario es el que debe escribirlos. Además, no se hace de forma remota ya que se ejecuta en el *cluster*, por lo que la movilidad aportada por esta aplicación es prácticamente nula.

- *eXludus Micro-Virtualization*, [3]: esta herramienta ha sido desarrollada por la empresa *eXludus Technologies, Inc.*. Se dedica a optimizar los recursos de un *cluster*, pero no los hace más accesible para el usuario y, desde luego, no trata el tema de conexiones. De esta forma, introduce una cierta organización al tratar con perfiles de trabajo que facilita la tarea para el usuario que, aún así, debe tener conocimientos acerca de la ejecución de trabajos por medio del gestor de colas.

- *Gridwise Tech Grid Engine-Globus Toolkit Adapter*, [4]: solución desarrollada por *GridwiseTech*, ofrece una adaptación entre un gestor de colas llamado SGE (*Sun Grid Engine*, y del que se dará más detalles a lo largo de la memoria) y *Globus Toolkit*, que son herramientas *software* que simplifican la programación de aplicaciones en *clusters*. Así, lo que hace esta aplicación es introducir un gestor de trabajos y un sistema de ficheros más simple para el programador, pero no para el que hace uso de los recursos computacionales que ofrece el *cluster*; además, tampoco introduce ninguna forma integrada de acceder a las tareas de forma remota.

- *Google Compute Engine*, [5]: tecnología desarrollada por *Google* y cuyo objetivo principal es ejecutar trabajos de computación a gran escala en máquinas virtuales Linux alojadas en la infraestructura de *Google*.

  Es la herramienta más parecida a *Posidonia*, ya que permite el acceso a ingentes recursos computacionales de forma remota y segura, ya que todos los datos son encriptados. Sin embargo, tiene una serie de inconvenientes que hacen que *Posidonia* sea una clara alternativa cumpliendo los objetivos marcados en este proyecto:

  - *Google* no proporciona una interfaz gráfica que mejore la experiencia de usuario sino que pone a disposición sus HPCC para la ejecución de trabajos: el usuario debe estar familiarizado con la forma de lanzar estas tareas o desarrollar él mismo una interfaz gráfica que simplifique y acelere el envío de estos proyectos. De hecho, la simple ejecución

de un problema implica definir un gran número de parámetros como *firewalls* que, a priori, no son necesarios para este escenario tan básico.

- *Google* no incluye los programas que son usados por el usuario, de forma que éste es el que tiene que subir el programa y configurar el *cluster* que va a utilizar su problema y luego, lanzar su ejecución. Esto, aunque hace más flexible los trabajos que puede ejecutar un usuario, es muy ineficiente en tanto que los programas que son usados para estos problemas tan grandes son, normalmente, muy pesados.
- *Google* da muchas opciones de configuración del entorno y de las máquinas virtuales que el usuario desea ejecutar; sin embargo, *Posidonia* se enfoca a restringir estos parámetros haciendo que el acceso al *cluster* sea mucho más sencillo, y con muchas menos opciones de configuración.
- Esta aplicación está aún en fase experimental, mientras que *Posidonia* se encuentra ya en una versión estable aunque, obviamente, en un entorno más controlado como es el *cluster* del que se dispone para estos problemas.

En definitiva, no hay actualmente ninguna herramienta desarrollada en el sector que se adecúe a las necesidades de este proyecto. Por otro lado, se ha observado que no hay nada parecido a lo que implementa la aplicación desarrollada, que no es más que la ejecución remota de trabajos en un gestor de colas, haciéndolo de forma segura y eficiente y, además, proporcionando una gran movilidad ya que el envío y control de los trabajos enviados al *cluster* se puede realizar desde cualquier equipo, incluido dispositivos móviles.

Esto último enlaza con el concepto de computación en la nube, de forma que *Posidonia* lleva el *cluster* a la nube situándose en la vanguardia de este tipo de aplicaciones, tal y como demuestra el intento de *Google* de hacer algo similar y a gran escala.

Para concluir esta introducción, se detalla la estructura que va a seguir la memoria de este Proyecto Fin de Carrera:

- En el Capítulo 2 se exponen cuáles son los objetivos de la herramienta desarrollada, y el entorno en el que ésta se encuentra. Además, se comentarán las características y funcionalidades de la aplicación, y se explicarán las tres soluciones desarrolladas: para problemas de FortranMod, para códigos de Matlab y para Android, presentando estos programas y sus particularidades.

- A continuación, en el Capítulo 3 se detalla cómo se realiza el envío de un trabajo al gestor de colas por medio de Java.
- Por otro lado, en el Capítulo 4 se presenta el protocolo de comunicaciones que se ha desarrollado para que el envío y control de las tareas que un cierto usuario ha enviado al *cluster* se hagan de forma remota y segura.
- Posteriormente, en el Capítulo 5 se explican algunos detalles sobre las interfaces gráficas programadas.
- Y para finalizar, en el Capítulo 6 se exponen las conclusiones que se han obtenido con el desarrollo de este proyecto, y las líneas de investigación futuras que se pueden seguir partiendo de esta herramienta.

Capítulo 2

Principales funcionalidades de la aplicación.

En este capítulo se van a detallar los motivos del desarrollo de esta aplicación, *Posidonia*, los objetivos que pretende alcanzar, y las principales funcionalidades que ofrece la aplicación en sus tres versiones:

- Integrada con GiD (véase [6]), que es una interfaz gráfica de pre y postprocesado para la ejecución de problemas de simulación complejos con programas como FortranMod, tal y como se verá más adelante.
- Herramienta autónoma para trabajos de Matlab (para más información sobre este programa se puede consultar [7]), que es una aplicación utilizada para la ejecución eficiente de operaciones matriciales entre otros muchos usos. También se podría haber usado Octave, [8], cuyas prestaciones son parecidas.
- Integrada en Android (véase [9]) para tareas de Matlab.

De esta forma, la estructura que sigue el epígrafe es la siguiente:

- Escenario actual y motivación, donde se verá el gestor de colas utilizado (SGE) y el valor añadido que supone el uso de *Posidonia*.
- Objetivos y funcionalidad general de *Posidonia*.
- Integración de *Posidonia* en GiD.
- Aplicación autónoma con Matlab.
- *Posidonia* en Android.

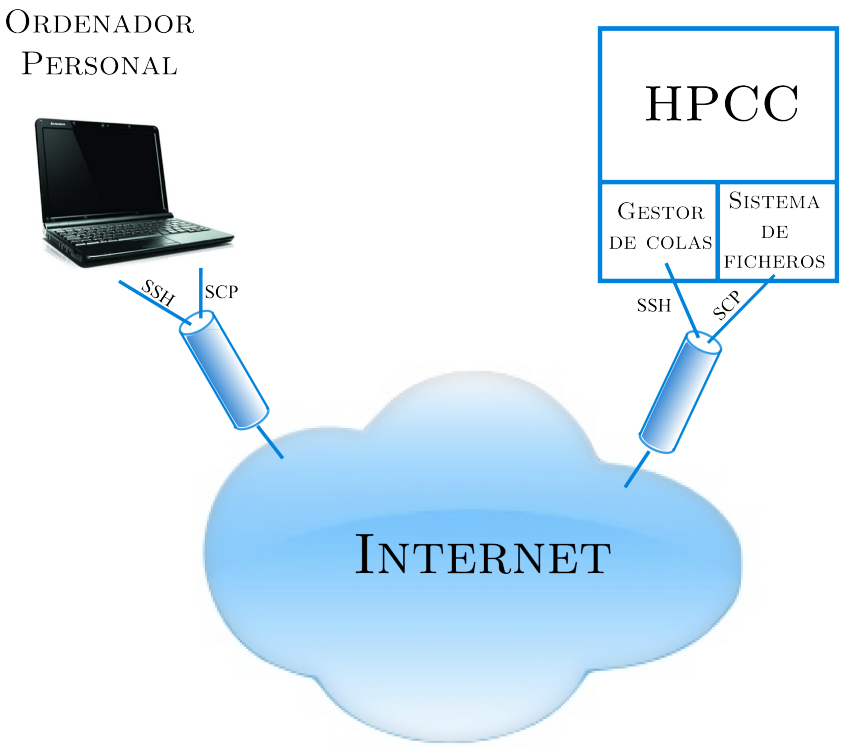


Figura 2.1: Escenario general.

2.1. Escenario actual y motivación.

El escenario que afronta la aplicación en su planteamiento es el que se muestra en la Figura 2.1, de forma que a lo largo de la sección se observará por qué surge la necesidad de desarrollar un proyecto como éste.

De esta manera, el escenario considerado a la hora de desarrollar la herramienta es muy general, ya que en el caso más común se tiene el entorno que se indica en la Figura 2.1 en el que se muestra la conexión con un *cluster* a través de Internet. En la imagen se muestran dos de los servicios del HPCC (*High Performance Computing Cluster*, que es el acrónimo de un *cluster* de altas prestaciones computacionales): un gestor de colas que se encarga de la gestión de los ficheros en ejecución y un sistema de ficheros en el que residen los archivos que son y serán usados por el *cluster*.

Hay dos protocolos fundamentales para la interacción con el HPCC de forma remota: a la consola de comandos se accede por medio del protocolo SSH (*Secure*

SHell, para más información véase [10, 11, 12]), de forma que se puede acceder de forma segura al gestor de colas por medio de comandos propios del servicio, mientras que al sistema de ficheros se entra mediante SCP (*Secure Copy*, que es una aplicación de SSH que garantiza la transferencia segura) enviando y recibiendo toda clase de archivos entre el *cluster* y el punto de acceso, considerado como un ordenador personal con recursos computacionales limitados o, incluso, como un dispositivo Android como se observará en posteriores apartados. Como es lógico, para el uso de estas herramientas hay que tener acceso a una conexión a Internet; en general, el *cluster* se encuentra accesible desde cualquier punto de la red y la seguridad viene proporcionada por el uso de SSH y SCP.

Posidonia busca que esta interacción con el *cluster* sea automatizada y mucho más sencilla para usuarios inexpertos en las herramientas descritas anteriormente. Como se observará a continuación, la ejecución simple de trabajos por medio del gestor de colas requiere de un proceso muy pesado y ciertamente complejo si no se conocen los comandos apropiados; con la aplicación se consigue que el *cluster* sea mucho más accesible para la gran mayoría de usuarios que desean hacer uso de sus servicios. Por otro lado, también se verá que el uso de *Posidonia* facilita el acceso al *cluster* en cualquier lugar con acceso a Internet, incrementando la movilidad.

Sin embargo, la aplicación desarrollada se encuentra en un entorno más específico que, en concreto, se muestra en la Figura 2.2. El *cluster* se encuentra en el dominio del departamento de Teoría de la Señal y Comunicaciones, TSC, de la Universidad, y no se encuentra accesible desde fuera del dominio, por lo que para poder entrar a él hay que usar una VPN (*Virtual Private Network*, o Red Privada Virtual), que es una tecnología mediante la cual se puede acceder a los servicios del dominio TSC por medio del ordenador personal aunque no estén abiertos a Internet.

Así, el uso de la VPN permite que el usuario se encuentre en la red de TSC aunque físicamente no tenga posibilidad de conectarse directamente a ella; es decir, permite disponer de una red privada aunque se disponga únicamente de un acceso a Internet. Hay distintos tipos de redes privadas virtuales, fundamentalmente:

- VPN de acceso remoto: el usuario es capaz de conectarse a un cierto dominio desde puntos de acceso remotos utilizando Internet como infraestructura. Una vez en la red privada pueden hacer uso de todos los elementos de red del dominio al que se conectan siendo, a todos los efectos, como si estuvieran físicamente en el dominio.
- VPN punto a punto: se establece una conexión virtual entre dos ordenadores, de forma que es equivalente a una conexión física punto a punto tradicional.

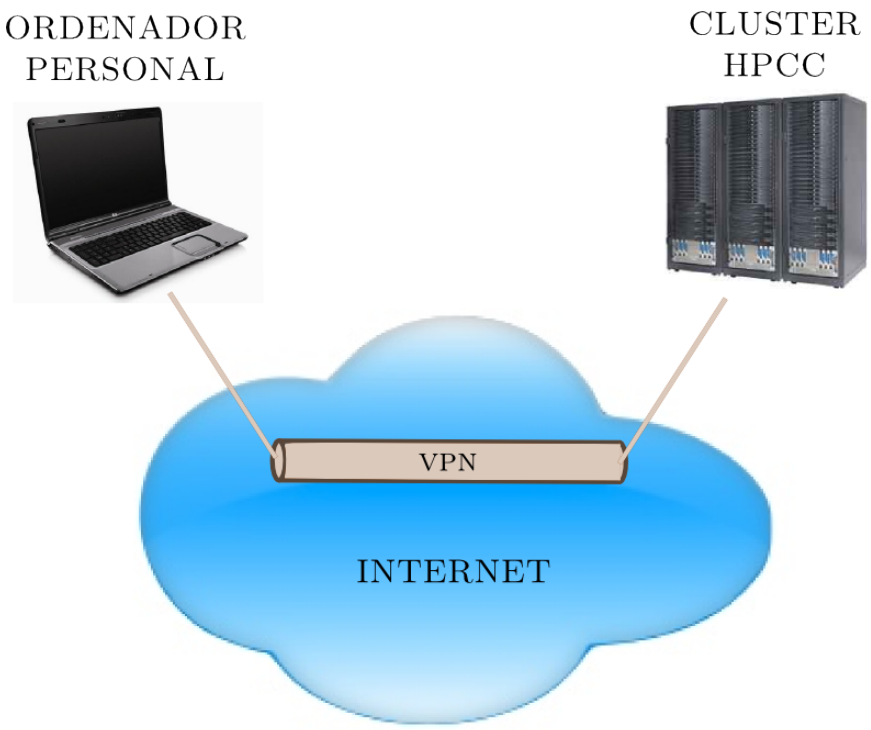


Figura 2.2: Escenario considerado con VPN.

- VPN sobre LAN (*Local Area Network*): la funcionalidad que ofrece es la misma que la VPN de acceso remoto pero la conexión se hace a través de una LAN, o red de área local en vez de hacerse a través de Internet. Así, la VPN no está disponible para cualquier acceso a Internet, sino que es necesario entrar físicamente en la red local; sin embargo, las prestaciones de una LAN son normalmente más elevadas que las que ofrece Internet.

La VPN que se considera en este escenario es la del primer tipo, y es la que aporta más movilidad ya que permite el acceso al *cluster* con una simple conexión a Internet, de forma que *Posidonia* puede ser usado en cualquier equipo con esa conectividad. Además, obligar a la utilización de la VPN añade seguridad, ya que así sólo pueden hacer uso de los servicios de TSC aquellos que dispongan de la red privada.

Por otro lado, la Figura 2.2 es muy descriptiva e incide en las ideas que se han explicado: se observa cómo la VPN se provee mediante Internet, por lo que es necesario el acceso a esta red para tener entrada al *cluster*, algo que no es necesario con otros tipos de VPN como se ha visto. Por otro lado, el uso de esta herramienta

se facilita por medio de herramientas como OpenVPN ([13]) y otras, de forma que el departamento ofrece la solución que mejor se adecúe al usuario incluyendo un amplio soporte técnico.

De esta forma, el entorno que se planteará en el desarrollo de la aplicación será este último, debido a las ventajas que ya se ha observado que ofrece (y que ofrecerá como se detallará a lo largo de esta memoria) y a la expansión de esta solución para el acceso a herramientas del departamento de TSC.

Ya explicado el escenario, a continuación se va a proceder a una breve ilustración de cómo se ejecuta un trabajo simple de Matlab en el escenario de la Figura 2.2. Ello ayudará a comprender la motivación de la aplicación desarrollada, *Posidonia*.

La tarea que se va a enviar al gestor de colas es un código en Matlab (programa ampliamente expandido en entornos de ingeniería y utilizado, entre otros usos, para el cálculo eficiente de pesadas operaciones matemáticas; para más información véase [7]) que se compone de las siguientes entradas:

- Un script, `prueba.m`, mostrado en el Fichero 2.1 que es una secuencia de comandos que se van a ejecutar y que usa diferentes archivos `.m`, que serán funciones a las que se le pasan ciertos parámetros y devuelven resultados, y un archivo `.mat`, que es un contenedor en el que se guardan algunas variables junto con sus valores. En general, aunque no sea el caso, también puede usar archivos `.p`, que son funciones `.m` que ya se encuentran compiladas, y son como ejecutables en Windows, o librerías en Java.
- Funciones de Matlab llamadas por el script, `cuadratica.m` y `lineal.m`.
- Un contenedor de variables llamado `plots.mat`

Y que tiene las siguientes salidas:

- Imágenes en el formato `.jpg` que son las representaciones que se han llevado a cabo en el Fichero 2.1: `plot1.jpg`, `plot2.jpg`, `polar.jpg`, `plot3d.jpg`, `cont.jpg` y `plot3d2.jpg`.
- Un contenedor de variables en el que se almacenan todas las variables existentes al acabar el script y que se denomina `script2.mat`. En el caso que se está proponiendo, es clave generar este contenedor para que los resultados del script no se pierdan al acabar la ejecución del trabajo; de otra forma, los valores asignados a las variables, e incluso éstas, no se guardan en el *cluster*.

Fichero 2.1: prueba.m

```
% Se usa el contenedor.
load plots.mat

x = lineal(1:10);
y = cuadratica(1:10);

% Algunas representaciones.
h = plot(x,y);

saveas(h, 'plot1.jpg');

h2 = plot(x,x);
saveas(h2, 'plot2.jpg');

t = 0:.01:2*pi;

h3 = polar(t,sin(2*t).*cos(2*t),'--r');
saveas(h3, 'polar.jpg');

th = (0:127)/128*2*pi;

x = cos(th);

y = sin(th);

f=abs(fft(ones(10,1),128));

h4 = stem3(x,y,f');
saveas (h4, 'plot3d.jpg')

[X,Y] = meshgrid(-2:.2:2,-2:.2:3);

Z = X.*exp(-X.^2-Y.^2);

[c, h5] = contour(X,Y,Z);
saveas(h5, 'cont.jpg')

[X,Y] = meshgrid(-3:.125:3);

Z = peaks(X,Y);

h6 = mesh(X,Y,Z);
saveas(h6, 'plot3d2.jpg')

save script2.mat
```

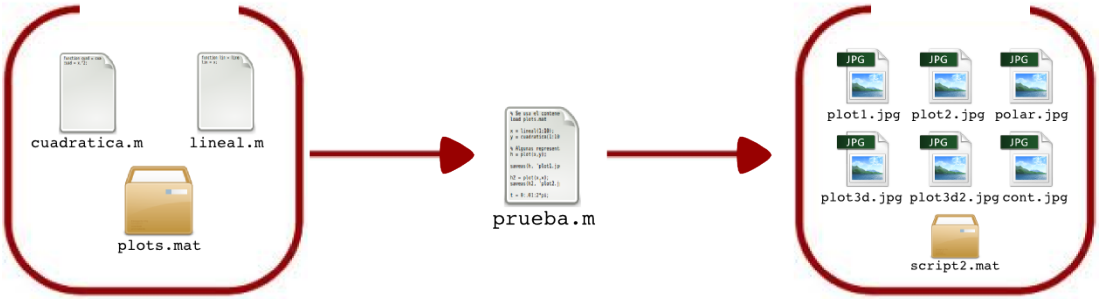


Figura 2.3: Ejemplo de ejecución con Matlab.

Así, el esquema que tiene la ejecución de esta tarea es el que se muestra en la Figura 2.3.

Una vez descrito el trabajo, se van a detallar los pasos a seguir para la ejecución de `prueba.m` en el *cluster*:

1. En primer lugar, se deben enviar los ficheros al *cluster* para que se pueda ejecutar el trabajo en él.

   Para esto, por consola se puede hacer uso del comando `scp`, ya que en el *cluster*, localizado en el *host* `ash25b` (nombre que representa al *cluster* en la red del departamento), se encuentra activo un servidor SSH, algo indispensable al estar SCP basado en este protocolo.

   De esta forma, considerando que se tiene acceso a la VPN que conecta directamente el ordenador personal a `ash25b`, hay que enviar los archivos al *cluster*. Así, si se tienen los ficheros de entrada de la Figura 2.3 en la ruta local `/home/adrian/Documentos/Ejemplo`, y se quieren enviar a, por ejemplo, la ruta en `ash25b` `/home/aamor/Ejemplo`, se puede ejecutar:

```
~$ cd /home/adrian/Documentos/Ejemplo/
~/Documentos/Ejemplo$ ls -l
  total 16
  -rw-r--r-- 1 adrian adrian  43 2012-06-29 18:49 cuadratica.m
  -rw-r--r-- 1 adrian adrian  34 2012-06-29 18:48 lineal.m
  -rw-r--r-- 1 adrian adrian 219 2012-06-23 04:21 plots.mat
  -rw-r--r-- 1 adrian adrian 628 2012-06-29 18:51 prueba.m
~/Documentos/Ejemplo$ scp /home/adrian/Documentos/Ejemplo/*
    aamor@ash25b:/home/aamorlocal/Ejemplo
  cuadratica.m    100%   43     0.0KB/s   00:00
  lineal.m        100%   34     0.0KB/s   00:00
  plots.mat       100%  219     0.2KB/s   00:00
  prueba.m        100%  628     0.6KB/s   00:00
```

   Y se puede observar que, en efecto, los archivos han llegado al directorio correspondiente de `ash25b`. Para ello se realiza una conexión SSH a `ash25b`, en la que también se crea un directorio en la ruta `/home/aamor/Ejemplo`,

Fichero 2.2: scriptMatlab.sh

```
1   #!/bin/bash
    #$ -V
    #$ -cwd
    #$ -j y
    #$ -S /bin/bash
6   #$ -N scriptMatlab
    #$ -o scriptMatlab.o
    #$ -q all.q
    #$ -l h_vmem=4G
    #$ -pe orte 1
11  ## Ejecutamos el script prueba.m y salimos
    echo prueba | /opt/matlab/bin/matlab -nosplash -nojvm -nodisplay
```

para lo que es necesario disponer de un usuario en `ash25b` que, en este caso, es `aamor`. Así los archivos ya están disponibles para que puedan ser ejecutados por el gestor de colas.

2. Una vez disponibles los archivos de entrada, a continuación hay que ejecutar el script `prueba.m` con el gestor de colas. Para ello hay que hacer uso de los comandos propios de dicho gestor, que en este caso concreto es SGE (*Sun Grid Engine*, véase [14, 15, 16]) y con los que hay que tener ciertas consideraciones:

   a) La ejecución se hace por medio de un script escrito por medio del lenguaje *shell script*. De esta forma hay que crear, en primer lugar, el fichero `.sh` que recoge algunas opciones de ejecución, y luego enviarlo a SGE por medio del comando `qsub`.

   b) El estado del trabajo se obtiene invocando a `qstat`, por medio del que se obtiene información acerca de las tareas enviadas por el usuario.

   c) Finalmente, en el caso de que se quisiera eliminar el trabajo que se está ejecutando actualmente, se ejecutaría el comando `qdel` acompañado del identificador del trabajo a eliminar.

   Todo esto se explica más detalladamente en el Capítulo 3; aquí se verá únicamente cómo se ejecuta el ejemplo que se está tratando. Así, en primer lugar se tiene que crear el script. Para un usuario inexperto en SGE, como es el que se considera en este proyecto y en este ejemplo, es ciertamente complicado programarlo si no se tienen unos conocimientos mínimos de *shell script*; sin embargo, hay plantillas que pueden ser bastante útiles como la que se incluye en el Fichero 2.2.

   Las principales opciones que se indican en ese script son, en la línea 7, el nombre del fichero de salida que genera SGE y que es clave para detectar si ha habido algún error en la ejecución del script, y en la línea 11, la máxima memoria virtual que es capaz de ocupar el trabajo (y es algo muy

importante debido al carácter de Matlab, que hace un uso intensivo de esa memoria) que se limita a 4 gigabytes. El resto de opciones se explican con detalle en el Capítulo 3, al igual que los comandos más importantes de SGE para el usuario.

Ya creado el script `scriptMatlab.sh`, lo que se ha de hacer es enviarlo al gestor de colas para que éste lo ejecute con Matlab. Para esto se ejecuta `qsub scriptMatlab.sh` en consola:

```
[aamor@ash25b Ejemplo]$ qsub scriptMatlab.sh
  Your job 4075 ("scriptMatlab") has been submitted
```

Se puede comprobar que el trabajo ha llegado al gestor SGE ya que si se ejecuta `qstat`, que es un comando en el que se ve el estado de todos las tareas enviadas por el usuario, se obtiene lo siguiente:

```
[aamor@ash25b Ejemplo]$ qstat
  job-ID  prior   name       user  state submit/start at     queue slots
  -----------------------------------------------------------------------
    4075 5.00500 scriptMatl aamor qw    07/01/2012 03:01:27         1
```

La información más importante para el usuario que se obtiene con `qstat` es la que aparece en el campo `state`, en este caso `qw`, es decir, *queue waiting* o esperando en cola a que el gestor le asigne recursos para su ejecución; si se tiene más interés, el significado de todas estas opciones aparece reflejado en el Capítulo 3.

De esta forma, si se ejecuta posteriormente el comando `qstat` se puede observar que el trabajo cambia su estado a `r` (*running*), lo que significa que la tarea ya ha conseguido los recursos que solicitaba y se está ejecutando en el *cluster*, como se observa en la siguiente salida por consola, donde se muestra por sencillez los campos que han cambiado con respecto a lo anterior:

```
[aamor@ash25b Ejemplo]$ qstat
  state  submit/start at      queue                        slots
  ---------------------------------------------------------------
    r    07/01/2012 03:02:51  all.q@compute-0-11.local     1
```

Así, el trabajo ya se está ejecutando y lo único que le queda al usuario es esperar a que termine, momento en el que puede ser notificado si ha configurado el script `scriptMatlab.sh` de forma que se envíe un correo electrónico cuando la tarea acabe de ejecutarse, algo que en el Fichero 2.2 no se ha determinado.

3. Finalmente, ya acabada la ejecución, en el directorio en el que se ha lanzado el trabajo se encuentran los ficheros de salida que son, por un lado, los que genera el propio script en Matlab `prueba.m` y, por otro lado, los que genera la ejecución en SGE que, en este caso, dada la configuración del

Fichero 2.3: scriptMatlab.o (sin errores)

```
                                    < M A T L A B (R) >
3                         Copyright 1984-2008 The MathWorks, Inc.
                             Version 7.6.0.324 (R2008a)
                                  February 10, 2008


8                  This is a Classroom License for instructional use only.
                   Research and commercial use is prohibited.

               To get started, type one of these: helpwin, helpdesk, or demo.
               For product information, visit www.mathworks.com.
13
             >> >>
```

script `scriptMatlab.sh`, simplemente es el fichero `scriptMatlab.o`. En este archivo se pueden encontrar fallos relacionados con la ejecución en Matlab, es decir, fallos en sintaxis o en ejecución de `prueba.m`, o con la gestión de SGE, como no encontrar los recursos que ha solicitado la tarea.

Así, si se listan los ficheros que hay en el directorio en el que se enviaron los archivos al gestor de colas, se obtiene lo siguiente:

```
[aamor@ash25b Ejemplo]$ ls -l
  total 74212
  -rw-r--r-- 1 aamor aamor    22047 Jul  1 03:05 contorno.jpg
  -rw-rw-r-- 1 aamor aamor       34 Jun 30 18:31 cuadratica.m
  -rw-rw-r-- 1 aamor aamor       43 Jun 30 18:31 lineal.m
  -rw-r--r-- 1 aamor aamor    14115 Jul  1 03:05 plot1.jpg
  -rw-r--r-- 1 aamor aamor    13530 Jul  1 03:05 plot2.jpg
  -rw-r--r-- 1 aamor aamor    51125 Jul  1 03:05 plot3d2.jpg
  -rw-r--r-- 1 aamor aamor    42924 Jul  1 03:05 plot3d.jpg
  -rw-r--r-- 1 aamor aamor      219 Jun 30 18:31 plots.mat
  -rw-r--r-- 1 aamor aamor    27772 Jul  1 03:05 polar.jpg
  -rw-r--r-- 1 aamor aamor      436 Jul  1 03:05 scriptMatlab.o
  -rw-rw-r-- 1 aamor aamor      249 Jul  1 03:01 scriptMatlab.sh
  -rw-r--r-- 1 aamor aamor      628 Jun 30 18:31 prueba.m
  -rw-r--r-- 1 aamor aamor 75686145 Jul  1 03:05 script2.mat
```

Se puede observar lo que se anticipó anteriormente, ya que aparecen tanto los archivos de salida propios de Matlab (los `.jpg` y `script2.mat`) como los que ha generado SGE (`scriptMatlab.o`). Como la ejecución ha transcurrido con normalidad, el Fichero 2.3 muestra una consola de Matlab sin ningún error; si el script en Matlab fuera sintácticamente incorrecto se obtendría un archivo `.o` tal y como se muestra en el Fichero 2.4. Así queda manifiesta la importancia que tiene este archivo `.o` para comprobar la correcta ejecución de las tareas enviadas al *cluster*.

Una vez localizados los archivos de salida, si el usuario quisiera descargar esos ficheros a su directorio local de trabajo, debería hacer lo mismo que se hizo para enviar los archivos de entrada de `prueba.m` por medio, por

Fichero 2.4: scriptMatlab.o (con errores)

```
                                    < M A T L A B (R) >
                          Copyright 1984-2008 The MathWorks, Inc.
                             Version 7.6.0.324 (R2008a)
 5                                February 10, 2008


                  This is a Classroom License for instructional use only.
                  Research and commercial use is prohibited.
10
              To get started, type one of these: helpwin, helpdesk, or demo.
              For product information, visit www.mathworks.com.

            >> ??? Error: File:
15          /home/aamor/Ejemplo/prueba.m
            Line: 2 Column: 1
            The input character is not valid in MATLAB statements or
                expressions.
            Error in ==> prueba at 4
```

ejemplo, del comando `scp` desde `ash25b`.

En definitiva, se puede observar que para un usuario sin un gran dominio de herramientas de comunicación como son `ssh` o `scp`, el simple hecho de enviar los archivos de entrada y descargar los archivos de salida puede ser bastante complicado y consumir una ingente cantidad de tiempo. Por otra parte, si se considera un usuario con conocimientos básicos de `ssh` y `scp`, también debe saber cómo enviar un trabajo al *cluster* y cómo conocer cuándo ha acabado la tarea enviada. Finalmente, un usuario con soltura en el manejo de los comandos descritos anteriormente encontrará este proceso bastante tedioso.

También hay que considerar que el ejemplo detallado en este apartado se ha realizado en un entorno Linux en el que, generalmente, es más sencillo el uso de `ssh` y `scp`. Para Windows hay herramientas específicas como Putty (que puede desempeñar la función de cliente SSH, véase [17]) o WinSCP (que es un cliente SFTP (*SSH File Transfer Protocol*) gráfico y que hace las mismas funciones que SCP, véase [18]) que pueden ser usados como sustitutivos pero que, a la vez, requieren la familiarización con sus interfaces gráficas.

De esta forma se tiene un *cluster*, que es un recurso muy potente que puede acelerar notablemente la ejecución de ciertos trabajos, accesible a un conjunto reducido de usuarios y cuyo uso es bastante pesado y dependiente de la plataforma. Parece interesante desarrollar una herramienta que facilite el uso de este recurso ya que, por un lado, se amplía el círculo de posibles usuarios del *cluster* y, por otro, conlleva un uso más cómodo y, sobre todo, más rápido, de una herramienta bastante útil en los ámbitos de trabajo del departamento.

2.2. Objetivos y funcionalidades generales de *Posidonia*.

En este esta sección se van a explicar qué metas se marca la aplicación y sus principales características, así como las funcionalidades que proporciona *Posidonia*.

Dada la motivación que se ha detallado en el apartado anterior, el principal objetivo de *Posidonia* es el hacer accesible el *cluster* a la mayor cantidad posible de usuarios del departamento, de forma que el envío de trabajos sea fácil y rápido. Se desea, además, que se pueda controlar el estado de las tareas enviadas y que guarde un histórico de las que ya han sido ejecutadas por el usuario.

De esta forma, las principales características que se van a tener en cuenta en todo momento a la hora de desarrollar esta aplicación son las siguientes:

- Extensibilidad y generalidad. Se hace una programación modular de forma que el código desarrollado sea reutilizable de una forma sencilla, para lo que se crean una serie de librerías con una documentación adecuada. También se procura que los parámetros de configuración más importantes sean modificables por el usuario, como la dirección en la que se encuentra el *cluster*, para que sea fácilmente adaptable a otros escenarios más generales, como el que se detalla en la Figura 2.1.

- Seguridad. Dadas las necesidades de privacidad que se tienen actualmente en Internet, las comunicaciones siempre se hacen de forma segura gracias al protocolo SSH para que los datos del usuario nunca corran peligro.

- Multiplataforma. Como se quiere abrir el uso del *cluster* al mayor número de usuarios posible, se desarrolla una aplicación ejecutable en todas las plataformas del mercado (Windows, Linux, Mac...).

- *User-friendly*. Este término, en el entorno de las aplicaciones informáticas, hace referencia a la facilidad de uso de un cierto programa. En todo momento se busca que *Posidonia* sea sencilla, con una interfaz amigable y que no requiera de ningún tutorial para su uso, es decir, que sea completamente intuitiva para todos los usuarios.

- Movilidad. Se busca que el acceso al *cluster* pueda hacerse desde cualquier sitio y dispositivo siempre y cuando se tenga instalada una de las versiones de *Posidonia*.

- Eficiencia. La aplicación busca consumir el menor número de recursos posibles. Por ello, no se utiliza ninguna espera activa ni nada similar, tal y como se explicará en el Capítulo 3, y el tráfico que circula por la red asociado a *Posidonia* es el menor posible, como se detallará en el Capítulo 4.

- Organización por capas. La herramienta se ha desarrollado con una estructura dividida por capas en base a la función desempeñada como se muestra en la Figura 3.1, con lo que el código es más inteligible de esta forma y más reutilizable. Por ejemplo, gracias a esta arquitectura las tres interfaces gráficas que se han implementado utilizan la misma capa de comunicaciones y de ejecución de trabajos.

Así, *Posidonia* ha sido desarrollado en el lenguaje Java (véase [19]), lo que tiene ciertas ventajas e inconvenientes:

1. En primer lugar, una ventaja importante de Java es su programación orientada a objetos, lo que hace que su extensibilidad sea más sencilla que en el caso de una programación funcional como, por ejemplo, la que aporta el lenguaje C. Por otro lado, para garantizar la seguridad en las comunicaciones se va a usar el protocolo SSH que está muy implementado en Java: de hecho, se usará, tal y como se mostrará en el Capítulo 4, la librería JSch (véase [20]) que se encuentra muy expandida, con todas las ventajas que ello conlleva.

2. Por otra parte, también hay que destacar el carácter multiplataforma de Java, ya que sus códigos se ejecutan en una máquina virtual (JVM, *Java Virtual Machine*) disponible para los principales sistemas operativos del mercado. En lo que respecta a la movilidad, dado que Android (véase [9]), basado en Java, es uno de los sistemas más expandidos en el entorno de móviles y *tablets*, es conveniente desarrollar *Posidonia* en este lenguaje para que el salto a la aplicación móvil sea mucho más suave y se pueda reutilizar gran parte del código ya programado.

3. Finalmente, uno de los principales inconvenientes que antes ofrecía Java es que el hecho de ser multiplataforma hacía que su interfaz gráfica fuera bastante pobre, amenazando así el carácter *user-friendly* de la aplicación. Sin embargo, esto con las últimas versiones de Java se ha ido corrigiendo y, de hecho, tal y como se verá en las siguientes secciones, se adapta perfectamente al entorno en el que se ejecute *Posidonia*.

De esta forma, los requisitos de sistema que tendrá la aplicación serán únicamente los que exija la instalación de la última versión de la máquina virtual de Java,

Java 7, ya que se han usado funcionalidades especiales que se incluyen en esta *release*. También se asumirá que la conexión directa al *cluster* al que se envían los trabajos está disponible, ya sea por medio de una VPN o por estar el *host* en el que se encuentra el HPCC disponible desde la conexión de red del usuario.

Por otro lado, como se ha comentado en la introducción de este capítulo, *Posidonia* tiene tres interfaces gráficas que demuestran la generalidad del código desarrollado, ya que:

- Por una parte, se utilizan dos aplicaciones distintas (Matlab y FortranMod, programa de simulación que se presentará en siguientes secciones, véase [21], integrado en GiD) de forma que cambian los parámetros de ejecución de los trabajos y los ficheros de entrada y de salida involucrados.
- Por otra, se desarrollan dos tipos de interfaces gráficas:
  - Un programa de escritorio multiplataforma, es decir, independiente del sistema operativo utilizado, por medio de Java para los dos programas expuestos anteriormente.
  - Una aplicación Android para trabajos de Matlab y que no es directamente exportable a otras arquitecturas móviles.

Sin embargo, las funcionalidades que tienen estas tres versiones de la aplicación son muy similares, y se pueden resumir en las siguientes:

- Envío de trabajos. Es la funcionalidad más básica, y la que se ha explicado pormenorizadamente en la sección anterior. La ejecución de tareas en el *cluster* se convierte en algo más sencillo y rápido como se observará en las siguientes secciones.
- Configuración de las opciones de ejecución. Para los usuarios más familiarizados con los *clusters* de alto rendimiento, se proporciona una pestaña para cambiar parámetros avanzados del trabajo que se va a enviar.
- Control y monitorización de las tareas en ejecución. Se pueden consultar los trabajos que se han mandado al *cluster* y que aún no han terminado de ejecutarse, de forma que pueden ser eliminados por el usuario.
- Notificación de trabajos finalizados. Si una tarea acaba su ejecución mientras la aplicación esté activa, una notificación es mostrada al usuario y se descargan los resultados automáticamente. Por otro lado, si una tarea ha acabado mientras *Posidonia* ha estado inactiva, al volver a lanzar la aplicación, se lanza una notificación en la que se indican los trabajos que han acabado cuando *Posidonia* no estaba ejecutándose.

- Histórico de trabajos. Se tiene un historial de las tareas que ha mandado un cierto usuario al *cluster* de manera que se pueden descargar los ficheros de entrada y de salida de cada trabajo. También es posible borrar una tarea del histórico de trabajos, de forma que se borra todo rastro de ella en el *cluster*.

  Esta funcionalidad contribuye enormemente a una de las principales características de *Posidonia* como es la movilidad, ya que funciona como un repositorio en el que se tiene acceso a las entradas y salidas de las tareas ejecutadas en cualquier dispositivo en el que se encuentre instalada la aplicación.

En las siguientes secciones se detallará cómo se adaptan las características y funcionalidades de *Posidonia* a los entornos que se han descrito anteriormente. Por otra parte, la estructura de la aplicación y su implementación se explican en los Capítulos 3, 4 y 5.

2.3. Integración de *Posidonia* en GiD.

En este apartado se explica cómo se lleva a cabo la integración de la aplicación *Posidonia* en GiD, que es una interfaz que facilita la integración de programas de simulación electromagnética como FortranMod (véase [21]), desarrollado en el grupo de radiofrecuencia, al que a partir de ahora se denominará GRF, al que está adscrito este proyecto.

GiD es un pre y postprocesador *user-friendly* para simulaciones numéricas en ciencia e ingeniería. En este departamento se utiliza para representar estructuras de radiofrecuencia que se simulan por medio de FortranMod pero puede ser usado en muchos otros casos como cálculos aerodinámicos o procesos industriales, ya que está preparada para utilizarse en un gran número de ámbitos. Si se desea más información sobre este programa, se puede consultar [6].

En definitiva, GiD proporciona únicamente un interfaz amigable para hacer el pre y el postprocesado de un problema de simulación; el programa, en el caso considerado, es el desarrollado en el GRF, FortranMod.

Es de destacar que las simulaciones que lleva a cabo FortranMod son muy intensivas en lo que se refiere a memoria y potencia de computación; de hecho, es habitual que sus cálculos duren horas e incluso días en función del dispositivo que se esté empleando. Por ello, es muy aconsejable contar con un equipo potente o

tener acceso a un HPCC, es decir, a un *cluster* de alto rendimiento para, por un lado, reducir el tiempo de ejecución del problema y, por otro lado, poder seguir trabajando con normalidad en el equipo del usuario. Así, GiD proporciona una interfaz gráfica que es intuitiva para el usuario haciendo que las simulaciones con FortranMod sean más confortables y sencillas pero exige que el ordenador del usuario esté encendido durante prolongados intervalos de tiempo, que serán más o menos largos en función de las características del equipo.

De esta forma, la inclusión de *Posidonia* en GiD supone un avance en la dirección que marca este interfaz: hace que las simulaciones con programas como FortranMod sean más sencillas y cómodas. De hecho, los principios de GiD son muy similares a los que se marca *Posidonia*: GiD pretende ser una herramienta universal, adaptativa y *user-friendly* mientras que *Posidonia* toma las mismas características y les añade la seguridad, la movilidad y el carácter multiplataforma.

Por todos estos motivos, la integración con GiD es más que conveniente: con *Posidonia* no sólo se tiene una interfaz gráfica intuitiva para llevar a cabo sus cálculos y simulaciones sino que además se pueden enviar éstas a un HPCC (si dispone de los permisos adecuados para acceder a un recurso de este tipo) y controlar el estado de su ejecución. De esta forma, el usuario configura un trabajo, lo envía al *cluster* y puede seguir trabajando normalmente en su equipo mientras su simulación está siendo ejecutada en el HPCC (lo que además es más rápido normalmente dadas las características de este tipo de equipos: paralelización, potencia de cálculo, etcétera); cuando ésta acabe, *Posidonia* lanzará una notificación y descargará los resultados automáticamente.

Por otra parte, el uso de *Posidonia* también facilita al usuario poder trabajar con la misma simulación en equipos distintos: puede lanzar la tarea, por ejemplo, con el ordenador de sobremesa de su lugar de trabajo y descargarse tanto los archivos de entrada como los resultados en su domicilio, observando si los cálculos son los esperados o hay que realizar pequeños cambios, en cuyo caso podría modificarlos en su ordenador personal (siempre y cuando disponga de GiD y FortranMod en ambos lugares, obviamente), lanzar la simulación y, al día siguiente, comprobar los nuevos resultados en su lugar de trabajo. De esta forma, el uso de este programa con *Posidonia* es mucho más cómodo que ejecutar el trabajo en un solo equipo y esperar a que finalice, sobre todo teniendo en cuenta el carácter de este tipo de tareas.

Además, el uso de *Posidonia* ofrece una gran ventaja que es la de tener un repositorio en el que se pueden guardar automáticamente distintos resultados de una misma simulación, de forma que se puede volver a una anterior versión si es mejor que la actual sin tenerla almacenada localmente. Esto es posible gracias al

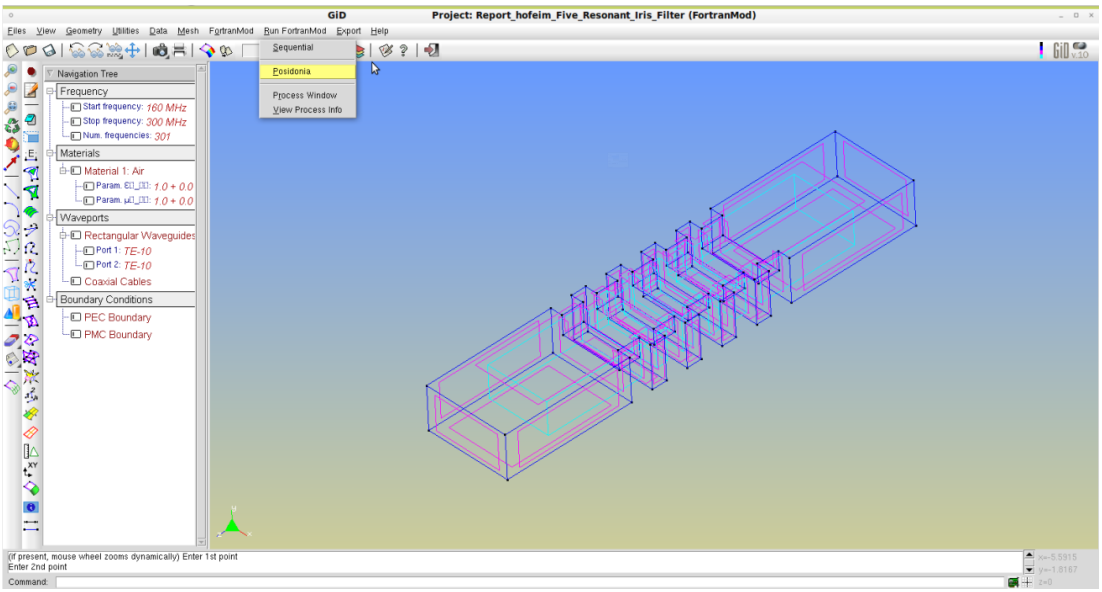


Figura 2.4: Integración de *Posidonia* con GiD.

histórico de archivos que incluye la aplicación y que permite descargar tanto los archivos de entrada como los de salida, evitando posibles pérdidas de versiones que funcionaban mejor que las más recientes y pudiendo trabajar con cualquier versión en distintos lugares.

Una vez explicadas las numerosas utilidades que proporciona el uso de *Posidonia* en conjunción con FortranMod, se va a mostrar cómo presenta la aplicación las funcionalidades que se han descrito en este apartado.

En primer lugar, la integración con GiD sigue uno de los principios básicos de ambos programas que es mantener siempre un comportamiento *user-friendly*, de forma que en la Figura 2.4 se observa que simplemente es una opción dentro del menú de FortranMod, de forma que la inclusión de *Posidonia* en el programa es sencilla y completamente mimetizada con el entorno de GiD.

El ejemplo que se propone en esta sección es el análisis de un filtro de cinco cavidades en guiaonda, cuya simulación requiere un consumo de recursos elevado para un ordenador personal de gama media.

De esta forma *Posidonia* es accesible para el usuario de una forma rápida una vez que haya configurado el problema, pudiendo elegir entre una simulación local, es decir, en el ordenador en el que se encuentra lanzado GiD y FortranMod, o mediante *Posidonia*, esto es, ejecutándolo de forma remota en el *cluster*. Los archivos de entrada que genera el programa son los siguientes:

- Archivo `<nombre_proyecto>.in`, que incluye los parámetros y opciones de ejecución del programa.
- Archivo `<nombre_proyecto>.msh`, en el que se determina el mallado del elemento a simular.
- Archivo `<nombre_proyecto>.bc`, que contiene las condiciones de contorno asociadas a la estructura bajo análisis.

Mientras que los ficheros de salida son, por otro lado:

- Archivo `<nombre_proyecto>.gi`, en el que se encuentra la solución asociada a los grados de libertad del problema, que se pueden relacionar con los puntos de la malla.
- Archivo `<nombre_proyecto>.gi.in`, que incluye los parámetros y opciones necesarias para el postprocesado del problema.
- Archivo `<nombre_proyecto>.gi.msh`, en el que se determina el mallado que se emplea en el postprocesado.
- Archivo `<nombre_proyecto>.gi.bc`, que contiene las condiciones de contorno, en el caso de que existan, para el postprocesado de la solución.

El uso de GiD consigue que el uso y manejo de estos archivos sea transparente para el usuario, ya que las entradas se usan para ejecutar la simulación y las salidas para mostrar los resultados de ésta; sin embargo, si se desea enviar el trabajo a un HPCC, se debe conocer la existencia de estos ficheros para enviarlos al *cluster*, generar el *script* y descargar los resultados, como ya se indicó al principio de este capítulo.

La inclusión de *Posidonia* en el programa supone que la ejecución del proyecto de GiD y FortranMod en un *cluster* también sea transparente para el usuario, de forma análoga a como lo es si se escoge simularlo de forma local; y no sólo esto, sino que la interfaz de *Posidonia* es lo más sencilla posible sin perder funcionalidad, como se muestra en la Figura 2.5. El usuario únicamente debe introducir su usuario y contraseña del HPCC en el que quiere ejecutar su proyecto (por defecto, `ash25b`) y presionar el botón `Submit job`.

Por otro lado, si se desean cambiar parámetros de configuración tales como:

- El número de recursos paralelos que puede consumir el proyecto, accesible mediante la opción `Number of MPI processes`.

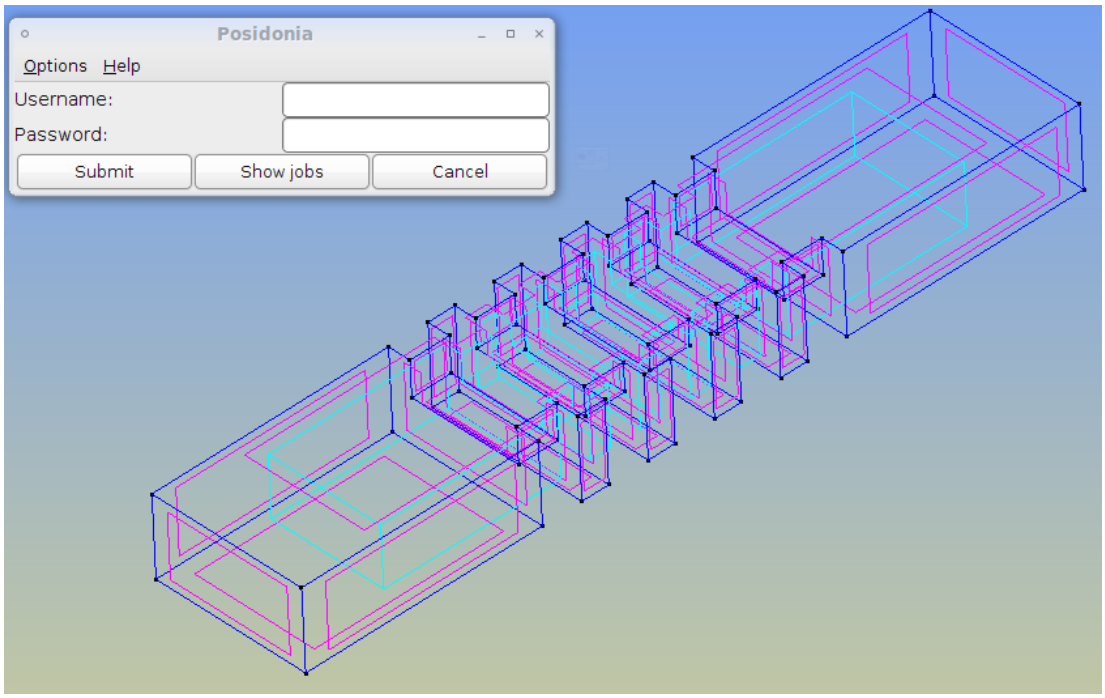


Figura 2.5: Pantalla principal de *Posidonia* en GiD.

- Notificación vía *email* a la dirección de correo especificada cuando el trabajo comience, acabe y/o sea abortado.
- La cola a la que se desea enviar el trabajo.
- El *host* en el que se encuentra el HPCC que, en el caso del escenario de la Figura 2.21, es `ash25b`.
- La máxima memoria que puede consumir el proyecto.

Únicamente hay que abrir el menú `Options/Advanced` como se presenta en la Figura 2.6. Se muestran los valores por defecto que son los que usará normalmente el usuario no experto.

Cabe destacar que el envío de los ficheros al *cluster* se hace de forma transparente al usuario por medio de un protocolo de comunicaciones que se establece entre el equipo local y el *cluster* que se detalla en el Capítulo 4, mostrando una notificación de escritorio cuando el trabajo haya sido recibido por el gestor de colas, tal y como se expone en la Figura 2.7.

Además, la transferencia de los ficheros se hace de forma segura gracias al protocolo SSH (véase [10]), estableciéndose una conexión segura con el *login* especificado

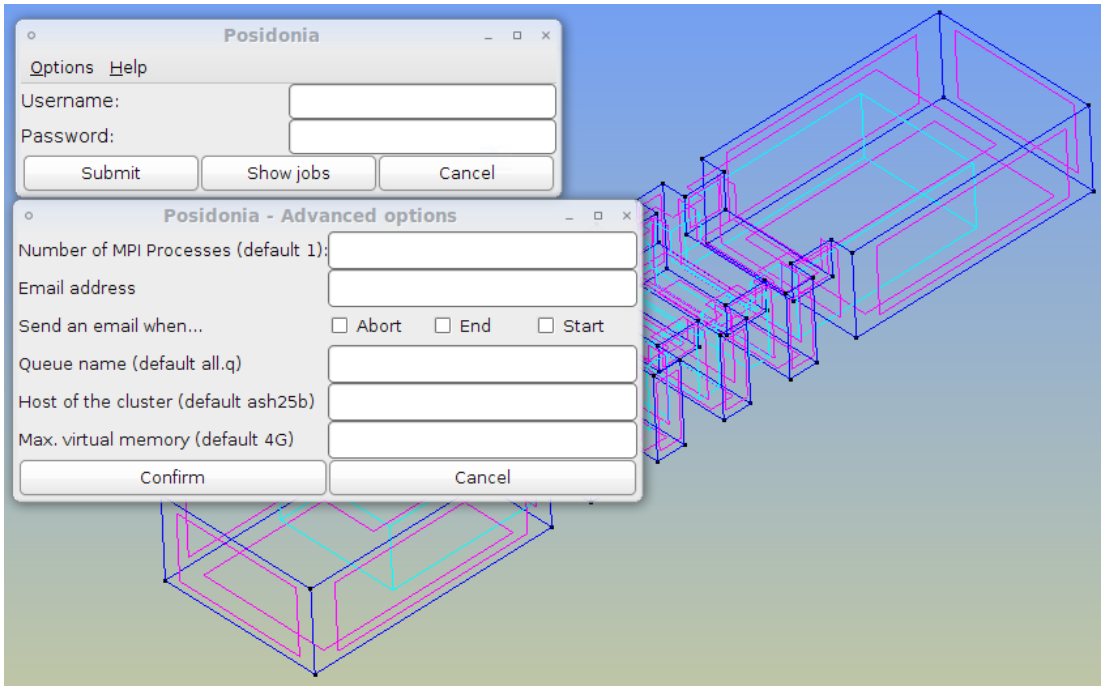


Figura 2.6: Opciones avanzadas de *Posidonia* en GiD.

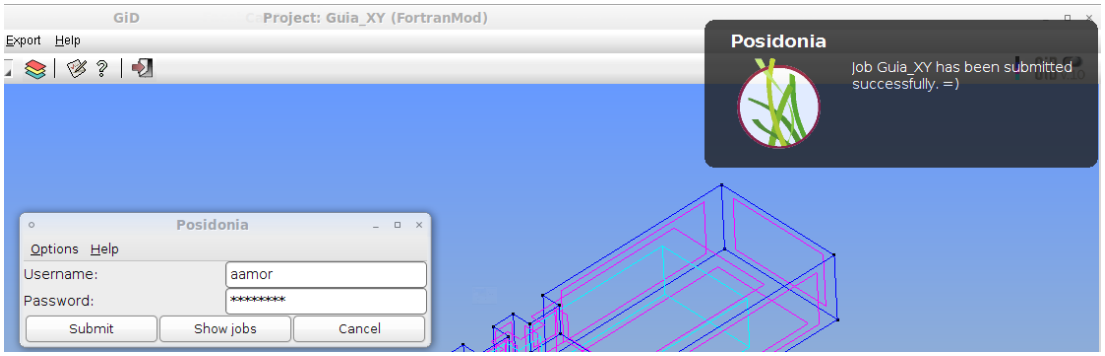


Figura 2.7: Notificación de recepción del trabajo en el *cluster*.

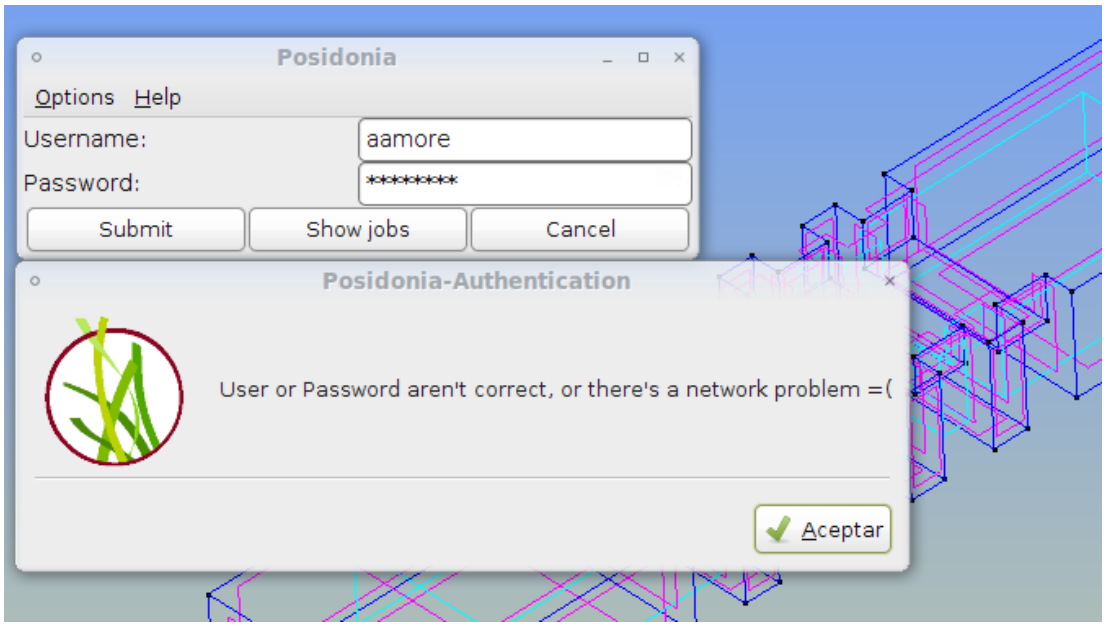


Figura 2.8: Error de autenticación en *Posidonia*.

por el usuario en la pantalla principal de *Posidonia*. Como es de esperar en una aplicación *user-friendly*, se muestra un mensaje de error en el caso de que se produzca un error de autenticación o no se encuentre disponible ninguna conexión a Internet como se observa en la Figura 2.8. Los detalles acerca de cómo se realizan estas comprobaciones se explican en el Capítulo 4.

Finalmente, cuando la ejecución del trabajo haya concluido, una notificación aparece en pantalla tal y como se muestra en la Figura 2.9, para lo que es necesario que un servidor TCP (o *Transmission Control Protocol*, véase [22]) esté activo como se explica en el Capítulo 4. Con esto, se descargan los ficheros de forma automática en la carpeta en la que aparecerían los archivos de salida si se simulara el proyecto de forma local, por lo que el análisis de los resultados es exactamente igual al que se hace en GiD sin *Posidonia*, lo que se puede ver en la Figura 2.10.

Con todo esto se ha observado el envío de un trabajo a un HPCC y la descarga automática de resultados, todo realizado de forma segura; sin embargo, *Posidonia* ofrece mucho más como es el control de los proyectos de GiD y FortranMod enviados por el usuario, para lo que éste debe introducir su *login* y presionar el botón `Show jobs`. Así aparece una nueva ventana en la que se muestra una tabla donde se encuentran los trabajos que actualmente están ejecutándose en el *cluster* como se expone en la Figura 2.11.

En esta pantalla se puede hacer un seguimiento de los trabajos que se están ejecutando actualmente, ya que aparecen datos tan relevantes como el estado de

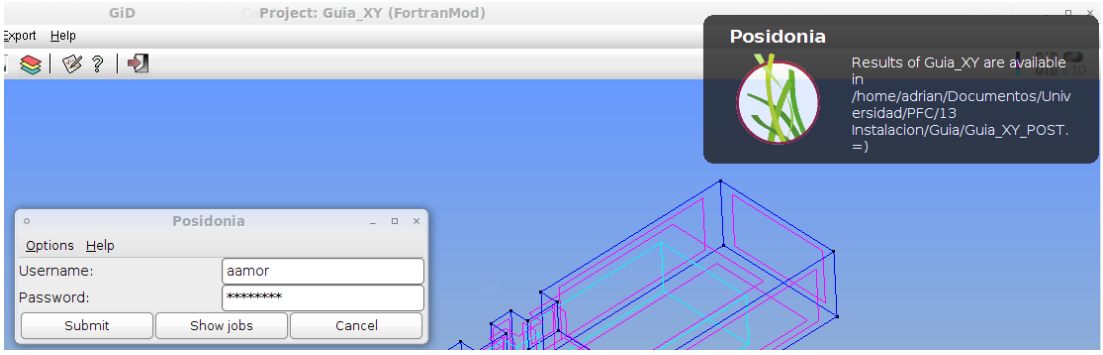


Figura 2.9: Finalización de un trabajo con *Posidonia* en GiD.

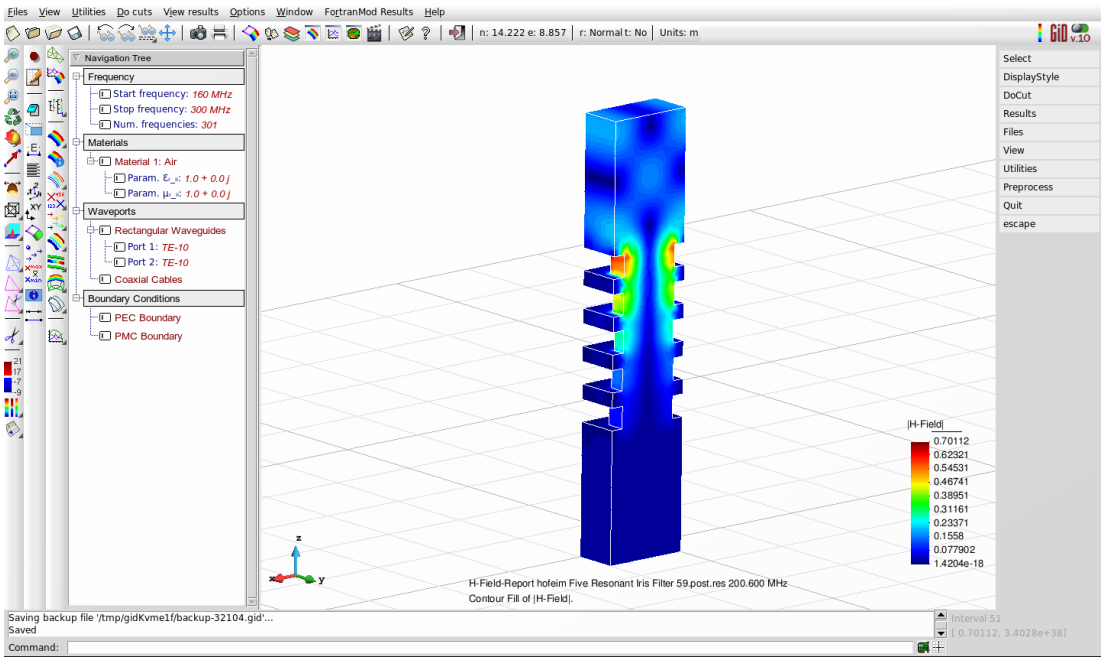


Figura 2.10: Análisis de resultados en GiD.

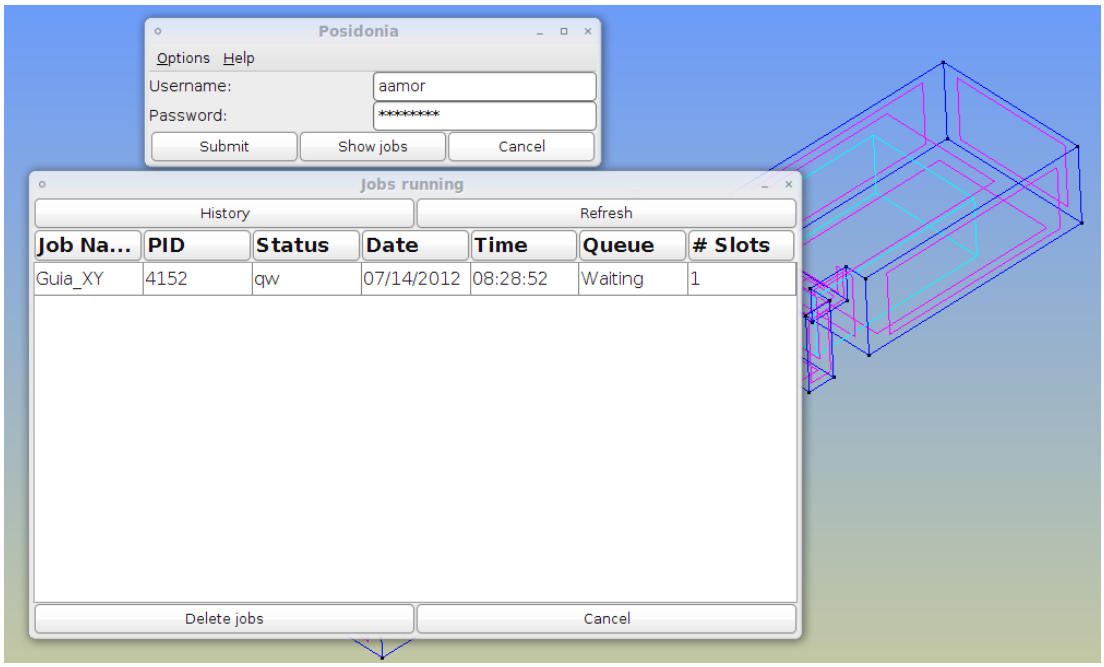


Figura 2.11: Monitorización y control de los trabajos en ejecución.

cada uno de ellos (si está en ejecución, si está esperando a ser asignado por el gestor de colas, etcétera), la fecha en la que se envió o su prioridad; en definitiva, toda la información que ofrece el gestor de colas sobre cada uno de ellos. Además, *Posidonia* también permite un cierto control sobre los trabajos enviados por el usuario ya que posibilita la eliminación de cualquiera de ellos, tal y como se muestra en la Figura 2.12.

También se debe destacar que la información de esta tabla no se actualiza en tiempo real, de forma que se deja al usuario que refresque la información cuando crea conveniente, de forma que se evita tráfico inútil en la red. Esta decisión se explicará pormenorizadamente en el Capítulo 4, junto con la forma de obtener y procesar esta información.

Por último, si se presiona el botón `History` en esta pantalla, aparece el histórico de los trabajos que ha enviado el usuario al *cluster* vía *Posidonia* y cuya ejecución ya ha finalizado, como se observa en la Figura 2.13.

En esta pantalla se tienen más opciones: por un lado, se puede eliminar un trabajo de este historial si se considera que ya no tiene ninguna utilidad, de forma que desaparece todo su rastro en el *cluster*; y por otro, se pueden descargar los ficheros de entrada y los de salida en el directorio indicado por el usuario tal y como se presenta en la Figura 2.14.

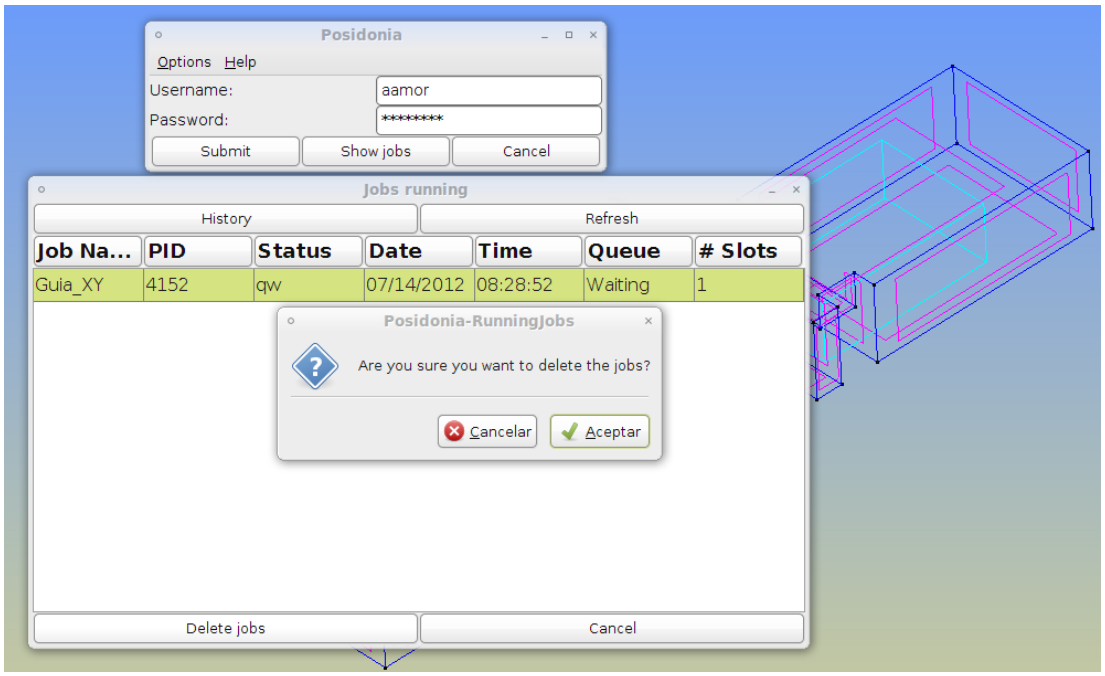


Figura 2.12: Eliminación de un trabajo en ejecución.

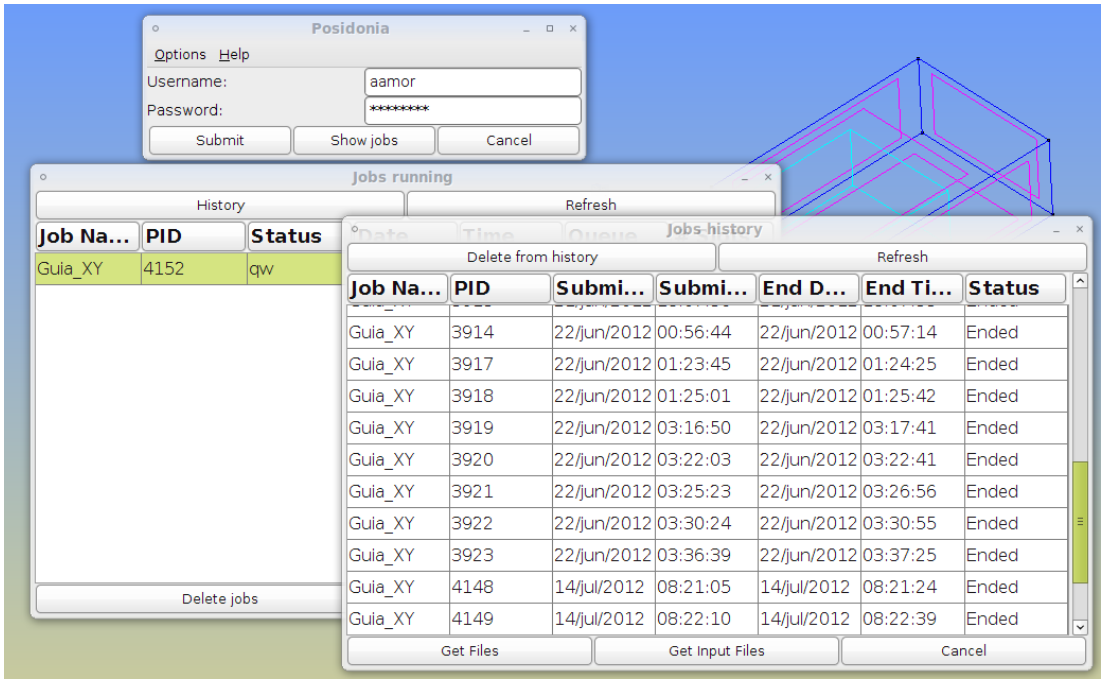


Figura 2.13: Historial de los trabajos ejecutados en el *cluster*.

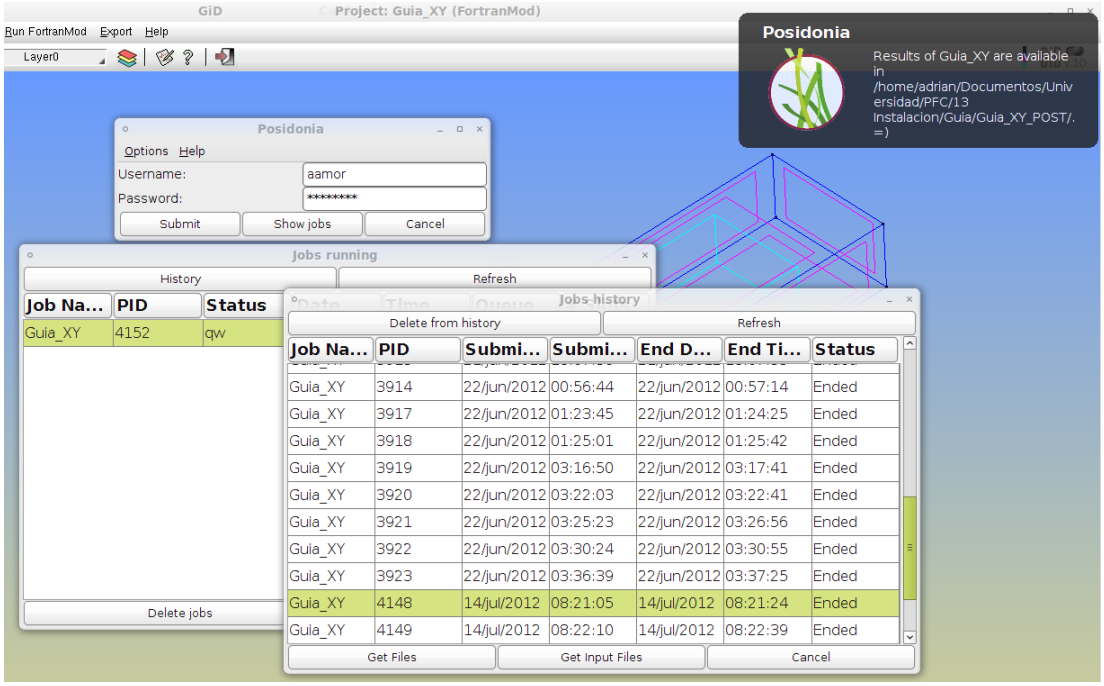


Figura 2.14: Descarga de ficheros del historial.

Así, cuando los ficheros han acabado de descargarse aparece una notificación similar a la que emerge al acabar la ejecución de un trabajo en el HPCC. De nuevo, esta tabla no se actualiza de forma automática para minimizar el tráfico circulante en la red lo que se explica, junto con la forma de adquirir y tratar la información, en el Capítulo 4.

En conclusión, en esta sección se ha mostrado cómo se ha efectuado la integración de *Posidonia* en una interfaz como GiD con una potente herramienta de simulación como es FortranMod y cómo mejora notablemente las prestaciones de este programa, avanzando en los principios que se marca GiD y mostrando la flexibilidad y fácil adaptabilidad de *Posidonia*.

2.4. Aplicación autónoma con Matlab.

En esta sección se va a detallar cómo se implementan las funcionalidades que se describieron en el Apartado 2.2 en el caso de la ejecución en el *cluster* de códigos Matlab. En concreto, se va a continuar con el mismo ejemplo que se detalló en el Apartado 2.1 y se van a observar cómo se manifiestan las principales características de *Posidonia* en el programa desarrollado.

En primer lugar, el simple hecho de tener la misma aplicación para entornos de programación completamente distintos es fiel reflejo de la extensibilidad y generalidad con la que se ha desarrollado el proyecto: como se detallará en capítulos posteriores, el código se hace de forma general para que las dos aplicaciones hasta ahora descritas reutilicen la mayor parte de código posible. Como es lógico y veremos en este apartado, hay características particulares que se reflejan en la interfaz gráfica y que diferencian a ambas aplicaciones, pero la aplicación es, en esencia, la misma.

Por otro lado, también se procura que la apariencia de la aplicación esté lo más adaptada posible a la del sistema en la que se encuentra. Así, como se observa en la Figura 2.15, *Posidonia* adquiere el actual tema de escritorio y es el que usa para presentar la información. Esta figura también sirve para mostrar el carácter multiplataforma del programa al presentarse a la izquierda la aplicación en Windows y a la derecha la aplicación en dos temas distintos de Linux. Esto es facilitado gracias a la programación en Java como ya se ha explicado en apartados anteriores.

En la Figura 2.15 también se muestra la sencillez de *Posidonia*, ya que sus campos se reducen a la mínima expresión: por una parte, permite el fácil acceso a las opciones avanzadas del trabajo, como aparece en la Figura 2.16 y, por otra, las principales funcionalidades se agrupan en dos botones requiriendo cada una de ellas la autenticación indicada en los campos de texto.

Es importante notar que las ventajas que se tienen en GiD con el envío de las simulaciones al *cluster* son prácticamente las mismas que se consiguen con Matlab. Hay códigos que son muy intensivos y cuya ejecución requieren horas e incluso días; por ello, además de la movilidad que ofrece *Posidonia* (que se ve notablemente acentuada con el desarrollo de la aplicación Android como se verá en la siguiente sección) se pueden lanzar tareas muy pesadas computacionalmente y seguir trabajando con normalidad en el equipo personal.

También cabe destacar que, dada la generalidad que se ha buscado en este proyecto, la aplicación contiene las mismas funcionalidades que se detallaron en la sección anterior; sin embargo, hay diferencias fundamentales que se hacen patentes en la interfaz:

- La aplicación es autónoma, es decir, que no se encuentra integrada en un entorno de Matlab. Esto conlleva que lo que era antes transparente al usuario (ficheros de entrada y de salida) ahora ya no lo es, por lo que hay que especificar ciertas opciones que se explicarán más adelante.

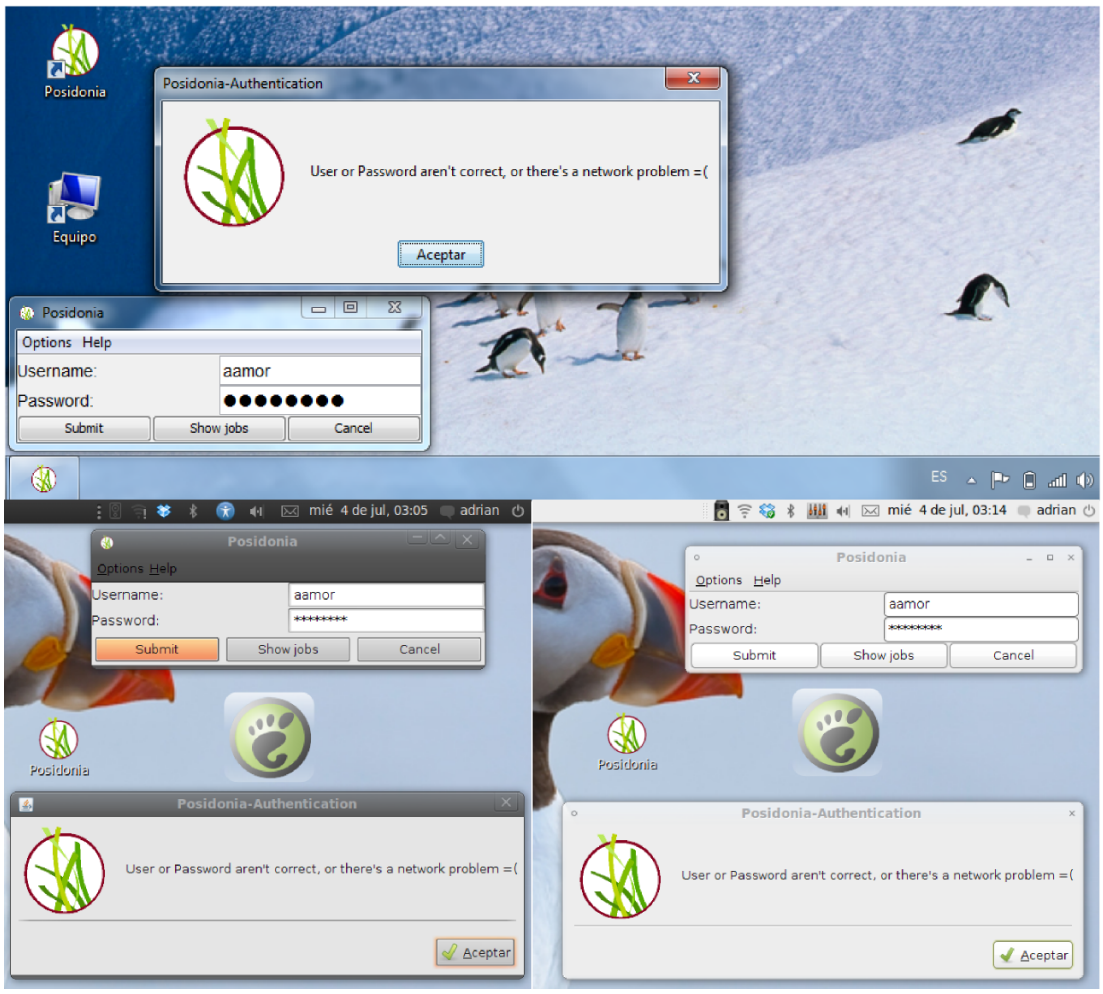


Figura 2.15: Distintas apariencias para un mismo interfaz.

- Los resultados no son previsibles como en GiD, en el que se sabía que el simulador generaba cuatro archivos de salida y el gestor de colas generaba el archivo `.o`. Ahora los resultados son, en general, ficheros `.mat` como contenedores de los resultados de la ejecución; sin embargo, también pueden aparecer otros tipos de archivos como imágenes, tal y como ocurre en el ejemplo que se va a seguir en esta sección.

Todo esto produce que se tenga que añadir una ventana como la que se presenta en la Figura 2.16 en la que se le pide al usuario que especifique información relativa a su proyecto en Matlab:

1. El nombre del trabajo. Se recomienda que sea descriptivo para el usuario para que sepa identificar qué proyecto se está ejecutando. Es indiferente que

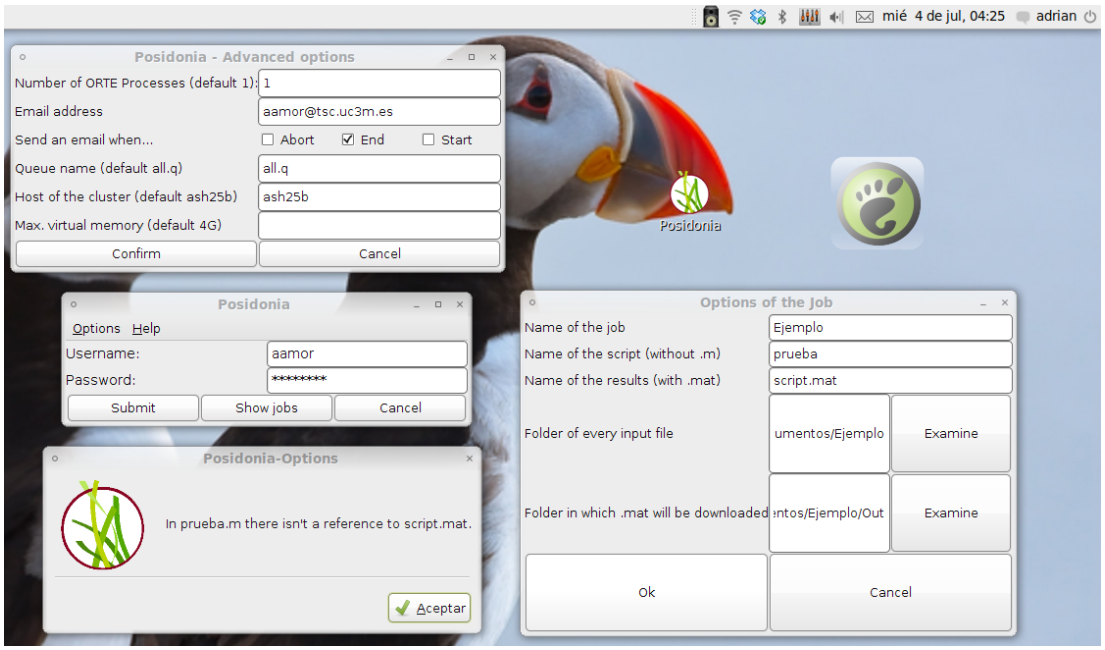


Figura 2.16: Envío de un trabajo de Matlab con error en el contenedor de salida.

se llame igual que otra tarea mandada con anterioridad, puesto que *Posidonia* usa identificadores unívocos tal y como se explicará en el Capítulo 4 y en la interfaz cada nombre viene acompañado por la fecha y hora en la que se envió. Por otra parte, se comprueba que no incluya caracteres especiales tales como /, *, ·, ñ...

2. El nombre del script. Aquí se debe indicar el script de Matlab que se va a ejecutar; para evitar pequeños fallos, se explora el directorio de entrada en el que el usuario indica que está el archivo para ver si realmente existe.

3. El nombre del contenedor en el que se guardan las variables y sus valores en la ejecución del script. *Posidonia* no necesita este nombre para funcionar correctamente; sin embargo, se fuerza al usuario a que indique su nombre para que tenga almacenados los resultados del script porque, de otra forma, toda la ejecución se pierde. De la misma forma, se comprueba que en el script indicado hay una referencia a este contenedor evitando así, sobre todo, fallos tipográficos. Al igual que ocurre con los otros posibles errores, se muestra por pantalla un mensaje instando al usuario a que corrija la información aportada, como se observa en la Figura 2.16.

4. El directorio en el que se encuentran todos los archivos de entrada. Su elección se hace por medio de un *file chooser*, o seleccionador de archivos, lo que es más cómodo tanto para el usuario, que no tiene que teclear ninguna

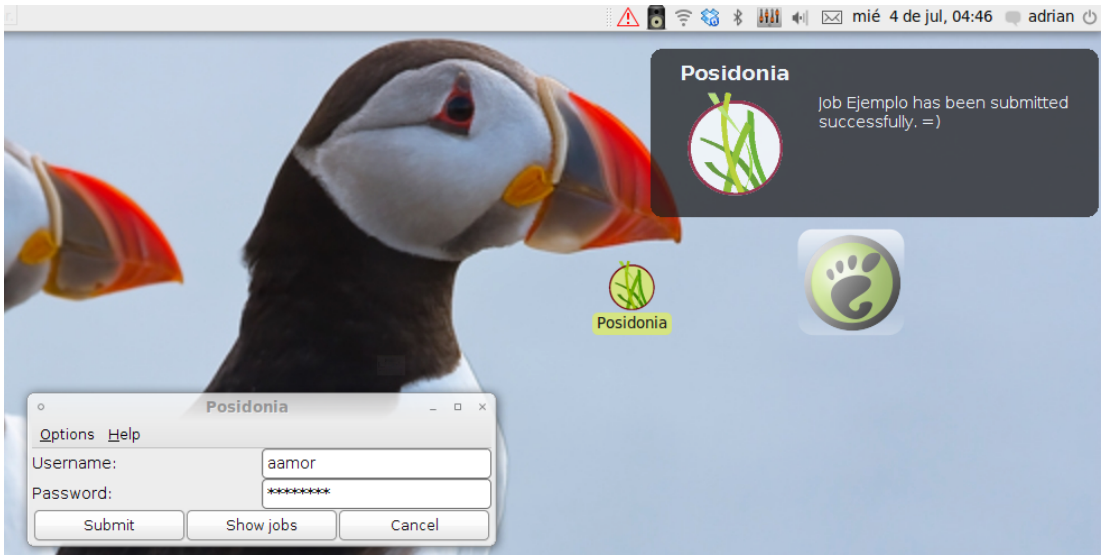


Figura 2.17: Llegada del trabajo al *cluster*.

ruta, como para la aplicación, que evita posibles fallos relacionados con la no existencia del directorio.

5. El directorio en el que se desean descargar los resultados si la ejecución acaba antes de cerrar *Posidonia*. Esta información se indica de la misma forma que se aporta en el caso del directorio de los ficheros de entrada.

Como se puede observar, los datos solicitados son los mínimos necesarios para poder lanzar la ejecución. Esto es clave para que el envío de trabajos al *cluster* sea confortable, sencillo y, sobre todo, rápido.

Si se contemplan todos los campos correctamente, se procede al envío del proyecto al HPCC y, cuando éste ya ha sido recibido y procesado de forma satisfactoria, una notificación aparece por pantalla, como se muestra en la Figura 2.17.

A partir de este punto, las funcionalidades son las mismas que se explicaron en la sección anterior. Para no repetir los mismos conceptos, se va a hacer un seguimiento de un trabajo enviado observando cuáles son las principales diferencias con respecto al Apartado 2.1.

Una vez enviado el proyecto Matlab, si se presiona en la pantalla principal el botón `Show Jobs` aparecerá el trabajo que se está ejecutando, tal y como se presenta en la Figura 2.18. Los cambios son drásticos con respecto a lo que se tiene sin la inclusión de *Posidonia*: además de que los archivos de entrada y de

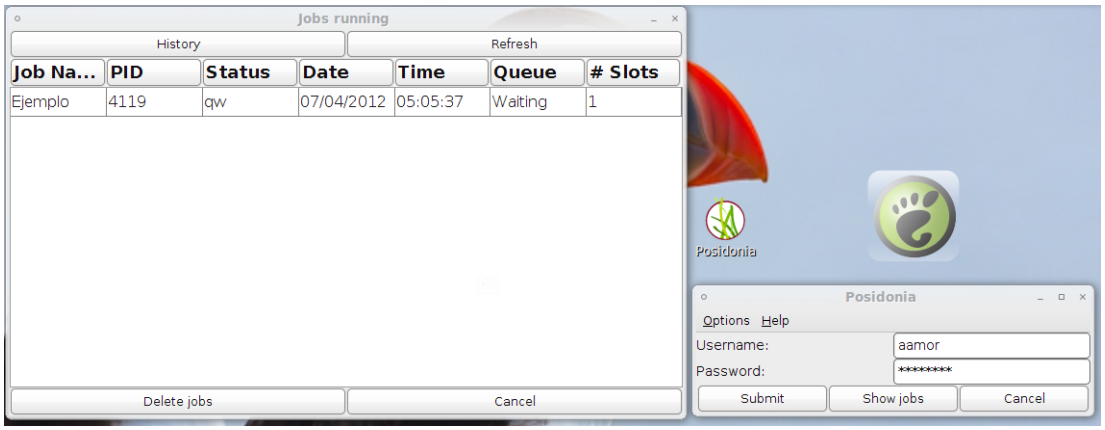


Figura 2.18: Tabla de tareas enviadas y aún ejecutándose.

salida son enviados y descargados automáticamente, la presentación del estado de los trabajos enviados es notablemente mejor a la que se obtiene por consola de comandos. Por otra parte, se muestra una notificación como la que aparece en la Figura 2.19 cuando uno de los proyectos en ejecución ha finalizado, y se descargan automáticamente los resultados a la carpeta indicada por el usuario; además, si se pulsa en la notificación mientras ésta aún está activa, se lanza un explorador de archivos en el directorio en el que se han descargado las salidas.

De nuevo, como ocurría en la interfaz integrada en GiD, es necesario que un servidor TCP esté activo para la recepción de estos avisos, algo que se lleva a cabo automáticamente tal y como se expone en el Capítulo 4.

Para finalizar, se va a mostrar el historial de proyectos que ha enviado el usuario y ha decidido mantener en el repositorio. En la Figura 2.19 aparecen las tareas que ya han acabado su ejecución, teniendo las mismas opciones que se indicaron en la sección anterior. De esta forma, en la Figura 2.20 se muestra cómo eliminar un trabajo del repositorio (se pide confirmación como es habitual al tratarse de una decisión irreversible) y la ventana en la que el usuario selecciona dónde desea descargar los ficheros, ya sean de entrada (presionando `Get Input Files`) o de salida (pulsando `Get Files`). Cuando los archivos estén disponibles en el sistema de ficheros local, se lanza una notificación similar a la que se mostró en la Figura 2.19 con las mismas funcionalidades que ya se explicaron.

Con todo esto se ha explicado brevemente cuáles son las principales características de *Posidonia* en su versión de aplicación autonóma con proyectos de Matlab. Se han observado las grandes ventajas que aporta su utilización y que ya se han ido desgranando a lo largo de este capítulo, haciendo el *cluster* mucho más accesible para un gran número de usuarios y el envío y control de trabajos un proceso mucho más rápido y sencillo.

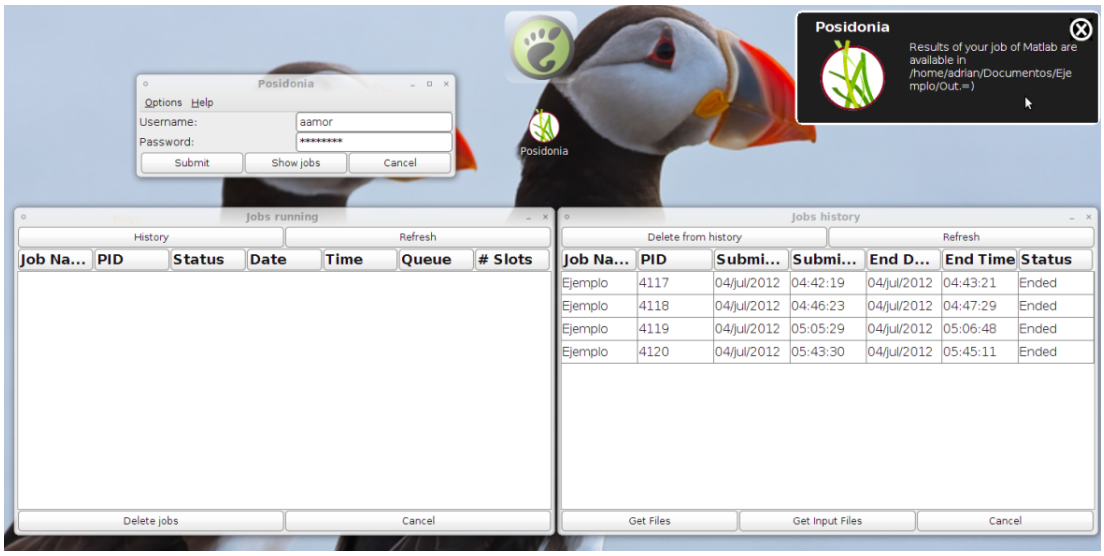


Figura 2.19: Histórico de trabajos y finalización de uno de ellos.

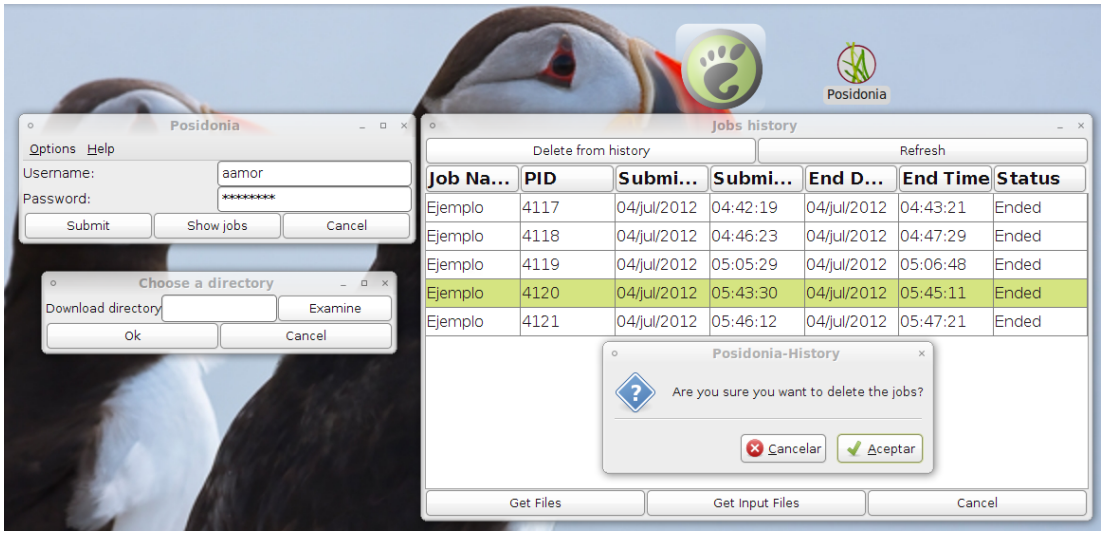


Figura 2.20: Eliminación de un trabajo en el repositorio y descarga de ficheros.

2.5. Aplicación en Android de *Posidonia*.

Finalmente, para cerrar el capítulo se va a detallar la última de las interfaces que se han planteado de *Posidonia*: una aplicación en Android que mantiene la mayoría de las funcionalidades de la aplicación autónoma con Matlab y que vuelve a demostrar la extensibilidad y generalidad del código desarrollado e incide notablemente en la movilidad que busca el proyecto.

Es interesante el desarrollo de una aplicación de este tipo dado la increíble expansión de dispositivos móviles como *smartphones*, *tablets*... que se ha venido produciendo en los últimos años. De esta forma, se implementa la práctica totalidad de las funcionalidades que se han presentado en secciones anteriores aunque lo que se plantea en un principio es, únicamente, el control y seguimiento de las tareas en ejecución del *cluster*.

Ya se explicó en el Apartado 2.2 que Android está basado en Java y, por ello, la adaptación es más sencilla al usar el mismo lenguaje. Sin embargo, hay una serie de diferencias notables a la hora de desarrollar esta aplicación:

- Pese a que el lenguaje es el mismo, el entorno de programación cambia casi por completo: hacer una interfaz gráfica para un dispositivo Android difiere bastante frente a usar las librerías habituales de Java.
- Cuando se ha desarrollado la aplicación no era tan sencillo realizar una conexión VPN por medio de Android por diferentes motivos: en primer lugar, en el departamento es habitual el uso de OpenVPN como ya se explicó anteriormente y esta aplicación aún no está disponible para este tipo de equipos; y, por otro lado, se tiene en cuenta un escenario en el que el dispositivo se mueva entre distintas conexiones a Internet, algo que dificulta el establecimiento de una VPN.

  Así, la aplicación desarrollada se encuentra en un entorno más específico aún que el de la Figura 2.2; en concreto, el que se muestra en la Figura 2.21. El *cluster* se encuentra en el dominio del departamento de TSC de la universidad y no se encuentra accesible fuera del dominio por lo que, si no se dispone de una VPN, para poder entrar a él hay que usar un servidor dedicado con salida al exterior que actúe como pasarela.

  De esta forma habrá que efectuar un "doble salto" que no afectará a la funcionalidad de la aplicación pero sí a su programación y a la interfaz: hay que hacer pasar los archivos y los comandos por el servidor, por lo que hay que autenticarse tanto en él como en el *cluster*. Este escenario tan particular se detallará en el Capítulo 4 en el que se verá la complejidad adicional que supone para la aplicación este entorno.

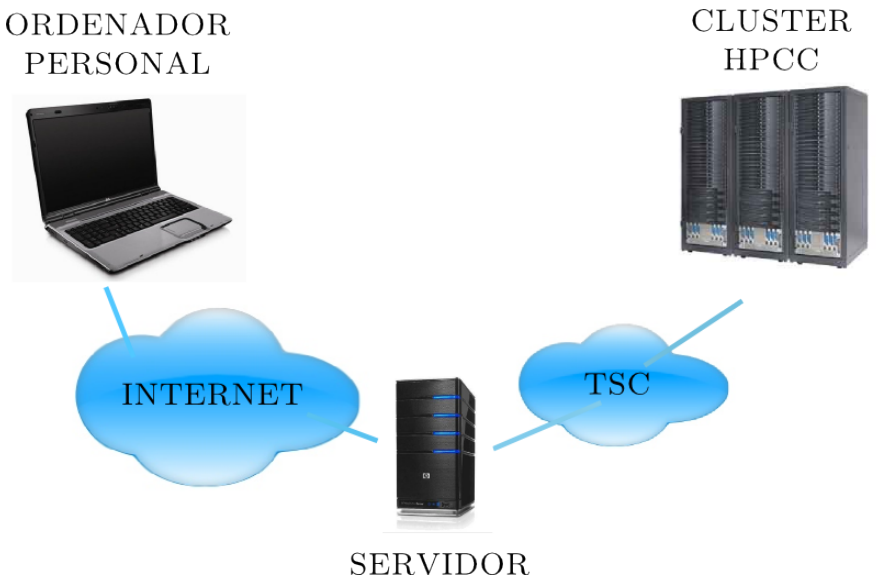


Figura 2.21: Escenario considerado sin VPN.

- Debido a la imposibilidad de usar una VPN para la conexión al *cluster*, tampoco es posible levantar un servidor TCP que sea accesible desde `ls6`, de forma que las notificaciones que se reciben en las aplicaciones de escritorio no pueden llegar al dispositivo Android.

  Esto se debe a la configuración del acceso a Internet: en general, al conectarse por WiFi o por redes móviles y no estar en una VPN, depende del operador la decisión de dejar pasar el tráfico hacia puertos no registrados como el de *Posidonia*.

  Por este motivo, la única funcionalidad que no se mantiene con respecto a los programas anteriores es la notificación de que el trabajo ha finalizado; todas las demás sí se mantienen como se verá a continuación.

De aquí en adelante se van a comentar brevemente las características de la aplicación desarrollada. En primer lugar, la integración con la plataforma Android es total, como es lógico: hasta ahora, se ha insistido en el carácter multiplataforma de *Posidonia*; sin embargo, en dispositivos móviles no se puede hacer nada parecido de momento y hay que elegir el sistema en el que se quiere lanzar la aplicación. Se ha escogido Android por su expansión imparable en los últimos años, tal y como se muestra en [23], y por basarse en Java que es el lenguaje en el que se han desarrollado los programas de escritorio.

De esta forma, la integración con el sistema Android es total, como se observa en la Figura 2.22. La instalación de la aplicación es por medio de un archivo `apk`

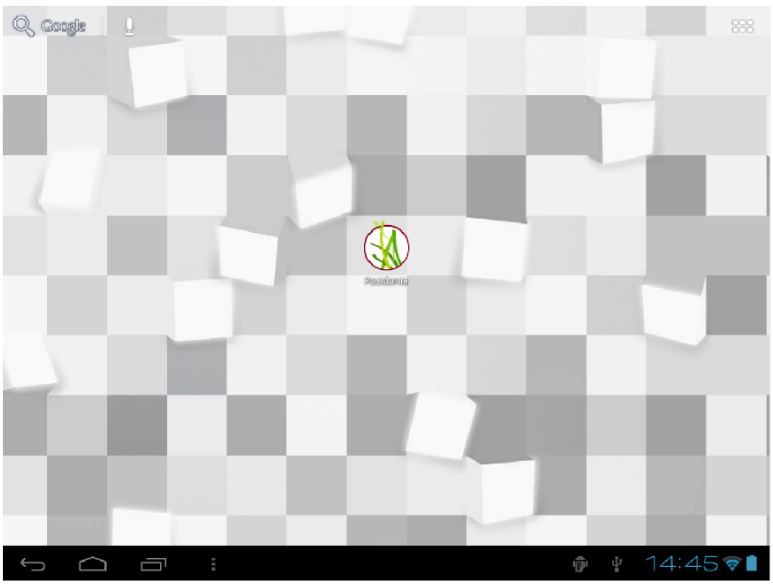


Figura 2.22: Integración de *Posidonia* en Android.

como los que se descargan en el Market de Android, por lo que se hace de forma muy sencilla, y pulsando en el icono se lanza la pantalla principal de la aplicación que es la que aparece en la Figura 2.23.

Se puede observar lo que ya se anticipaba anteriormente: es necesario hacer un "doble salto", primero a `ls6` y luego al *cluster* en este caso concreto, y eso influye en la interfaz al necesitarse los datos de autenticación para ambos equipos. Esto puede llegar a ser algo pesado, y lo que se ha procurado en todo momento a lo largo de este proyecto es que el envío y control de trabajos se haga de forma rápida, por lo que existe la opción de guardar los actuales datos de autenticación de forma que se introduzcan una vez al iniciar la aplicación y no se vuelvan a introducir hasta que sea desinstalada.

Si se presiona el botón `Submit Job`, se llega a una pantalla similar a la que se tenía en la aplicación autónoma de *Posidonia*: es necesario que el usuario dé una serie de datos relacionados con el trabajo para que la ejecución se haga de forma correcta. Así, hay que indicar dónde se encuentran los archivos en el dispositivo Android (por medio de un explorador de archivos, como el que se muestra en la Figura 2.25), con qué nombre queremos que se identifique el proyecto, etcétera. Así, las comprobaciones que se hacían en la aplicación de escritorio se hacen de la misma forma aquí, tal y como se ve en la Figura 2.24.

Por otro lado, no es habitual que un usuario se dedique a hacer un código en Matlab en un dispositivo Android, por lo que el envío de trabajos no es, a priori,

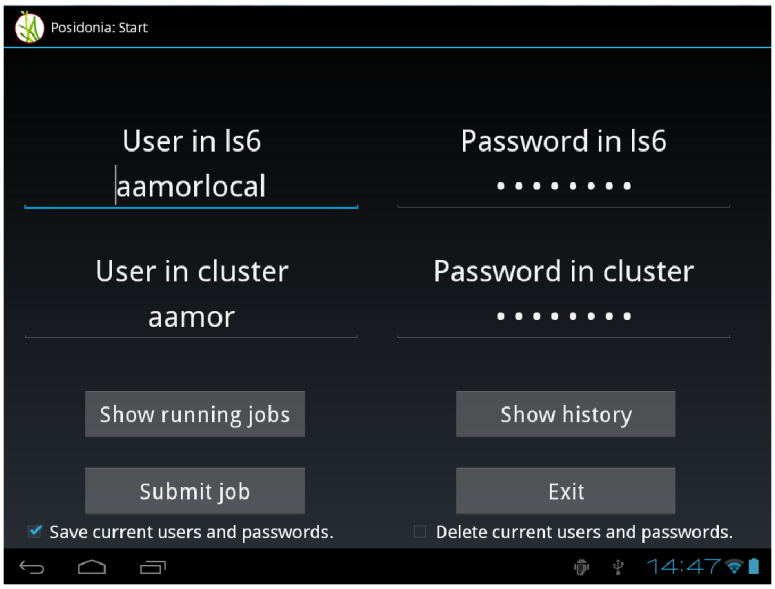


Figura 2.23: Pantalla principal de *Posidonia*

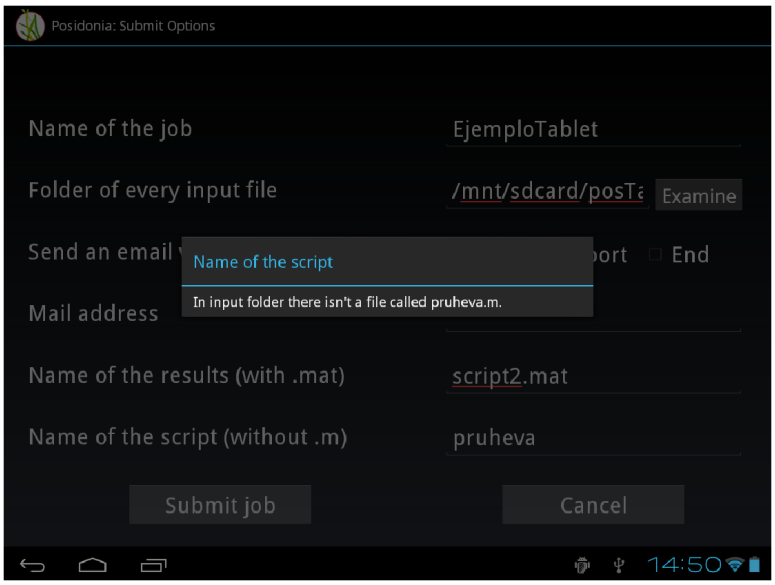


Figura 2.24: Error en las opciones del trabajo de Matlab.

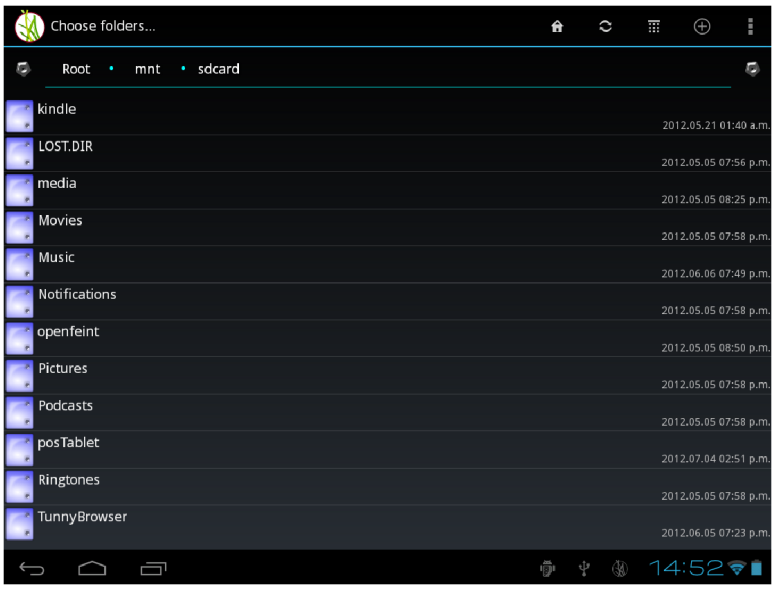


Figura 2.25: Selección del directorio en el que se encuentran los ficheros de entrada.

muy útil. Hay dos argumentos de suficiente peso en contra de esta afirmación: en primer lugar, la evolución de estos equipos móviles (como por ejemplo, en las *tablets*) hace pensar que será cada vez más habitual que se trabaje sobre este tipo de dispositivos; y, en segundo lugar, dada la funcionalidad de repositorio que la aplicación incluye, se pueden visualizar los resultados de la ejecución (como, por ejemplo, gráficas) y, si no son los esperados, descargar los ficheros de entrada, modificar pequeñas partes de código y volver a lanzar el trabajo desde el dispositivo móvil sea cual sea el lugar en el que se esté, siempre y cuando se tenga una conexión a Internet.

De esta forma, no es necesario instalar en el equipo móvil un programa tan pesado como Matlab, sino que simplemente con los datos de autenticación necesarios y la aplicación instalada tenemos la potencia de un programa como este en un *cluster* de altas prestaciones por medio de Internet, de una forma rápida y sencilla.

Así, si se rellenan correctamente las opciones del código en Matlab que se desea ejecutar, el trabajo es enviado al *cluster* pasando por `ls6`, algo que es transparente para el usuario y que se detallará en el Capítulo 4. De igual forma a como se mostró en secciones anteriores, cuando el trabajo ha llegado al gestor de colas del *cluster*, una notificación es mostrada en la barra de tareas de Android (lo que demuestra, de nuevo, la integración plena con el sistema, ya que en las aplicaciones de escritorio la notificación es una ventana propia de *Posidonia*) que, si se pulsa, se accede a la tabla de trabajos en ejecución para poder monitorizar su estado,

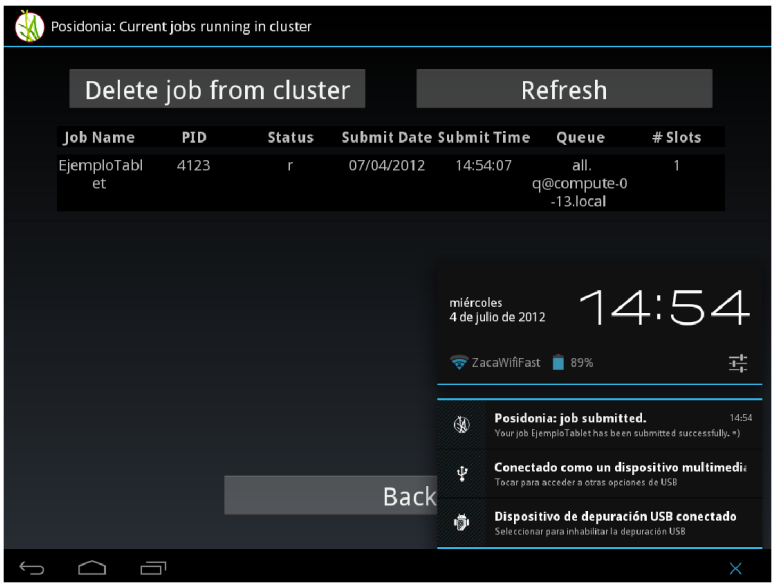


Figura 2.26: Notificación de llegada del trabajo al gestor de colas y tabla de tareas en ejecución.

como se muestra en la Figura 2.26. Esta ventana también es accesible por medio del botón de la pantalla principal `Show running jobs`.

Finalmente, cuando acabe el trabajo no se recibirá ninguna notificación integrada en Android, aunque si se desea se puede configurar el correo electrónico en la pantalla de la Figura 2.24 para que un mail sea enviado cuando el trabajo haya finalizado.

De esta manera, si se presiona en la ventana principal el botón `Show history`, se accede al histórico de los trabajos enviados por el usuario por cualquiera de las dos aplicaciones, la de escritorio o la de Android. Y es aquí donde reside la movilidad llevada en su grado más extremo: se pueden enviar trabajos desde un ordenador personal por medio de la aplicación de escritorio y se verán en el dispositivo Android, hayan acabado (si es así, en el historial) o no (en ese caso, en los archivos en ejecución), y viceversa. De esta forma, se tiene acceso a toda la información que puede ofrecer el gestor de colas en cualquier lugar siempre y cuando se tenga conexión a Internet, y si el control de trabajos era algo pesado en un ordenador personal (conexión SSH y conocimiento de comandos de SGE, tal y como se detalló en el Apartado 2.1), en un dispositivo móvil esa característica se acentúa, por lo que es conveniente el desarrollo de una aplicación como *Posidonia*.

Esta última característica del proyecto desarrollado está directamente relacionada con el *cloud computing*, o computación en la nube (si se desea más información se

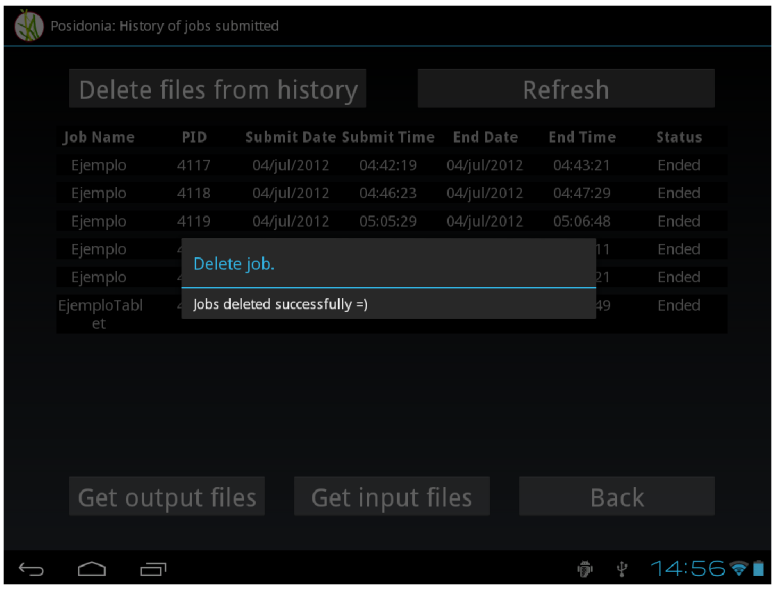


Figura 2.27: Tabla de tareas enviadas por el usuario y ya finalizadas, y borrado de una de ellas.

puede consultar [24, 1]), que es una tendencia en auge en el sector de la informática y las telecomunicaciones con numerosos proyectos de las empresas más punteras como Apple (véase [25]) o como Google (como aparece en [26]), tal y como se comparó en el capítulo anterior.

Por otra parte, las funcionalidades que se tenían en esta ventana en la aplicación autónoma con Matlab se mantienen en Android, de forma que se pueden eliminar trabajos del historial, como se muestra en la Figura 2.27, o descargar los archivos de entrada y de salida, como se hace en la Figura 2.28. En este caso, sí se muestra una notificación cuando los ficheros ya han sido descargados, y si se pulsa en ella, aparece un explorador de archivos en el que se pueden visualizar los resultados, o los ficheros de entrada.

Como se persigue a lo largo del proyecto, se busca en todo momento la comodidad del usuario de forma que la visualización de los resultados se puede hacer de forma directa desde *Posidonia*. Así, si se utiliza el mismo ejemplo que se ha venido siguiendo en este capítulo, al seleccionar uno de los ficheros de imagen resultado de la ejecución se abre la galería, como se presenta en la Figura 2.29.

También se ha de destacar que se controla que haya conexión a Internet cada vez que se necesite acceder a un recurso de red, es decir, cada vez que se realice una conexión por medio de SSH o SCP. Como se muestra en la Figura 2.30, un mensaje de error aparece en el caso de que no esté disponible el acceso a la red.

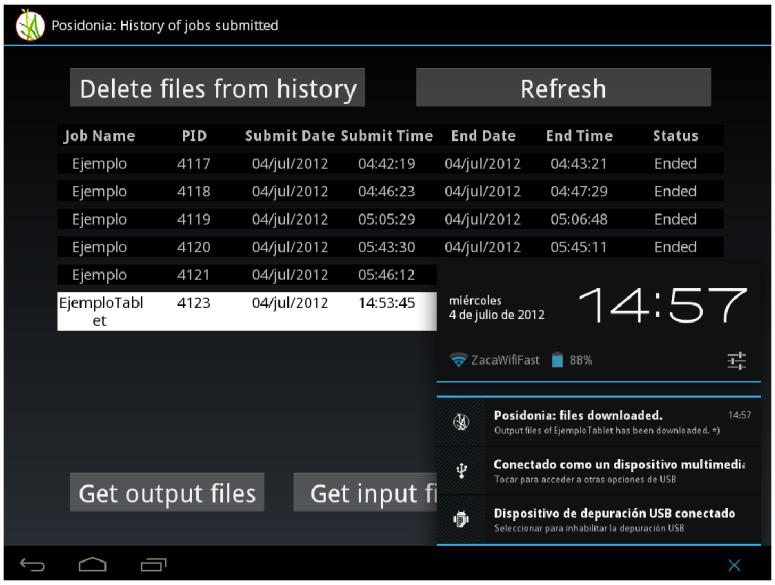


Figura 2.28: Descarga de los ficheros de salida del trabajo seleccionado, y notificación.

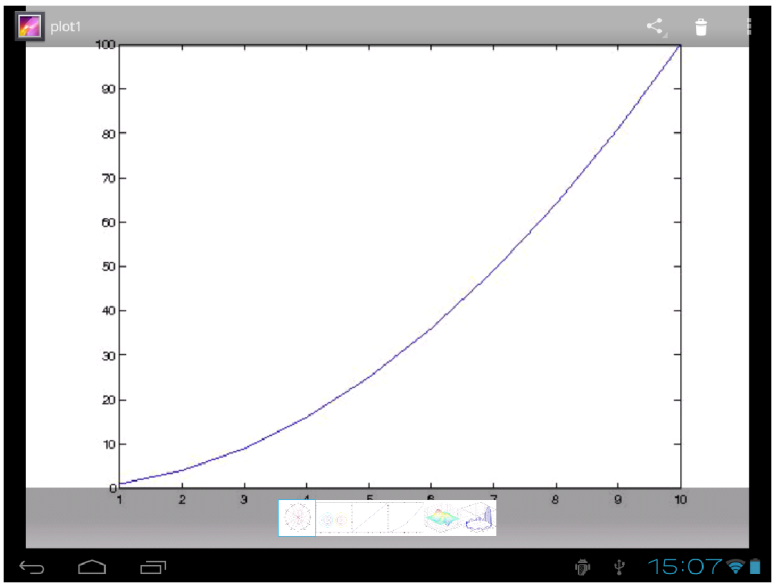


Figura 2.29: Visualización de los ficheros de salida.

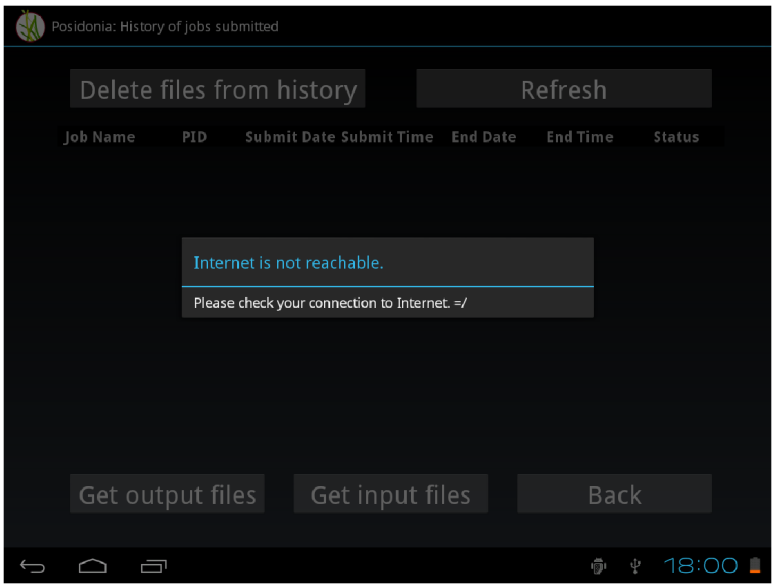


Figura 2.30: Error lanzado al no estar disponible la conexión a Internet.

Para el desarrollo de esta aplicación se han usado librerías externas de Android como son un *file chooser*, o seleccionador de archivos, que es el que se indica en [27], y un explorador de archivos que permite la visualización directa de los ficheros de entrada y de salida llamado *Blackmoon File Browser* (véase [28]).

En conclusión, en este capítulo se han hecho patentes las características del proyecto desarrollado, tanto su flexibilidad y extensibilidad (hay tres versiones distintas de *Posidonia* que se basan en las mismas librerías, como se detallará en los siguientes capítulos) como la movilidad que ofrece gracias al desarrollo de aplicaciones en dispositivos móviles como Android y a la característica de funcionamiento de *Posidonia* que, como hemos visto, no almacena datos localmente y accede a ellos de forma remota. Todo esto se consigue manteniendo las principales funcionalidades de un gestor de colas, la sencillez de la aplicación, la seguridad en las comunicaciones y la eficiencia en la red, evitando el envío de mensajes inútiles.

En posteriores capítulos se detallará cómo se han implementado las funcionalidades que se han presentado a lo largo de este epígrafe y las principales decisiones que se han tomado a la hora de programar esta aplicación.

Capítulo 3

Ejecución de trabajos en un *cluster* mediante Java.

Una vez explicada la funcionalidad de *Posidonia* en el Capítulo 2, en las siguientes secciones se detallará cómo se estructura la aplicación y cómo se implementa. En la Figura 3.1 se muestra un esquema simplificado de las principales partes en las que se divide el código desarrollado, centrándose este epígrafe en la parte resaltada de la figura.

De esta forma, el primer hito que debe afrontar este proyecto es la ejecución de trabajos en un *cluster* por medio de Java ya que, como se ha indicado en el Capítulo 2, la aplicación se codifica enteramente en Java por su orientación a objetos y su carácter multiplataforma, entre otras muchas razones.

Con este objetivo se ha hecho uso de un API (*Application Programming Interface*) que proporciona una plataforma para la ejecución de trabajos por medio de Java: este API se denomina DRMAA, o *Distributed Resource Management Application API* (véase [29]), que permite la ejecución y control de trabajos en el gestor de colas utilizado, que, en este caso concreto, es SGE aunque se observará que se puede adaptar fácilmente a otros tipos de gestores.

Es de destacar que la base de DRMAA es que todo lo que se puede controlar mediante los comandos nativos del gestor de colas se puede hacer por medio de este API. De esta forma, se mantiene toda la funcionalidad que se puede conseguir por medio de la consola de comandos pero con programas Java, lo que aporta grandes ventajas como es la integración total con el resto de *Posidonia* ya que el programa está enteramente codificado en este lenguaje.

Así, en este capítulo se va a detallar:

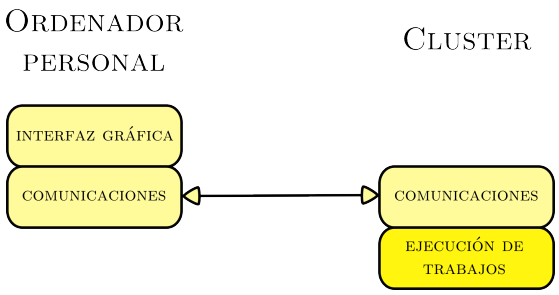


Figura 3.1: Estructura simplificada del código de *Posidonia*.

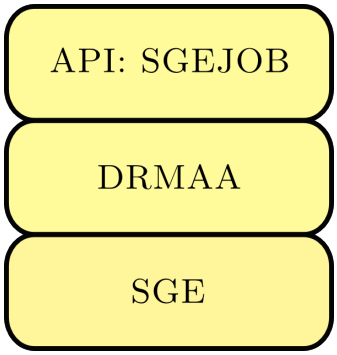


Figura 3.2: Capas de la ejecución de trabajos en un *cluster* mediante *Posidonia*.

- Cómo se realiza la ejecución de un trabajo por medio de las herramientas propias del gestor de colas.
- Cómo se lleva a cabo la ejecución de un trabajo por medio de DRMAA, y los problemas que han surgido para ello.
- La definición de un API intermedio para corregir los contratiempos que aparecen en DRMAA.

Como se observará a lo largo del capítulo, la forma de estructurar la ejecución de trabajos de la Figura 3.1 conduce, análogamente a cómo se estructuran las pilas de protocolos, a la pila de la Figura 3.2. Así, cada sección del capítulo se corresponde con una de las capas mostradas en la figura.

3.1. Herramientas del gestor de colas.

En este apartado se va a explicar cómo se ejecuta un trabajo mediante las herramientas que proporciona el gestor de colas del *cluster* de estudio, en este caso,

Fichero 3.1: scriptSGE.sh

```
#!/bin/bash
#$ -M aamor@tsc.uc3m.es
#$ -m e
#$ -cwd
#$ -pe mpi 16
#$ -l mf=31G
#$ -l h_vmem=3.6G
#$ -j y
#$ -S /bin/bash
#$ -q all.q
#$ -V
MACHINES=$TMPDIR/machines
TMPDIR=/tmp
mpiexec -machinefile $MACHINES -np $NSLOTS hola/Hello
```

SGE (*Sun Grid Engine*, ahora llamado OGE, o *Oracle Grid Engine*, tras la adquisición de Sun por Oracle; véase [14, 15, 16]).

En la estructura por capas que se expone en la Figura 3.2, este epígrafe se corresponde con la inferior, es decir, a la ejecución de trabajos por medio de consola de comandos.

Como ya se explicó en la introducción, las funcionalidades que ofrece DRMAA son, básicamente, las que se pueden conseguir por medio de los comandos nativos de SGE. Por este motivo, parece más que conveniente familiarizarse con estas herramientas para saber qué se debería poder controlar con DRMAA.

La primera, y la más básica de todas, es el envío de un trabajo, *job*, a la cola. Para ello se usa el comando `qsub` acompañado de un código en *shell script* que es el que se indica en el Fichero 3.1, esto es:

```
$ qsub scriptSGE.sh
Your job 3085 ("scriptSGE.sh") has been submitted
```

Se observa claramente que ese fichero tiene una cierta estructura:

- En la primera línea se indica el tipo de *shell script* que se está utilizando, en este caso `bash` (véase [30]).
- A continuación, las líneas que comienzan con los caracteres `#$` son relativas a las opciones de ejecución del trabajo en SGE. Se hará enfásis en este aspecto más adelante.
- Finalmente, los últimos comandos son los que van a ser ejecutados en el *cluster* bajo el entorno de SGE. En este caso se está ejecutando un fichero de prueba en Fortran90 que se indicará posteriormente.

Así, las opciones de ejecución con las que cuenta un trabajo en SGE son parámetros que analiza el gestor de colas y que determinan la memoria virtual que puede consumir el trabajo en cada nodo, la cola a la que se va a enviar la tarea... entre otras muchas variantes. Las principales opciones que admite SGE son las siguientes:

- `-S shell`: campo obligatorio en un script de este tipo, indica el intérprete de comandos que se está utilizando.
- `-cwd`: lanza la tarea en el directorio desde donde es ejecutada. Por defecto, es el `$HOME` de cada usuario.
- `-pe mpi/orte cores`: establece las opciones del entorno paralelo, que puede hacerse por medio de recursos como `mpi`, usado por ejemplo en FortranMod, o como `orte`, utilizado en Matlab. Se definen así variables y se configura el tipo de ejecución. Por otra parte, el parámetro `cores` indica en cuántos núcleos (que es una abstracción ya que un nodo puede tener varios núcleos) se distribuye la tarea por medio de herramientas como `mpiexec` o `mpirun`; en el caso concreto de `mpi`, el gestor de colas repartirá el trabajo en el número de `cores` indicado por parámetro intentando llenar los `cores` de un nodo antes de usar otro `core` de un siguiente nodo. Para más información véase [31, 32].
- `-M mailAddress@site.ext`: se manda un correo a la dirección indicada por parámetro según la configuración indicada en la opción `-m`.
- `-m a/b/e`: indica cuándo se envía un correo a la dirección determinada por la opción `-M`. Puede ser cuando el trabajo sea abortado (`a`), cuando comience (`b`) y/o cuando acabe (`e`).
- `-N nombre`: determina el nombre del trabajo. Por defecto, es el nombre del `script` que se va a enviar.
- `-o nombre_fichero`: fija el nombre del fichero de salida del trabajo. Por defecto es el nombre del `script` y acaba en `.o[pid]`, donde el `pid` es el número que se le asigna al trabajo al entrar en el gestor de colas.
- `-e nombre_fichero`: indica el nombre del fichero de error del trabajo en el caso de que los errores no estén incluidos en la salida `.o` mediante la opción `-j`. Si no se asigna, su nombre es el del `script` acabado en `.e[pid]`.
- `-j y`: opción que fuerza a que el fichero de error `.e` esté incluido en el fichero `.o`.
- `-i [host:/directorio/] nombre_fichero`: determina el fichero de entrada que sustituye a la entrada estándar.

- `-v var1[=val1][,var2[=val2],...]`: carga variables de entorno en la ejecución del trabajo; sin embargo, es aconsejable cargarlas en el `script` como comandos, tal y como se hace en 3.1.
- `-V`: carga todas las variables de entorno actuales, y siempre es recomendable que esté presente para evitar problemas de ejecución.
- `-h`: hace que el trabajo se encole en estado de *parada*; de esta forma, se queda en espera hasta que se ejecuta el comando de SGE `qalter -hU`.
- `-l h_rt=hh:mm:ss`: delimita el tiempo de ejecución a `hh` horas, `mm` minutos y `ss` segundos.
- `-l mf=memoria`: fija unos requisitos de memoria de tal forma que, hasta que no estén disponibles esos recursos, la tarea no se ejecuta.
- `-l h_vmem=memoria`: determina la memoria virtual por unidad de recurso que puede solicitar el trabajo enviado; así, si el proceso intenta reservar más recursos, se devolverá un error. Es posible usar el sufijo `G` para indicar la cantidad en gigabytes.
- `-l h_fsize=tamaño_fichero`: especifica el tamaño máximo del fichero que el trabajo puede escribir en disco.
- `-q cola`: fija una cola determinada en la que se incluirá la tarea.
- `-q cola@nodo`: hace que la tarea sea lanzada en el `nodo` de la `cola`.

Una vez expuestas las principales opciones con las que un cierto trabajo se puede lanzar, queda explicar la misión de las tres últimas líneas del Fichero 3.1. Las dos primeras son variables de entorno que se deben declarar en ejecución para el correcto funcionamiento de la última línea; finalmente, ésta indica la ejecución por medio de `mpiexec` un simple programa en Fortran90 (véase [33]) que es el referido en el Fichero 3.2.

Así, la ejecución de este programa imprime por pantalla un mensaje de texto que se enseña a continuación:

```
Hello world! I am     0  out of    16  in compute-0-9   .
Hello world! I am     1  out of    16  in compute-0-9   .
Hello world! I am     2  out of    16  in compute-0-9   .
Hello world! I am     3  out of    16  in compute-0-9   .
Hello world! I am     5  out of    16  in compute-0-9   .
Hello world! I am     4  out of    16  in compute-0-9   .
Hello world! I am     9  out of    16  in compute-0-9   .
Hello world! I am     6  out of    16  in compute-0-9   .
Hello world! I am     7  out of    16  in compute-0-8   .
Hello world! I am     8  out of    16  in compute-0-8   .
Hello world! I am    10  out of    16  in compute-0-8   .
Hello world! I am    11  out of    16  in compute-0-8   .
```

Fichero 3.2: HelloWorld.F90

```
1   program main
    include 'mpif.h'
    character (LEN=20) : : name
    integer ierr, my_id, nproc
    c Rutina de inicializaci n.
6   c ierr devuelve un mensaje de error
    call MPI_INIT( ierr )
    c Esta rutina devuelve el handler por
    c el cual invocamos al proceso, y nos
    c devuelve su identificador, que ser
11  c un numero entre 0 y el nproc-1.
    call MPI_COMM_RANK( MPI_COMM_WORLD, my_id, ierr )
    c Esta rutina nos devuelve el n mero de
    c procesos que maneja el handler MPI_COMM_WORLD.
    c ierr es el valor de error de retorno.
16  call MPI_COMM_SIZE( MPI_COMM_WORLD, nproc, ierr )
    c Aqu  se imprime nuestro identificador y
    c todos los procesos que estamos usando.
    print *, 'Hello World! I am ', my_id, ' out of ', nproc, ' in ', name, ' .'
    c As  liberamos toda la memoria que hayamos podido usar.
21  c Despu s de esta funci n no se puede ejecutar ninguna rutina m s.
    call MPI_FINALIZE( ierr )
    end
```

```
Hello world! I am     13   out of     16   in compute-0-8    .
Hello world! I am     12   out of     16   in compute-0-8    .
Hello world! I am     14   out of     16   in compute-0-8    .
Hello world! I am     15   out of     16   in compute-0-8    .
```

Esta salida de pantalla muestra que se han lanzado 16 procesos (ya que se indicó 16 como opción de `-pe mpi` en el Fichero 3.1) de forma paralela gracias a MPI (*Message Passing Interface*, véase [31, 32]); cada proceso imprime por pantalla su orden y el nodo en el que ha sido ejecutado. Como se observa, hay 8 *cores* por nodo, poniéndose de manifiesto la abstracción que ya se indicó anteriormente. Con esto se demuestra que la ejecución de tareas paralelas con el gestor de colas es posible en un *cluster* como el que se ha usado, pero no se ahondará más en este aspecto por no ser el objeto principal de este proyecto; si se desea más información véase [34].

Otro comando importante es `qstat`, que permite obtener el estado de todos los trabajos que ha enviado un cierto usuario al gestor de colas y que aún no han finalizado. Así, devuelve información como, entre otros parámetros, el PID (*Process IDentifier*, equivalente a *job-IDentifier*), la fecha en la que se envió la tarea y la cola y el nodo en el que se ejecutará el trabajo. Un ejemplo de su funcionamiento es el siguiente:

```
$ qstat
job-ID  prior   name       user    state  submit/start at     queue slots
-----------------------------------------------------------------------------
  3874 0.00000 scriptSGE. aamor   qw     06/21/2012 02:52:53          16
```

En este caso se observa que el estado de la tarea enviada, cuyo nombre en este caso es `scriptSGE.sh`, es `qw`, lo que significa que está *queue waiting*, es decir, esperando en el gestor de colas a que sea asignado a un nodo concreto; por otro lado, se puede ver que el campo `slots` indica los recursos paralelos que se van a consumir.

Hay que indicar también que si el trabajo ya ha sido admitido por SGE, la ejecución de `qstat` ofrece resultados ligeramente distintos tal y como se muestra en la siguiente salida por consola en la que, por motivos de espacio, se muestran únicamente los campos que han cambiado:

```
$ qstat
state  submit/start at      queue                      slots
-------------------------------------------------------------
  r    06/21/2012 02:53:08  all.q@compute-0-8.local     16
```

Así, las principales diferencias que se muestran en esta salida de pantalla son que cambia el tiempo de `submit/start at`, ya que en el primer caso reflejaba cuándo había recibido SGE la tarea, y ahora indica cuándo ha comenzado la ejecución del trabajo en el *cluster*; que cambia el estado de la tarea a `r` debido a que el trabajo ya se está ejecutando; y que el campo `queue` ha dejado de estar vacío, indicando ahora `nombre_cola@nombre_nodo`, es decir, en qué nodo se ha lanzado y qué cola lo ha recogido. Por otro lado, hay que notar que el campo `state` tiene otros posibles valores; sin embargo, se muestran los más relevantes.

Por otro lado, el comando `qdel` hace que un cierto trabajo sea eliminado de SGE esté en el estado que esté. Un ejemplo de invocación a `qdel` es el siguiente:

```
$ qsub scriptSGE.sh
Your job 3876 ("scriptSGE.sh") has been submitted
$ qstat
job-ID  prior   name       user    state  submit/start at     queue   slots
-----------------------------------------------------------------------------
  3876 0.00000 scriptSGE. aamor   qw     06/21/2012 03:04:46             16
$ qdel 3876
aamor has deleted job 3876
$ qstat
```

De esta forma, mediante la llamada a `qdel pid_trabajo` se consigue que la tarea enviada con el pid indicado por parámetro sea eliminada del gestor de colas.

Por último, se va a explicar brevemente una herramienta gráfica que ofrece SGE denominada `qmon`, con la que se puede conseguir una interfaz gráfica con el aspecto de la figura 3.3.

Pudiendo acceder y controlar por medio de una interfaz gráfica todos los parámetros que se pueden controlar por línea de comandos; por ejemplo, a la hora de

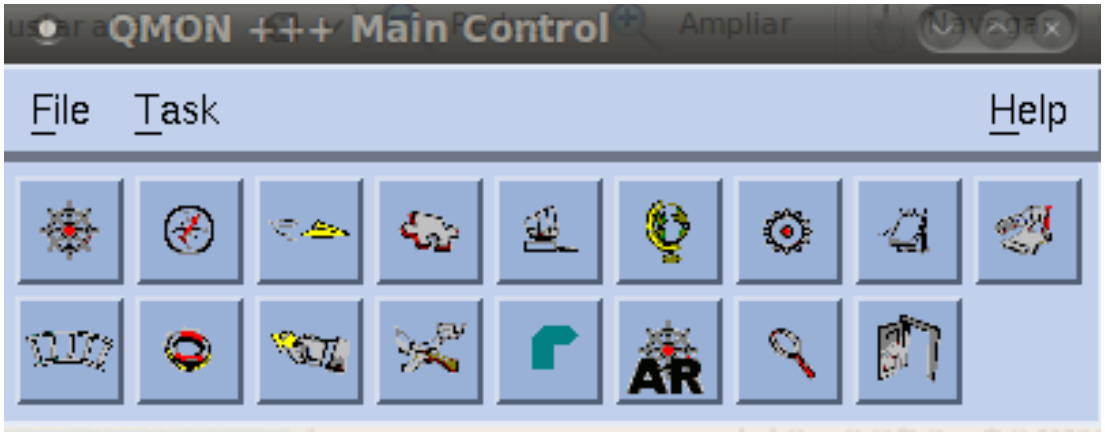


Figura 3.3: Pantalla principal de `qmon`.

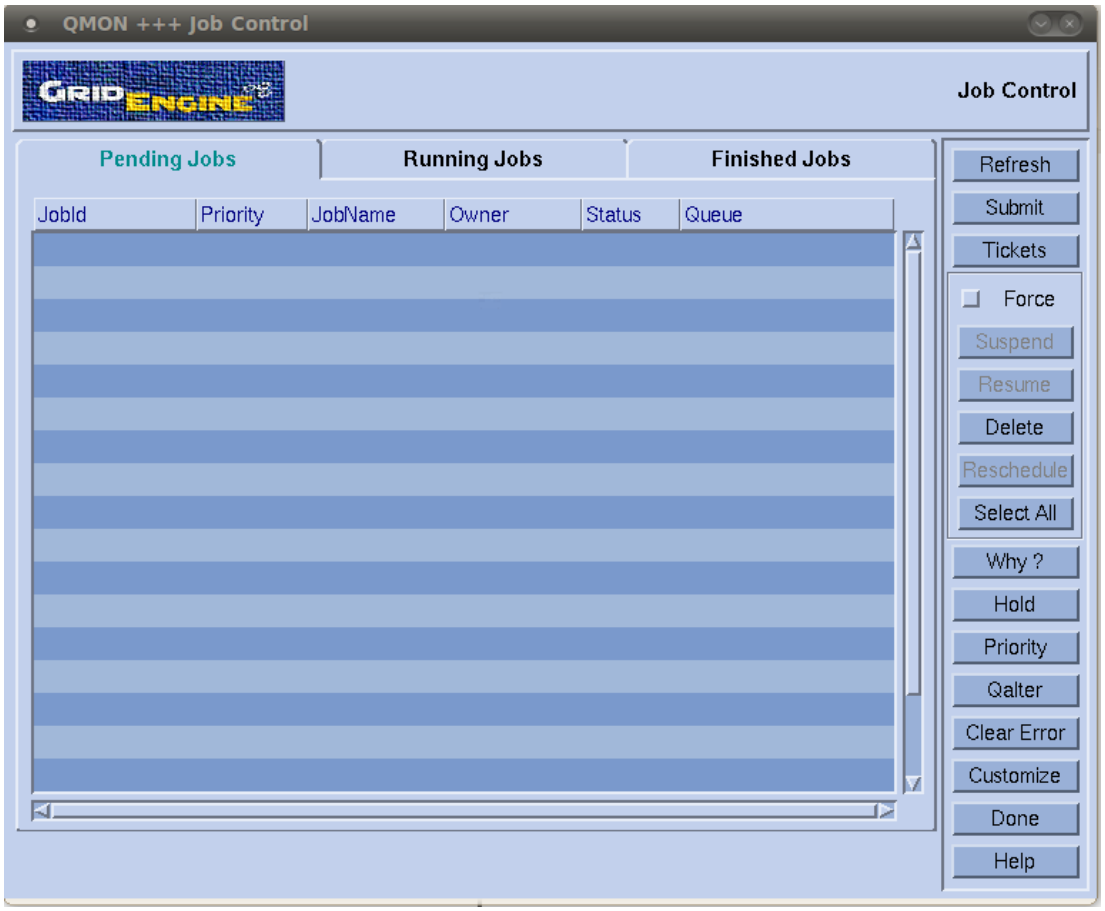


Figura 3.4: Control de trabajos por medio de `qmon`.

controlar trabajos se tienen todas las opciones posibles, como se observa en la figura 3.4

Sin embargo, esta herramienta está más enfocada a administradores de sistemas que a usuarios, de forma que incluye todas las opciones posibles que, en general, un usuario corriente desconoce. Así, esta herramienta es poco útil para este proyecto ya que el objetivo es conseguir algo más sencillo y manejable para el usuario, que pueda ejecutar con seguridad sin perder las principales funcionalidades de SGE y que no esté en remoto, ya que de esa forma la velocidad de ejecución depende directamente de la velocidad de conexión al sistema.

Por otra parte, si bien `qmon` se crea para que el manejo de un gestor de colas como SGE no sea tan complejo, tiene inconvenientes importantes como, por ejemplo, que se lanza en remoto de forma que su ejecución es bastante pesada de forma que el control de trabajos por medio de `qmon` es muy lento en comparación con el que se puede conseguir por medio de línea de comandos.

Quedan explicados así los principales comandos de SGE y que se van a emplear en este trabajo:

- `qsub nombre_script`, para mandar un trabajo con ciertas opciones de ejecución al gestor de colas SGE.
- `qstat`, para observar el estado de los tareas que un usuario ha enviado y aún no han finalizado.
- `qdel pid_trabajo`, para eliminar un trabajo que aún no haya finalizado.

3.2. DRMAA

En esta sección se va a usar DRMAA (véase [29]) para ejecutar y controlar los trabajos enviados al gestor de colas, SGE, desde Java, buscando así conseguir la misma funcionalidad que se ha descrito en el apartado anterior (comandos y opciones de ejecución).

De esta forma, este apartado explica la capa intermedia en la pila que se muestra en la Figura 3.2.

DRMAA es un API desarrollado por un grupo de trabajo del OGF (*Open Grid Forum*) que se emplea para controlar y enviar trabajos a uno o más sistemas DRM (*Distributed Resource Management*) que, básicamente, son los que planifican y

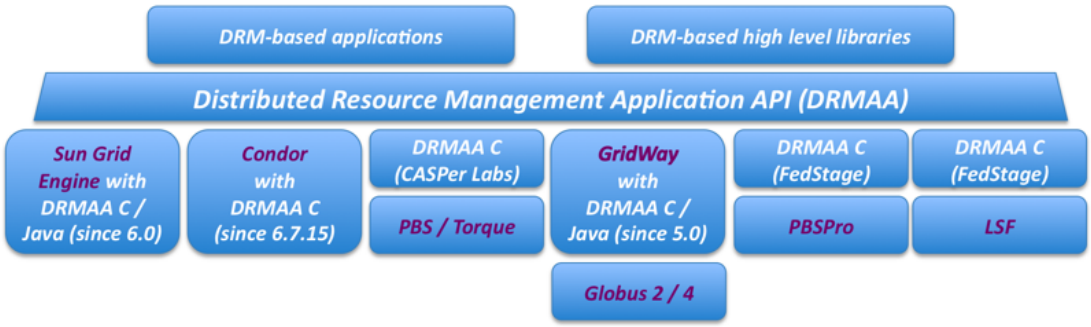


Figura 3.5: Situación de DRMAA sobre los gestores de colas.

controlan las tareas que se realizan en un *cluster*, o lo que es lo mismo, en un conjunto de recursos autónomos que actúan de forma cooperativa. Se podría decir, así, que un DRM es un *scheduler* o planificador muy sofisticado.

Se podrá observar que las propias características de DRMAA introducirán algunos inconvenientes ya que se trata de un proyecto ambicioso pero aún en desarrollo, de forma que hay algunas funcionalidades que no se comportarán como deben.

Esos inconvenientes se solucionarán por medio de la creación de un API sobre DRMAA, es decir, una capa más tal y como se observa en la Figura 3.2 y que se detallará en la siguiente sección.

Sin embargo, el uso de DRMAA merece la pena no sólo por el hecho de controlar y ejecutar trabajos de un gestor de colas desde Java, sino porque implementa una funcionalidad que es muy interesante para el usuario, que es la captura del evento de finalización del trabajo. En efecto, gestores de colas como SGE tiene como mecanismo de notificación al usuario el envío de un correo cuando el trabajo haya empezado, haya finalizado o haya sido abortado; DRMAA, por otro lado, además de posibilitar esta notificación, captura el evento de finalización de la tarea por código, pudiendo implementar así mecanismos de aviso al usuario alternativos al correo que, en cierta medida, es algo tedioso.

Además, la utilización de DRMAA también introduce otro tipo de ventajas, que es la independencia del gestor de colas empleado. De esta forma, DRMAA es un API que proporciona una capa de abstracción sobre SGE, haciéndolo compatible con otros sistemas como se observa en la Figura 3.5, extraída de [29].

Se puede ver en la figura que hay dos implementaciones de DRMAA, una en C y otra en Java. Así, se escogió el desarrollo en Java por los motivos que se indicaron en el Capítulo 2 de forma que hay compatibilidad con SGE y con Gridway, que es otro sistema de gestión de colas (véase [35]). La evolución natural

de este desarrollo es que futuras versiones de DRMAA expandan este escenario; sin embargo, el proyecto se encuentra algo estancado por lo que no hay fechas ni plazos. Por otra parte, hay ciertos mecanismos que permiten la compatibilidad entre Java y C, de forma que es sencillo llamar a funciones de C desde Java por medio una interfaz denominada JNI, o *Java Native Interface*, cuya definición se detalla en [36].

Así, DRMAA ofrece una interfaz de alto nivel para todas aquellas aplicaciones que necesiten ponerse en contacto con un gestor de colas, ya sea para interrumpir un trabajo ya enviado, para mandar uno nuevo o para verificar su estado. En el escenario propuesto en el Capítulo 2, el *scheduler* es SGE que es un sistema de encolamiento de procesos distribuido; sin embargo, la transición a otro planificador no debería introducir cambios en *Posidonia* al usar DRMAA que hace una abstracción sobre el gestor de colas empleado, de forma que sería igual usar SGE u otros *schedulers* como Condor [37], Xgrid [38], EGEE (LCG2 / gLite) [39], Platform LSF [40], UNICORE [41], Kerrighed Cluster Framework [42], IBM Tivoli Workload Scheduler LoadLeveler [43] y SLURM [44], es decir, la mayor parte de los sistemas de colas que se emplean en la actualidad. Por otro lado, la implementación en Java de DRMAA permite llegar a entornos Grid [45], ya que GridWay [35] y Globus [46] lo soportan.

En primer lugar, se va a explicar cómo lanzar un trabajo de forma similar a lo detallado en la sección anterior; se verán los problemas que surgen con esta implementación y cómo se resolverán, en la siguiente sección, con la creación de un API que abstrae las funcionalidades de DRMAA.

De esta forma, se intenta replicar el envío del trabajo que se hizo por línea de comandos, aunque la primera aproximación será, simplemente, la ejecución del programa en Fortran90 que se expuso en el Fichero 3.2. Para ello, hay tutoriales muy interesantes en Internet, sobre todo en las referencias [47, 48], en las que se basa el código del Fichero 3.3, que es autoexplicativo, aunque se van a detallar las líneas más importantes. En primer lugar, se observa que DRMAA está orientado a sesión de forma que todas las interacciones con el gestor de colas se hacen a través de un objeto `Session`, del que sólo puede existir uno simultáneamente: así, en la línea 12 se obtiene el objeto y en la línea 36 se libera. Por otra parte, los trabajos se configuran por medio de un objeto `JobTemplate` creado por la sesión y en el que se pueden modificar, a priori, todas las opciones de ejecución que se detallaron en el apartado anterior, además de ser capaz de incluir los comandos que se desean ejecutar en el entorno SGE por medio de la instrucción de la línea 25.

Una vez se tiene el fichero fuente en Java, hay que compilarlo para poder ejecutarlo en la máquina virtual. Como se usa DRMAA, hay que indicar además, por medio

Fichero 3.3: HolaFortran.java

```
1  import java.util.Collections;
   import org.ggf.drmaa.DrmaaException;
   import org.ggf.drmaa.JobTemplate;
   import org.ggf.drmaa.Session;
   import org.ggf.drmaa.SessionFactory;
6  public class HolaFortran {
       public static void main(String[] args) {
           // Necesitamos un objeto de tipo Session para interactuar
           // con la cola DRM (en nuestro caso, ahora SGE), para lo
           // cual necesitamos una SessionFactory de la que extraer la Session.
11         SessionFactory factory = SessionFactory.getFactory();
           Session session = factory.getSession();
           try {
               // Inicializamos la sesion. De momento lo hacemos sin parametros.
               // La llamada a init crea una sesion y activa un listener que
16             // recibe actualizaciones del gestor de colas y del estado de la
                   cola.
               session.init("");
               // Creamos una JobTemplate, que es un objeto que almacena
                   informacion
               // acerca del trabajo que vamos a dar a la cola.
               JobTemplate jt = session.createJobTemplate();
21             // Aqui indicamos donde se encuentra el archivo que vamos a agregar
               // a la cola. En nuestro caso, HelloFortran. Le pasamos la ruta
               // absoluta; si le pasamos la ruta relativa, debemos tener
               // cuidado con WorkingDirectory.
               jt.setRemoteCommand("/home/aamor/hola/HelloFortran");
26             String [] nulo = {""};
               jt.setArgs(nulo);
               // Ponemos en la cola nuestro trabajo. Ya da igual como acabe el
               // programa, el trabajo estar  en la cola.
               String id = session.runJob(jt);
31             System.out.println("Your job has been submitted with id " + id);
               // Esta accion borra la plantilla del trabajo y asi se evitan
               // fugas de memoria.
               session.deleteJobTemplate(jt);
               // Cerramos la comunicacion con el gestor de colas.
36             session.exit();
           } catch (DrmaaException e) {
               System.out.println("Error: " + e.getMessage());
           }
       }
41 }
```

de la directiva `-cp`, dónde están los paquetes `org.ggf.drmaa`, que en este caso se encuentran en el mismo directorio del fichero `.java` por medio de un contenedor en java `.jar`; de esta forma la compilación queda así:

```
$ javac -cp ./drmaa.jar HolaFortran.java
```

Y para la ejecución, gracias a `$CLASSPATH:./drmaa.jar`, se añade a los actuales valores de `classpath` (que es una variable de entorno que le muestra a la máquina virtual dónde buscar los ficheros compilados `.class`) el sitio en el que se encuentra el paquete DRMAA, tal y como se muestra a continuación:

```
$ java -cp $CLASSPATH:./drmaa.jar HolaFortran
```

De esta manera el código indicado en el Fichero 3.3 es ejecutado obteniendo la siguiente salida por pantalla que muestra que el trabajo ya ha llegado al gestor de colas:

```
Your job has been submitted with id 4051
```

Con el programa `HolaFortran.java` la función de DRMAA finaliza con la recepción del trabajo en el gestor de colas, de forma que la salida que se obtiene al acabar la tarea es un fichero con extensión `.o+pid`, donde `pid` es el identificador del trabajo que en este caso es `4051`, y que es el resultado de la ejecución del Fichero 3.2; y otro con extensión `.e+pid`, que es el archivo de error que se obtiene en el caso de producirse algún fallo. Así, el contenido del fichero `HolaFortran.o4051` es el siguiente:

```
Hello world! I am     0   out of    1   in compute-0-9    .
```

Lo que indica que se ha lanzado un único proceso, al no haber cambiado el parámetro `-pe` del trabajo, y que se ha ejecutado de forma correcta ya que el fichero de error, `HolaFortran.e4051`, está vacío.

Por último, es importante destacar que para la ejecución de un trabajo simple mediante DRMAA como el que se está poniendo como ejemplo hay dos vías:

1. Crear un script como el del Fichero 3.1 e indicar al gestor de colas que debe ejecutarlo por medio del método `setRemoteCommand` que aparece en el Fichero 3.3. Se verá posteriormente que esto, en el caso que se contempla, sólo sirve para los comandos que deben lanzarse en SGE, es decir, las tres últimas líneas del script, ya que el gestor no reconoce las opciones de ejecución del trabajo por esta vía.

2. Indicar las opciones de la tarea por medio de métodos propios de DRMAA y los comandos a ejecutar por medio de `setRemoteCommand`; también se observará que esto no es posible por problemas de implementación de la librería DRMAA.

En el ejemplo del Fichero 3.3, por su simplicidad, se está usando la segunda vía aunque también se podría seguir la primera sin grandes dificultades.

Una vez que se sabe cómo ejecutar una tarea por medio de la librería DRMAA para Java y a través de SGE, lo siguiente que se debe implementar es la ejecución del script de comandos del Fichero 3.1 para demostrar la capacidad de cambiar cualquier opción de ejecución del trabajo. Con este objetivo, se consulta como referencia el *Javadoc* (que es la documentación asociada a una cierta librería o fichero Java si se comenta con un cierto formato) de DRMAA que se encuentra en [49], aunque no es una fuente del todo fiable ya que tiene ciertas diferencias de implementación con nuestra versión.

A continuación se muestran los principales métodos, además de los ya vistos como `setRemoteCommand` o `setArgs`, que deberían permitir replicar cada parte del script:

- `setNativeSpecification`: a esta función se le pasa un `String`, o cadena de caracteres, opaco que es interpretado por DRMAA. Gracias a esto, se podrían pasar todas las opciones de ejecución del trabajo (que se detallaron en el Apartado 3.1) introduciendo la siguiente línea en el código:

  ```
  jt.setNativeSpecification("-cwd -j y -pe mpi 16 -m e -M aamor@tsc.uc3m.
      es -S /bin/bash -V -q all.q -l mf=31G -l h_vmem=3.6G");
  ```

  Donde `jt` es el nombre de un objeto de la clase `JobTemplate` que representa una tarea de SGE. De esta forma, se consiguen pasar los parámetros de configuración de la tarea al gestor; sin embargo, no es una programación limpia porque la llamada al método no se entiende si no se conocen las opciones de ejecución de un trabajo en el gestor de colas, de forma que la interpretación del código por un tercero se hace bastante más complicada.

  De esta forma, se debe evitar en la medida de lo posible la llamada a `setNativeSpecification`; no obstante, DRMAA fuerza a utilizarlo para parámetros como `-V`, `-q`, `-pe` o `-m`, ya que no se ofrece ningún otro método para configurar estas opciones como pudiera ser, por ejemplo, para `-q`, una función llamada `setQueue`.

  Además, aunque no aparece en la documentación, si se introduce una opción no reconocida por el sistema DRM (en este caso, SGE) salta la siguiente excepción interrumpiendo la ejecución del código:

```
org.ggf.drmaa.DeniedByDrmException: invalid option argument "-j i"
```

- Rutas de entrada y de salida: el API proporciona diversos métodos para controlar la ruta de los archivos que devuelve la ejecución de los trabajos, como `setOutputPath` o `setInputPath`. De esta forma, si se desea especificar el nombre del fichero de salida, por ejemplo, se haría de la siguiente manera:

  ```
  jt.setOutputPath(":/home/aamor/hola/HolaFortran.o");
  ```

  Es clave escribir ':' al principio de la ruta (que, además, debe ser completa) por razones de implementación de DRMAA, y de forma similar se procede para determinar el nombre del fichero de error. También hay que notar que el fichero no se crea cada vez que se escribe sino que se añade al archivo ya existente si éste ya hubiera sido creado, sin posibilidad de cambiar esta opción; de esta manera, si se quiere hacer una cierta gestión de los ficheros de salida hay que hacerla de forma externa a la librería.

  Por otro lado, se debe destacar que la documentación asociada al API determina que se cree un objeto de la clase `FileTransferMode` para indicar que se especifican rutas de error y de salida concretas, y no las asociadas por defecto al trabajo; esto se haría con los siguientes comandos en Java:

  ```
  FileTransferMode ftm = new FileTransferMode();
  ftm.setErrorStream(true);
  ftm.setOutputStream(true);
  ftm.setInputStream(false);
  jt.setTransferFiles(ftm);
  ```

  No obstante, esto no funciona ya que al ejecutar esas llamadas se lanza una excepción del tipo `java.lang.IllegalArgumentException`. Posteriores pruebas demostraron que la implementación de DRMAA no necesita que se indique explícitamente que se van a usar unas rutas determinadas de error y de salida sino que, si estas son especificadas, SGE las toma automáticamente desechando los nombres y rutas por defecto de los archivos de salida y de error.

- Correo electrónico de aviso: con respecto a esto se dispone del método `setEmail`, que recibe como parámetro un array de `String` al contrario de lo que se indica en la documentación empleada, que afirma que se debe usar un objeto de la clase `Set` en vez de dicho array. La llamada a esta función se hace de la siguiente forma:

  ```
  String [] mailString = {"aamor@tsc.uc3m.es"};
  jt.setEmail(mailString);
  ```

  Mediante este método se determina a qué dirección de correo hay que enviar el aviso; sin embargo, la configuración del correo electrónico (es decir, cuándo se notificará al usuario por medio de este mecanismo, en el inicio, finalización o aborto del trabajo) se debe hacer por medio de `setNativeSpecification` como se ha visto anteriormente.

No obstante, `setEmail` no está correctamente implementado en DRMAA ya que SGE no envía ningún correo a la dirección indicada por parámetro; de esta forma, para habilitar esta opción hay que hacer uso del *flag* `-M` en `setNativeSpecification` tal y como se ha comentado en el primer apartado.

- Introducción de variables en el entorno de trabajo: se consigue con el método `setJobEnvironment`, de forma que la antepenúltima y penúltima línea del script de refencia, el Fichero 3.1 se tienen que introducir mediante esta función.

Con todas estas funciones se debería poder enviar sin problemas el trabajo que se lanzó en el Apartado 3.1; sin embargo, no se consiguen los mismos resultados por lo que, para detectar el error, se ejecutarán algunos comandos simples como:

- `ls`: en primer lugar se va a lanzar este comando sin parámetros, de forma que en el Fichero 3.3 se modifican los métodos `setRemoteCommand`, `setNativeSpecification` y `setArgs` de la siguiente manera:

```
// Ponemos el campo obligatorio, cargamos el entorno y
// usamos la cola all.q
jt.setNativeSpecification("-S /bin/bash -V -q all.q");
// El comando que ejecutaremos sera ls.
jt.setRemoteCommand("ls");
```

  El método `setArgs` se debe omitir al contrario de lo que se indica en la documentación consultada, en la que se afirma que si no se usan argumentos se deben introducir como parámetros a `setArgs Strings` con valor nulo. Con esta configuración el comando ejecuta como cabe esperar, listando los ficheros que encuentra en su carpeta.

- `ls -l -a`: a continuación se desea ejecutar un comando con parámetros, es decir, con las opciones `-l -a`. Esto se prueba porque debido a la documentación aportada, se desconoce si para ejecutar el comando `mpiexec` del Fichero 3.1 se debe poner como un `String` en `setRemoteCommand` incluyendo la última línea entera del script, o incluyendo `mpiexec` en `setRemoteCommand` y, a continuación, el resto de la línea como parámetros de `mpiexec` en `setArgs`. Así, ahora se modifica el Fichero 3.3 con lo siguiente:

```
jt.setNativeSpecification("-S /bin/bash -V -q quad.q");
jt.setRemoteCommand("ls");
// Los argumentos son:
String [] argStr = {"-l", "-a"};
jt.setArgs(argStr);
```

  Con estas llamadas la salida también es la que cabe esperar en un comando de este tipo; por otra parte, hay que destacar que en la documentación de referencia se indica que la entrada de la función `setArgs` es un objeto de la clase `List`, y no un array de `String`.

Gracias a la ejecución de estas instrucciones sencillas se ha depurado el funcionamiento mal documentado de DRMAA, de modo que ya se está en condiciones de intentar replicar el script del Fichero 3.1. De esta forma, en el Fichero 3.3 se modifica lo que ya se ha cambiado con la ejecución de los comandos `ls` y `ls -l -a` con lo siguiente:

```
// Necesitamos pasar de manera explicita los valores de
// TMPDIR y de MACHINES, necesarios para la ejecucion de mpiexec.
Properties env = new Properties ();
env.put ("TMPDIR", "/tmp");
env.put ("MACHINES", "`$TMPDIR/machines`");
// Con esto hacemos que en el entorno de SGE esten los
// valores de esas variables.
jt.setJobEnvironment (env);
// Campo obligatorio, cargamos el entorno y usamos la
// cola -q all.q. Ademas, ponemos 16 en paralelo.
jt.setNativeSpecification("-S /bin/bash -V -pe mpi 16 -q all.q");
// El comando que usamos es mpiexec.
jt.setRemoteCommand("mpiexec");
// Y ahora le pasamos los argumentos (que es lo que sigue a mpiexec)
String [] argStr =
  {"-machinefile", "$MACHINES", "-np", "$NSLOTS", "$PWD/HolaFortran"};
jt.setArgs(argStr);
```

En esa porción de código se observa que:

- La introducción y definición de variables en el entorno se hace a través del método `setJobEnvironment`.
- Todo lo que sigue al comando `mpiexec` son parámetros, por lo que hay que pasarlos por medio de `setArgs` tal y como se vio con la ejecución del comando `ls -l -a`.

Sin embargo, con esas modificaciones el Fichero 3.3 no funciona. Después de una depuración exhaustiva del código bastante complicada debido a la pésima documentación de DRMAA, se encuentra el error: Java pasa al `bash`, es decir, al intérprete de comandos, las instrucciones de la siguiente manera:

```
'mpiexec' '-machinefile' '$MACHINES' '-np' '$NSLOTS' '$PWD/HolaFortran'
```

Lo que no es lo mismo que `mpiexec -machinefile $MACHINES -np $NSLOTS $PWD/HolaFortran`, ya que en este último caso `bash` interpreta la instrucción, es decir, accede al valor de `MACHINES` ya que así se le indica por medio de `$`, así como al valor de `NSLOTS` y `PWD`; de la otra forma, `bash` no interpreta el comando y lo toma como tal, es decir, intenta abrir, por ejemplo, la carpeta `$PWD/HolaFortran` que, obviamente, no existe en el sistema.

Esto supone un obstáculo inabordable, ya que es un problema de implementación de la librería DRMAA y no se puede hacer nada al respecto por medio de llamadas a las funciones propias del API. Se podría evitar el uso de variables como

`MACHINES` o `PWD`; sin embargo, eso no es posible ya que el valor de, por ejemplo, `MACHINES`, es distinto en cada ejecución en función del nodo y la cola en la que se ejecuten los procesos.

Por otra parte, también se ha detectado que al pasar los valores de las variables de entorno `MACHINES` y `TMPDIR` se hace de la misma forma, es decir:

```
'MACHINES=$TMPDIR/machines'
```

Lo que es igualmente inasumible, ya que `bash` no interpreta la instrucción e intenta encontrar la carpeta `$TMPDIR/machines` que, lógicamente, no existe.

De esta forma, se ha encontrado un obstáculo inevitable a la hora de usar DRMAA por lo que, a priori, no se podría usar esta librería para los propósitos del proyecto desarrollado. Sin embargo, hay otra vía que se puede explorar como ya indicamos anteriormente: DRMAA puede enviar al gestor de colas un script que es el que debe ejecutar. En este caso, el intérprete de comandos, `bash`, sí entiende lo que se le pasa y lo interpreta. No obstante, si se intenta ejecutar directamente el script del Fichero 3.1 el código en Java tampoco funciona porque SGE no reconoce las opciones de ejecución que sí entiende con el uso de `setNativeSpecification` o `setOutputPath`, entre otros.

Así, para poder usar DRMAA se va a desarrollar un API propio, es decir, una capa por encima, que se detalla en la siguiente sección y que recoge toda la experiencia explicada en este apartado para implementar lo que debería hacer DRMAA por *Posidonia*.

3.3. API desarrollada: SGEJob

Finalmente, en esta parte se detalla cómo se desarrolla un API propio, llamado SGEJob, para implementar las funciones de DRMAA en un enfoque orientado a objetos. Así, aquí se explica la capa superior de la pila que se muestra en la introducción de este capítulo en la Figura 3.2.

La necesidad de la creación de este API se ha pormenorizado en la sección anterior y se puede resumir en los siguientes puntos:

1. La implementación en Java de DRMAA hace que el intérprete de comandos, `bash`, no interprete las instrucciones que se le pasan por medio de los métodos `setNativeSpecification` y `setJobEnvironment`, de forma que

es inabordable la correcta ejecución de scripts como el indicado en el Fichero 3.1. Sin embargo, el gestor de colas sí recibe las opciones de ejecución del trabajo de forma adecuada.

2. La ejecución de los comandos por medio de un script sí es interpretada por `bash` y es por este motivo por el que no se abandona el uso de DRMAA ya que se puede construir un script con las instrucciones que se deben ejecutar.

3. La documentación de la librería es insuficiente y, en algunos casos, incorrecta, lo que hace muy difícil su uso.

4. El uso de `setNativeSpecification` permite la inclusión de cualquier opción de ejecución de los trabajos pero, sin embargo, no hace transparente para el usuario de DRMAA la utilización de estas opciones; por ejemplo, para cambiar el nombre de la tarea el usuario debería utilizar un método `setJobName` y no escribir como parámetro en `setNativeSpecification` el `String -N nombre_trabajo`.

5. La gestión de los ficheros de error y de salida no es tan completa como se podría desear, ya que no hay opción de sobreescribir el archivo si éste ya ha sido creado.

De esta forma, las características del API desarrollado dan solución a todos los problemas que han surgido del uso de DRMAA:

1. La correcta ejecución de scripts como el del Fichero 3.1 por medio de la combinación de métodos de DRMAA como `setJobEnvironment` y la ejecución de comandos por medio de la creación de un script que se lanza por medio de `setRemoteCommand`.

2. Una documentación completa, correcta y detallada de la funcionalidad que ofrece el API.

3. Uso del API estilo Java, es decir, que es transparente para el usuario de esta librería la utilización de métodos como `setNativeSpecification` y sus parámetros. Se usarán métodos como `setJobName` para indicar el nombre de la tarea, o `setQueue` para indicar la cola a la que se desea enviar el trabajo.

4. Gestión completa de los ficheros de error y de salida, de forma que hay posibilidad de sobreescribir el archivo si éste ya existe.

Con esto se consigue un código más comprensible conservando la funcionalidad que *Posidonia* espera de DRMAA, es decir, ser capaz de cambiar todas las opciones de ejecución del trabajo que ofrece SGE y capturar el evento de finalización que es algo que en DRMAA sí que funciona bien y por eso no se ha explicado en el anterior apartado.

Así, la estructura del código del API es la siguiente:

- Métodos `set` para modificar los parámetros de ejecución de la tarea.
- Métodos `get` para obtener las opciones de configuración del trabajo, hayan sido definidas por el usuario mediante los `set` o conserven el valor por defecto.
- Una función `runJob` en la que se ejecuta el trabajo con las opciones de configuración que hayan sido determinadas por medio de las funciones `set` del API.

El gran número de opciones de una tarea hace imposible adjuntar el código completo del API desarrollado; sin embargo, en el Fichero 3.4 se muestra la documentación asociada a los métodos `get` y `set` y el código necesario para especificar cuál es la ruta del fichero de salida del trabajo y la gestión de archivos que se indicó como característica del API desarrollado. Los nombres de las funciones son más descriptivos para que el uso de este API sea lo más sencillo posible.

Por otro lado, en el Fichero 3.5 se puede observar cómo se han resuelto, en el código, los problemas que conlleva el uso de DRMAA y cómo se implementa la espera del evento de finalización de la ejecución de la tarea (dejando a decisión del usuario si esta espera se realiza o no). Esto último es clave en la aplicación *Posidonia*, ya que es muy cómodo para el usuario saber cuándo ha acabado el trabajo que ha enviado al *cluster* sin tener que estar pendiente de ello o recibir un correo a la dirección que haya indicado, lo que resulta engorroso.

Sin la captura de este evento, la única forma de solucionarlo es con una espera activa, es decir, tener un proceso que esté preguntando siempre, o cada cierto tiempo, si la tarea ha acabado. Este mecanismo es muy rudimentario pero, sobre todo, consume recursos del sistema de forma inútil y lo ralentiza; gracias al uso de DRMAA se consigue evitarlo como aparece en el Fichero 3.5.

La longitud del método impide adjuntarlo entero: se muestran los métodos más relevantes para la ejecución indicando con '. . .' las partes que han sido omitidas. Se puede observar que la estructura básica es la misma que se ha utilizado en el

Fichero 3.4: set/getOutputFile en SGEJob.java

```
/**
 * Class equivalent to JobTemplate in DRMAA library. It's possible
 * to define characteristics of the job before sending it to the
 * queue manager and, moreover, sending it to the queue
 * manager.
 * @version 2.0 14/12/2011
 * @author Adri n Amor.
 */
public class SGEJob {

  // Object from DRMAA API to give characteristics to the work
  // which will be sended to the queue.
  private JobTemplate actual_job;

...

  // Output Path.
  private String outputPath = "";

...

  /**
   * Sets output path of the job. It's possible to append
   * to an existing file, or create a new one (if the file
   * exists, function deletes it and creates a new one).
   * @param outPath Output path for the job.
   * @param append If true, output appends the file (if it exists).
   */
  public void setOutputPath(String outPath, boolean append) {
    // It checks if the file exists in case if
    // it's necessary to delete the file.
    if (!append) {
      try {
        File outFile = new File (outPath);
        outFile.delete();
      }
      catch (Exception e) {
        System.out.println("File doesn't exist or it's invalid.");
        e.printStackTrace();
      }
    }
    // We change outputPath.
    outputPath = outPath;

  }

  /**
   * Gets output path of the job.
   * @return String which is output path for the job.
   */
  public String getOutputPath () {
    return outputPath;
  }
```

apartado anterior y la vía intermedia que se ha tomado para obtener un buen funcionamiento del código: al gestor de colas se le pasan las opciones indicadas por el usuario por medio del método `setNativeSpecification`, y se crea un script con los comandos que se desean ejecutar. Finalmente, con `session.wait(id, Session.TIMEOUT_WAIT_FOREVER)` se espera a que el trabajo con el identificador pasado por parámetro acabe, ya sea correctamente o por alguna incidencia excepcional.

Ya explicada la implementación del API que se ha desarrollado, a continuación se va a detallar cómo hacer uso correctamente de él:

1. En primer lugar, se deben indicar los parámetros de configuración del trabajo, es decir, el máximo de memoria virtual que puede utilizar, la ruta de salida, el nombre, el comando a ejecutar, etcétera.

2. Una vez indicados todos los parámetros deseados, se ejecuta el método `runJob` por medio del cual el trabajo será enviado al *cluster* mediante DRMAA. Esta función es la única que utiliza llamadas a DRMAA; el resto de métodos se utilizan para configurar las opciones de ejecución de la tarea. En cierta medida, el API es un contenedor en el que se pueden cambiar los valores por defecto del trabajo y, una vez modificados los que se deseen, se envía al gestor de colas.

Un ejemplo de uso de este API es el que se expone en el Fichero 3.6, en el que se observa cómo se cumplen las características que se introdujeron al principio de la sección: los nombres de los métodos son descriptivos, de forma que en todo momento se sabe qué se está modificando sin tener conocimiento de cómo son concretamente las opciones de SGE; el código es mucho más comprensible que el que se consigue con el uso de `setNativeSpecification` y, sobre todo, todos los métodos están probados y funcionan correctamente.

Finalmente, se debe destacar que este API, SGEJob, es prácticamente el que se utiliza en *Posidonia* salvo ciertos retoques necesarios para el envío de notificaciones y la actualización de ciertos archivos que se explicarán más adelante.

3.4. Conclusiones.

En conclusión, en este capítulo se ha detallado cómo se lanza un trabajo por medio de Java en un *cluster* que cuente con un gestor de colas como SGE. Además, se han observado detenidamente cuáles son los problemas del uso de librerías como

Fichero 3.5: runJob en SGEJob.java

```
1  ...
   /** Sends a job to the queue.
     * @param wait Boolean that, if true, it blocks the execution in
     * this function until job is finished.
     */
6    public void runJob(boolean wait){
       // We get the session from the factory...
       SessionFactory factory = SessionFactory.getFactory();
       Session session = factory.getSession();
       try
11     {
         // This line initialize the session.
         session.init(null);
         // We create a template for the job.
         JobTemplate jt = session.createJobTemplate();
16       // Now, we have to define NativeSpecification
         // (with flags of DRMAA)
         // Environment...
         if (environment)
           nativeSpec += "-V ";
21       // Join error and output files...
         if (joinErrorOut)
           nativeSpec += "-j y ";
         ...
         // Native Specification is done.
26       jt.setNativeSpecification(nativeSpec);
         System.out.println("nativeSpec: "+nativeSpec);
         PrintWriter scriptStream = new PrintWriter(new
               BufferedWriter(new FileWriter(actualPath+"/scriptSGE.sh")));
         scriptStream.println("#!/bin/bash");
31       scriptStream.println("MACHINES=$TMPDIR/machines");
         scriptStream.println("TMPDIR=/tmp");
         scriptStream.println(remoteCommand);
         scriptStream.flush();
         scriptStream.close();
36       // Now, we execute the script.
         jt.setRemoteCommand(actualPath+"/scriptSGE.sh");
         System.out.println ("Job with id " + id + " has been submitted!");
         if (wait) {
           // With this we wait for job to finish.
41         JobInfo info = session.wait(id, Session.TIMEOUT_WAIT_FOREVER);
           if (info.wasAborted()) {
             System.out.println("Job " + info.getJobId() + " never ran");
           }
           else if (info.hasExited()){
46           System.out.println("Job " + info.getJobId() + " finished
                    regularly with exit status " + info.getExitStatus());
           }
           else if (info.hasSignaled())
             System.out.println("Job " + info.getJobId() + " finished
51                   due to signal " + info.getTerminatingSignal());
           else
             System.out.println("Job " + info.getJobId() + "
                                  finished with unclear conditions");
           session.deleteJobTemplate(jt);
56         session.exit();
           System.out.println("Session Exit success");
       }
       catch (DrmaaException e) {e.printStackTrace(); }
       catch (IOException e) { e.printStackTrace(); }
61 }
```

Fichero 3.6: SGEJobExample.java

```
import org.ggf.drmaa.*;
import java.util.*;
// Our API is imported.
import SGE.SGEJob;
public class SGEJobExample
{
  public static void main (String[] args)
  {
    // Create a new instance of the class SGEJob.
    SGEJob job = new SGEJob();
    // We want to use flag -V.
    job.loadEnvironment(true);
    // A mail to aamor@tsc.uc3m.es will be sended when
    // job has finished (if we don't call setEmailConfig).
    job.setEmail(new String [] {"aamor@tsc.uc3m.es"});
    // File in which error output will be written, and
    // if script.e exists, program will delete it and create it
    job.setErrorPath ("/home/aamor/API/script.e", false);
    // The same with output.
    job.setOutputPath ("/home/aamor/API/script.o", false);
    job.setJobName ("APITry");
    // -q all.q (the only queue available currently).
    job.setQueue ("all.q");
    // Flag -pe mpi 8.
    job.setParallelEnvironment(8, "mpi");
    // Environmental variables mandatory for the job.
    job.setEnvironmentVariable("TMP", "/tmp");
    job.setEnvironmentVariable("MACHINES", "$TMP/machines");
    // Variables will be written on screen.
    job.printEnvironmentVariables();
    // Command which will be executed.
    job.setRemoteCommand("mpiexec -machinefile $MACHINES
                                                -np $NSLOTS $PWD/Hello");
    // true: we want to wait for the job to finish.
    job.runJob(true);
  }
}
```

DRMAA que, si bien permite la captura del evento de finalización de la ejecución del trabajo, lo que es fundamental para *Posidonia*, tiene ciertos inconvenientes que son solucionados por medio de la creación de `SGEJob`, un API que proporciona un acceso mucho más limpio a las funcionalidades de DRMAA.

Capítulo 4

Protocolo de comunicaciones.

La ejecución de trabajos en un *cluster* ya se ha explicado en el Capítulo 3, mientras que las particularidades que presenta el diseño de la interfaz gráfica se detallarán en el Capítulo 5. Así, en este punto se va a detallar cómo acceder a un *cluster* de forma remota por medio de Java.

De esta forma, la capa a explicar se enmarca dentro de *Posidonia* tal y como se muestra en la Figura 4.1, que es similar a la Figura 3.1 resaltando en cada caso el punto a tratar. El principal cometido del protocolo de comunicaciones es conseguir, en el ordenador personal y de forma segura, la misma funcionalidad que se tiene en el *cluster* con DRMAA, es decir, el envío y control de los trabajos que se están ejecutando por un cierto usuario mediante SGE. Sin embargo, esta capa no se conforma con eso sino que aporta una serie de características que son determinantes para el desarrollo de *Posidonia*, como la implementación de los trabajos en ejecución de los que dispone un cierto usuario o el historial de los proyectos enviados y ya ejecutados.

Además, este protocolo también se extiende para la aplicación Android con una serie de particularidades debidas al entorno diferente que se encuentra un dispositivo móvil frente a un ordenador personal, como ya se explicó en el Capítulo 2.

Así, la estructura que se va a seguir es la siguiente:

- Características del protocolo, en la que se explicarán las principales funcionalidades que aporta, en general, el uso de esta capa.
- Conexiones básicas SSH y SCP, donde se detallará la librería utilizada y la forma de acceder de forma segura a un servidor SSH.

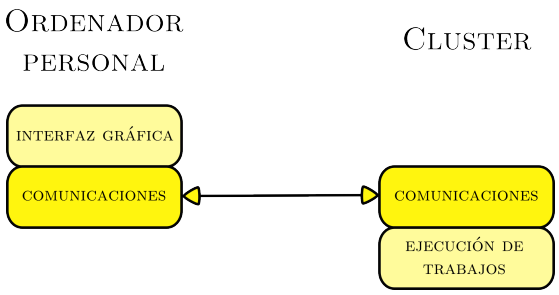


Figura 4.1: Estructura simplificada del código de *Posidonia*.

- Funcionalidades del protocolo en la aplicación de escritorio, exponiendo los servicios más importantes que ofrece la capa de comunicaciones.
  - Para ofrecer las funcionalidades de este protocolo se detallará, en primer lugar, el esquema de clases Java desarrolladas.
  - A continuación, se presentarán los principales ficheros de los que hace uso la capa.
  - Implementación de las funcionalidades, incluyendo diagramas de secuencia, en las que se mostrarán los mensajes intercambiados entre cliente y servidor para los casos de uso más importantes.
- Diferencias del protocolo en la aplicación Android, explicando los cambios que hay que introducir inevitablemente por el cambio de escenario. De la misma forma que en la sección anterior, se estructura:
  - Cómo se divide el código desarrollado para soportar los servicios desarrollados.
  - Implementación de las funcionalidades, presentando algunos diagramas de secuencia entre el dispositivo móvil, el servidor dedicado que actúa como pasarela y el *cluster*.

4.1. Características del protocolo.

Aquí se van a detallar las principales características con las que cuenta el protocolo de comunicaciones desarrollado y que, como es lógico, afectan a las propiedades de *Posidonia* que se explican en el Capítulo 2.

Así, el protocolo cuenta con:

- Seguridad: debido a la evolución de Internet es imprescindible que la comunicación con el *cluster* garantice la seguridad y la confidencialidad de los datos transferidos.

  Esta característica debe cubrir dos puntos:

  - El acceso al servicio SSH, en este caso el *cluster* `ash25b`, debe estar disponible únicamente para los usuarios que tengan esos privilegios, es decir, que puedan autenticarse en `ash25b`.
  - Los datos que se transfieren entre cliente y servidor deben estar encriptados para que sean únicamente comprensibles para las entidades involucradas.

  En relación con el primer punto se tiene una doble seguridad en el escenario de la Figura 2.2 del Capítulo 2, ya que para poder utilizar el *cluster* hay que, por un lado, tener acceso a la red del departamento (con una VPN) y, por otro, poseer el usuario y la contraseña necesarios para acceder a los servicios que proporciona el *cluster*. Para demostrar esto último, se usa el protocolo SSH (véase [11]), que está muy extendido para este tipo de casos.

  Con respecto al segundo punto, se usa SCP (para maś información véase [50]) que es una aplicación de SSH que garantiza la transferencia segura de archivos para el usuario autenticado por ese protocolo.

  La implementación de esta característica se hace mediante Java, que no incluye actualmente en ninguna de sus librerías la posibilidad de conectarse por medio de SSH y SCP. Por este motivo, se usa la librería JSch que es una implementación pura de este protocolo (se puede ver en [20]) en Java cuyas ventajas se detallarán en la siguiente sección.

- Extensibilidad: se desea que esta capa simplifique el desarrollo de cualquier interfaz gráfica u otro tipo de aplicación Java que quiera usar la funcionalidad proporcionada por sus métodos. De esta forma, el código se estructura básicamente en paquetes Java (que es una forma de organizar el código en base a sus funcionalidades) que, en el *cluster* se denominará `SSHSGE`, y que en el ordenador personal se llamará `Connections`; es necesario tener un paquete en cada extremo del protocolo puesto que `Connections` se comunicará con `SSHSGE` para usar los servicios proporcionados por el API SGEJob en el *cluster*, y `SSHSGE` se pondrá en contacto con `Connections` para el envío de ciertas notificaciones, como se observará en las siguientes secciones. Por otro lado, en el caso de la aplicación Android el esquema es más particular y se explicará en el último punto del capítulo.

- Arquitectura cliente-servidor y servidor-cliente: el ordenador personal (o el dispositivo móvil) será normalmente el cliente que pida ciertos servicios como, por ejemplo, la ejecución de un cierto trabajo en SGE, al servidor, que

es el *cluster*. Así se configura una arquitectura clásica cliente-servidor; sin embargo, dadas las características de la aplicación desarrollada, *Posidonia*, hay que incluir una estructura servidor-cliente ya que en el extremo del cliente se levantará un servidor TCP (*Transmission Control Protocol*, véase [22]) para recibir ciertas notificaciones como son las relacionadas con el acceso concurrente a ciertos ficheros o la finalización de los trabajos en ejecución.

Este tipo de servidores no tienen ninguna seguridad inherente a su uso como sí la tienen los servidores SSH; no obstante, ahora no es necesaria la seguridad porque los mensajes que se transmiten son del tipo `Job Ready` o `File Available`, que no aportan información que deba ser tratada de forma confidencial.

Cabe destacar también que en la aplicación Android será imposible habilitar un servidor TCP para la recepción de estas notificaciones por lo que se perderán algunas de las funcionalidades que ofrece la aplicación de escritorio como es el aviso al usuario de que un trabajo ha acabado su ejecución.

- Orientación a usuario y no a conexión: gracias a que toda la información sobre las tareas en ejecución y ejecutadas se encuentran en remoto, esto es, en el *cluster*, es indiferente que el usuario esté conectado o no para que se mantenga la funcionalidad sobre estos trabajos.

  Esta característica es fundamental debido a las particularidades de los proyectos con los que trata *Posidonia* que son, normalmente, de una duración prolongada en el tiempo de forma que es bastante habitual que el usuario no esté conectado durante todo el intervalo de ejecución de la tarea.

- Adaptabilidad y movilidad: como esta capa, al igual que toda la aplicación, está escrita íntegramente en Java y usa librerías de este mismo lenguaje, se puede utilizar en prácticamente cualquier entorno de trabajo siempre y cuando se tenga acceso a una máquina virtual de Java, JVM. Por este motivo, la adaptación de esta capa a Android es bastante sencilla ya que este sistema operativo está basado en Java, como se observará más adelante.

  Por otra parte, la orientación a usuario del protocolo proporciona una movilidad casi absoluta, ya que no depende del equipo que utilice *Posidonia* porque las funcionalidades y la información mostrada será exactamente la misma. El desarrollo de la capa de comunicaciones para entornos Android fortalece esta característica, ya que se puede acceder a la información y el control que ofrece la aplicación desde cualquier dispositivo móvil que utilice este sistema.

- Eficiencia: al tratarse de un protocolo de comunicaciones y dada la situación actual en la que el ancho de banda es un recurso limitado, los mensajes enviados por esta capa son los mínimos imprescindibles para proporcionar la

funcionalidad que se busca. Además, esta propiedad es importante también para que el *cluster* pueda atender múltiples peticiones a la vez de distintos usuarios. Por ejemplo, si los datos de autenticación de un usuario son correctos no se vuelven a comprobar para cada mensaje siempre y cuando no se cambien.

De esta forma, la programación de la capa es más compleja pero se consigue un protocolo lo más eficiente posible.

- Acceso concurrente a los ficheros empleados: para proporcionar las funcionalidades que se describieron en el Capítulo 2 se hace uso de tres archivos almacenados en el *cluster* para cada usuario que son:

  1. `runningJobs`, que básicamente almacena información sobre cada trabajo que está en ejecución en el *cluster*.
  2. `history`, que guarda datos sobre las tareas que ya han sido ejecutadas y que el usuario ha decidido mantener en el historial.
  3. `pendingJobs`, que sirve para informar al usuario de los trabajos que han finalizado en el *cluster* mientras ha estado desconectado, es decir, entre sesión y sesión de *Posidonia*.

  Estos ficheros residen en la carpeta utilizada por la aplicación y llamada `Posidonia`. Más concretamente, dentro de este directorio hay una división en base a la aplicación utilizada (hasta ahora, FortranMod o Matlab), de forma que estos tres archivos se encuentran, desde la carpeta `home` de cada usuario, en `/Posidonia/Matlab` o en `/Posidonia/FortranMod`.

  Por otro lado, este acceso concurrente también se emplea a la hora de acceder a la *cookie* de un trabajo, que es un fichero en el que se guardan opciones relacionadas con cada tarea como las rutas de los ficheros de salida o el estado del trabajo, y cuya utilidad se detallará en siguientes secciones.

  El hecho de que estos ficheros estén alojados en remoto aporta la movilidad y la orientación a usuario que caracteriza a la aplicación; sin embargo, puede ocurrir que acabe un trabajo mientras se está descargando `runningJobs`.

  Para evitar esto se ha desarrollado un sistema de acceso concurrente a estos archivos usando clases propias de Java y el servidor TCP en el equipo local de forma que se eviten casos como el propuesto que produzcan un comportamiento incorrecto de la aplicación.

Todas estas características son las que hay que tener en cuenta a la hora de desarrollar este protocolo de comunicaciones, así que en las siguientes secciones se detallará la funcionalidad que aporta esta capa y la implementación de estas propiedades.

4.2. Conexiones básicas SSH y SCP.

Como se ha explicado en el apartado anterior, para garantizar la seguridad en el acceso al *cluster* y la confidencialidad en la transferencia de información entre cliente y servidor se ha decidido utilizar el protocolo SSH (véase [11, 10, 12]) y la aplicación SCP (véase [50]).

Esta elección no es casual: en primer lugar, su amplia expansión hace que los problemas a la hora de implementar una solución de este tipo sean resueltos con más facilidad; además, en el *cluster* hay lanzado un servidor SSH que permite el uso de este protocolo y sus aplicaciones; y, por último, hay multitud de librerías en Java que implementan este tipo de conexiones.

Ya se ha expuesto en anteriores apartados la decisión de desarrollar íntegramente la aplicación en lenguaje Java debido a las numerosas ventajas que ello conlleva; por este motivo, se ha elegido una librería Java para establecer conexiones seguras por medio de SSH al no incluir Java esta conectividad en sus librerías nativas. Cabe destacar que no es razonable desarrollar una librería propia para esta aplicación puesto que SSH está muy expandido y no es el propósito principal de este proyecto cómo hacer conexiones seguras entre un cliente y un servidor.

Se analizan así varias alternativas, entre las que se pueden destacar:

1. Zehon, para más información véase [51]. Es una librería que está ampliamente documentada y con una programación orientada a objetos, es decir, que la integración con Java es muy sencilla. Sin embargo, incluye únicamente la implementación de aplicaciones relacionadas con la transferencia de ficheros y el protocolo de comunicaciones necesita también el uso de SSH, por lo queda descartada.

2. SSHTools, si se desea saber más véase [52]. Esta librería implementa tanto SSH como SCP y la programación es también fácilmente integrable en Java. No obstante, no se encuentra muy expandida y por ello, el soporte es peor que en la librería escogida, JSch.

3. JSch, cuyo sitio oficial es [20]. JSch es una librería que implementa puramente el protocolo SSHv2 (que mejora el anterior SSH por algunos problemas de seguridad) y se encuentra ampliamente expandida ya que es usada por programas como Eclipse (programa muy conocido para el desarrollo de códigos Java, véase [53]) o NetBeans (alternativa a Eclipse, véase [54]), lo que supone que el soporte en la red es abundante.

   Sin embargo, también tiene ciertos inconvenientes:

- JSch se programa en la filosofía de *self documenting code*, o código que se explica solo. De esta forma, no hay ninguna documentación por parte de los desarrolladores ya que se piensa que para conocer qué hace un método basta con observar su código.
- La programación no es tan orientada a objetos como las otras librerías, lo que supone que la integración con Java no es tan inmediata y el código es algo más complejo.

No obstante, estos problemas se solucionan gracias, por un lado, a la documentación no oficial que se referencia en [55] y que es tremendamente útil a la hora de implementar los códigos desarrollados en esta capa; y, por otro, a que el uso de las funciones de JSch es bastante similar al que requiere DRMAA para utilizar sus servicios, por lo que la familiarización es rápida.

Con todas estas opciones, se decide el uso de JSch por sus ventajas y a continuación se va a detallar cómo se realiza una conexión básica por medio de esta librería.

La forma de compilar y ejecutar un código Java que utiliza JSch es análoga a cómo se usa DRMAA en el Capítulo 3, por lo que no se insistirá en este aspecto.

Así, el código del Fichero 4.1 implementa una conexión SSH en la que se listan los archivos de la carpeta actual por medio del comando `ls -l`. En él se puede observar que, al igual que se hacía en DRMAA con `SessionFactory`, la clase `JSch` funciona como un creador de sesiones por medio de las cuales se establecen los parámetros de configuración de la conexión SSH que hay que especificar antes de proceder a la conexión por medio de `session.connect()`. Entre estos parámetros se encuentran el comando que se desea ejecutar, por medio de `setCommand`, y el `login` del usuario que desea acceder al `host`. Además, JSch permite mostrar por pantalla el resultado de la ejecución del comando en el *cluster* lo cual será indispensable para saber si ha sucedido algún error a la hora de ejecutar la instrucción enviada.

Por otro lado, analizando el código se puede ver que para mejorar la eficiencia del protocolo de comunicaciones, el objeto `session` debe mantenerse para todas las funciones de red de la aplicación ya que es el que establece la conexión con el servidor SSH del `host`. Sin embargo, `channelExec` no se puede conservar puesto que el comando que se desea ejecutar no es el mismo, en general, para todas las tareas de red de *Posidonia*.

Finalmente, en el Fichero 4.2 se muestra cómo se realiza una conexión SCP en la que se descarga el fichero `ejemplo` del `host` indicado por parámetro. Se puede

Fichero 4.1: Conexión SSH con JSch

```
1
  /** In this method files are listed in current directory.
    * @param user The name of the user.
    * @param host The name of the host (cluster).
    * @param password The password of the user in the cluster.
6   */
  public void unlockCookie (String user, String host, String password) {
    // SSH Port.
    Integer port = 22;
    // JSch is a class in which we get Session.
11  JSch jsch = new JSch();
    try {
      // This is the same to do ssh user@host to the port port.
      Session session = jsch.getSession(user, host, port);
      // UserInfo gives a interface with user.
16    UserInfo ui = new UserInfoSSH();
      session.setUserInfo(ui);
      session.setPassword(password);
      // Connection is opened with default timeout.
      session.connect();
21    // With this, we can execute commands in cluster.
      ChannelExec channelExec = (ChannelExec)session.openChannel("exec");
      // This has to be called before connect.
      InputStream in = channelExec.getInputStream();
      channelExec.setCommand("ls -l");
26    // Execution channel is opened.
      channelExec.connect();
      // With this we read whatever is written on screen in cluster.
      BufferedReader reader = new BufferedReader(new InputStreamReader(in));
      String linea = null;
31    int index = 0;
      while ((linea = reader.readLine()) != null) {
        ++index;
        System.out.println(index + " : " + linea);
      }
36    // Both sides are closed.
      channelExec.disconnect();
      session.disconnect();
    }
    catch(Exception e) {
41    e.printStackTrace();
    }
  }
```

Fichero 4.2: Conexión SCP con JSch

```
1
/** In this method the file "Ejemplo" is downloaded
  * to the current local directory.
  * @param user The name of the user.
  * @param host The name of the host (cluster).
6 * @param password The password of the user in the cluster.
  */
public void unlockCookie (String user, String host, String password) {
  // SSH Port.
  Integer port = 22;
11 // JSch is a class in which we get Session.
  JSch jsch = new JSch();
  try {
    // This is the same to do ssh user@host to the port port.
    session = jsch.getSession(user, host, port);
16  // UserInfo gives a interface with user.
    UserInfo ui = new UserInfoSSH();
    session.setUserInfo(ui);
    session.setPassword(password);
    // Connection is opened with default timeout.
21  session.connect();
    // SFTP Channel is open to transfer files.
    ChannelSftp sftp = (ChannelSftp)session.openChannel("sftp");
    sftp.connect();
    // Current local directory is changed with value of $PWD in local.
26  sftp.lcd(sftp.lpwd());
    // Current remote directory is changed with value of $PWD in remote.
    sftp.cd(sftp.pwd());
    // File "Ejemplo" is downloaded to the current local directory.
    sftp.get(sftp.pwd()+"/Ejemplo", sftp.lpwd()+"/Ejemplo");
31  // Both sides are closed.
    sftp.disconnect();
    session.disconnect();
  }
  catch(Exception e) {
36  e.printStackTrace();
  }
}
```

observar que la estructura es muy parecida a la de la conexión SSH, de forma que lo que se ha comentado sobre la eficiencia del protocolo es válido aquí con la salvedad de que el objeto `sftp` se puede conservar entre funciones de red.

Además, en el Fichero 4.2 se pone de manifiesto que la implementación de SSH es pura, ya que se puede hacer en remoto las mismas operaciones que se pueden hacer en el sistema local de archivos: `cd, pwd`... Si se desea más información sobre los métodos que se pueden usar del objeto `sftp`, se puede consultar [55].

En resumen, la librería Java que se decide utilizar para implementar la conexión segura al *cluster* es JSch ya que se encuentra muy expandida y su uso es muy similar al de DRMAA del Capítulo 3, garantizando así, por un lado, la seguridad en el acceso al *cluster* y, por otro, la confidencialidad en los datos transferidos gracias a SCP.

4.3. Funcionalidades del protocolo en la aplicación de escritorio.

En este apartado se van a detallar cuáles son los servicios que ofrece la capa de comunicaciones, y cómo se lleva a cabo su implementación. Para esto último se explicarán cuáles son las clases Java que aportan esta funcionalidad y los ficheros que son usados por el protocolo para transferir información entre el ordenador personal y el *cluster*.

En primer lugar, es conveniente recordar el entorno en el que se trabaja, que no es más que el referido en la Figura 2.2. De esta forma, aunque `ash25b`, que es el *host* en el que reside el *cluster* no está abierto a Internet, gracias al uso de la VPN es como si el ordenador personal estuviera en la red en la que se encuentra `ash25b`, así que se tiene acceso directo a él.

Con esto, el protocolo de comunicaciones debe proporcionar, de forma segura (para lo que se usa la librería JSch tal y como se detalló en la sección anterior) la siguiente funcionalidad:

- Envío de trabajos al gestor de colas: es el principal servicio que ofrece esta capa y su implementación se verá más adelante. Hay que diferenciar varias partes en esta funcionalidad:

  1. La creación de dos ficheros, `JobConfig` que sirve para comunicar al *cluster* las opciones de configuración de la tarea; y una *cookie* que es un fichero relacionado con el trabajo que guarda datos específicos sobre él.
  2. La transferencia de los ficheros de entrada asociados al proyecto, y los ficheros `JobConfig` y la *cookie* del trabajo para que el *cluster* conozca los parámetros de ejecución relacionados con la tarea y otros datos que se incluyen en la *cookie*.
  3. El envío del trabajo al gestor de colas usando los ficheros de entrada transferidos y los parámetros indicados por `JobConfig` y la *cookie*.
  4. Cuando la tarea ya ha llegado al gestor, se avisa al usuario por medio de una notificación como la que se muestra en la Figura 2.7.

- Notificación al usuario de un trabajo que finaliza su ejecución: se ha decidido que se haga en esta capa porque estas notificaciones son las mismas para todas las aplicaciones que puedan usar este protocolo de comunicaciones, esto es, tiene el mismo formato y varía mínimamente el mensaje mostrado.

Esta funcionalidad es posible gracias a la característica de DRMAA, e incluida en el API SGEJob que posibilita esperar a que la tarea finalice su ejecución para notificar al usuario por medio de un código Java tal y como se observó en el Capítulo 3.

En concreto, este servicio se hace modificando ligeramente el API desarrollado de forma que los pasos que se han de seguir son los siguientes:

1. Cuando el trabajo ha finalizado su ejecución, se ejecuta un código en el que, entre otras instrucciones, se envía una notificación al servidor TCP que ha lanzado el protocolo en el equipo local. La localización de este servidor (es decir, su dirección IP que indica la interfaz de red en la que se encuentra el usuario, y su número de puerto) se obtiene analizando la *cookie* del trabajo, que también indica si está disponible. Si la notificación no se puede enviar, significa que el servidor TCP está desactivado así que no se puede avisar al usuario por esta vía.
2. La notificación se recibe en el servidor TCP, de forma que los resultados de la ejecución pueden descargarse automáticamente a la ruta que indicó la capa superior; si no, simplemente salta un aviso con el aspecto de la Figura 2.9

- Notificación al usuario de un trabajo que ha finalizado su ejecución mientras ha estado desconectado: para esto se usa un archivo residente en el *cluster* llamado `pendingJobs` que incluye las tareas que han acabado su ejecución mientras el servidor TCP ha estado inhabilitado. Se deja a la capa superior que el procesamiento del archivo se haga en el momento en el que se desee: por ejemplo, a la hora de acceder a la tabla de trabajos en ejecución; el protocolo de comunicaciones únicamente ofrece la información que se introduce en `pendingJobs`.

- Comprobación de los datos de autenticación introducidos: se comprueba antes de realizar ninguna operación de red que el *login* del usuario en el *cluster* es correcto. Para ello se hace una conexión de prueba por medio de JSch y si arroja un error de autenticación, se comunica a la capa superior que el *login* es erróneo.

  Por otro lado, la eficiencia de la que se hablaba como característica del protocolo en el caso de esta comprobación se deja a la capa superior; es decir, la interfaz gráfica, por ejemplo, debe saber cuándo es necesario que se vuelvan a comprobar los datos de autenticación.

- Acceso concurrente a los ficheros: para evitar problemas que ya se especificaron al hablar de las características del protocolo, la capa de comunicaciones garantiza la concurrencia en el acceso a los ficheros de la siguiente forma:

1. Si se desea modificar desde el equipo local a un archivo del *cluster*, se bloquea el acceso al fichero por medio de una clase Java llamada `FileLock` que sirve para este tipo de casos. Una vez que se consigue este acceso, se envía una notificación al servidor TCP del equipo local para que sepa que ya está disponible el archivo.
2. Cuando se recibe la notificación, se descarga el archivo, se hacen los cambios pertinentes y se vuelve a enviar al *cluster*.
3. Una vez finalizado el envío, se libera este bloqueo por medio de la misma clase Java.
4. Si la modificación se hace desde el propio *cluster*, se usa `FileLock` para bloquear el acceso hasta que los cambios hayan sido realizados. La ventaja del uso de esta clase es que el bloqueo se hace entre distintas llamadas a `java` que es como se accede a los archivos.

Esta funcionalidad se explicará con más detalle al final de esta sección.

- Identificación unívoca de los trabajos: para que no haya conflictos de nombres ya que es bastante habitual que el usuario nombre de la misma forma a dos proyectos, en el protocolo de comunicaciones se trabaja con un identificador unívoco que consiste en la concatenación del nombre dado por el usuario y la fecha, en milisegundos, en la que se envía el trabajo desde la capa superior a la capa de comunicaciones. Así se evita cualquier tipo de conflicto y, además, es transparente para la capa superior puesto que uno de los campos de la *cookie* asociada al trabajo incluye el nombre indicado por esa capa.

- Datos sobre qué trabajos se encuentran aún en ejecución: por medio del acceso concurrente al fichero `runningJobs` se obtiene cierta información sobre todos los trabajos actualmente en ejecución enviados por un usuario y que permite obtener todos los datos asociados a cada tarea. Así, relacionado con esto se ofrece la siguiente funcionalidad:

  - Información asociada a un trabajo en ejecución: el protocolo desarrollado da toda la información que se puede conseguir gracias al comando de SGE (gestor de colas utilizado en el caso considerado) `qstat` sobre una cierta tarea en ejecución; la forma de procesar y mostrar la información al usuario se deja a la capa superior.
  - Control de los trabajos en ejecución: esta capa permite eliminar una cierta tarea que aún esté en ejecución en el gestor de colas. Además, aparte de sacar al trabajo del gestor de colas (función equivalente a la desempeñada por el comando de SGE `qdel`) elimina todo su rastro en el *cluster*, es decir, borra todos sus archivos de entrada y de configuración lo que contribuye al uso óptimo de un recurso limitado como es el disco duro de uno de estos equipos.

- Historial de trabajos: gracias al archivo `history`, se conservan ciertos datos sobre los trabajos que han sido ejecutados en el *cluster* por un usuario a través de *Posidonia* y que ha decidido mantener. Con respecto a este historial se ofrecen las siguientes posibilidades:

  - Borrado de una tarea del historial: si el usuario decide eliminarla definitivamente, aparte de sacar al trabajo del fichero `history`, se elimina todo su rastro en el *cluster* de forma similar a lo que se hacía en las tareas en ejecución.
  - Descarga de los ficheros de entrada y de salida asociados a un trabajo: se habilita una función de repositorio para todas las tareas presentes en el historial de forma que se puede obtener con qué entradas se hizo la ejecución de ese proyecto. Para ello, la capa superior debe aportar cuál es la ruta local en la que se quieren descargar los archivos.

- Disponibilidad del acceso a Internet y conectividad desde el *cluster*: el protocolo ofrece la posibilidad de saber si Internet es alcanzable desde el equipo local (lo que se puede usar para la depuración de fallos en la capa superior) y qué dirección IP de las que tiene asignadas el usuario es la que es accesible desde el *cluster*.

  Para lo primero se intenta contactar con una página web del dominio en el que se encuentra alojado el *cluster* y, para lo segundo, se hace un `ping` (herramienta muy usada en interconexiones de redes para saber si hay conectividad entre dos máquinas) desde el *cluster* a las direcciones IP del equipo local; si la ejecución de este comando es correcta, significa que esa IP es alcanzable desde el *cluster* por lo que podrá enviar notificaciones al servidor TCP levantado en local.

Se puede observar que la funcionalidad que ofrece el protocolo de comunicaciones es muy amplia: esto se debe a una de las propiedades de *Posidonia*, que es la extensibilidad: la capa superior, que en el caso de esta aplicación es la interfaz gráfica, se encarga únicamente de dar unos ciertos datos de entrada (como los parámetros de configuración del trabajo o qué tarea se desea eliminar de los proyectos en ejecución o del historial) y procesar la información obtenida por medio de la capa de comunicaciones y mostrarla de una cierta forma al usuario.

Por otro lado, a la hora de describir los servicios que ofrece esta capa ya se ha explicado brevemente su implementación; en siguientes secciones, se detalla cómo funciona este protocolo de comunicaciones y, en los casos más importantes, el paso de mensajes entre el equipo local y el *cluster* necesario para ofrecer esta funcionalidad.

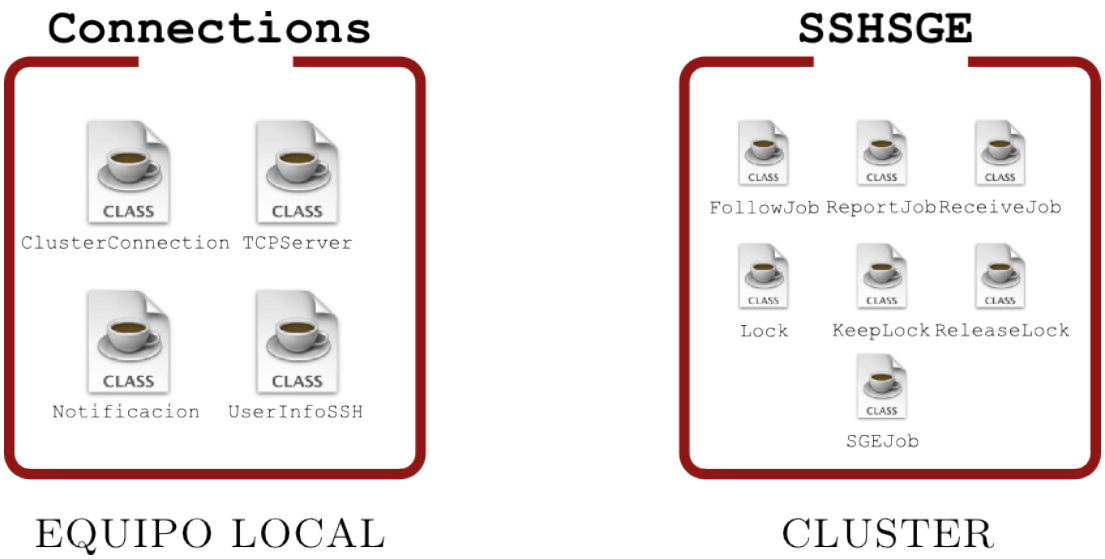


Figura 4.2: Esquema general de la estructura de clases del protocolo de comunicaciones.

4.3.1. Esquema de clases Java desarrolladas.

En esta subsección se va a mostrar cuál es la estructura de las clases Java desarrolladas en cada paquete.

Como ya se ha comentado, hay dos paquetes: uno en el equipo local llamado `Connections`, y otro en el *cluster* llamado `SSHSGE`. El esquema general se muestra en la Figura 4.2; a continuación se explica brevemente cuál es el cometido de cada clase:

- Paquete `Connections`: es el que se encuentra en el equipo local y, por tanto, el que se encarga de ofrecer las funcionalidades que se han descrito anteriormente en colaboración con `SSHSGE`; en concreto, se encarga de todas las transferencias de archivos necesarias (como no podía ser de otra manera ya que desde el *cluster* no se puede enviar ningún fichero al equipo local mediante SCP porque, para ello, debería habilitarse un servidor SSH en ese extremo, lo que no se hace por su complejidad) y de ejecutar los comandos necesarios para implementar los servicios ofrecidos.

  Las clases que contiene este paquete son:

  1. `ClusterConnection`: es el núcleo del protocolo y reúne los métodos que implementan la práctica totalidad de los servicios ofrecidos por la

capa de comunicaciones. Toda la conectividad con el *cluster*, es decir, todas las llamadas a JSch, se reúnen en esta clase.

2. `TCPServer`: implementa el servidor TCP necesario para recibir las notificaciones del *cluster*. Utiliza las librerías nativas de Java para el establecimiento de este tipo de comunicaciones, que están basadas en *sockets*. También debe incluir el procesamiento indispensable para los mensajes recibidos, de forma que interactúa con `ClusterConnection` en el acceso concurrente y la finalización de trabajos.

   Además, hay que destacar que el servidor TCP no se lanza en un único puerto ya que *Posidonia* no se trata de una aplicación registrada. Así, el servidor intenta lanzarse en el puerto 2012 (puerto por defecto) y, si no lo consigue, escanea secuencialmente hacia arriba los puertos hasta encontrar uno que sí pueda utilizar.

3. `Notificacion`: es una clase eminentemente gráfica en la que se muestra la notificación con el icono de *Posidonia* y un cierto mensaje que depende de la capa superior.

4. `UserInfoSSH`: es necesaria para la implementación de las conexiones JSch.

- Paquete `SSHSGE`: radica en el *cluster* e incluye el API desarrollada en el Capítulo 3, por lo que su función es, principalmente, el envío de trabajos al gestor de colas y la modificación de los archivos `runningJobs`, `pendingJobs` y `history`. También incluye las clases necesarias para garantizar el acceso concurrente a los ficheros.

  Los archivos que incluye este paquete son los siguientes:

  1. `SGEJob`: clase encargada principalmente de establecer los parámetros de configuración del trabajo que se manda al gestor de colas y del propio envío tal y como se explicó ampliamente en el Capítulo 3. Por otro lado, también se encarga de interactuar con `ReportJob` para notificar al usuario y con `FollowJob` para actualizar los archivos `pendingJobs`, `runningJobs` y `history`.

  2. `ReceiveJob`: es utilizada por `ClusterConnection` para enviar un trabajo a SGE; así, una vez que se ha enviado el fichero `JobConfig` y la *cookie* asociada a la tarea al *cluster*, se ejecuta este código para que se procesen los parámetros de configuración y se envíe el trabajo al gestor de colas por medio de `SGEJob`.

  3. `ReportJob`: la funcionalidad que ofrece esta clase es empleada por `SGEJob` cuando ha finalizado para actualizar algunos campos de la *cookie* asociada a la tarea y para enviar una notificación al servidor TCP.

4. `FollowJob`: es llamada por `SGEJob` cuando ha finalizado un trabajo, de forma que:
   - Añade una tarea al fichero `runningJobs` cuando llega al gestor de colas.
   - Introduce un trabajo al fichero `history` cuando éste ha acabado su ejecución y lo elimina de `runningJobs`.
   - Si la notificación al servidor TCP no ha podido ser enviada, añade la tarea al fichero `pendingJobs`.

   Todo esto se hace garantizando el acceso concurrente a los ficheros modificados.
5. `Lock`: su cometido es usar la clase `FileLock` de Java para bloquear el acceso a un cierto fichero si está siendo modificado, y liberar este bloqueo cuando han terminado de hacerse los cambios.
6. `KeepLock`: es utilizada por `ClusterConnection` para bloquear al acceso a un archivo que va a ser modificado. De esta forma, esta clase es básicamente un método `main` (que se invoca por línea de comandos al ejecutar `java KeepLock ...`) que se ejecuta por medio de una conexión SSH desde el equipo local.
7. `ReleaseLock`: es empleada por `ClusterConnection` de forma análoga a `KeepLock` para liberar el bloqueo de un cierto fichero porque los cambios que se estaban realizando ya están disponibles en el *cluster*.

Se puede observar claramente que se ha buscado seguir una modularidad en el desarrollo del código en base a las funciones que implementa cada clase, de forma que la comprensión de la arquitectura empleada es más sencilla si cada clase realiza un conjunto delimitado de funciones.

4.3.2. Principales ficheros del protocolo.

En este apartado se van a detallar cuáles son los y qué información aportan los ficheros que ya se han ido introduciendo en las anteriores secciones.

Para garantizar la movilidad, todos estos ficheros residen en el *cluster* de forma que se descargan por medio de SCP al equipo local en el que se encuentre el usuario en cada momento. Además, para asegurar que la información que contienen es fiable y está actualizada, se implementa un mecanismo de acceso concurrente a los archivos que ya se ha comentado brevemente y que se detallará en la siguiente sección.

Así, hay que distinguir entre dos clases de archivos:

Fichero 4.3: runningJobs

```
1 Ejemplo1341977924699/4135/Ejemplo
```

- Asociados al usuario, que son los que se utilizan para conseguir información sobre los trabajos de un cierto usuario que están ejecutándose o que ya han finalizado. Son tres, cada uno relacionado con una funcionalidad distinta:

  1. `runningJobs`, que incluye los trabajos que se encuentran actualmente en ejecución. Se puede observar un ejemplo de este archivo en el Fichero 4.3, en el que se muestran los campos (delimitados por el caracter '/') que contiene:

     - Identificador unívoco del trabajo `Ejemplo` en el protocolo de comunicaciones. Se utiliza para saber en qué carpeta se encuentran los ficheros relacionados con esta tarea, ya que la organización del sistema de ficheros es:
       - Una carpeta `Posidonia` que contiene todo lo relacionado con la aplicación.
       - Dentro de `Posidonia`, carpetas de aplicación como por ejemplo `FortranMod` o `Matlab`,
       - En el interior de uno de estos directorios en función del tipo de programa al que pertenezca la tarea enviada, una carpeta llamada igual que el identificador unívoco de la tarea en la que se encuentran todos los archivos de entrada y de salida relacionados con el trabajo.

       De esta forma, se consigue una organización de ficheros limpia y ordenada, y poco molesta para un usuario que acceda con normalidad al *cluster*.
     - `PID` de la tarea, es decir, el identificador del trabajo en el gestor de colas. Es necesario en el caso de que se desee eliminar el proyecto antes de que acabe su ejecución, y también puede ser presentado al usuario por la interfaz gráfica.
     - Nombre dado por el usuario a la tarea `Ejemplo`, utilizado para mostrar al usuario su denominación del trabajo y no su identificador unívoco.

  2. `history`, que contiene todos los trabajos que han sido enviados mediante esta aplicación y que ya han acabado su ejecución. Un ejemplo de este archivo aparece en el Fichero 4.4, donde se pueden observar los campos, de nuevo delimitados por el caracter '
     ' que incluye el fichero:

Fichero 4.4: history

```
Ejemplo1341369739452/Ejemplo/1341369801463/4117/Ended
Ejemplo1341369983299/Ejemplo/1341370049355/4118/Ended
Ejemplo1341371129955/Ejemplo/1341371208292/4119/Ended
Ejemplo1341373410543/Ejemplo/1341373511306/4120/Ended
Ejemplo1341373572707/Ejemplo/1341373641314/4121/Ended
EjemploTablet1341406425093/EjemploTablet/1341406489122/4123/Ended
Ejemplo1341977924699/Ejemplo/1341977982488/4135/Ended
```

Fichero 4.5: pendingJobs

```
Ejemplo/ended
```

- El identificador unívoco del trabajo, por los motivos que se incluía en `runningJobs`.
- El nombre dado por el usuario a la tarea, por las mismas razones que en `runningJobs`.
- El instante en el que el trabajo acabó su ejecución, en milisegundos. Esto puede ser empleado por la capa superior para mostrar la fecha de finalización de la tarea.
- `PID` de la tarea, que puede ser utilizado por la interfaz gráfica para mostrárselo al usuario.
- El estado en el que acabó el trabajo. `Ended` supone que ha finalizado correctamente, aunque no asegura que los resultados sean los esperados.

Además, en el Fichero 4.4 se puede ver que la información de cada tarea está inscrita en la misma línea, de forma que la separación entre trabajos se hace renglón a renglón, al igual que sucede en `runningJobs` y `pendingJobs`.

3. `pendingJobs`, que comprende todos los trabajos que han acabado su ejecución mientras el servidor TCP estaba inhabilitado. De nuevo, se muestra una instancia de este archivo en el Fichero 4.5, en el que se pueden ver los campos, separados por '/' que comprende el fichero:
   - El nombre real del trabajo.
   - El estado en el que finalizó la tarea.

   Este fichero se utiliza principalmente para mostrar una notificación al usuario indicándole que un trabajo ha terminado mientras ha estado desconectado y que los resultados pueden descargarse desde el historial, por lo que no se necesita ningún campo más.

- Asociados al trabajo, que se emplean para una única tarea y contienen

Fichero 4.6: JobConfig

```
lE
jEO
cwd
name Ejemplo1341977924699
VMem 4G
Queue all.q
pE orte 1
rC echo prueba | /opt/matlab/bin/matlab -nosplash -nojvm -nodisplay
user aamor
app Matlab
realName Ejemplo
```

información relacionada con ella. Hay dos archivos:

1. `JobConfig`, que se utiliza para especificar desde el equipo local cuáles son los parámetros de configuración de un cierto trabajo. Se expone un ejemplo de este archivo en el Fichero 4.6, y las opciones que acepta (una en cada línea) son las siguientes:
   - `jEO`, si se desea unir el fichero de error con el de salida.
   - `lE`, si todas las variables de entorno se cargarán en el entorno de SGE.
   - `cwd`: si se desea poner el trabajo en el mismo directorio que fue lanzado.
   - `email name@addr.ess`, si se va a enviar algún tipo de notificación a la dirección de correo especificada.
   - `emailConfig ase`, para especificar la configuración del envío de avisos a la dirección de correo electrónico. Por ejemplo, si se indica `emailConfig se`, un correo será enviado cuando el trabajo empiece y cuando acabe.
   - `name JobID`, para determinar el nombre del trabajo. Se emplea el identificador unívoco del protocolo.
   - `realName jobName`, para conservar el nombre dado por el usuario.
   - `FSize size`, para limitar un tamaño máximo de memoria que la tarea puede escribir en disco.
   - `VMem size`, para acotar un tamaño máximo de memoria virtual que la tarea puede usar en cada nodo.
   - `MinMem size`: se puede utilizar si se desea que el trabajo no se ejecute hasta que el tamaño de la memoria especificado por parámetro no esté disponible.
   - `NodeQueue Node Queue`: para forzar a que un trabajo se mande a una cierta cola de un cierto nodo.

Fichero 4.7: Cookie asociada a la tarea

```
JobName Ejemplo1341977924699
Connection true
Host 192.168.111.26
InputPath /home/aamor/Posidonia/Matlab/Ejemplo1341977924699/lineal.m
InputPath /home/aamor/Posidonia/Matlab/Ejemplo1341977924699/cuadratica.m
InputPath /home/aamor/Posidonia/Matlab/Ejemplo1341977924699/plots.mat
InputPath /home/aamor/Posidonia/Matlab/Ejemplo1341977924699/prueba.m
RealName Ejemplo
Port 2012
JobState ended
OutputPath /home/aamor/Posidonia/Matlab/Ejemplo1341977924699/
    Ejemplo1341977924699.o
OutputPath /home/aamor/Posidonia/Matlab/Ejemplo1341977924699/script2.mat
ErrorPath joined
DownloadResults yes
```

- `pE peType nSlots`: si se desea usar un cierto mecanismo de paralelización como `mpi` y el número de *slots*, o *cores*, que se desean utilizar.
- `Queue name`: para obligar a que la tarea se envíe a la cola indicada por parámetro.
- `tLim hour min secs`: para que el trabajo tenga un tiempo límite de ejecución.
- `rC RemoteCommand`: que incluye el comando que se va a ejecutar mediante SGE.
- `app appName`: indica el programa que se va a utilizar, como por ejemplo `FortranMod` o `Matlab`, tal y como se mostró en el Capítulo 2.

Básicamente, se incluyen todas los parámetros de configuración que ofrece SGE y que se detallaron en el Capítulo 3. De esta forma, este fichero se puede eliminar una vez que el trabajo se encuentra en el gestor de colas.

2. Una *cookie* asociada a cada tarea (denominada con la concatenación del identificador unívoco del trabajo con `cookie`) que guarda información de estado sobre ella como si el usuario está conectado o dónde enviar la notificación al acabar el trabajo. Se puede observar un ejemplo de un archivo de este tipo en el Fichero 4.7, de forma que los campos que incluye son los siguientes:
   - `JobName name`: el identificador unívoco del trabajo.
   - `Connection true/false`: para indicar al *cluster* si el servidor TCP está habilitado.
   - `Host value`: muestra la dirección IP en la que se encuentra lanzado el servidor TCP.

- `Port value`: como la aplicación no tiene un puerto fijo debido al escaneo de puertos que se comentó anteriormente, es necesario para la recepción de notificaciones que el *cluster* sepa en qué puerto se lanzó el servidor.
- `RealName name`: el nombre que le dio el usuario al trabajo.
- `InputPath path`: se utiliza para que el *cluster* sepa dónde están los ficheros que se deben ejecutar. Si hay más de un archivo, se escribe cada uno en una línea al igual que para `OutputPath`.
- `OutputPath path`: se emplea para que el equipo local sepa dónde se encuentran los resultados que hay que descargar.
- `ErrorPath path`: si el fichero de error no está unido al de salida (como en el ejemplo), aquí se muestra en qué ruta se encuentra dicho fichero.
- `JobState value`: con esto se conoce el estado en el que se encuentra actualmente el trabajo.
- `DownloadResults yes/no`: este campo se utiliza si el trabajo no acaba en la misma sesión en la que fue enviado, ya que en ese caso, dada la movilidad que ofrece la herramienta desarrollada, no se conoce dónde descargar los ficheros.

Con esto ya se han mostrado cuáles son los archivos que usa el protocolo de comunicaciones y cuál es el cometido de cada uno de ellos. En la siguiente subsección, se mostrará cómo se utilizan en las funcionalidades que ofrece esta capa de la aplicación y que se han comentado anteriormente.

4.3.3. Implementación de las funcionalidades.

En este apartado se detallará cómo se ofrecen las funcionalidades más importantes que se comentaron en la introducción de esta sección, ya que la implementación de algunos servicios menores ya se explicaron con el suficiente detalle. Así, se mostrarán algunos diagramas de secuencia en los que aparecerán el paso de mensajes entre el equipo local y el *cluster*.

En primer lugar, se va a tratar el caso de uso más básico que es la ejecución de un trabajo en el *cluster* y que finalice antes de que el servidor TCP se inhabilite. Los pasos, que se muestran en la Figura 4.3, son los siguientes:

1. Una vez que la capa superior aporta todos los datos necesarios (incluida el *host* en el que se encuentra lanzado el servidor TCP), hay que crear los

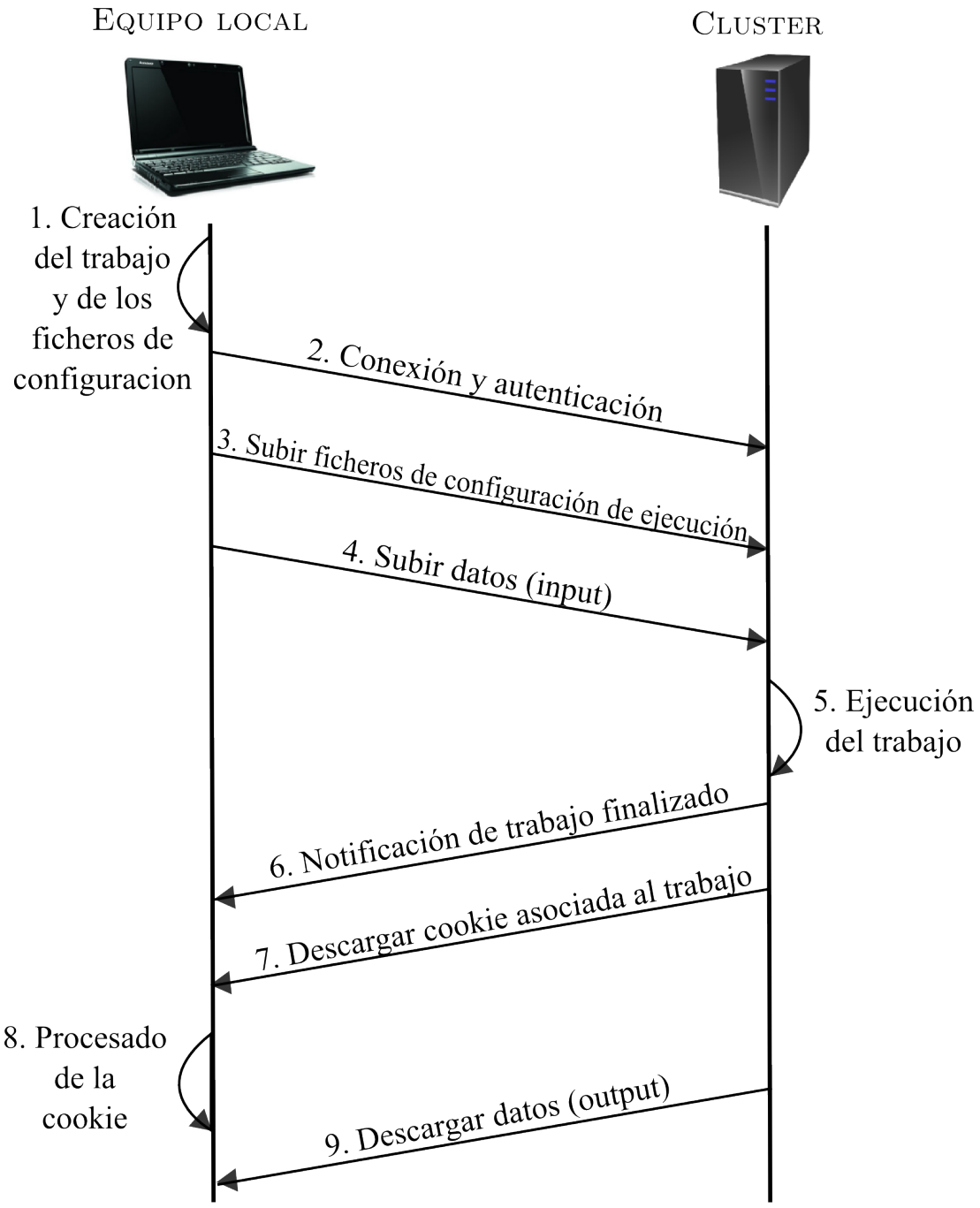


Figura 4.3: Intercambio de mensajes entre los agentes involucrados para el envío de un trabajo al *cluster*, y descarga de resultados.

ficheros de configuración `JobConfig` y la *cookie* asociada al trabajo, lo que se hace a través de métodos de `ClusterConnection`.

2. Posteriormente, hay que hacer la conexión del objeto de la clase `Session` que se vio en el Apartado 4.2 y la autenticación mediante `ClusterConnection`. Este paso puede omitirse si ya se han hecho anteriormente por motivos de eficiencia, como se ha comentado anteriormente, aunque se deja como opción a la capa superior.

3. A continuación, hay que enviar los ficheros de configuración que se han creado en el primer paso para que el envío del trabajo al gestor de colas se haga correctamente.

4. También hay que subir los ficheros de entrada que necesita el trabajo para poder ejecutarse.

5. Con todos los archivos subidos, en el equipo local `ClusterConnection` ejecuta, por SSH, `ReceiveJob` de forma que `JobConfig` es procesado y el trabajo enviado al gestor de colas. Esto se hace por medio de la siguiente línea de código en `ClusterConnection`:

```
channelExec.setCommand("java -cp $CLASSPATH:SSHSGE/drmaa.jar SSHSGE.
    ReceiveJob "+"$PWD/Posidonia/"+appName+"/"+jobID+"/"+jobCookie+ " "
    + "$PWD/Posidonia/"+appName+"/"+jobID+"/"+jobTemplate);
```

   Donde `appName`, `jobID` y `jobCookie` son los nombres de la aplicación, el identificador unívoco del trabajo y el nombre de la *cookie* respectivamente. Con ese código se le dice a la clase `ReceiveJob` del paquete `SSHSGE` dónde se encuentran los ficheros de configuración `JobConfig` y la *cookie* relacionada.

6. Cuando la tarea ha finalizado su ejecución, se notifica al servidor TCP (cuya localización se obtiene leyendo la *cookie* desde `ReportJob`) que el trabajo ya ha terminado y se actualizan los ficheros `runningJobs` y `history`.

   Para esto es necesario introducir unos ligeros cambios en el API desarrollado en el Capítulo 3 que afectan únicamente al método `runJob` de `SGEJob` y que se muestran en el Fichero 4.8. En esa modificación se usa `ReportJob` y `FollowJob` para los casos que ya se han comentado:

7. Al ser notificado el servidor TCP, éste interactúa con `ClusterConnection` para que se descargue la *cookie* asociada al trabajo finalizado. Se sabe qué trabajo ha finalizado gracias a que en el mensaje de aviso al servidor se incluye el identificador unívoco de la tarea.

8. Se descarga la *cookie*. Para garantizar que la información descargada es fiable, se usa un esquema muy similar al de la Figura 4.4, salvo que en este caso no se actualiza el archivo bajado.

Fichero 4.8: Método runJob modificado

```
...
/**
  * Sends a job to the queue.
  * @param wait Boolean that, if true, it blocks the execution in
  * this function until job is finished.
  * @param pathCookie String that shows where we have to update
  * the cookie, if this is necessary.
  * @param realName String which has the real name of the job, not its ID.
  */
  public void runJob(boolean wait,String pathCookie,String realName){
      ...
    String id = session.runJob(jt);
    // We have to update "runningJobs".
    FollowJob.addJobToFile(this.getJobName(), id, realName, app);
    System.out.println("directorio actual "+System.getProperty("user.dir"));
    System.out.println ("Job with id " + id + " has been submitted!");
    if (wait) {
      // With this we wait for job to finish.
      JobInfo info = session.wait(id, Session.TIMEOUT_WAIT_FOREVER);
      if (info.wasAborted()) {
        // "runningJobs" is updated.
        FollowJob.removeJobFromFile(this.getJobName(), id, realName, app);
        ReportJob rJ = new ReportJob();
        connected = rJ.checkUpdateCookie(pathCookie+"/"+jobName+"cookie",
            outputPath, errorPath, "aborted", app);
        if (connected) {
          sended = rJ.notifyClient("Aborted "+jobName);
          if (!sended) {
            rJ.updateCookieClientDown(pathCookie+"/"+jobName+"cookie");
            // pendingJobs file is updated.
            FollowJob.addJobToPending(realName, "aborted", app);
          }
        }
        FollowJob.addJobToHistory(this.getJobName(),
          realName, new Long(System.currentTimeMillis()).toString(),
          app, id, "Aborted");
      }
      else if (info.hasExited()){
        System.out.println("Job " + info.getJobId() + " finished
          regularly with exit status " + info.getExitStatus());
        // "runningJobs" is updated.
        FollowJob.removeJobFromFile(this.getJobName(),id,realName,app);
        ReportJob rJ = new ReportJob();
        connected = rJ.checkUpdateCookie(pathCookie+"/"+jobName+"cookie",
          outputPath, errorPath, "ended",app);
        if (connected) {
          sended = rJ.notifyClient("JobReady "+jobName);
          if (!sended) {
            rJ.updateCookieClientDown(pathCookie+"/"+jobName+"cookie");
            // pendingJobs file is updated.
            FollowJob.addJobToPending(realName, "ended", app);
          }
        }
        FollowJob.addJobToHistory(this.getJobName(), realName, new Long(
            System.currentTimeMillis()).toString(), app, id, "Ended");
      }
    }
    ...
}
```

9. Se procesa la *cookie*, de forma que `ClusterConnection` sabe qué archivos hay que descargar gracias al campo `OutputPath` de la *cookie*.

10. Todos los ficheros que se indican en `OutputPath` son bajados, y una vez hecho esto se muestra la notificación al usuario.

Hay que notar que este caso es un caso específico, pero refleja bien el funcionamiento general del protocolo de comunicaciones. Así, por ejemplo, en el caso de que se quisieran descargar los ficheros de entrada o de salida de un trabajo en el historial, se siguen los tres últimos pasos.

El otro caso que se va a detallar por medio de un diagrama de secuencia es el del acceso concurrente a un fichero. En este protocolo esto se utiliza tanto para la modificación de una *cookie* como para la descarga de *runningJobs* o `history`, que son los archivos más sensibles a este problema. En este supuesto, como se muestra en la Figura 4.4, los pasos a seguir son los siguientes:

1. En primer lugar, `ClusterConnection` pide el acceso al archivo, para lo que ejecuta la siguiente línea dentro de una conexión SSH como la que se mostró en el Apartado 4.2:

   ```
   channelExec.setCommand("java SSHSGE.KeepLock $PWD"+pathArchivo+" "+
       IPAddress+" "+portServerTCP);
   ```

   Así, hace uso de la clase `KeepLock` para conseguir bloquear el archivo. Debe indicar la ruta donde se encuentra el archivo que desea bloquear, y cómo localizar al servidor TCP que va a recibir la notificación de que el archivo ya está disponible.

2. La clase `KeepLock` hace uso de la clase nativa de Java `FileLock` para conseguir el bloqueo del fichero entre distintas instancias de `java`, es decir, para todas las ejecuciones de Java menos ésta. Si el archivo estuviera abierto por, por ejemplo, `SGEJob`, esperaría hasta que estuviera disponible.

3. Una vez que consigue el bloqueo, `KeepLock` envía una notificación al servidor TCP indicándole que el archivo ya está disponible. Hay que destacar que desde que `ClusterConnection` ha solicitado el acceso, se ha quedado esperando a que el servidor TCP reciba el aviso y despierte a `ClusterConnection`, que ahora ya puede descargar el fichero de forma fiable.

4. Se procede a la descarga del fichero solicitado, que en este protocolo puede ser `runningJobs`, `history` o una *cookie*.

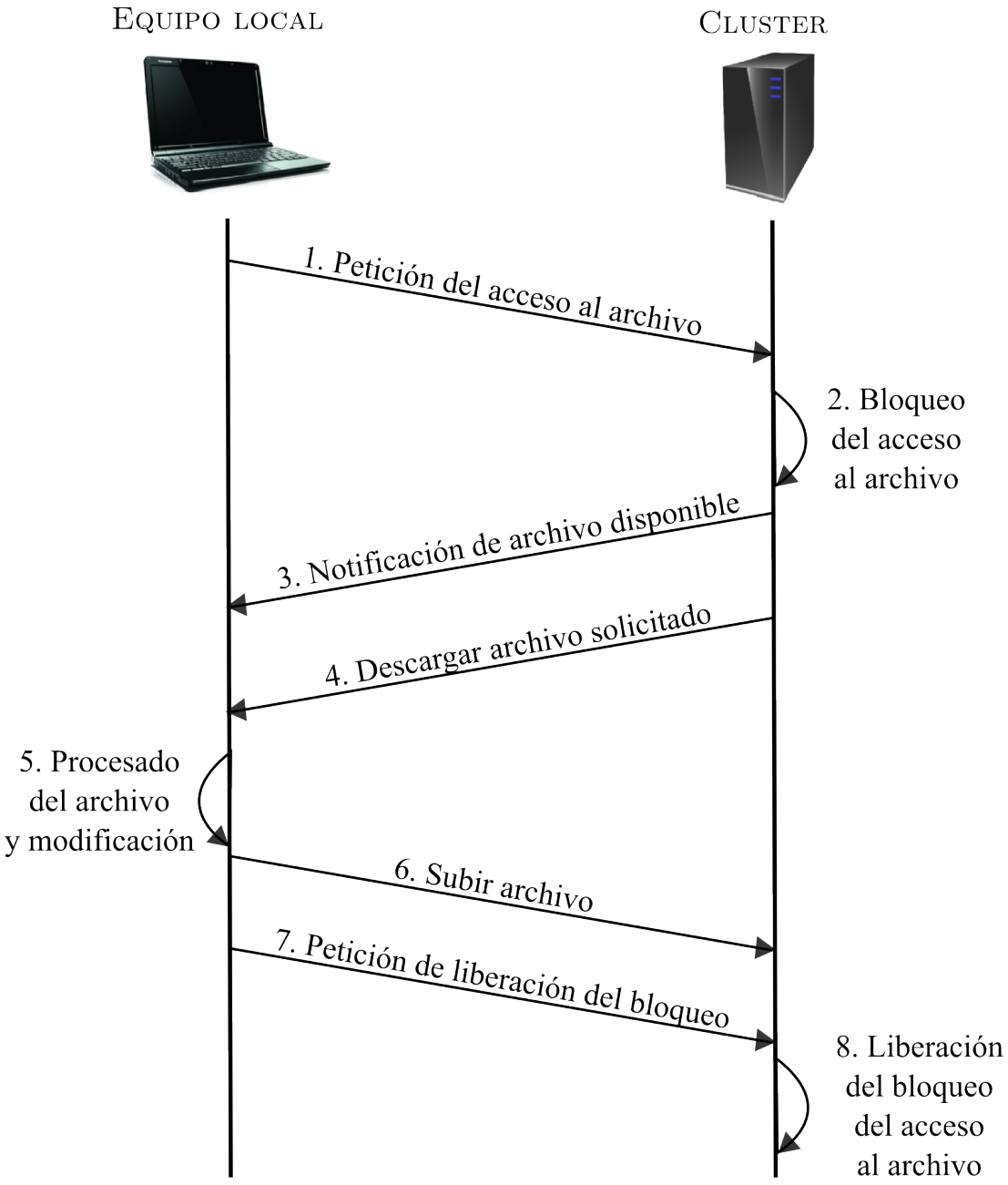


Figura 4.4: Intercambio de mensajes entre cliente y servidor para el acceso concurrente a un fichero.

5. Se procesa el archivo descargado y se hacen los cambios que se crean necesarios. Por ejemplo, en el caso de que se decida eliminar un trabajo en ejecución, se llamaría al comando de SGE `qdel`, que lo sacaría del gestor de colas, se eliminaría su carpeta del *cluster* y se modificaría *runningJobs* de esta forma que se está describiendo para eliminar su entrada en el fichero.
6. Se envía el archivo, ya modificado, al *cluster*.
7. A continuación, se ejecuta la clase `ReleaseLock` de forma muy similar a como se hizo la ejecución de `KeepLock`.
8. La ejecución de `ReleaseLock` hace que se libere el bloqueo del archivo modificado, de forma que se puede acceder desde cualquier otro sitio.

Hay que destacar que queda a decisión de la capa superior implementar una seguridad adicional en este tipo de accesos, tal y como se hace en la interfaz gráfica de escritorio a la hora de eliminar un trabajo en ejecución que ya ha finalizado. Esto último se detallará en el Capítulo 5.

4.4. Funcionalidades del protocolo en Android.

En esta sección se van a presentar cuáles son las características del protocolo de comunicaciones en la aplicación Android desarrollada, y cómo se han implementado.

Hay que destacar que, dado que la extensibilidad y reusabilidad es una de las principales características tanto de *Posidonia* como de este protocolo, sería de esperar que la capa de comunicaciones fuera exactamente igual tanto si la capa superior se encuentra desarrollada en Android como si es una interfaz gráfica de escritorio. Sin embargo, esto no puede ocurrir puesto que, tal y como se explicó en el Capítulo 2, el entorno en el que tiene que trabajar el protocolo de comunicaciones es distinto en estos dos casos. El escenario que se considera es el que aparece en la Figura 2.21.

En este entorno la programación es más compleja que en el de la aplicación de escritorio puesto que no hay conexión directa con el *cluster* debido a la imposibilidad (dado el carácter móvil del dispositivo Android) actual de tener conexión a una VPN. De esta forma, hay que implementar un "doble salto" de manera que el equipo móvil se comunica con una pasarela, implementada por un servidor dedicado, y ésta es la que interactúa con el *cluster*.

La manera de implementar este doble salto es la siguiente:

1. El dispositivo Android se comunica, de forma segura gracias a SSH, con la pasarela ejecutando una clase Java de un paquete específico del protocolo de comunicaciones llamado `ConnTablet`. Además, enviará algunos archivos necesarios para la clase Java que, en cada caso, se ejecutará.

2. El servidor dedicado interactúa con el *cluster* llevando a cabo la función definida en la clase. El esquema de ficheros, como se verá más adelante, del paquete `ConnTablet` está muy orientado a la funcionalidad; es decir, que cada clase ejecuta una funcionalidad, como por ejemplo la descarga de ficheros de salida de un determinado trabajo, o el envío de una tarea al *cluster*.

3. Una vez que el servidor ha intercambiado todos los mensajes y archivos necesarios con el *cluster* el equipo móvil, dependiendo de la funcionalidad, procede a descargar los ficheros que ha solicitado y a borrarlos de la pasarela.

Como se observa, el algoritmo es bastante más complejo que el que se describió en la sección anterior, pero sin embargo se consigue mantener prácticamente toda la funcionalidad de una forma transparente al usuario a pesar de que, por ejemplo, los resultados deben descargarse primero a la pasarela y, posteriormente, al equipo móvil.

Sin embargo, debido a la movilidad de estos dispositivos el servidor TCP no es accesible desde el servidor dedicado que actúa como pasarela; es decir, el servidor se puede levantar en el equipo móvil pero debido a que ahora la conexión no se hace por medio de una VPN, normalmente la conexión es rechazada por alguno de los intermediarios en la red (normalmente el operador rechaza todas las comunicaciones de servicios no reconocidos y, obviamente, *Posidonia* no se encuentra registrado en la lista de los reconocidos). Así, hay una parte importante de la funcionalidad que no se puede implementar:

- La notificación al usuario de que la tarea ya ha finalizado, ya que se hacía enviando una notificación al servidor TCP. Debido a esto, si el usuario quiere ser avisado de que un trabajo que ha enviado ha acabado su ejecución, deberá usar la vía del correo electrónico que, por otra parte, *Posidonia* facilita.

- El acceso concurrente a los ficheros, que también hacía uso del servidor TCP levantado. De esta forma, la información de los ficheros descargados puede no ser fiable, o la más actualizada posible, aunque no es algo crítico en las funcionalidades desarrolladas en este protocolo.

El resto de servicios que ofrece la capa se mantiene, incluida la notificación de que un trabajo ya ha llegado al gestor de colas. Esto es posible gracias a la librería JSch que es utilizada porque, tal y como se observó en el Fichero 4.1, los resultados que aparecen en la consola de comandos se pueden capturar en el código. Así, se puede esperar a que SGEJob imprima por pantalla `Your job with id has been submitted`, y así el equipo móvil sabrá que el trabajo ya es responsabilidad de, en el caso contemplado, SGE.

De igual manera se puede proceder con el resto de funcionalidades, como la descarga de los archivos, estableciéndose un canal de comunicaciones entre el dispositivo Android y la pasarela. Así, los archivos sólo serán descargados al equipo móvil cuando estén disponibles en la pasarela, y para ello no es necesario el uso de un servidor TCP gracias al canal creado con la conexión SSH.

Con todo lo explicado hasta ahora se sientan las bases de la capa de comunicaciones en Android: en definitiva, nada cambia salvo la introducción del "doble salto" que introduce cambios que ya han sido comentados. De esta forma, los ficheros de configuración del protocolo (`runningJobs`, `pendingJobs`, `history` para cada usuario y `JobConfig` y la *cookie*, para cada trabajo) son los mismos y ya quedaron suficientemente explicados en la sección anterior; en las siguientes subsecciones se expondrán:

- El esquema de ficheros Java que ofrece la funcionalidad explicada hasta ahora.
- La implementación de las funcionalidades más importantes, aunque por norma general se sigue el esquema de comunicación con el *cluster* por medio de la pasarela que ya se ha detallado.

4.4.1. Esquema de clases Java desarrolladas.

Aquí se explicarán cuáles son las clases Java que soportan la funcionalidad ofrecida por esta capa de comunicaciones Android.

Debido al escenario tan particular que se ha presentado en la Figura 2.21 ahora hay tres paquetes, situados en cada uno de los tres agentes involucrados (dispositivo Android, pasarela y *cluster*) tal y como se muestra, simplificadamente, en la Figura 4.5. De forma más detallada, a continuación se explican las funcionalidades que desempeñan cada una de las clases:

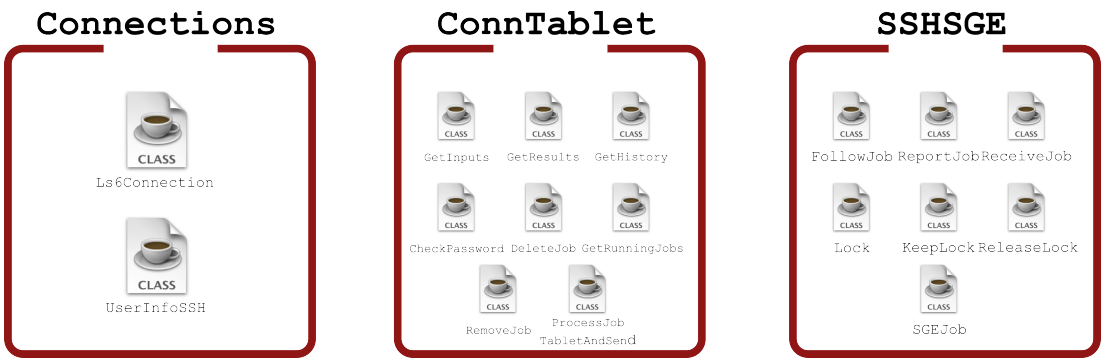


Figura 4.5: Esquema general de la estructura de clases del protocolo de comunicaciones para Android.

- Paquete `Connections`: situado en el equipo móvil y que se encarga de establecer la comunicación entre el dispositivo móvil y el servidor dedicado que desempeña la función de pasarela. Las clases Java que se incluyen son:

  1. `Ls6Connection`: clase similar a `ClusterConnection` en la que se establecen las conexiones SSH para ejecutar clases Java y comandos en la pasarela, y las sesiones SCP para enviar las entradas de un trabajo (al mandarlo al gestor de colas) o descargar ficheros como los resultados de ejecución o las entradas de un trabajo, o como los archivos de configuración `history` o `runningJobs`.

     Da opción, tal y como hacía `ClusterConnection`, a que estas comunicaciones sean eficientes, es decir, usando una sola sesión para todas las conexiones SSH, o una sola instancia de sesión SCP para la transferencia de ficheros.

  2. `UserInfoSSH`: necesaria para el objeto de la clase `Session` de JSch.

- Paquete `ConnTablet`: alojado en la pasarela, su cometido es hacer de intermediario entre las órdenes enviadas por el dispositivo Android mediante `Ls6Connection` y el *cluster*.

  Cada clase se encarga de una de las funciones ofrecidas por el protocolo de comunicaciones y su forma de uso ya se ha comentado: cada una de ellas es un método `main` que se ejecuta por consola de comandos, y de forma remota gracias a la conexión SSH que ofrece la librería JSch.

  Así, el progreso de estas operaciones se puede seguir por medio del canal establecido, siendo posible conocer cuándo un archivo del *cluster* ya está disponible en la pasarela o el momento en el que el trabajo ha llegado al gestor de colas del *cluster*.

  Las clases contenidas en este paquete son las siguientes:

1. `CheckPassword`: como existe un doble salto, hay que comprobar tanto la autenticación para la pasarela como para el *cluster*. Como el equipo móvil no tiene acceso directo al *cluster*, la autenticación para éste debe hacerse a través de la pasarela con la ejecución de esta clase.
2. `DeleteJob`: esta clase se encarga, con su ejecución, de borrar un trabajo en ejecución cuyo identificador es pasado por parámetro en la consola de comandos, es decir, especificado por `Ls6Connection`. Esta eliminación tiene las mismas repercusiones que en el caso de la aplicación de escritorio: se saca la tarea del gestor de colas si aún no ha acabado, se actualiza `runningJobs` y se elimina todo rastro de él en el *cluster*.
3. `GetHistory`: su cometido es descargar el archivo `history`, almacenado en el *cluster*. La clase `Ls6Connection` lo descargará cuando esté disponible, algo que se puede conocer gracias al canal establecido en la conexión SSH.
4. `GetInputs`: su función consiste en descargar todos los ficheros de entrada de un determinado trabajo pasado por parámetro.
5. `GetResults`: el servicio que desempeña es el mismo que el de la clase anterior, pero para los resultados de una cierta tarea.
6. `GetRunningJobs`: esta clase implementa lo mismo que `GetHistory` pero con el fichero de estado `runningJobs`.
7. `ProcessJobTabletAndSend`: se utiliza cuando se desea enviar un trabajo al *cluster*; para ello deben estar disponibles los ficheros de configuración del trabajo, `JobConfig` y su *cookie* asociada, y sus ficheros de entrada.
8. `RemoveJob` se encarga de eliminar un trabajo del historial, borrando del fichero `history` y eliminando toda su presencia en el *cluster*.

Cabe destacar que todos los ficheros que descarga este paquete son eliminados por `Ls6Connection` cuando son descargados al dispositivo Android, por lo que este servidor dedicado actúa como contenedor temporal de los archivos.

- Paquete `SSHSGE`, que no se va a explicar porque no cambia con respecto al de la capa de comunicaciones explicado en la sección anterior. Esto refleja la extensibilidad del código desarrollado que, a pesar de los cambios que introduce el nuevo escenario considerado, no introduce ninguna modificación en el *cluster* ya que éste sigue siendo el mismo: lo único que cambia es cómo se realiza el acceso.

Con todo esto ha quedado explicada la estructura del código que se ha desarrollado. A continuación, y para finalizar el capítulo, se explicará cómo se implementan

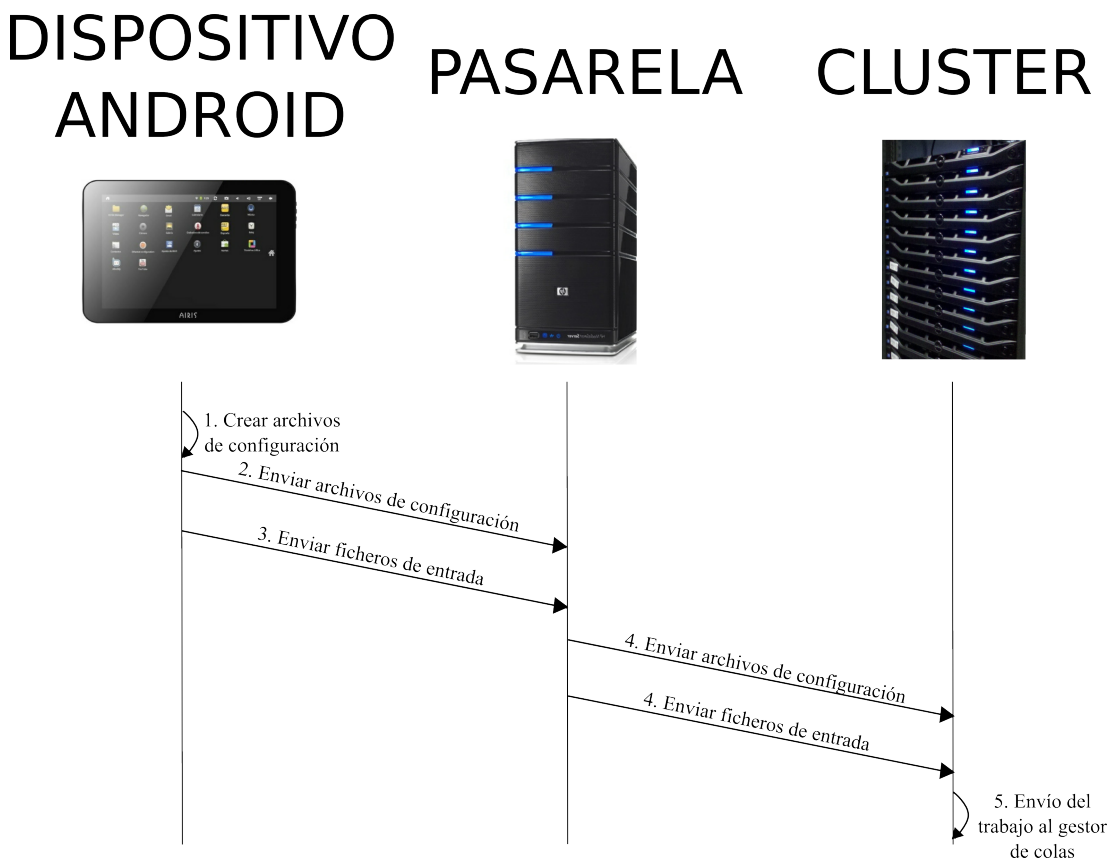


Figura 4.6: Intercambio de mensajes entre los agentes involucrados para el envío de un trabajo al *cluster*.

las funcionalidades más importantes por medio de estas clases Java.

4.4.2. Implementación de las funcionalidades.

En esta subsección se detallará cómo se llevan a cabo paso a paso las principales funcionalidades que ofrece la capa de comunicaciones en este escenario, ya que el esquema básico de funcionamiento se explicó al principio de esta sección.

En primer lugar, se va a explicar cómo se lleva a cabo el servicio más básico del protocolo, que es el envío de un trabajo al gestor de colas del *cluster*. El diagrama de secuencia de la Figura 4.6 muestra qué mensajes se intercambian entre los agentes involucrados: dispositivo Android, la pasarela y el *cluster*; paso a paso ocurre lo siguiente:

1. En primer lugar, la capa superior comunica al paquete `Connections` dónde están los ficheros de entrada del trabajo y cuáles son las opciones de configuración de la tarea. Así, se crean los ficheros de configuración: `JobConfig` y la *cookie* relacionada con la tarea. En este escenario, la conexión y autenticación se ha hecho anteriormente (ya que lo recomendable es que se haga una sola vez, tal y como se explica en el Capítulo 5).

2. A continuación, se envían los dos archivos de configuración del trabajo.

3. También se envían los ficheros de entrada necesarios para la ejecución de la tarea.

4. Una vez que están disponibles todos los ficheros en la pasarela, se ejecuta por medio de una conexión SSH implementada en `Ls6Connection` la clase `ProcessJobTabletAndSend`, que se encarga de procesar y mandar estos archivos al *cluster*. Así se envían los ficheros de configuración del trabajo y las entradas necesarias.

5. Con todo esto, ya se encuentran disponibles todos los archivos indispensables para la ejecución del trabajo mediante el gestor de colas. Para ello, `ProcessJobTabletAndSend` hace uso de la clase `ReceiveJob` tal y como se detalló en la sección anterior.

Se puede observar que el esquema de funcionamiento es el mismo que en la sección anterior, salvo que lo que antes se hacía directamente desde el equipo local ahora hay que realizarlo en la pasarela y, por ello, enviar los ficheros a ese servidor para que puedan ser recibidos en el *cluster*. Por otro lado, no cambia nada en `SSHSGE`, por lo que la notificación se seguirá enviando al servidor TCP; como no hay ninguno disponible, lo único que ocurrirá es que el trabajo se guardará en `history` y en `pendingJobs` para que sea visible en la aplicación de escritorio.

Así, el simple hecho de mantener la funcionalidad en las dos capas de comunicaciones y de que el paquete en el *cluster* sea el mismo para ambas hace que la movilidad sea plena, ya que los archivos de estado que se modifican, `runningJobs`, `pendingJobs` y `history` se alojan en el *cluster* y tienen la misma información en ambas capas: lo único que varía es la forma de presentar la información al usuario, y eso depende de cada interfaz gráfica. Esta forma de tratar los ficheros hace que este proyecto esté muy relacionado con las actuales tendencias de *cloud computing*, tal y como se observó en el Capítulo 1.

Para acabar, se va a detallar cómo se realiza la descarga de un archivo cualquiera por medio de este protocolo de comunicaciones. Este fichero puede ser `runningJobs`, `history` o las entradas y salidas de un cierto trabajo solicitado por la capa superior al paquete `Connections`. Así, se muestran los mensajes intercambiados en la Figura 4.7 siguiendo los pasos que se detallan a continuación:

Figura 4.7: Intercambio de mensajes entre los agentes involucrados para la descarga de un fichero desde el *cluster* al dispositivo Android.

1. Desde `Ls6Connection` se ejecuta, por medio de una conexión SSH, uno de los ficheros Java de `ConnTablet` como, por ejemplo, `GetHistory`. De nuevo, se considera que la conexión SSH y la autenticación (tanto para la pasarela como para el *cluster*) se ha hecho con anterioridad.

2. De esta forma, esta clase se descarga el fichero solicitado del *cluster* a la pasarela por medio del establecimiento de una sesión SCP entre este servidor y el *cluster*.

3. Una vez que el fichero se encuentra en el servidor que funciona como intermediario, `Ls6Connection` se lo descarga mediante otra sesión SCP que, normalmente, ya estará activa y no habrá que establecer por motivos de eficiencia. Se sabe que el archivo está disponible por medio del canal establecido en la conexión, como ya se ha comentado anteriormente.

4. Finalmente, cuando el fichero ya se encuentra en el equipo móvil, se elimina de la pasarela porque este servidor no utiliza el archivo para nada.

Con todo esto quedan explicadas la implementación de las principales funcionalidades que ofrece esta capa de comunicaciones en el entorno Android. Se ha visto que, pese al uso de una pasarela como intermediario para la comunicación entre el equipo local y el *cluster*, la dinámica de trabajo es la misma porque se busca que el protocolo desarrollado sea lo más extensible posible. Además, aunque este escenario hace que la programación sea más compleja y se pierdan algunas funcionalidades como las que implementaba el servidor TCP, hace que el usuario tenga mayor movilidad al no necesitar el acceso VPN para poder utilizar el *cluster*.

Capítulo 5

Interfaz gráfica.

Una vez que se ha explicado el núcleo de la aplicación, correspondiente a la ejecución de trabajos por medio de DRMAA y a la capa de comunicaciones, en este capítulo se va a detallar, como aparece en la Figura 5.1, cuáles son los principales objetivos de la interfaz gráfica y cómo se implementan algunas de sus funcionalidades.

De esta forma, tal y como se detalló en el Capítulo 2, hay tres interfaces gráficas desarrolladas, cuya clasificación ya quedó establecida. A efectos de implementación, la principal diferencia entre los trabajos de FortranMod y los de Matlab son los archivos de entrada y de salida involucrados, pero la interfaz gráfica es básicamente igual: la principal diferencia, tanto por apariencia como por entorno de trabajo, está entre las interfaces de escritorio y la desarrollada en Android. Por este motivo, para aclarar la explicación, en primer lugar se van a exponer las características que comparten todas las interfaces:

- Inteligencia limitada: para garantizar la extensibilidad de la aplicación, la funcionalidad que aporta la interfaz gráfica se encuentra muy restringida: la práctica totalidad de las características de *Posidonia* son aportadas por la capa de comunicaciones tal y como se vio en el Capítulo 4.
- *User-friendly*: la interfaz con el usuario debe ser intuitiva y cómoda. Así, se prefieren no incluir todas las posibles parámetros de configuración del trabajo sino que en una ventana aparte, para usuarios avanzados, se pueden modificar los principales opciones de ejecución tal y como se observa en la Figura 2.6.

  De esta forma, la pantalla principal de la interfaz es lo más simple posible en todos los formatos, ya sea en escritorio, donde se muestran únicamente dos

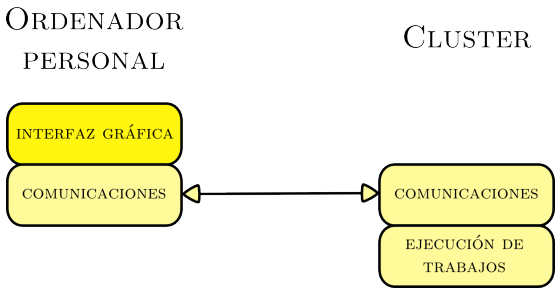


Figura 5.1: Estructura simplificada del código de *Posidonia*.

cuadros de texto para introducir los datos de autenticación al *cluster* (como se observa en la Figura 2.5, o en Android, en la que aparecen cuatro campos porque hay que efectuar el doble salto del que se habló en el Capítulo 4, como aparece en la Figura 2.23.

- Presentación clara de los datos: la información que se obtiene con la capa de comunicaciones, como los trabajos en ejecución o las tareas del historial, debe ser mostrada con todos los campos relevantes para el usuario tales como la fecha de envío (útil para discriminar trabajos con el mismo nombre) o el estado actual de ejecución. Para esto se decide usar una tabla, que es la forma más diáfana de exponer la información, tanto para los trabajos en ejecución, como se observa en la Figura 2.18 y en la Figura 2.26 para Android; como para el historial, tal y como se presenta en la Figura 2.19 y en la Figura 2.27.

- Notificación de lo más relevante: para asegurar la comodidad del usuario, se le muestran avisos únicamente cuando se cree necesario, es decir:

  - Cuando el trabajo ha llegado al gestor de colas, porque de esta forma se indica al usuario que el envío de la tarea se ha hecho de forma satisfactoria, tal y como aparece en la Figura 2.7 o en la Figura 2.26.
  - Cuando la tarea ha acabado su ejecución, en la aplicación de escritorio, para notar que los resultados ya están disponibles, como se presenta en la Figura 2.9.
  - Cuando los ficheros de entrada o de salida han acabado de descargarse, de forma que el usuario sabe cuándo puede consultarlos en la ruta que se haya indicado. Esto se puede observar en la Figura 2.14, y en la Figura 2.28.

  Cabe destacar que las notificaciones son lo menos agresivas posible: incluso, en la aplicación Android los avisos están integrados en la barra de tareas, algo que no es posible en el programa de escritorio para garantizar su comportamiento multiplataforma.

- Eficiencia en el uso de conexiones seguras: desde la interfaz gráfica se pide a la capa de comunicaciones el menor número de conexiones posibles para que no haya sobrecarga en la red y, sobre todo, para que el *cluster* pueda atender peticiones de muchos usuarios a la vez. Más adelante se detallará cómo se hace esta implementación.

Sin embargo, las diferencias entre las interfaces gráficas de escritorio y la de Android hace que las siguientes funcionalidades se presenten únicamente en las primeras:

- Adaptación completa al entorno de trabajo: gracias a la programación por medio de Java, la aplicación de escritorio desarrollada se mimetiza con el tema del escritorio sea cual sea la plataforma en la que se ejecute, tal y como se muestra en la Figura 2.15.
- Actualización del estado: para todos los trabajos que se encuentren aún en ejecución, esta capa debe actualizar su estado de desconectado a conectado e indicar, para cada tarea, si los resultados deben descargarse automáticamente o no.

De esta forma quedan detalladas cuáles son las principales funcionalidades que ofrece la interfaz gráfica: en definitiva, se busca que tenga la menor inteligencia posible para que sea la capa de comunicaciones la que ofrezca la mayor parte de las características. Así, su principal objetivo es que el usuario sea capaz de interactúar con el *cluster* de la forma más sencilla, sin tener que saber ni siquiera qué es un *cluster*, ya que únicamente quiere usarlo como servicio.

Para ilustrar el cambio que supone desarrollar la interfaz gráfica en Java o en Android (aunque se base también en Java) se muestra cómo se implementa el aviso al usuario de que los ficheros de salida ya han sido descargados. De este modo, en el Fichero 5.1 aparece parte del código necesario para ello y que se encuentra en la clase `Notificacion.java` correspondiente al paquete `Connections`. Se pueden observar algunas de las características descritas anteriormente: se busca que sea multiplataforma, para lo que se adapta la notificación de una forma u otra en función del sistema operativo, y para evitar posibles incompatibilidades se crea una ventana aparte para el aviso en vez de integrar éste en la barra de tareas.

Por otra parte, para la interfaz de Android la implementación es mucho más sencilla ya que se implementa de forma nativa en el sistema. De esta forma, no se busca la característica multiplataforma que es obligada en las interfaces de escritorio sino que el objetivo es la integración completa en el dispositivo, por lo

Fichero 5.1: showNotification

```
/**
 * Show the notification with the message passed as
 * as parameter.
 * @param message String which is going to be showed on screen.
 * @param time Time (in milliseconds) notification is going to
 * be shown on screen.
 * @param error boolean which, if true, shows that notification is an
 * error notification.
 */
public void showNotification(String message,String results_dir,
                                                    int time,boolean error) {
  //Create and set up the window.
  frame = new JFrame("Posidonia");
  // External frame is deleted.
  frame.setUndecorated(true);
  // Not resizable.
  frame.setResizable(false);
  // Black background.
  frame.getContentPane().setBackground(new Color (30,30,30));
  ...
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  ...
  // Get size of the screen.
  Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
  int X = 0;
  int Y = 0;
  // Southeast corner.
  if (this.isWindows()) {
    X = screen.width - frame.getWidth();
    Y = screen.height- frame.getHeight();
  }
  // Northeast corner.
  else {
    X = screen.width - frame.getWidth()-3;
    Y = 40;
  }
  frame.setLocation(X,Y);
  //Display the window.
  frame.setVisible(true);
}
```

que se usa la clase `Notification` que es provista por las librerías de Android, tal y como se puede observar en el Fichero 5.2. Así, por ejemplo, el icono cumple con las especificaciones impuestas por el sistema para que se mimetice completamente con el resto de notificaciones mostradas por el dispositivo, y la llamada es más sencilla que la creación de una nueva ventana como se hace en el Fichero 5.1.

Finalmente, de manera análoga a como se hizo la descripción de las características de esta capa, se van a exponer en primer lugar cómo se implementan algunas de las funcionalidades comunes expuestas anteriormente:

- Eficiencia en las conexiones: la capa de comunicaciones da como opción que las conexiones sean eficientes; es decir, que el establecimiento de la

Fichero 5.2: Notificación en Android

```
...
// Se obtiene el administrador de notificaciones.
String ns = Context.NOTIFICATION_SERVICE;
NotificationManager mNotificationManager =
        (NotificationManager) getSystemService(ns);
// Se fija el icono, cu ndo se lanza y el texto que aparece
// al saltar la notificaci n.
int icon = R.drawable.notification_icon;
CharSequence tickerText = "Posidonia";
long when = System.currentTimeMillis();
Notification notification = new Notification(icon, tickerText, when);
// Se obtiene el contexto para especificar par metrso de la notificaci n.
Context context = getApplicationContext();
CharSequence contentTitle = "Posidonia: files downloaded.";
CharSequence contentText = "Output files of "+
  jobIDs.elementAt(rowSelected.elementAt(0)).substring
  (0,jobIDs.elementAt(rowSelected.elementAt(0)).length()-13)+" has
  been downloaded. =)";
...
// Se integra con el sistema con el sonido por defecto.
notification.defaults |= Notification.DEFAULT_SOUND;
notification.flags |= Notification.FLAG_AUTO_CANCEL;
// Tipo de notificaci n.
final int HELLO_ID = 1;
// Se indica al administrador de notificaciones qu  es lo que ejecuta.
mNotificationManager.notify(HELLO_ID, notification);
```

conexión se haga una sola vez dentro de una sesión, que es algo que permite el protocolo SSH. Así, en el Fichero 4.2 se invocaría a `session.connect()` y a `sftp.connect()` únicamente la primera vez que se ejecuten, por medio de un método `connect` de `ClusterConnection`, o de `Ls6Connection`.

Esto es muy positivo para que no haya sobrecarga en la red, pero es obligatorio en el caso de Android, ya que en el entorno en el que se trabaja existe un doble salto en el que es necesario que se sea eficiente en el uso de las conexiones SSH: no se puede iniciar una nueva conexión cada vez que se quiera descargar un recurso remoto porque el servidor dedicado como pasarela detecta un abuso de conexiones SSH y por seguridad bloquea la dirección IP de la que ha procedido el exceso.

En esta línea también hay que realizar la autenticación del usuario, ya que ésta se hace a través de una conexión SSH de prueba y se tiene el mismo problema. Para evitarlo, se comprueba el *login* sólo cuando es necesario, es decir:

- Cuando el usuario aún no se ha autenticado, o los datos introducidos son incorrectos.
- Cuando la información de *login* ha variado con respecto a la última conexión.
- En el caso de la aplicación Android, si los datos de autenticación se

han obtenido a partir de un fichero (es decir, se han guardado de una sesión a otra) como ya están verificados no se vuelve a realizar la comprobación.

También hay que destacar que esta capa es la responsable de verificar si hay conexión a Internet o no por medio del método proporcionado por `ClusterConnection`, de forma que cada vez que se necesite hacer uso de alguna conexión SSH se debe comprobar, en primer lugar, si hay acceso a la red. En caso contrario, se obtendría una notificación como la de la Figura 2.30.

Además, el lanzamiento del servidor TCP en el caso de las interfaces de escritorio también corresponde a éstas y se debe hacer de forma inteligente, es decir, una sola vez y lo antes posible (es decir, cuando se han verificado los datos de autenticación) para indicar al *cluster* que el usuario se encuentra conectado.

- Actualización manual de la información: se ha decidido que sea el usuario el que refresque la información de la tabla de los trabajos en ejecución y del historial para evitar mensajes inútiles en la red. Sin embargo, esto conlleva ciertos problemas asociados a no disponer de la información completamente actualizada: por ejemplo, se puede intentar eliminar un trabajo en ejecución cuando éste ya ha concluido y ha salido del gestor de colas, por lo que no puede ser eliminado.

  Ese error es capturado y se le notifica al usuario, como se puede observar en la Figura 5.2, de forma que si se quiere eliminar el trabajo y borrar todo rastro de él en remoto, se deberá acceder al historial y eliminarlo allí.

A continuación, se va a exponer cómo se realizan algunas de las funcionalidades propias de las interfaces de escritorio y que se mostraron en el Capítulo 2:

- Integración con GiD: en la Figura 2.4 se puede observar que la inclusión de *Posidonia* en la interfaz gráfica GiD es natural, ya que en ese menú se puede elegir entre simular el problema de forma local o por medio de la aplicación desarrollada.

  Para conseguir esta integración se cambia uno de los scripts incluido en el programa de simulación FortranMod. Ya se explicó en el Capítulo 2 que GiD sólo es una interfaz de pre y postprocesado en problemas de simulación; el núcleo es un código desarrollado en el Grupo de Radiofrecuencia de la Universidad y llamado FortranMod. Éste ejecuta una serie de scripts que lo incluyen en GiD, y uno de los scripts que usa es `Run.tcl`, en el que se determinan las dos opciones que se le dan al usuario: resolverlo de forma

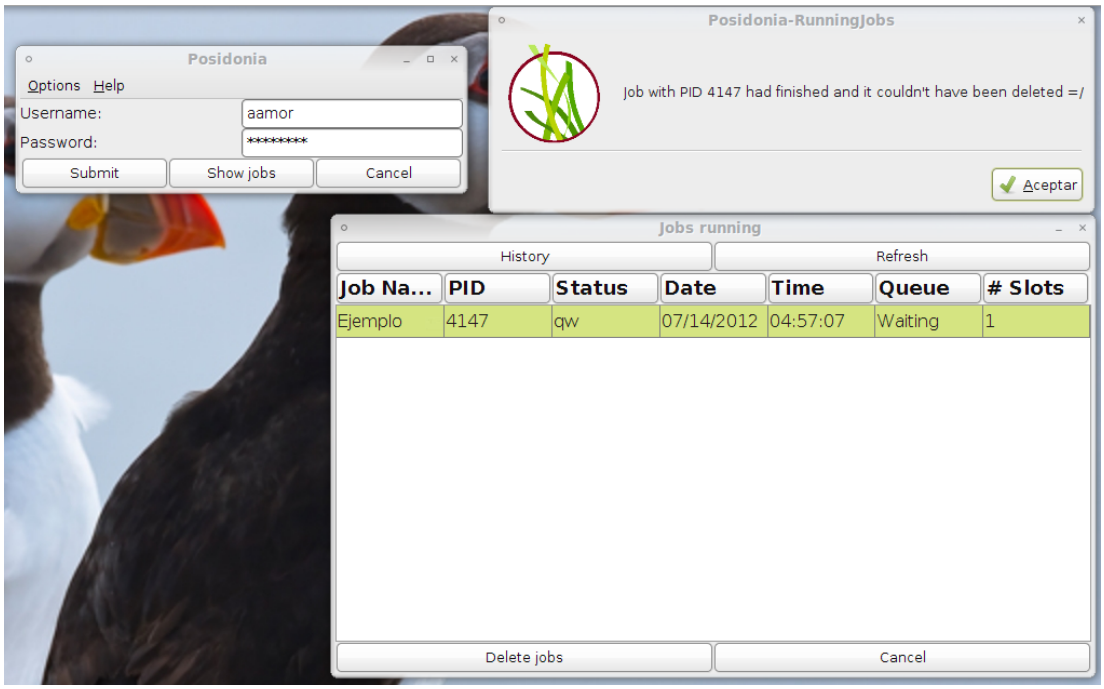


Figura 5.2: Error al intentar eliminar un trabajo en ejecución que ya ha finalizado.

local y ejecutar *Posidonia*, con lo que nos saldría la pantalla principal de la aplicación.

- Única instancia de *Posidonia*: para evitar problemas relacionados con tener más de un usuario activo en la misma máquina como, por ejemplo, la compartición de un servidor TCP para todas las instancias activas, se ha forzado que haya sólo una aplicación *Posidonia* en ejecución. Para ello, se ha hecho uso de un fichero que funciona como cerrojo: así, la aplicación al arrancar observa si este fichero existe y, en caso afirmativo, informa al usuario de que no puede ejecutarse porque hay una instancia de *Posidonia* ya activa.

Y para concluir, se va a detallar la implementación de alguna de las funcionalidades que sólo aparecen en la interfaz de Android tal y como se expuso en el Capítulo 2:

- Almacenamiento de los datos de autenticación: dado que se tiene un escenario particular en el caso de Android, hay que rellenar cuatro campos de texto para conseguir acceder a los servicios integrados en *Posidonia* y, además, es más pesado al hacerlo a través de un dispositivo móvil. De esta

forma, se da la opción al usuario, tal y como se ve en la Figura 2.23, de guardar y borrar la información asociada a la autenticación si es correcta.

Para ello se usa la carga de ficheros propia de Android y en modo privado, de forma que esa información sólo es accesible (y visible) para *Posidonia*, salvo si se *rootea* el dispositivo, algo que no es habitual. De esta forma, se ha decidido no introducir una encriptación adicional debido a la que Android aporta de forma nativa.

En resumen, la interfaz gráfica tiene una cierta inteligencia asociada que es muy restringida en comparación con la que tiene el protocolo de comunicaciones pero que es indispensable para ofrecer al usuario una buena experiencia de uso.

Capítulo 6

Conclusiones y líneas futuras.

En este proyecto se ha visto cómo se ha desarrollado una herramienta para acercar el *cluster* al usuario esté donde esté, de forma que éste puede ejecutar códigos computacionalmente muy costosos de forma simple desde un equipo remoto.

Los principales objetivos que se han alcanzado son los siguientes:

1. Sencillez y rapidez a la hora de enviar los trabajos: la interfaz gráfica es muy intuitiva para que la ejecución simple de tareas en el *cluster* sea lo más rápida posible.

2. Recepción de los resultados de forma automática y aviso al usuario: gracias al uso de la librería DRMAA, como se explica en el Capítulo 3, es posible capturar el evento de salida en el código y, de esta forma, enviar una notificación al usuario que se encuentra en remoto y mandarle automáticamente los resultados generados.

3. Seguridad: todas las conexiones se encuentran encriptadas por el protocolo SSH, lo que hace que el acceso remoto al *cluster* no suponga ninguna amenaza ni para los datos del usuario ni para el propio *cluster*.

4. Integración con los programas desarrollados: la integración de *Posidonia* con FortranMod, tal y como se detalla en los Capítulos 2 y 4 es total, de forma que es transparente para el usuario el hecho de simular su problema en su equipo o en el *cluster*.

5. Extensibilidad y generalidad: como demuestran las tres versiones de *Posidonia* desarrolladas, el código se ha programado de tal forma que sea muy sencilla la implementación de las mismas funcionalidades en distintos entornos y aplicaciones.

6. Control de los trabajos: todo lo que un usuario puede realizar con una tarea enviada al gestor de colas puede llevarse a cabo por medio de la aplicación *Posidonia* de una forma cómoda y ágil.

7. Movilidad: gracias a la programación de una aplicación en Android, todas las funcionalidades que un usuario busca cuando accede a un *cluster* para ejecutar un proyecto están accesibles por medio de un dispositivo móvil. Además, se puede trabajar de igual forma en cualquier equipo que cuente con *Posidonia* debido a que toda la información utilizada por *Posidonia* se encuentra en remoto.

8. Eficiencia: para evitar sobrecargas en la red, tanto del usuario como del *cluster*, los mensajes que se intercambian son los imprescindibles para mantener la funcionalidad que introduce *Posidonia*.

9. Historial de los proyectos ejecutados: es una función tremendamente útil para el usuario, ya que le permite tener disponibles distintas versiones de un mismo trabajo de forma ordenada y fácilmente accesible. Además, se pueden descargar tanto ficheros de entrada como de salida en cualquier equipo, sea fijo o móvil, que tenga instalado *Posidonia*.

En definitiva, se ha desarrollado una herramienta muy potente, que no acaba aquí su recorrido puesto que es fácilmente reutilizable y que une dos conceptos que aún no se han juntado en el sector: la ejecución de trabajos en un *cluster* de forma remota y segura.

A continuación se detallan algunas de las futuras líneas de investigación que se abren con el desarrollo de esta aplicación:

- Implementación de *Posidonia* en distintos *clusters* y entornos: dados los medios limitados con los que se ha contado a la hora de desarrollar este proyecto (acceso a un solo *cluster*), es interesante investigar la compatibilidad con otros *clusters* y gestores de colas, introduciendo los pequeños cambios que habría que realizar, ya que esta aplicación se ha desarrollado de forma muy general y el uso de la librería DRMAA hace que sea prácticamente igual usar un gestor u otro.

- Desarrollo de otras funcionalidades como puede ser la notificación de mensajes de administrador en *Posidonia* como revisiones técnicas y problemas de conexión.

- Creación de una web en la que se mantengan las principales funcionalidades de *Posidonia* para incrementar la movilidad que ya aporta la herramienta desarrollada.

- Programación de la aplicación en otros entornos móviles como *iOS* o *Windows Phone* ya que, si bien el carácter multiplataforma hace que *Posidonia* valga para cualquier sistema operativo de ordenador, en dispositivos móviles esto no ha sido posible por lo que habría que desarrollar aplicaciones nativas para cada entorno.

- Automatización de la conexión por medio de claves SSH: se pueden usar conjuntos de claves asociadas con el protocolo SSH para que no sea necesario la introducción de los datos de autenticación cada vez que se quiera utilizar *Posidonia*. Dadas las características de la aplicación desarrollada, se debería hacer de forma transparente al usuario y que sea compatible con los principales sistemas operativos.

Capítulo 7

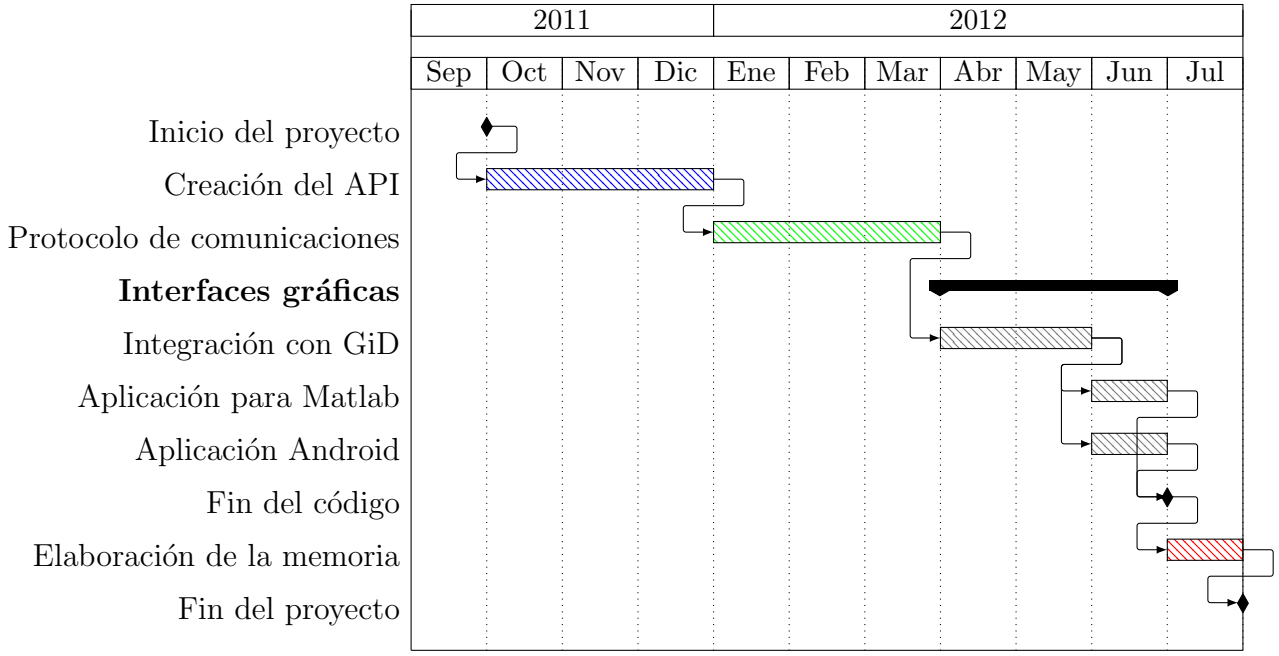
Presupuesto.

Tal y como se ha visto a lo largo de esta memoria, este proyecto implementa una herramienta para que la mayoría de usuarios puedan usar los servicios de un *cluster* de altas prestaciones computacionales estén donde estén, de una forma sencilla y eficiente.

Asimismo, se ha dividido el desarrollo del proyecto en las siguientes fases:

- Creación del API para la ejecución de trabajos en el *cluster* por medio de Java, que es lo que se expone en el Capítulo 3.
- Implementación del protocolo de comunicaciones para la interacción entre el equipo local y el *cluster*, tal y como se presenta en el Capítulo 4.
- Programación de la interfaz gráfica para ofrecer una fácil interacción con el usuario, como se describe en el Capítulo 5. Esta fase se puede dividir, a su vez, en otras tres:
  - Integración con FortranMod en GiD.
  - Aplicación autónoma de escritorio para trabajos con Matlab.
  - Aplicación Android para trabajos con Matlab.
- Una vez realizadas estas tareas se llega al hito de fin del código.
- Elaboración de la memoria del presente proyecto.

De forma que se crea el siguiente diagrama de Gantt detallando el desarrollo cronológico del proyecto:



Y finalmente, el presupuesto considerando todos los costes posibles (personas involucradas en el desarrollo, luz eléctrica, equipos empleados) es el que se muestra a continuación:

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor:

Adrián Amor Martín.

2.- Departamento:

Teoría de la Señal y Comunicaciones.

3.- Descripción del Proyecto:

- Titulo: **Herramienta de Simulación Remota en un *Cluster* de Computación Científica.**
- Duración (meses): **10 meses.**
- Tasa de costes Indirectos: **20%**

4.- Presupuesto total del Proyecto (valores en Euros):

Euros: 82227

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a titulo informativo)	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
Adrián Amor Martín.		Ingeniero Junior	6	2.450,00	14.700,00	
Ignacio Martínez Fernández.		Administrador Senior	3	2.600,00	7.800,00	
					0,00	
					0,00	
					0,00	
		Hombres mes	**9**	**Total**	22.500,00	

[a] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Laptop	840,00	100	10	24	350,00
Cluster del Grupo de Radiofrecuenci	163.498,00	50	10	24	34.062,08
Servidor Dedicado	4.800,00	50	10	24	1.000,00
Tablet Android	84,00	100	10	24	35,00
		100		60	0,00
					0,00
				Total	35.447,08

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
Energia electrica	Universidad Carlos III de Madrid	10.575,00
	Total	10.575,00

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,...

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	22.500
Amortización	35.447
Subcontratación de tareas	0
Costes de funcionamiento	10.575
Costes Indirectos	13.704
Total	82.227

Así, el presupuesto total de este proyecto asciende a la cantidad de 82.227 euros.

Leganés, a 26 de Julio de 2012.

El ingeniero proyectista.

Glosario

A

API *Application Programming Interface.*, pág. 47.

D

DRM *Distributed Resource Management.*, pág. 56.

DRMAA *Distributed Resource Management Application API.*, pág. xvi.

G

GRF *Grupo de Radiofrecuencia de la universidad.*, pág. 21.

H

HPCC *High Performance Computing Cluster.*, pág. 1.

I

IP *Internet Protocol.*, pág. 83.

J

JNI *Java Native Interface.*, pág. 58.

JSch *Java Secure Channel.*, pág. 19.

JVM *Java Virtual Machine.*, pág. 19.

L

LAN *Local Area Network.*, pág. 10.

M

MPI *Message Passing Interface.*, pág. 52.

O

OGE *Oracle Grid Engine.*, pág. 49.

OGF *Open Grid Forum.*, pág. 56.

P

PID *Process IDentifier.*, pág. 52.

S

SCP *Secure Copy.*, pág. 9.

SFTP *SSH File Transfer Protocol.*, pág. 17.

SGE *Sun Grid Engine.*, pág. 3.

SSH *Secure SHell.*, pág. 8.

T

TCP *Transmission Control Protocol.*, pág. 27.

TSC *Teoría de la Señal y Comunicaciones.*, pág. 9.

V

VPN *Virtual Private Network.*, pág. xv.

X

XML *Extensible Markup Language.*, pág. 2.

Bibliografía

[1] G. Reese, *Cloud Application Architectures: Building Applications and Infrastructure in the Cloud.* O'Reilly, 2009.

[2] "Página oficial del proyecto *Workflow System*," http://tigr-workflow.sourceforge.net/, Jul. 2012.

[3] "Página oficial de *eXludus*," http://www.exludus.com/, Jul. 2012.

[4] "Página oficial de *GE-GT Adapter*," http://www.gridwisetech.com/ge-gt/, Jul. 2012.

[5] "Página oficial de *Google Compute Engine*," http://cloud.google.com/products/compute-engine.html, Jul. 2012.

[6] "Página de GiD." http://gid.cimne.upc.es/, Jul. 2012.

[7] "Página oficial de Matlab," http://www.mathworks.es/products/matlab/, Jul. 2012.

[8] "Sitio oficial de Octave," http://www.gnu.org/software/octave/, Jul. 2012.

[9] "Página oficial de Android," http://www.android.com/, Jul. 2012.

[10] "Grupo de trabajo de SSH2," http://datatracker.ietf.org/wg/secsh/charter/, Jul. 2012.

[11] "RFC de SSH," http://www.ietf.org/rfc/rfc4251.txt, Jul. 2012.

[12] D. J. Barrett, R. E. Silverman, and R. G. Byrnes, *SSH, The Secure Shell. The Definitive Guide.*, 2nd ed. O'Reilly, 2005.

[13] "Página oficial de la herramienta *OpenVPN*," http://openvpn.net/, Jul. 2012.

[14] *Begginer's Guide to Sun Grid Engine 6.2*, 2008.

[15] "Página web de *Sun Grid Engine*," http://gridengine.sunsource.net/, Jul. 2012.

[16] "Web de *Oracle Grid Engine*," http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html, Jul. 2012.

[17] "Página oficial de la aplicación *Putty*," http://www.putty.org, Jul. 2012.

[18] "Página oficial del programa *WinSCP*," http://winscp.net, Jul. 2012.

[19] "Sitio oficial de Java en castellano," http://www.java.com/es/, Jul. 2012.

[20] "Página de la librería JSch," http://www.jcraft.com/jsch/, Jul. 2012.

[21] D. G. Doñoro, C. S. Maíz, L. E. G. Castillo, and I. G. Revuelto, "Implementación del Método de los Elementos Finitos en Fortran 90 basada en el Paradigma de Programación Orientada a Objetos." *URSI 2009*, Sep. 2009.

[22] J. Postel, "Transmission Control Protocol (TCP)," RFC 793 (Standard), Sep. 1981.

[23] "Noticia sobre el mercado de los dispositivos móviles," http://money.cnn.com/2012/05/16/technology/smartphones/index.htm, Jul. 2012.

[24] "Artículo informativo sobre *cloud computing*," http://www.infoworld.com/d/cloud-computing/what-cloud-computing-really-means-031, Jul. 2012.

[25] "Página oficial de la nube de apple, *iCloud*," http://www.apple.com/icloud/, Jul. 2012.

[26] "Página oficial de la nube de Google," http://cloud.google.com/, Jul. 2012.

[27] "Seleccionador de archivos de Android," http://code.google.com/p/android-filechooser/, Jul. 2012.

[28] "Página de información sobre *Blackmoon File Browser*," https://play.google.com/store/apps/details?id=com.blackmoonit.android.FileBrowser, Jul. 2012.

[29] "Grupo de trabajo de DRMAA," http://www.drmaa.org/, Jul. 2012.

[30] C. Newham and B. Rosenblatt, *Learning the bash Shell, Third Edition*. O'Reilly, 2005.

[31] "MPI, Message Passing Interface, Standard," http://www.mpi-forum.org/, Jul. 2012.

[32] "MPICH home page," http://www.mcs.anl.gov/research/projects/mpi/mpich1/, Jul. 2012.

[33] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Martin, and J. L. Wagener, *Fortran 90 Handbook*. McGraw-Hill, 1992.

[34] I. M. Fernández, "Creación y Validación de un Cluster de Cálculo Científico Basado en Rocks." Master's thesis, Universidad Carlos III de Madrid, 2009.

[35] "Sitio oficial de Gridway," http://www.gridway.org/doku.php?id=about, Jul. 2012.

[36] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[37] "Condor Project," http://research.cs.wisc.edu/condor/, Jul. 2012.

[38] "Sitio oficial de Xgrid," http://www.apple.com/macosx/server/, Jul. 2012.

[39] "Sitio web de EGI (*European Grid Infrastructure*)," http://www.egi.eu/, Jul. 2012.

[40] "Sitio oficial de Platform LSF," http://www.platform.com/workload-management/high-performance-computing, Jul. 2012.

[41] "Sitio oficial de UNICORE (*UNiform Interface to COmputing REsources*)," http://www.unicore.eu/, Jul. 2012.

[42] "Sitio oficial de *Kerrighed Cluster Framework*," http://kerrighed.org/wiki/index.php/Main_Page, Jul. 2012.

[43] "Sitio oficial de textitIBM Tivoli Workload Scheduler LoadLeveler," http://www-01.ibm.com/software/tivoli/products/scheduler-loadleveler/, Jul. 2012.

[44] "Sitio oficial de SLURM (*Simple Linux Utility for Resource Management*)," https://computing.llnl.gov/linux/slurm/, Jul. 2012.

[45] I.Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*. Elsevier, 2004.

[46] "Sitio oficial de Globus," http://www.globus.org/, Jul. 2012.

[47] "Tutorial sobre DRMAA," http://arc.liv.ac.uk/SGE/howto/drmaa_java.html, Jul. 2012.

[48] "Tutorial de Oracle sobre DRMAA," http://docs.oracle.com/cd/E19957-01/820-0699/gejso/index.html, Jul. 2012.

[49] "Javadoc sobre DRMAA," http://arc.liv.ac.uk/SGE/javadocs/jdrmaa/index.html, Jul. 2012.

[50] "Manual de SCP," http://linux.die.net/man/1/scp, Jul. 2012.

[51] "Página oficial de *Zehon*," http://www.zehon.com/, Jul. 2012.

[52] "Página oficial de *SSHTools*," http://sourceforge.net/projects/sshtools/, Jul. 2012.

[53] "Página oficial de Eclipse," http://www.eclipse.org/, Jul. 2012.

[54] "Página oficial de NetBeans," http://netbeans.org/, Jul. 2012.

[55] "Documentación Javadoc sobre la librería JSch," http://epaul.github.com/jsch-documentation/, Jul. 2012.