Universidad de Oviedo

Memoria del Trabajo Fin de Máster realizado por

HÉCTOR RODRÍGUEZ CAMPO

para la obtención del título de

Máster en Ingeniería de Automatización e Informática Industrial

CONTROL DE PROCESOS EN UN ENTORNO DE COMPUTACIÓN DISTRIBUIDA

MAYO 2017

Índice

Índice de figuras

Índice de tablas

1. Introducción

1.1. OBJETIVOS Y ALCANCE

El objetivo de este trabajo es plantear, y posteriormente desarrollar, un sistema distribuido de control de procesos para sistemas satelitales de observación de la Tierra. Estos sistemas requieren como mínimo dos segmentos:

- Segmento de Vuelo, que prepara el hardware y software de un satélite.
- Segmento de Tierra, en el cual se implementará el sistema propuesto en este documento.

Este último se encarga de recibir, procesar, distribuir, almacenar y analizar los datos recibidos de dicho satélite.

Los datos tomados por un satélite pueden comprender:

- Datos relacionados con el propio objetivo del satélite, la denominada “carga útil”, como por ejemplo un conjunto de imágenes tomadas en un momento dado.
- Datos concernientes al estado del satélite, oomo uso de memoria, planificación de maniobras...
- Pueden también ser datos generados en procesos intermedios, tales como información sobre la calidad del procesado de las imágenes, o el porcentaje de efectividad en el indexado o la distribución de éstas.

Debido al gran tamaño de datos a manipular y al hecho de que las misiones de observación de la Tierra suelen incluir a muchas partes interesadas, es importante definir arquitecturas de hardware, software y de red que sean capaces de aportar alta disponibilidad y rendimiento. Estas características conllevan la necesidad de uso de sistemas distribuidos, que repartan la carga de procesado entre múltiples dispositivos sin afectar al uso normal del sistema.

En este trabajo se plantea como objetivo el diseño e implementación de un sistema distribuido que pueda implantarse para cualquier tipo de proceso relacionado con un Segmento de Tierra de cualquier misión, cumpliendo estos requisitos además de proporcionar:

- Un servicio web que un usuario pueda utilizar para solicitar la ejecución de procesos, así como conocer el estado de la ejecución y el resultado de éstos.
- Una aplicación software encargada de la coordinación de los distintos nodos que formen la arquitectura distribuida.
- Una serie de módulos internos de la aplicación que se encarguen, por separado, de gestionar los recursos disponibles, implementar los estándares y servicios requeridos, y en general de lanzar los procesos solicitados por el usuario.
- Facilidades en la inicialización y parada en modo seguro del sistema, además de disponer de una fácil integración con herramientas de análisis.

Debido a la necesidad de utilizar el sistema de control de procesos enmarcado en un sistema de procesado general de mayor escala, será necesario ajustarse a los estándares utilizados en el sector aeroespacial, en este caso los propuestos por el *Open Geospatial Consortium* (OGC). También será necesario que el código fuente resultante de este proyecto esté en inglés en su totalidad, para facilitar la transmisión de conocimientos y la mantenibilidad del código en el ámbito de una empresa del sector espacial.

Para esta especificación, el desarrollo se ajustará al estándar OGC ya que existe la necesidad de proveer al usuario de una interfaz web, en concreto mediante servicios de procesado web remotos (WPS o Web Processing Services)[2], explicados con detalle en el apartado 1.2. Este estándar se implementará mediante software de terceros (framework WPS) seleccionado tras una comparativa de las diferentes opciones disponibles en la actualidad. Esta elección conllevará que el resto del desarrollo vaya acorde a su estructura.

Todo el sistema se desarrollará con la intención de desplegarlo en máquinas con un sistema operativo de tipo GNU/Linux, para favorecer una integración lo más eficiente posible. La interoperabilidad del sistema propuesto (es decir, su capacidad de ser integrado en diferentes entornos) se garantizará mediante un desarrollo apoyado en llamadas al sistema de tipo POSIX[3].

Como objetivos secundarios no funcionales se plantean:

- La generación un sistema de *logging* que registre mensajes de funcionamiento (para su posterior análisis) organizados por relevancia para el usuario (información, errores, *warnings*...). Este sistema de *logging* vendrá acompañado de los medios necesarios para permitir el acceso a los registros por parte de otras herramientas.

- Garantizar la disponibilidad del sistema. Se propondrá una arquitectura distribuida de alto nivel que solucione está cuestión en la medida de lo posible, teniendo en cuenta los efectos que la solución tenga en la eficiencia y funcionamiento general del sistema.

Finalmente, se realizarán simulaciones con la intención de analizar el comportamiento por separado del framework WPS, así como cada uno de los módulos software desarrollados, pudiendo comprobar el funcionamiento del sistema en conjunto.

1.2. OGC Y EL ESTÁNDAR WPS

1.2.1. OGC, The Open Geospatial Consortium

El Consorcio Geoespacial[1] es una organización sin ánimo de lucro dedicada a crear estándares de alta calidad orientados al *Open Source* y destinados al uso en la comunidad geoespacial. El objetivo de los estándares es mejorar la compartición de los datos geoespaciales en todo el mundo mediante un proceso de consenso que permita la mayor interoperabilidad posible.

Los estándares definidos por OGC son utilizados en una gran variedad de campos entre los que se encuentran el Análisis del Medio Ambiente, Defensa, Salud, Agricultura, Meteorología o Desarrollo Sostenible y cubren las diferentes necesidades de:

- Catalogado: Estándar CSW)[4].
- Acceso y modificación: Estándar WPS.
- Uso de lenguajes de marcado específicos: GML[5].
- Métodos de planificación.

Los miembros de OGC provienen de diferentes gobiernos, agrupaciones comerciales, ONGs y organizaciones académicas y/o de investigación, eiendo en total 522 entidades, entre las cuales se encuentran (por ejemplo): ESA[6] (Agencia Espacial Europea), NASA[7], Google o Airbus.

La filosofía de desarrollo de estándares por parte de OGC, parte de la necesidad de establecer una serie de especificaciones abstractas que sirvan como el fundamento y referencia

para todos los estándares específicos. Esto permite una alta interoperabilidad entre diferentes módulos de software destinados al procesado, que podrían haber sido creados por diferentes entidades, lo que favorece el desarrollo de la industria geoespacial.

1.2.2. WPS: Web Processing Services

El estándar de *Web Processing Services* es esencialmente una implementación de las normas abstractas de OGC, aplicada al funcionamiento de entradas/salidas de un sistema de procesado de datos geoespaciales. Define, por lo tanto, la interfaz que permite a otro módulo de software comunicarse con dicho sistema.

Además de definir estas normas, también especifica otros conceptos como:

- Las estructuras mediante las cuales un cliente puede recibir información sobre los métodos disponibles de procesado de datos geoespaciales, así como los parámetros de entrada y salida en su ejecución.
- Los diferentes modos de ejecución de procesos (tanto síncronos como asíncronos).
- El método de publicación de los resultados de una operación de procesado, así como especificar el formato y destino de estos.

La interacción de un cliente con un servicio WPS se basa principalmente en la utilización de una arquitectura cliente/servidor que implementa un servicio web ligero de tipo REST[8], aprovechando llamadas GET y POST. Estas llamadas contendrán la información necesaria para hacer peticiones de información, ejecución o resultados. Esta metodología se explicará con más detalle en el siguiente apartado (1.2.3).

En el estándar WPS 1.0 existen tres tipos de peticiones. Posteriormente, el estándar WPS 2.0 añadió funcionalidad asíncrona, y dos tipos de peticiones adicionales. Las cinco peticiones resultantes son:

- GetCapabilities: No requiere parámetros específicos. Devuelve un resumen de los procesos disponibles en el sistema para su ejecución, así como metadatos sobre la organización o el sistema sobre el que WPS ha sido desplegado.
- DescribeProcess: Se requiere pasar como parámetro el proceso a describir. Devuelve un identificador, una lista de entradas y salidas, y descripciones detalladas que incluyen los tipos de datos (enteros, *strings*, coma flotante) de cada parámetro del proceso.

- Execute: Se requiere pasar como parámetros el proceso a ejecutar y las entradas de este. Existen parámetros opcionales que indican dónde se debe almacenar el resultado o el formato que debe tener este, existiendo a su vez dos opciones de ejecución:

  - Síncrona: Se espera por una respuesta de WPS que puede contener el resultado del proceso o la localización de este.

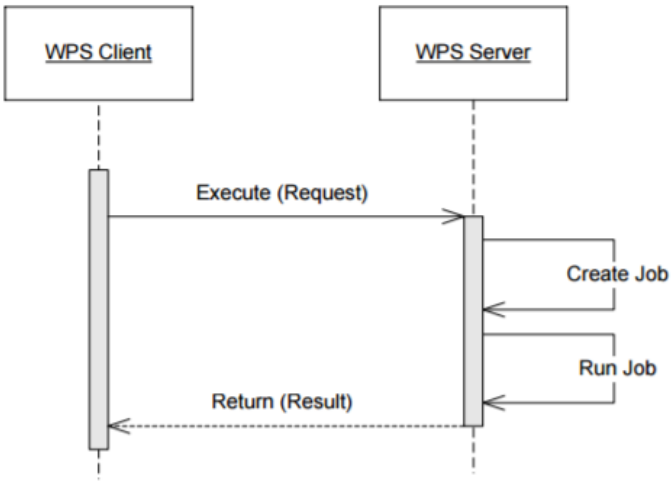

Figura 1.1: Funcionamiento síncrono de WPS.

Fuente:[2]

  - Asíncrona (Sólo en WPS 2.0): Se recibe por lo general una confirmación de que la herramienta de procesado se encuentra en ejecución, junto a un identificador que permite consultar su estado. Esta estrategia es una implementación de las metodologías de *polling* (el cliente se encarga de "preguntar" al servidor el estado de un proceso).

- GetStatus (Sólo en WPS 2.0): Se requiere pasar como parámetro el identificador de ejecución, y se recibe el estado del proceso. Puede ser STARTED, RUNNING, SUCCEEDED o FAILED. Mientras el proceso está en ejecución, es posible configurar información sobre el porcentaje o tiempo estimado de ejecución. También es posible indicar mensajes de error en caso de que el estado sea FAILED.

- GetResult (Sólo en WPS 2.0): Es posible acceder a la salida generada por la herramienta de procesado indicando el identificador de ejecución, siempre y cuando el estado del proceso sea SUCCEEDED.

Hay que tener en cuenta que el tipo de comunicación síncrona corresponde a una metodología basada únicamente en peticiones y respuestas, con la versión 2.0 y la ejecución

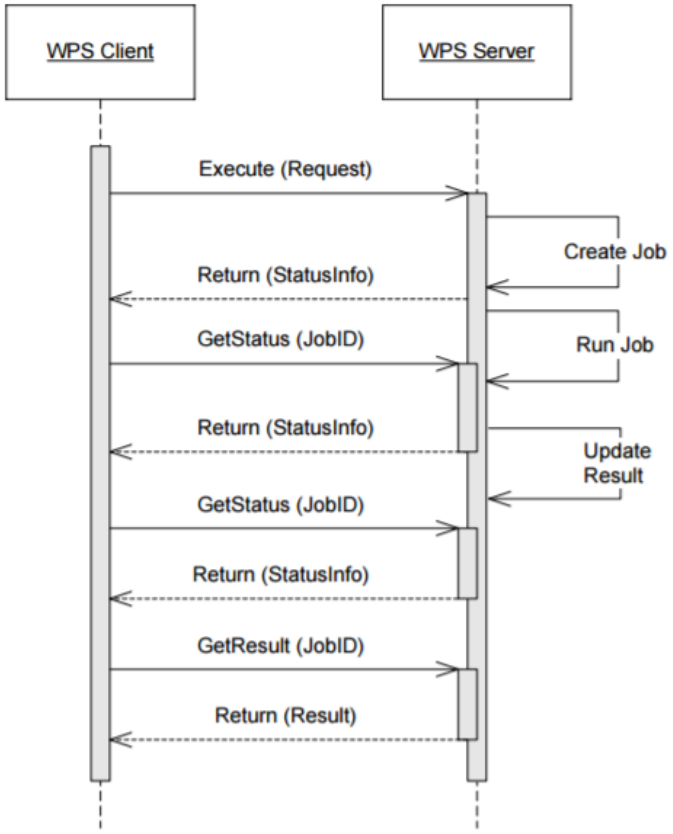

Figura 1.2: Funcionamiento asíncrono de WPS.

Fuente:[2]

asíncrona se introduce una metodología basada en *polling* mediante la ejecución periódica de peticiones de tipo GetStatus por parte del cliente.

También cabe destacar que el estándar no favorece esta opción en concreto, ya que añade parámetros de configuración opcionales que permiten indicar una fecha de caducidad de las respuestas dadas por el servidor, permitiendo consultar el estado de la ejecución de la herramienta de procesado una vez dicha fecha se cumpliera.

1.2.3. Servicios REST

Los servicios REST forman una arquitectura software utilizada en la *World Wide Web* como protocolo de comunicación ligero, es decir, que requiere pocas modificaciones si se parte del protocolo HTTP. Sus principales características son:

- Protocolo cliente/servidor sin almacenamiento de estado (toda la información va encapsulada en las peticiones y respuestas).

- Unas operaciones bien definidas. Estas son:
  - GET: Obtención de un recurso.
  - PUT: Generación de un recurso.
  - POST: Actualización de un recurso, o en su defecto, generación de dicho recurso si está permitido.
  - DELETE: Eliminación de un recurso.
- Sintaxis universal. Aunque no se trata de un estándar, su uso siguiendo un conjunto de normas único está muy generalizado.
- Uso de hipermedios (HTTP y XML).

Para los propósitos de este trabajo, se requiere utilizar REST para la comunicación entre el usuario y el sistema. En este caso, REST se implementa en los Web Processing Services (WPS) mediante dos tipos de operaciones:

- GET junto a parámetros introducidos en la URL/URI: utilizado en las peticiones tipo GetCapabilities, DescribeProcess, GetStatus y GetResult.
- POST junto a formularios XML que contienen los parámetros de ejecución: utilizado en peticiones tipo ExecuteProcess.

En el caso de las últimas, es posible implementar otras tecnologías orientadas a arquitecturas de mensajería como SOAP, aunque REST resulte una opción que aporta más eficiencia y poca sobrecarga en las peticiones (algo de especial interés al desplegar un servidor WPS).

2. Diseño del sistema

Según los objetivos definidos en el apartado 1.1 se ha planteado una arquitectura general del sistema, teniendo en cuenta los criterios de:

- Interoperabilidad. El resultado del diseño debe ser uno o varios módulos de software capaces de comunicarse e integrarse con facilidad en un sistema de mayor escala.
- Escalabilidad. En el caso de querer ampliar el número de nodos del sistema distribuido, no se debería percibir una disminución de la eficiencia. En este caso, se requiere que la comunicación y coordinación entre nodos no perjudique el rendimiento general del sistema.
- Extensibilidad. Es necesario que las funcionalidades de los módulos de software planteados puedan ser extendidas con la menor dificultad posible.

2.1. ARQUITECTURA DE ALTO NIVEL

2.1.1. Control de procesos

El sistema desarrollado se divide principalmente en varios módulos software, relacionados como se indica en la imagen 2.1:

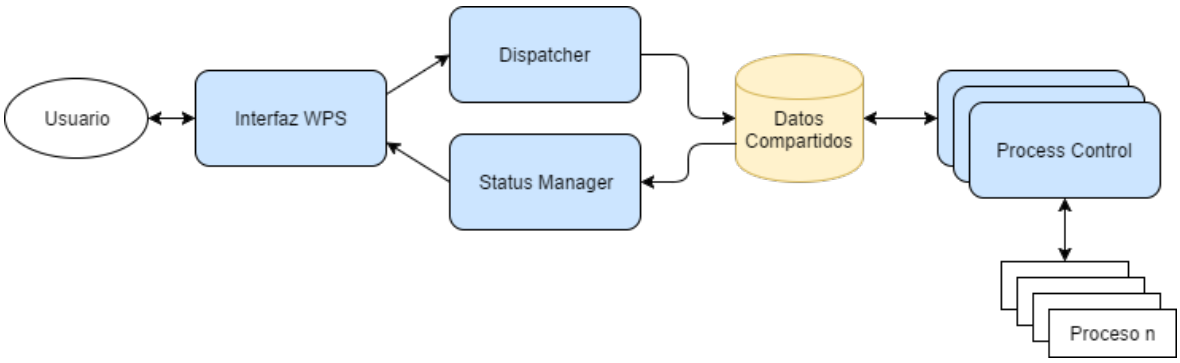

Figura 2.1: Diagrama de alto nivel del sistema.

El funcionamiento de los módulos (Figura 2.1, en color azul) se describe de forma detallada en los apartados que siguen a continuación (a partir de 2.3). La comunicación entre módulos se representa mediante las diferentes flechas que los interconectan, indicando a su vez el sentido de la comunicación.

2.1.2. Implementación de sistema distribuido

Los módulos desarrollados se pueden agrupar en dos tipos de nodos de red, maestro o esclavo, dando la capacidad a cada nodo de ejecutarse en ambos modos. Adicionalmente existe un área compartida de datos en la que se almacena información de control y los datos generados en el proceso.

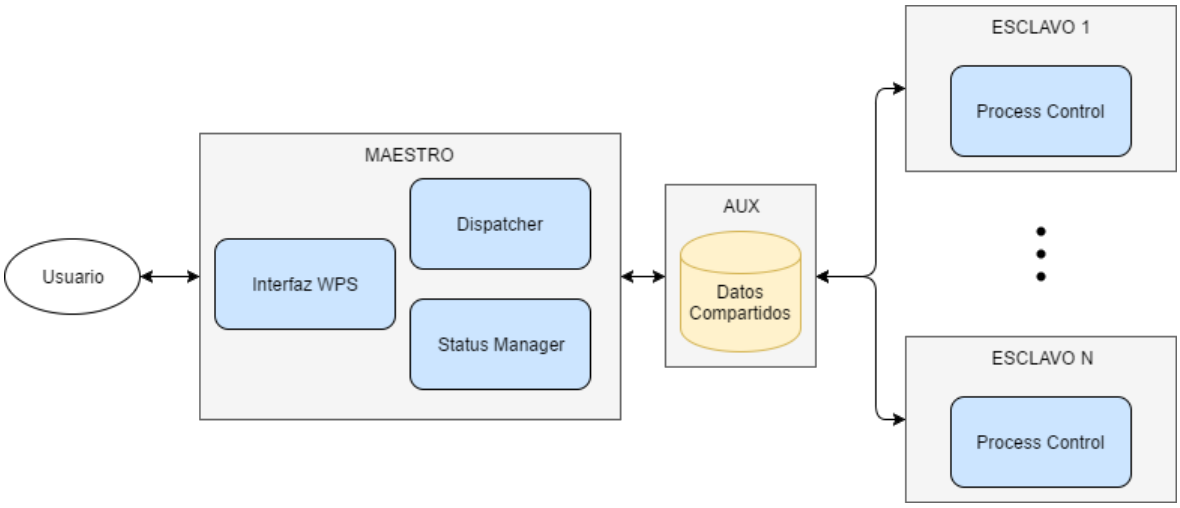

Figura 2.2: Diagrama de red.

La figura 2.2 muestra los módulos distribuidos en los diferentes componentes de la red planteada. Toda la lógica del sistema recae sobre un nodo maestro, mientras que la ejecución de procesos la realizan los nodos esclavo. Este diseño facilita tomar decisiones en torno a los recursos de los que debe disponer cada nodo. Adicionalmente se requiere un nodo que aloje el sistema de compartición de datos de control (AUX).

Una vez implementadas las técnicas de aumento de disponibilidad que se mencionan en 2.7, sólo sería necesario añadir redundancia y sistemas de *backup* en el nodo AUX.

2.2. EJECUCIÓN DE PROCESOS

La ejecución de procesos sigue una lógica secuencial de forma general, tal y como se indica en las figuras 2.3 y 2.4.

En el primer caso, la solicitud consiste en una petición tipo GET, recogida por el WPS. A continuación, haciendo uso de un archivo de configuración de servicios, responde al cliente con un resumen de aquellos que se encuentran disponibles.

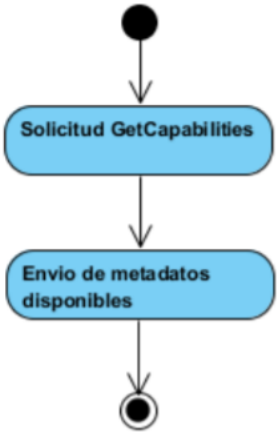

Figura 2.3: Secuencia simple de ejecución de una petición de tipo GetCapabilities.

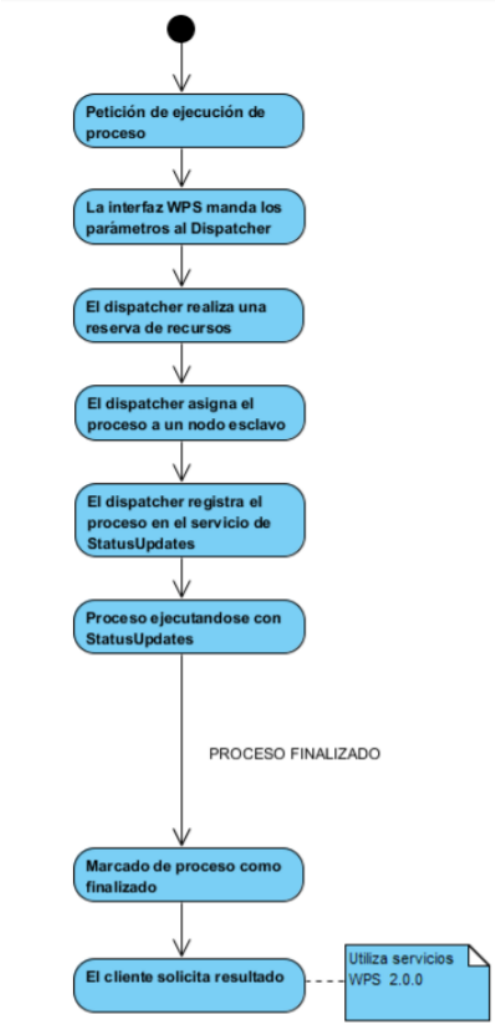

Figura 2.4: Secuencia de ejecución de una petición de tipo Execute.

En el segundo caso se presenta el flujo de ejecución de alto nivel de un proceso (mediante una petición del estándar de tipo “Execute”) visto desde el punto de vista de un nodo maestro. La ejecución del proceso sigue una estructura similar a la metodología asíncrona detallada en 1.2.2.

2.3. INTERFAZ WPS

La interfaz WPS implementa el estándar pero añade funcionalidad necesaria para la distribución de carga y la actualización de estados de procesos (Figura 2.5)

Figura 2.5: Funcionamiento de la interfaz WPS tras recibir una petición de tipo Execute.

Inicialmente la interfaz WPS se encuentra a la espera de recibir peticiones del cliente. Una vez recibida la petición POST de tipo *Execute*, lee los parámetros que se encuentran en el formulario adjunto y los prepara para enviarlos al *Dispatcher*. A continuación, abre una comunicación y se queda a la espera de información sobre el proceso. Cuando reciba dicha información, actualizará el estado del proceso para que el cliente pueda conocer la situación.

2.4. DISPATCHER

La funcionalidad del dispatcher resulta ser bastante simple (Figura 2.6): reserva los recursos en función de las peticiones del servicio WPS, interactúa con la zona de datos de control compartida y realiza una suscripción del proceso.

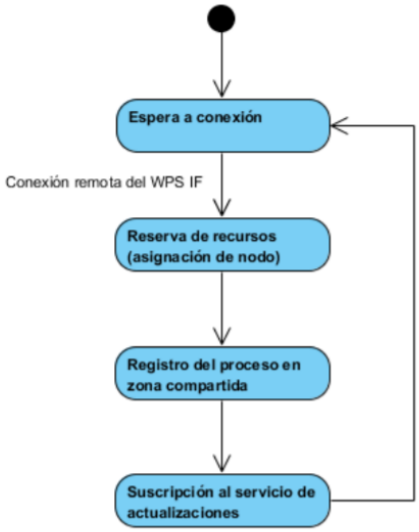

Figura 2.6: Diagrama de estado del módulo Dispatcher.

Cabe destacar que este módulo implementa un proceso transaccional, es decir, cualquier fallo en un paso intermedio no debe dejar rastro alguno (si no tenemos en cuenta los mensajes de error y la respuesta de tipo FAILED al cliente). De esta forma, no quedan bloqueados recursos del sistema, ni se sobreutiliza el servicio de actualizaciones de estado.

2.5. STATUS MANAGER

Este módulo se encarga de recorrer la zona de datos de control compartida en busca de procesos. Genera una lista actualizada de forma periódica que contiene identificadores de los procesos cuya ejecución ha sido solicitada por el cliente. Para cada elemento de la lista, comprueba que sigue existiendo una comunicación enviando en tal caso una actualización de estado.

De haberse cerrado la comunicación con el proceso, lo elimina de la lista y manda un mensaje de error a la interfaz WPS. Cuando ha terminado de recorrer la lista de procesos, se queda a la espera hasta la siguiente ejecución. Su funcionamiento se resume en la figura 2.7.

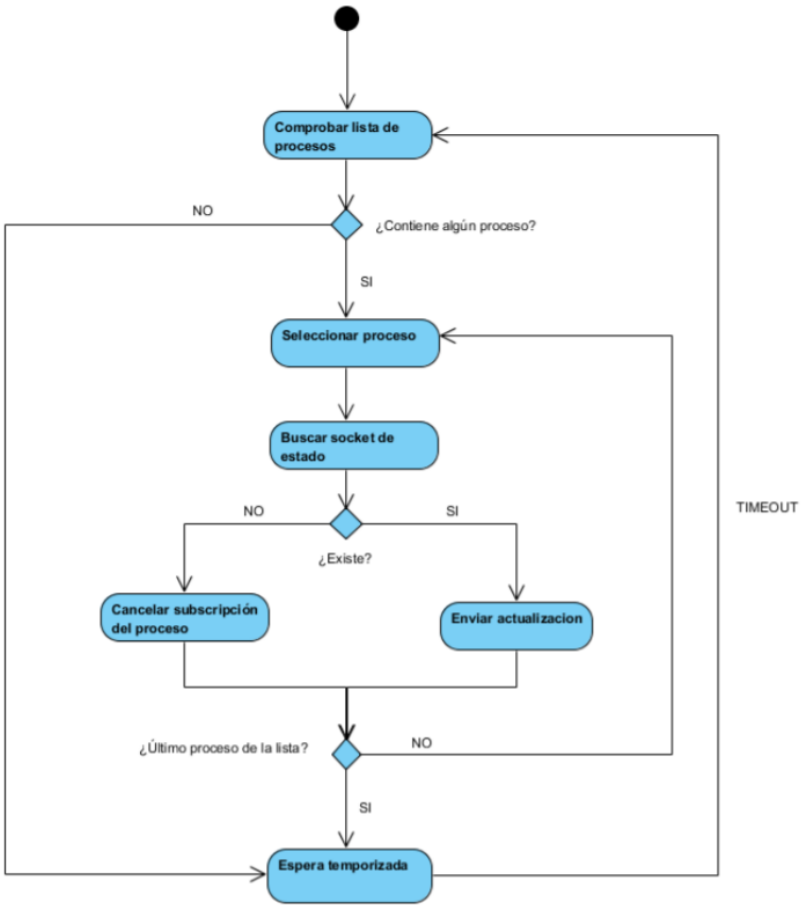

Figura 2.7: Diagrama de estado del módulo Status Manager.

2.6. PROCESS CONTROL

La zona de datos de control contiene la información necesaria para que este módulo pueda comprobar la existencia de nuevas peticiones de ejecución asignadas al nodo en el que se está ejecutando. Una vez detectada dicha petición, el proceso principal se bifurca dos veces generando dos procesos hijo que permiten leer el estado de la petición y actualizarlo para que el *Status Manager* haga uso de este. Esta división en subprocesos queda reflejada en la figura 2.8.

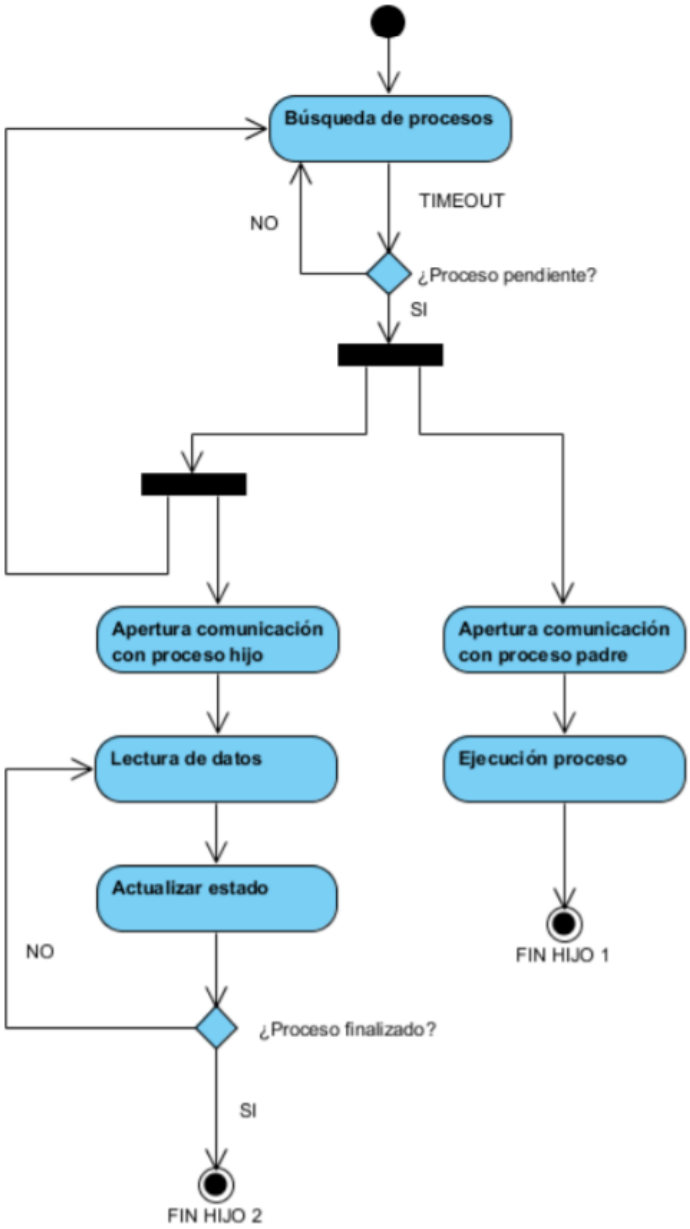

Figura 2.8: Diagrama de estado del módulo ProcessControl.

2.7. NET MANAGER

El *Net Manager* implementa la funcionalidad que permite crear una arquitectura distribuida con alta disponibilidad con un maestro y varios esclavos (apartado 2.1.2). Se plantea un sistema de inicialización con un envío de mensaje *broadcast* (envío de un mensaje genérico al resto de nodos de la red), protegido frente a pérdidas mediante una serie de reintentos, tal y como se observa en la figura 2.9.

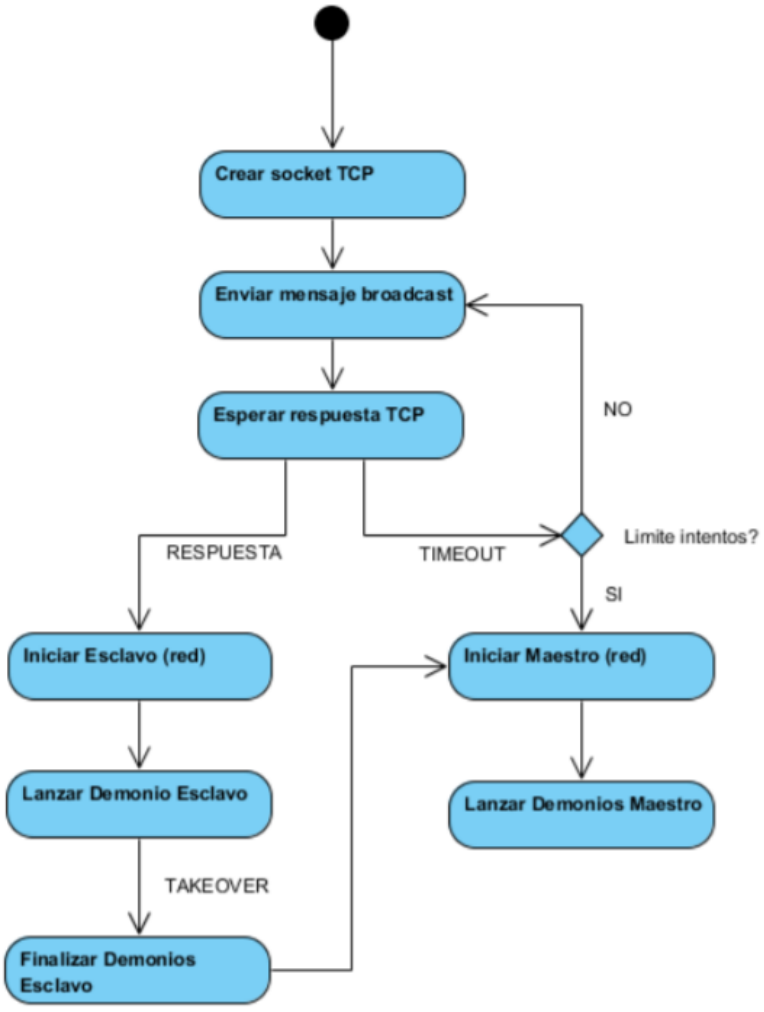

Figura 2.9: Inicio de un nodo del sistema distribuido.

También se detalla el funcionamiento del proceso de *Takeover* (Figura 2.10), mediante un sistema de prioridad en el que un esclavo (y sólo uno) puede tomar el rol de maestro en caso de fallo de disponibilidad del mismo. Este es un concepto clave del diseño del sistema, ya que elimina el riesgo de que procesos que se están ejecutando pierdan relevancia cuando el maestro deja de funcionar.

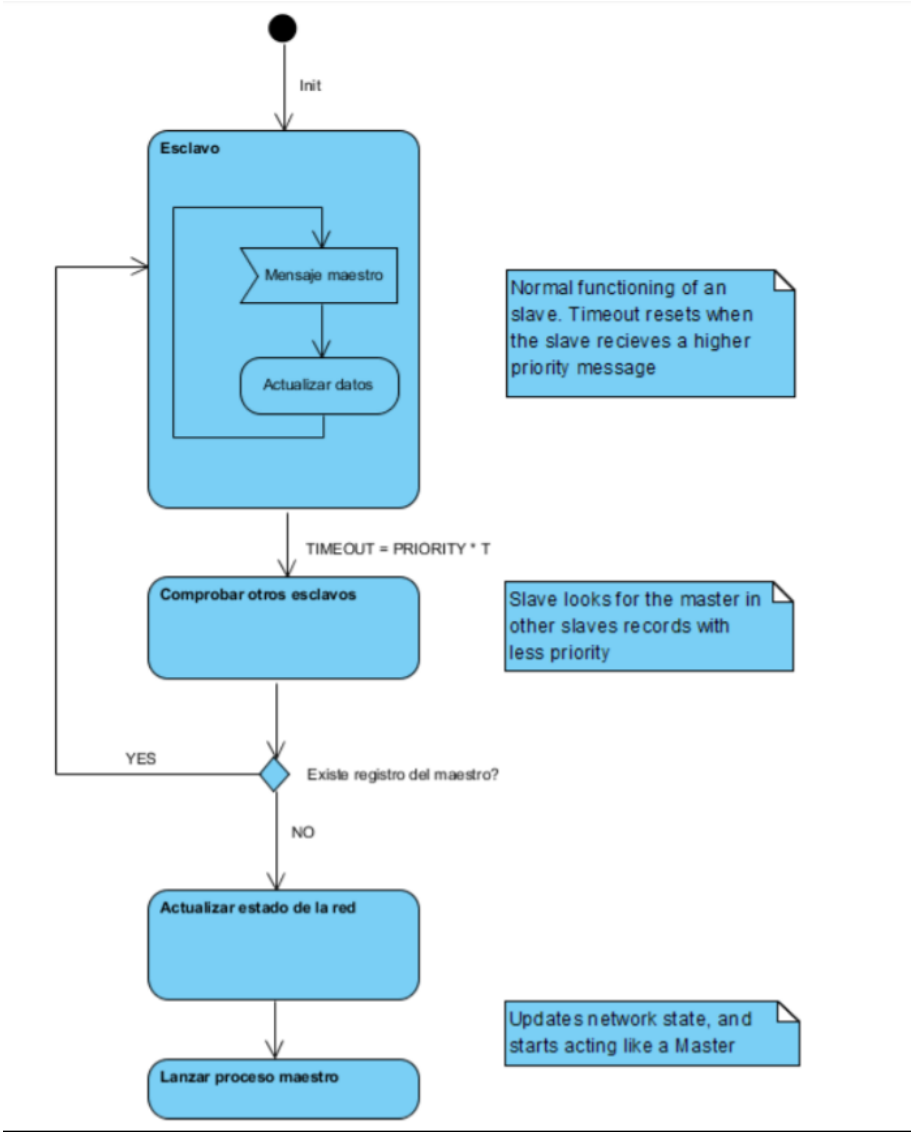

Figura 2.10: Toma de control del rol de maestro por parte de un esclavo

3. Implementación

La implementación tiene como objetivo obtener el comportamiento mencionado en el apartado 2 y que, siendo acorde a la arquitectura de alto nivel propuesta en 2.1, permita obtener un comportamiento adecuado cuando el cliente realice una petición WPS de tipo "Execute". La interacción entre los módulos propuestos se detalla en el diagrama de secuencia de la figura 3.1.

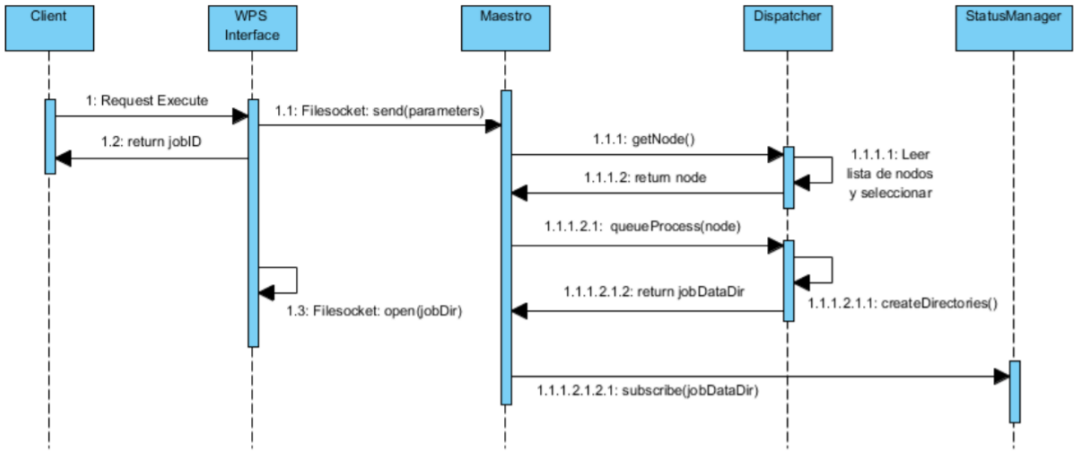

Figura 3.1: Diagrama de secuencia del sistema.

Como se puede observar en la figura, el cliente puede realizar una petición de ejecución de procesos, recibiendo un identificador de proceso con el cual puede conocer el estado de la tarea o su resultado. La interfaz WPS transmite la petición y sus parámetros al nodo Maestro, que hará uso del *Dispatcher* y el *Status Manager* para coordinar la ejecución y la actualización de estado del proceso.

Para ello, en los siguientes subapartados se detallan las tecnologías y metodología empleadas en la implementación del modelo propuesto en 2.

3.1. INTERFAZ WPS

En el caso de la interfaz WPS, al tratarse de un estándar muy detallado, se introduce un elemento de complejidad adicional en su implementación. En este caso, se ha decidido implementar WPS mediante la utilización de un framework. Idealmente, este framework será un módulo software predefinido que contendrá toda la funcionalidad incluida en el estándar.

3.1.1. Selección de *Framework* WPS

Los principales criterios de selección serán:

- Grado de implementación del estándar. Es esencial que el framework WPS seleccionado se ajuste a OGC, de forma que garantice la interoperabilidad característica del estándar, que forma parte de los requisitos del diseño.
- Complejidad de despliegue. Se pretende que el despliegue del sistema sea lo más simple posible, facilitando de esta forma su automatización (mediante instaladores de paquetes que automaticen el proceso).
- Dependencias de terceros. Se busca un framework que dependa lo menos posible de librerías de terceros que dificulten el mantenimiento a largo plazo.
- Documentación. Una comunidad de desarrollo y el acceso a manuales y entornos de aprendizaje suponen facilidades a la hora de implementar y mantener un framework WPS.
- Flexibilidad. Debe ser posible manipular el comportamiento de los servicios WPS para que estos puedan cumplir la funcionalidad de interfaz de comunicación que se pretende desarrollar en este trabajo.

Atendiendo a estos requisitos, se realiza un estudio de los frameworks WPS de licencia libre disponibles en la actualidad.

Inicialmente se considera PyWPS[9], basado enteramente en Python, un lenguaje de programación multiparadigma y orientado a objetos. Con Python, tanto el núcleo de funcionalidad WPS como el desarrollo de servicios dedicados resulta una tarea menos compleja, gracias a las extensas librerías e interfaces de diseño que proporciona.

PyWPS dispone de una documentación detallada y su instalación sobre un servidor Apache[10] es relativamente simple a la vez que permite un desarrollo de alto nivel gracias a sus librerías dedicadas de Python. Se presenta como la actual apuesta de futuro para el estándar, debido a su naturaleza *Open Source*, que permite implementaciones específicas como el framework WPS denominado COWS, realizada por el CEDA[11].

Sin embargo. se descarta ya que su principal desventaja es que tan solo implementa la primera versión del estándar, y requeriría manipular el código fuente y utilizar librerías de terceros para poder ampliar su funcionalidad hasta la siguiente versión (2.0).

Otra posible opción es GeoServer[12], una solución *Full Stack* que proporciona todas las herramientas para desplegar el sistema: el servidor, las librerías, los servicios y demás funcionalidades en un solo paquete indivisible. Está basado en Java con implementación mediante *servlets*. Esta tecnología permite ejecutar aplicaciones de Java en cualquier servidor, pero debe descartarse tanto por la caída en desuso de los *servlets* (debido a problemas de seguridad) como por la imposibilidad de manipular los servicios WPS o crear versiones propias de estos (último apartado de los requisitos).

Finalmente, se decide implementar ZOO[13], un *framework* que reúne prácticamente todos los requisitos enumerados: implementa la última versión del estándar, es relativamente sencillo de desplegar y es flexible. Aunque depende de como mínimo dos librerías de terceros, estas son de uso común y de mantenimiento regular (pueden ser obtenidas con cualquier instalador de paquetes de cualquier distribución Linux, tal y como se muestra en el Anexo I). Adicionalmente, permite el desarrollo de servicios y procesos en varios lenguajes de programación como Java, C++, Python.

Su principal punto débil es una documentación que no cubre toda la casuística de uso y que obliga a realizar pruebas con el objetivo de comprobar el funcionamiento correcto. Para poder implementar ZOO, se realiza una instalación sobre Red Hat[14] con Apache, detallada en el Anexo I (A.1) y se desarrolla el servicio basado en un script de Python propuesto a continuación:

```
# -*- coding: utf-8 -*-
#import pdb
import zoo
import time
import random
import os
import socket
import sys

def Multiply(conf,inputs,outputs):
  return ServiceHandler(conf,inputs,outputs)

def DelayPy(conf,inputs,outputs):
  return ServiceHandler(conf,inputs,outputs)
```

Este script se debe extender con una función para cada servicio que se añada (junto a los

archivos de configuración que correspondan).

Todos los servicios llamarán a una función genérica llamada *ServiceHandler*.

```
def ServiceHandler(conf,inputs,outputs):
   jobID = str(conf['lenv']['usid'])
   jobIDstr = "[JobID]\n"+jobID+"\n"
   identifier = "[Identifier]\n"+str(conf['lenv']['oIdentifier'])+"\n"
   inputsKeyValue = "[Inputs]\n"
   for key in inputs:
      inputsKeyValue = inputsKeyValue + str(key) + "=" + str(inputs[
         str(key)]['value']) + "\n"
   total = jobIDstr+identifier+inputsKeyValue
   f = open('/var/www/cgi-bin/Request','a')
   f.write(total)
   f.close()
   fail = sendProcessData(total)
   if fail:
      return zoo.SERVICE_FAILED
   outputs['result']['value'] = str(500)
   status = getUpdates(jobID,conf)
   print >>sys.stderr, 'FINAL status value is: "%d"' % status
   return zoo.SERVICE_SUCCEEDED
```

Este método se encarga de implementar la funcionalidad correspondiente a la interfaz WPS. Inicialmente recoge los parámetros del servicio y construye un *filesocket* de UNIX por el que los envía a cualquier proceso que esté escuchando (mediante el método *sendProcessData*). Tanto los *filesockets* como otras tecnologías cuyo uso sería posible en este entorno, se tienen en consideración en un apartado específico de este documento (3.1.3)

```
def sendProcessData(data):
   server_address = '/var/www/sockets/requests'

   # Create a UDS socket
   sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)

   # Connect the socket to the port where the server is listening
   try:
```

```
        sock.connect(server_address)
    except socket.error, msg:
        print >>sys.stderr, msg
      return 1
    try:
        print >>sys.stderr, 'sending "%s"' % data
        sock.sendall(data)
    finally:
      print >>sys.stderr, 'closing socket'
        sock.close()
```

Si no ocurre ningún error en la inicialización, comienza una nueva comunicación mediante la llamada al método *getUpdates*. En caso caso contrario, el servicio WPS retorna con valor 1, lo que propaga el error hasta la función anterior, que retorna el valor SERVICE_FAILED, marcando el proceso con un error que el cliente podrá observar con una petición GetStatus.

```
def getUpdates(jobID,conf):
    server_address = '/var/www/sockets/'+jobID
    # Make sure the socket does not already exist
    try:
        os.unlink(server_address)
    except OSError:
        if os.path.exists(server_address):
            return zoo.SERVICE_FAILED
    # Create a UDS socket
    sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    # Bind the socket to the port
    print >>sys.stderr, 'starting up on %s' % server_address
    sock.bind(server_address)

    # Listen for incoming connections
    sock.listen(1)

    status = 0
    finished = False
```

```
while (not finished):
      # Wait for a connection
      print >>sys.stderr, 'waiting for a connection'
      connection, client_address = sock.accept()
    try:
      total = ""
        print >>sys.stderr, 'connection from', client_address
        while True:
                data = connection.recv(1024)
                print >>sys.stderr, 'received "%s"' % data
                if data:
                  total = total + data
                else:
                  break
              status = updateStatus(total,conf)
      print >>sys.stderr, 'Status value is: "%d"' % status
      if status>0:
        finished = True

    finally:
      connection.close()

try:
          os.unlink(server_address)
     except OSError:
           if os.path.exists(server_address):
                 return zoo.SERVICE_FAILED
print >>sys.stderr, 'Socket has been closed. Returning value: "%d"'
     % status
return status
```

Este último método resulta más complejo, debido a que la interfaz WPS está actuando como lo que sería un "servidor" en las arquitecturas TCP/UDP más comunes. Las librerías de Python permiten implementar sin demasiada problemática los sistemas de "timeout" o los parámetros de una conexión por *filesocket*.

Cada vez que la interfaz WPS recibe un nuevo paquete (que proviene del *Status Manager*)

a través de esta conexión, ejecuta el método *updateStatus* cuyo retorno permite realizar un control de errores que se propaga hacia arriba en la jerarquía de ejecución.

```
def updateStatus(message,conf):
   endValue = 0
   percent = 0
   info = ""
   fields = message.split("|")
   print >>sys.stderr, 'Updating status to: "%s"' % fields[0]
   if fields[0]=="SUCCEEDED":
      endValue = 3
   if fields[0]=="FAILED":
      endValue = 4
   if fields[0]=="RUNNING":
      if len(fields)>2:
         percent = int(fields[1])
         info = fields[2]
      if len(fields)==2:
         percent = int(fields[1])
      zoo.update_status(conf,int(percent))
   return endValue
```

Este es el método encargado de actualizar el estado de un proceso en WPS, de forma que el usuario pueda realizar las peticiones de estado mediante el estándar, y recibir dicha información. Los valores de retorno (3 y 4) coinciden con los indicados por ZOO en su entorno de desarrollo, que permiten que su *kernel* (núcleo de la funcionalidad WPS) marque el proceso como exitoso (permitiendo que el cliente acceda a este con una petición GetResult) o como fallido.

3.1.2. Comprobaciones sobre el *Framework* seleccionado

Es necesario comprobar, antes de implementar sistemas de reparto de carga, que la interfaz WPS perteneciente a un único nodo va a ser capaz de soportar una determinada carga de peticiones.

Debido a que no es posible analizar el código fuente de ZOO, la mejor forma de comprobar su comportamiento es mediante tests de rendimiento y de estrés. Principalmente, interesa comprobar las capacidades de procesado concurrente del *kernel* o núcleo de ZOO.

El objetivo es lanzar un número alto de procesos (que requieran suficiente tiempo) para ver si ZOO los trata de ejecutar de forma secuencial o si los lanza de forma concurrente. Para eso se va a simular un servicio para que realice procesos de escritura en función de un parámetro pasado por el cliente WPS y guarde el tiempo empleado en un archivo. Luego se puede realizar una media de tiempos y se analiza el comportamiento de 1 a 100 peticiones en saltos de 10.

Número de peticiones	Tiempo transcurrido (Concurrente)	Tiempo transcurrido (Secuencial)
1	44,15184903 s	44,15184903 s
10	120,6031384 s	441,5184903 s
20	225,5183751 s	883,0369806 s
30	339,4547966 s	1324,555471 s
40	459,7724327 s	1766,073961 s
50	563,8365207 s	2207,592452 s
60	713,260539 s	2649,110942 s
70	819,3041601 s	3090,629432 s
80	952,9626749 s	3532,147923 s
90	1062,148803 s	3973,666413 s
100	1157,652121 s	4415,184903 s

Tabla 3.1: Tabla de tiempo transcurrido en las pruebas de ejecución realizadas.

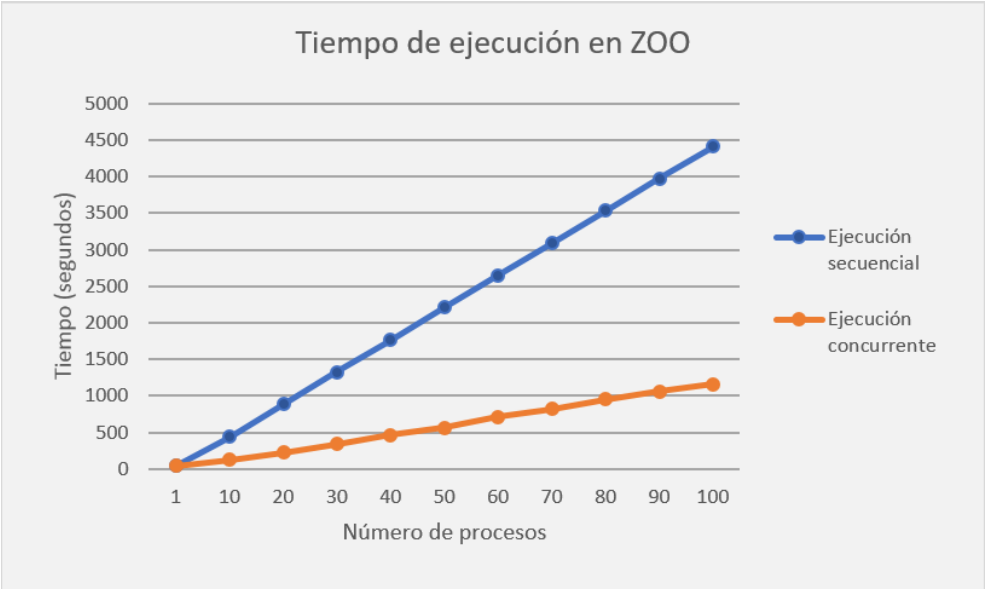

Figura 3.2: Comparativa de modos de ejecución de ZOO.

Como se puede comprobar con los datos de la figura 3.2, el factor limitante para esta interfaz reside enteramente en Apache y sus parámetros de configuración (número de hilos *workers*) y en última instancia en las limitaciones hardware del sistema, mientras que ZOO

apenas introduce sobrecarga en las operaciones normales. Esto permite modelar el resto del sistema teniendo en cuenta tan sólo las limitaciones correspondientes al resto de módulos.

Se realizan también pruebas para validar cada una de las funcionalidades WPS que se espera que implemente ZOO. En concreto las funcionalidades asíncronas y de actualización de estado tienen especial interés ya que son de gran importancia a la hora de implementar el funcionamiento general descrito en el apartado 3. Las peticiones se realizan mediante la librería de propósito general *Curl*[15] que permite emitir peticiones GET o POST.

Se han comprobado las siguientes peticiones [1]:

- Peticiones GET de tipo GetCapabilities y DescribeProcess
- Peticiones POST de tipo Execute con formulario XML de ejemplo (Anexo I, apartado A.2). Se comprueban peticiones pasando parámetros de forma directa o como referencia. También se envían formularios con protocolo SOAP para comprobar si admite funcionalidad.
- Peticiones GET de tipo GetStatus y GetResult

3.1.3. Comunicación con otros módulos

En este caso se va a implementar una funcionalidad de interfaz, en la que el servidor WPS actuará de "*relay*" entre el usuario del sistema y las funcionalidades de reparto de carga y comunicación de estado en la ejecución de estos. Esto quiere decir que se aplicará un patrón de diseño de tipo *proxy*, en el que la interfaz WPS hace de comunicador entre la funcionalidad interna y el cliente.

Para esto se requiere establecer una comunicación local entre los procesos que implementan la lógica del sistema y la interfaz WPS. Esta comunicación local en un entorno Unix puede realizarse mediante diferentes métodos. Los más utilizados en la actualidad son los *filesockets*, la memoria compartida o las *pipelines*[16]. Cada uno de ellos ofrece una funcionalidad distinta:

- *Filesockets*. Permiten implementar una comunicación entre diferentes aplicaciones ejecutadas sobre el mismo sistema operativo, muy similar a las conexiones TCP o UDP entre máquinas distintas, pero simulando una interfaz de red por medio de un archivo especial del sistema.

[1] Todas las pruebas validan el funcionamiento correcto de ZOO, y se adjutan por separado.

- Memoria compartida. Se destina una zona de memoria común entre ambos módulos, y se coordina su acceso. En esta zona de memoria se pueden escribir mensajes. Resulta el método más eficiente, porque no requiere archivos ni manipulación de entradas/salidas de un proceso.

- *Pipelines*. Consisten en la redirección de la salida de un proceso hacia la entrada de otro. Esta conexión tan simple puede dar lugar a sistemas de múltiples componentes entrelazados. Su uso es relativamente simple en sistemas Unix.

Se decide utilizar *filesockets* debido a que, aunque en este caso se plantea que ambos están siendo ejecutados en la misma máquina, la comunicación entre módulos no debería requerir que éstos compartan memoria. Tampoco es realmente necesario el uso de *pipelines*, ya que forzaría una separación de cada módulo en procesos independientes, añadiendo complejidad innecesaria.

La principal ventaja de utilizar *filesockets* es que permiten la implementación de protocolos personalizados. Puede ser interesante establecer una lógica de confirmación de llegada de información o un *timeout* que permita identificar errores en la ejecución. Tienen una baja eficiencia, al requerirse enviar cabeceras en los mensajes que a veces pueden ser más largas que la propia carga útil del mensaje. Sin embargo, este no es un factor limitante en la comunicación, que no requiere el envío de grandes cantidades de datos en una única conexión.

Cada petición consistirá en un enlace con el *Dispatcher* que está a la espera de conexiones, en la que se enviarán metadatos de la petición del usuario. A continuación, la conexión se cierra y se abre otra a la escucha de mensajes de estado enviados por parte del Status Manager.

Durante la ejecución del proceso, el cliente puede solicitar una actualización del estado de este. La implementación de esta funcionalidad presenta una arquitectura de tipo cliente/-servidor, en la que el usuario puede realizar peticiones (con respecto a un proceso concreto) a la interfaz WPS y esta le responde con el estado actual del proceso (Figura 3.3).

3.2. REPARTO DE CARGA: DISPATCHER

Para el caso del reparto de carga, es necesario que la reserva de recursos sea conocida por el maestro de la comunicación, incluido el caso en el que un esclavo tome el rol de nuevo maestro. Para ello, el maestro ejecutará un *Dispatcher* que se desarrolla teniendo en

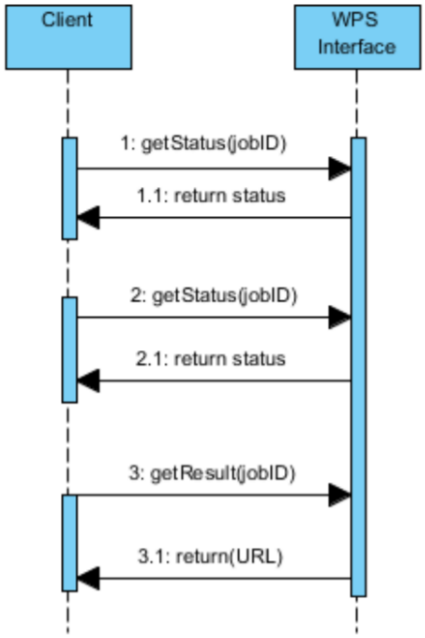

Figura 3.3: Implementación interna de las peticiones de tipo *GetStatus*.

mente la capacidad de retomar la ejecución en una máquina distinta sin que se resienta el funcionamiento general.

Esto es posible si se establece un método transaccional de reserva de recursos en el que, en caso de fallar la ejecución de un proceso, los recursos sean liberados como si la petición nunca se hubiera realizado. Esta reserva se realiza mediante la generación de archivos que contienen datos de los parámetros de la ejecución del proceso, identificador de proceso, tiempo estimado de ejecución y recursos reservados (memoria, tiempo de CPU y uso de funciones de entrada/salida).

Es importante también ajustarse a la concurrencia del servidor Apache de la interfaz WPS que recibe las peticiones (tal y como se demuestra en el apartado 3.1.2), para evitar que el Dispatcher actúe como un cuello de botella (limitador clave del rendimiento) en la aplicación.

Dicha concurrencia se implementa mediante *multi-threading* con un conjunto configurable de hilos (idealmente con la misma cantidad de hilos que los que tenga el servidor Apache) que irán sirviendo las peticiones, llevando a continuación el procesado de la petición: aplicación de la lógica de distribución, generación de archivos de control en recurso compartido y suscripción del proceso al servicio de monitorización de estado.

```
//OBJECT INSTANTIATION
//SOCKET OPENING


pthread_mutex_t dispatcherMutex;
pthread_mutex_t managerMutex;

pthread_mutex_init(&dispatcherMutex, NULL);
pthread_mutex_init(&managerMutex, NULL);

struct requestData data;
data.dispatcher = disp;        //Previously instantiated Dispatcher
data.statusManager = status;   //Previously instantiated Status
   Manager
data.sock = sock;              //Reference to WPS Interface socket
data.dispatcherMutex = dispatcherMutex; //Mutex for exclusive
   access to dispatcher methods
data.managerMutex = managerMutex; //Mutex for exclusive access to
   manager methods
data.logger = log;             //Logging object

pthread_t net;

for (int i=0;i<maxThreadCount;i++){
   pthread_create(&net, NULL, netRequest, (void*)&data);
}
```

Este código permite crear un número determinado de hilos que estarán a la escucha de peticiones a través de un socket UNIX. Cabe destacar que cada hilo recibe como parámetros tanto un objeto único de tipo “Dispatcher” como el objeto único “StatusManager” declarados ambos en la misma aplicación. El código de ejecución del hilo es el siguiente:

```
void * netRequest (void * params) {

      requestData *data = (requestData*) params;
      Dispatcher* disp = data->dispatcher;
      Logger* log = data->logger;
      int sock = data->sock;
```

```
StatusManager *manager = data->statusManager;
pthread_mutex_t dispatcherMutex = data->dispatcherMutex;
pthread_mutex_t managerMutex = data->managerMutex;

sockaddr_un remote;
uint len = sizeof(struct sockaddr_un);

int buffLen = 1024;
char buffer[buffLen];
memset( (void*) buffer, 0, buffLen);

while (true){
      string message = "";
      int conn = accept(sock, (sockaddr*)&remote, &len);
      cout << "[INFO] Connection received from remote"<<endl;
      log->logMessage("notice","Dispatcher","Connection
         received from remote");


      while (recv(conn, &buffer, buffLen-1, 0) > 0) {
         string str(buffer);
         message = message + str;
         memset( (void*) buffer, 0, buffLen);
      }

      string id = "DispatcherThread";
      stringstream ss;
      ss << conn;
      string str = ss.str();
      id = id + str;
      log->logMessage("notice",id,"Message received from WPS
         IF");

      pthread_mutex_lock(&dispatcherMutex);
      string node = disp->getNode();
      log->logMessage("notice",id,"Selected node: "+node);
      string jobDir = disp->queueProcess(node,message);
      pthread_mutex_unlock(&dispatcherMutex);

      pthread_mutex_lock(&managerMutex);
```

```
            manager->subscribeJob(jobDir);
            pthread_mutex_unlock(&managerMutex);
        }
        return 0;
}
```

Si bien es cierto que se persigue una arquitectura distribuida, el hecho de separar ambos módulos requeriría una comunicación adicional para que un único hilo realizara llamadas a ambos módulos y sería necesario realizar algún tipo de coordinación entre módulos. Esta opción se considera en el apartado de discusión (4).

3.2.1. Lógica de distribución

La lógica de distribución debe estar limitada por los recursos disponibles, de forma que pueda decidir en todo momento qué máquina debe ejecutar cada proceso. Para ello el Dispatcher conoce los recursos disponibles en cada máquina, establece cuáles podrían ejecutar el proceso y a continuación selecciona una de ellas (por ejemplo utilizando una lógica de tipo *round-robin*).

```
string Dispatcher::getNode(){

    if (currentNodes.size()>0){
        //int randomIndex = rand()  % currentNodes.size();
        int selectedNode = currentNode;
        currentNode++;
        if (currentNode==currentNodes.size())
            currentNode=0;
        return currentNodes[selectedNode];
    }
    else
        return "default";
}

string Dispatcher::queueProcess(string node, string message){
    processData data = parseMessage(message);
    string jobDir = createDirectories(node,data.jobID);
    createProcessFiles(jobDir,data);
    return jobDir;
}
```

Las funciones *getNode* y *queueProcess* se encargan de seleccionar el esclavo que debe ejecutar la tarea y se la asignan. Esta asignación se realiza mediante la creación de datos de control, descrita en el apartado 3.2.2.

3.2.2. Escritura de datos de control

La escritura en recursos compartidos no tiene por qué protegerse frente a condiciones de carrera generadas por la filosofía multi-hilo del Dispatcher. Esto es debido a que solo se realiza una escritura inicial y no se vuelve a manipular los archivos, que pasan a ser responsabilidad del *Status Manager* y del *Process Control*.

Sin embargo, si se debe tener en cuenta el uso de otros recursos compartidos, como es la lista de procesos suscritos en el *Status Manager*. Este método, tal y como se explica en el subapartado siguiente (3.3), manipula una lista de procesos y es susceptible a las problemáticas asociadas a accesos múltiples. Para evitar esto el acceso a ese método debe bloquearse mediante un mútex.

3.3. MONITORIZACIÓN DE ESTADO: STATUS MANAGER

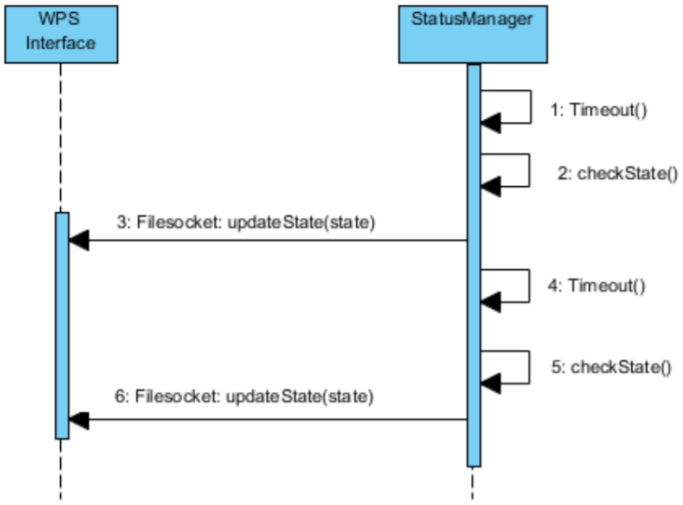

Figura 3.4: Comunicación entre la interfaz WPS y el módulo StatusManager.

El sistema de monitorización de estado se basa principalmente en una patrón de diseño de tipo publicación/subscripción de procesos a un servicio de actualizaciones. Para ello, se registra un proceso en la lista de actualizaciones, y es el propio servicio el encargado de cancelar esa suscripción una vez finaliza la ejecución de dicho proceso. Hasta que se cancela

la suscripción, el Status Manager comunica el estado del proceso de forma periódica a la interfaz WPS.

A continuación se muestran las funciones de suscripción y cancelación de la suscripción.

```
void StatusManager::subscribeJob(string jobD){
   vector<string> parts = split(jobD,'/');
   subs[parts[parts.size()-1]]=jobD;
   log->logMessage("notice","StatusManager","Job "+parts[parts.size
      ()-1]+" has subscribed.");
   if (subs.size()>10){
      log->logMessage("warn","StatusManager","Number of subscribers
          too high. It is possible that processes are not being
         executed");
   }
}
void StatusManager::unsubscribeJob(string jobID){
   subs.erase(jobID);
   log->logMessage("notice","StatusManager","Job "+jobID+" has
      unsubscribed.");
}
```

Este módulo interno se comunica directamente con la interfaz WPS, actualizando el porcentaje de finalización de la tarea, para que este pueda indicarlo como respuesta a peticiones de tipo *GetStatus* (Figura 3.3).

Este porcentaje se lee de un archivo de control compartido, perteneciente a la estructura comentada en el siguiente apartado (3.4). Esta lectura se realiza de forma periódica, para lo que se crea un hilo que ejecuta el código del método *statusUpdates*.

```
void * statusUpdates (void * params){
      updateResources *resources = (updateResources*) params;
      StatusManager * manager = resources->statusManager;

      manager->initialize();

      pthread_mutex_t managerMutex = resources->managerMutex;
      while(true){
         pthread_mutex_lock(&managerMutex);
         manager->sendUpdates();
```

```
            pthread_mutex_unlock(&managerMutex);
            sleep(2);
        }
        return 0;
}
```

La función *sendUpdates*, invocada cada 2 segundos, recorre la lista de suscriptores y envía un mensaje a la interfaz WPS para que actualice el estado que el usuario puede consultar.

```
void StatusManager::sendUpdates(){

    map<string, string>::iterator it;
    for (it = subs.begin(); it != subs.end(); it++) {
        bool accesible = true;
        string sockDir = socketsPATH+it->first;
        if( (access(sockDir.c_str(), F_OK ) == -1)) {
            unsubscribeJob(it->first);
            accesible = false;
        }
        if ((it->second!="")&&(accesible)){
            string msg = getStatus(it->second);
            if (msg!=""){
                sendMessage(msg,it->first);
            }
        }
    }
}
```

También es necesario almacenar una lista de procesos que están utilizando dicha subscripción al servicio de monitorización de estado. Esto permite proteger el sistema frente a caídas, a la vez que habilita un cambio de maestro sin pérdida de información crítica de estado.

3.4. COMPARTICIÓN DE DATOS DE CONTROL

La compartición de datos se implementa mediante NFS (*Network File System*)[17], un protocolo de nivel de aplicación que permite utilizar un sistema de archivos distribuido ac-

cesible por cualquier nodo. Este requiere un servidor específico independiente de la arquitectura WPS principal, que puede ser accesible mediante el sistema de archivos Unix.

La principal ventaja de esta implementación es que permite gestionar la fiabilidad de los datos de control de forma independiente al resto de componentes, además de que evita que los nodos esclavos tengan que almacenar y compartir la información de control. Es interesante evitar esto último, ya que requería envío periódico de mensajes que podrían saturar la red de una empresa u organización.

Los datos de control compartidos se organizan en dos tipos: funcionamiento general y gestión de procesos. En el caso de los datos de funcionamiento general, es necesario almacenar una lista de procesos suscritos al servicio de actualizaciones y, opcionalmente, una lista de dispositivos (IP y puertos de acceso) que actualmente forman parte del sistema (tanto si son maestros como esclavos). Para la gestión de procesos, se establece la siguiente jerarquía de archivos:

- *Slave*: Se trata de un directorio específico con un identificador de nodo esclavo.
  - *Resources*: archivo que almacena parámetros que indican los recursos utilizados. El *Dispatcher* los lee para determinar si el nodo es elegible para ejecutar un proceso.
  - *Process*: Un directorio por cada proceso que se ejecuta en el nodo.
    - *Input*: archivo indicando los parámetros iniciales, recursos a reservar, identificador.
    - *Status*: archivo indicando el estado actual de la tarea, que puede ser INIT (a la espera de ser ejecutado), RUNNING (acompañado de un porcentaje de avance en el proceso), SUCCEEDED o FAILED. Es modificado por el *Process Control* del nodo esclavo y monitorizado por el *Status Manager* del nodo maestro.
    - *Logging*: archivo con información de *logging*, marcas de tiempo, errores de ejecución...

En el caso de los directorios de tipo “Nodo esclavo” o “Proceso”, se requiere añadir una marca de tiempo de expiración que permita eliminarlos para no saturar la zona de compartición de datos.

3.5. LANZAMIENTO DE PROCESOS: PROCESS CONTROL

Tal y como se indica en el apartado de diseño, se ha decidido utilizar una estrategia de *forking*. La ejecución del control de procesos espera a que aparezca de un nuevo directorio que indica un nuevo proceso a tratar. Estos directorios de proceso, como se ha comentado previamente, se encuentran dentro de la carpeta asignada al nodo, a la que se asocia la ejecución de un único módulo de control de procesos. Este módulo se puede ejecutar desde cualquier nodo de la red, siempre y cuando apunte un directorio de control:

```
int main(int argc, char **argv) {
   if (argc<2){
         cerr << "[ERROR] Usage: ProcessControl <node_directory>"
            << endl;
         return 1;
   }
   string pathToNode = argv[1];
   string pathToNodeLog = pathToNode+"/processcontrol_log";
   Logger nodeLog = Logger(pathToNodeLog.c_str());

   while (true){
      findNewProcesses(pathToNode, nodeLog);
      sleep(2);
   }
   return 0;
}
```

Una vez se encuentra el directorio de un proceso nuevo (identificado mediante un estado de tipo INIT), se crean dos procesos hijo: el primero que ejecuta el script con los inputs indicados en el archivo *Input* a la vez que se comunica mediante *pipes* con el segundo, que recibe las actualizaciones de estado y las escribe en el archivo de *Status*.

3.6. TRAZABILIDAD DE ERRORES: LOGGING

En el caso de sistemas en los que se llevan a cabo operaciones concurrentes de control, es necesario tener un registro de actividad del sistema en el que también se registren errores de tal forma que puedan realizarse tareas de solución de errores y de mantenimiento, así como funcionamiento operacional.

Para ello se implementa también un módulo de *Logging* accesible por el resto de componentes anteriormente descritos, con una función compartida y con acceso bloqueante, que genera *logs* con un formato uniforme. Dicho formato es:

$$[FECHA][NIVEL][HOST/PROCESO][MENSAJE]$$

donde el nivel indica la relevancia del mensaje de *log* (a saber, error, warning o info). La principal función implementada para este módulo es *logMessage*, y puede ser invocada por cualquier otro módulo que incluya este módulo auxiliar.

```
void Logger::logMessage(string type, string process, string msg) {
   time_t   now = time(0);
   struct tm tstruct;
   char     buf[80];
   tstruct = *localtime(&now);
   strftime(buf, sizeof(buf), "[%b %d %X %Y] ", &tstruct);
   pthread_mutex_lock(&mutex);
   ofstream h(path,ios::app);
   h << buf << "[" << type << "] [" << process << "] " << msg <<
      endl;
   h.close();
   pthread_mutex_unlock(&mutex);
}
```

Estos logs pueden ser accesibles mediante un servicio REST separado para su representación en una interfaz gráfica (más detallado en el anexo 2).

3.7. MEJORA DE LA DISPONIBILIDAD: NET MANAGER

En este caso, tal y como se observa en el subapartado 2.1.2 se ha propuesto un sistema que al inicio emita mensajes *broadcast* y espere una respuesta por un puerto TCP indicado en dichos mensajes. Una vez recibida la respuesta a este mensaje broadcast, se determina si el nodo actua de esclavo o de maestro. A continuación se llevan a cabo envíos de mensajes de actualización del estado de cada nodo (su prioridad, disponibilidad, etc). Esta información es propaganda por parte de cada nodo hacia el resto de nodos que tengan menor prioridad. De esta forma, si un nodo deja de estar disponible, siempre habrá otro dispuesto a tomar su rol (en caso de que fuera maestro).

Sin embargo, esta solución no es apta para cualquier situación, ya que por lo general las redes empresariales suelen tener impuesta una limitación al tráfico en *broadcast*, para evitar la saturación de la red interna. Por ello, tal y como se indica en el apartado 3.4, debería ser posible almacenar dicha información en el área de control compartida. De esta forma todos los nodos podrían acceder a ella y no se requeriría un hilo en cada uno a la escucha de peticiones de información.

En cualquier caso, este módulo es en realidad el encargado de coordinar la ejecución del resto dentro de los nodos. Es decir, cuando el *Net Manager* detecte que el nodo debe esclavo, se debe ejecutar el *Daemon* Process Control, y si es maestro se ejecutarán el *Dispatcher* y el *Status Manager*. También sería el encargado de finalizar el primero de forma segura y lanzar los dos últimos en caso de tener que pasar de modo esclavo a maestro.

4. Discusión

El éxito en la implementación del sistema propuesto viene en gran medida dado por la capacidad de utilizar estándares previamente definidos, que ayudan a cumplir con el requisito clave de diseño: la interoperabilidad. La aplicación resultante, junto a sus módulos, puede usarse por cualquiera para gestionar procesos en cualquier ámbito del Segmento de Tierra de una misión espacial de observación terrestre.

Esto es debido en gran medida al uso de ZOO, un *framework* WPS que cumple en gran parte con todos los requisitos del estándar. Teniendo en cuenta las limitadas opciones, y la carencia de implementaciones suficientemente válidas de WPS, la decisión de usar ZOO ha permitido centrar gran parte de la atención del diseño al resto de requisitos descritos en los objetivos.

En general, los resultados obtenidos se ajustan a los objetivos propuestos en el apartado inicial de este documento. Aunque no se proporciona un desarrollo completo de toda la funcionalidad que se podría requerir en caso de utilizarse en una misión de observación de la Tierra, se han previsto la mayoría de ampliaciones del sistema.

Se ha obtenido también un sistema de fácil extensibilidad. Esto es así gracias a que el diseño lo ha tenido en cuenta desde el inicio, así como la capacidad de integrar el sistema en un otro de grado mayor, que gobierne el conjunto de funcionalidades de una misión.

Finalmente, uno de los puntos más importantes de este sistema es su usabilidad. Esto está estrechamente relacionado con el uso de módulo de *Logging* propuesto, que permite conocer el estado del sistema en todo momento, y la capacidad que este tiene de interactuar con tecnologías “Front-End” gracias a servicios adicionales propuestos en el Anexo II.

4.1. Análisis de las limitaciones del sistema

A lo largo del desarrollo del trabajo, se han identificado una serie de puntos de interés, relacionados tanto con decisiones de diseño como con limitaciones técnicas. En este apartado se detalla cada una de ellas.

En el caso de la interfaz WPS, es de especial interés crear una cola FIFO (*First-In-First-Out*) de procesos a ejecutar, que evitara problemas de sobrecarga en la interfaz (ya que el *filesocket* rechaza todas las conexiones a partir del número límite por defecto).

La reserva de recursos no se ha podido llevar a cabo con el *Dispatcher*, debido a que el entorno de pruebas estaba virtualizado de forma local, y existían limitaciones en el acceso a información sobre el hardware. Es importante tener en cuenta que este módulo tiene la capacidad de dar cohesión a una red de esclavos heterogénea, es decir, la configuración de los parámetros de recursos de cada nodo permitiría tratarlos a todos por igual sin riesgo de sobrecarga.

Por otro lado todos los módulos deberían estar protegidos frente a errores. Este es el caso del *Dispatcher* o de la interfaz WPS, pero no el caso del *Process Control*, ya que un fallo en el proceso padre que lanza otros procesos deja inutilizado el nodo esclavo, resintiéndose la eficiencia del sistema en general.

Debe tenerse en cuenta que, en un caso realista con integración completa en un sistema mayor, el módulo de control de procesos debería implementar muchas más funcionalidades, pero para el propósito de este trabajo se ha desarrollado una versión mínima que permita la integración con el sistema propuesto.

También es importante destacar la imposibilidad de desplegar operacionalmente (para empleo de usuarios o empresas) un *Net Manager* que use tráfico *broadcast* para iniciar el funcionamiento del sistema. Por lo general, las redes empresariales tienden a tener restringido el tráfico de este tipo, por motivos de seguridad y para evitar usos excesivos de ancho de banda.

Es por ello, que una solución viable sería mantener una lista de servidores disponibles en la zona de control y ampliar el protocolo del *Net Manager* para evitar lecturas “en falso” de dicho archivo (es decir, que un nodo lea una versión del archivo que ya se ha actualizado por otro nodo).

5. Conclusiones

A lo largo de este trabajo, el principal objetivo ha sido siempre que el sistema desarrollado pudiera utilizarse en el Segmento de Tierra de una misión espacial. Para ello, ha sido de vital importancia llevar a cabo un diseño teniendo en cuenta los siguientes aspectos claves:

- Implementar en todo momento las metodologías indicadas por los estándares utilizados en el sector aeroespacial.
- Definir un sistema fiable y que aporte una alta disponibilidad en la ejecución de procesos, mediante el diseño de un componente que gestiona la red de nodos y sus estados de disponibilidad.
- Priorizar un desarrollo basado en la extensibilidad teniendo en mente un uso operativo del sistema.
- Incluir una serie de funcionalidades que permitan una alta usabilidad del sistema.

La implementación de los estándares se ha podido complementar con la creación de módulos adicionales (como el sistema de gestión de estados de los procesos), mientras que la alta disponibilidad se ha podido garantizar mediante la compartición de datos de control y la implementación de un protocolo de comunicación y una arquitectura de red dedicada.

La extensibilidad se puede garantizar gracias a una arquitectura de alto nivel que permite añadir módulos adicionales que aprovechen un sistema común de comunicación. Finalmente, se ha tenido en cuenta la necesidad de informar sobre el estado del sistema, y se han creado servicios que permiten acceder a esta información.

Se debe tener en cuenta que el sistema resultante requiere extender algo más la funcionalidad para poder ser operativo, pero sirve como un punto de partida fiable sobre el que desarrollar un sistema que incluya todos los detalles restantes.

Todos estos aspectos se han tenido en cuenta, junto a las necesidades de funcionalidad requeridas en caso de un despliegue real, dando lugar a un resultado satisfactorio que cumple con los objetivos marcados al inicio del trabajo.

Referencias

[1] OGC: Open Geospatial Consortium.
www.opengeospatial.org

[2] WPS: Web Processing Service.
Open Geospatial Consortium
http://www.opengeospatial.org/standards/wps

[3] POSIX.1-2008
IEEE and The Open Group
http://pubs.opengroup.org/onlinepubs/9699919799/

[4] CSW: Catalogue Service.
Open Geospatial Consortium
http://www.opengeospatial.org/standards/cat

[5] GML: Geography Markup Language.
Open Geospatial Consortium
http://www.opengeospatial.org/standards/gml

[6] HMA Standards.
European Space Agency
https://earth.esa.int/web/gscb/hma-standards

[7] Standard Requirements and References.
National Aeronautics and Space Administration
https://earthdata.nasa.gov/user-resources/
standards-and-references

[8] RESTful Web Services Cookbook
Subbu Allamaraju
O'Really 2010.

[9] PyWPS 4.0.0
-
http://pywps.readthedocs.io/en/latest/

[10] Apache. The HTTP Server Project.
Apache Software Foundation
https://httpd.apache.org/

[11] CEDA OGC Web Services framework (COWS)
Centre of Environmental Data Archival
http://cows.ceda.ac.uk/

[12] GeoServer: An open-source server for geospatial data.
Open Source Geospatial Foundation
http://geoserver.org/

[13] ZOO project.
ZOO-Project team.
http://www.zoo-project.org/docs/intro.html

[14] Basic Guide for Enterprise Linux Servers
Richard Petersen
Osborne

[15] Everything curl.
Daniel Stenberg
https://www.gitbook.com/book/bagder/everything-curl/details

[16] The Linux Programming Interface
Michael Kerrisk
No Starch Press, 2010

[17] NFS Illustrated
Brent Callaghan
Addison-Wesley, 2014

[18] SELinux System Administration
Sven Vermeulen
Packt Publishing, 2013

A. Anexo I: Preparación de entorno ZOO en CentOS 6

A.1. PROCESO DE INSTALACIÓN

A.1.1. Adquisición de dependencias y código fuente de ZOO

Existen una serie de pre-requisitos y herramientas que facilitan la instalación. Su adquisición puede realizarse con cualquier gestor de paquetes (en este caso, por ser CentOS6, se usa yum):

- gettext (https://www.gnu.org/software/gettext/)
- cURL (http://curl.haxx.se)
- Flex & Bison (http://flex.sourceforge.net/ | http://www.gnu.org/software/bison/)
- libxml2 (http://xmlsoft.org)
- OpenSSL (http://www.openssl.org)
- autoconf (http://www.gnu.org/software/autoconf/)

Una vez adquiridos e instalados dichos paquetes, se pasa a realizar la instalación de dependencias como tal:

```
rpm -Uvh http://elgis.argeo.org/repos/6/elgis-release-6-6_0.noarch.
   rpm
rpm -Uvh \
http://download.fedoraproject.org/pub/epel/6/x86_64/epel-release
   -6-8.noarch.rpm
wget\
http://proj.badc.rl.ac.uk/cedaservices/raw-attachment/ticket/670/
   armadillo-3.800.2-1.el6.x86_64.rpm
yum install armadillo-3.800.2-1.el6.x86_64.rpm
yum install gcc-c++ zlib-devel libxml2-devel bison openssl \
  python-devel subversion libxslt-devel libcurl-devel \
  gdal-devel proj-devel libuuid-devel openssl-devel fcgi-devel
yum install java-1.7.0-openjdk-devel
```

Dentro del manual de instalación no figura una dependencia adicional sin la cual el sistema se descubrió que no funcionaba: hdf5.

```
yum install  hdf5-1.8.5.patch1-9.el6.x86_64
```

Finalmente, podemos descargar ZOO

```
svn checkout http://svn.zoo-project.org/svn/trunk zoo-src
```

Para generar un CGI (ejecutable para Apache) es necesario compilar la carpeta cgic (situada zoo-src/thirds/cgic-206/) modificando la variable LIBRARY_PATH del Makefile en caso de utilizar un sistema con arquitectura de 64 bits (el nuevo PATH debería ser /usr/-lib64/libfcgi.so). Para que esta librería sea accesible por el sistema es necesario añadir un archivo .conf (ejemplo zoo.conf) con la linea "/usr/local/lib" en /etc/ld.so.conf.d/ y ejecutar *ldconfig*

Una vez hecho esto, podemos realizar la instalación, que generará un CGI llamado *zoo_kernel*. Se debe indicar como parámetro la carpeta de datos de Apache a la que se accederá mediante peticiones:

```
cd zoo-project/zoo-kernel
autoconf
./configure --with-cgi-dir=/var/www/cgi-bin/ --with-etc-dir=yes --
   sysconfdir=/var/www/cgi-bin --with-fastcgi=/usr/local/ --with-
   python=yes
make
make install
```

A.1.2. Configuración de Apache

Es necesario añadir parámetros en el archivo de configuración del servidor Apache (llamado normalmente httpd.conf). Esta configuracion permite ejecutar archivos tipo ".cgi" (es decir, el ejecutable de ZOO):

```
ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"
AddHandler cgi-script .cgi

<Directory "/var/www">
   AllowOverride None
   Options +ExecCGI
```

```
   Order allow,deny
   Allow from all
</Directory>


<Directory "/var/www/cgi-bin">
   AllowOverride None
   Options +ExecCGI
   AddHandler cgi-script .cgi
   Order allow,deny
   Allow from all
</Directory>
```

A.1.3. Pasos finales

El paso A.1.1 genera un directorio con los archivos zoo_service.cgi y main.cfg que deben ser copiados al directorio que se haya configurado en Apache para ejecutar los servicios. Dicho directorio (en este caso el cgi-bin) debe tener asignado como dueño "apache" mediante el comando de UNIX "chown". Adicionalmente el servicio tiene que tener derechos de ejecución asignados con "chmod".

Finalmente, CentOS requiere configurar el entorno de seguridad de linux (SELinux) tal y como se indica en [18]. En caso de no realizarse esta configuración, los servicios de tipo CGI fallarán. Opcionalmente se puede desactivar SELinux con el comando:

```
setenforce 0
```

Para iniciar Apache, se requiere utilizar el siguiente comando:

```
service httpd start
```

A.2. *TESTING* DEL SISTEMA

A.2.1. Preparación del entorno de pruebas

Inicialmente se debe editar el archivo de configuración principal, llamado "main.cfg" para indicar valores de variables tal y como se indica en la documentación de ZOO. Un

ejemplo de dicha configuración sería la siguiente:

```
[headers]
X-Powered-By=ZOO@ZOO-Project

[main]
version=2.0.0
encoding=utf-8
dataPath=/mnt/hgfs/Shared
tmpPath=/mnt/hgfs/Shared
cacheDir=/mnt/hgfs/Shared
sessPath=/tmp
serverAddress=/mnt/hgfs/Shared
lang=en-ES,en-GB
language=en-US
msOgcVersion=1.0.0
tmpUrl=/

[identification]
keywords=t,ZOO-Project, ZOO-Kernel,WPS,GIS
title=ZOO-Project demo instance
abstract= This is ZOO-Project, the Open WPS platform.
accessConstraints=none
fees=None

[provider]
positionName=Developer
providerName=Deimos Space
addressAdministrativeArea=False
addressDeliveryPoint=Ronda de Pte., 19, 28760 Tres Cantos, Madrid
addressCountry=es
phoneVoice=+034918063450
addressPostalCode=28760
role=Dev
providerSite=http://www.deimos-space.com
phoneFacsimile=False
addressElectronicMailAddress=dummy@deimos-space.com
addressCity=Tres Cantos
individualName=Hector Rodriguez
```

Para poder realizar pruebas que demuestren el funcionamiento de ZOO, es necesario generar archivos que implementen servicios y sus metadatos. Estos archivos son:

- Fichero de metadatos del servicio. Es accedido por el kernel de ZOO cada vez que se recibe una petición de tipo “GetCapabilities” o “DescribeProcess”. Es imprescindible para el correcto funcionamiento, ya que sirve como referencia para descartar peticiones mal formadas. Un ejemplo de archivo de configuración es el siguiente:

```
[DelayPy]
 Title = Answer this call after a delay of 1 second.
 Abstract = Multiply two values and stor the result in Result.
 processVersion = 1
 storeSupported = true
 statusSupported = true
 serviceProvider = test_delay.py
 serviceType = Python
 <MetaData>
  title = Demo for testing delay
 </MetaData>
 <DataInputs>
  [a]
  Title = Delay amount
  Abstract = Amount of seconds sleeping before sending a
     response
  minOccurs = 1
  maxOccurs = 1
  <LiteralData>
   DataType = int
   <Default>
   </Default>
  </LiteralData>
 [Result]
  Title = JobID identifier
  Abstract = Shows internal variable for JobID
  <LiteralOutput>
   DataType = string
   <Default>
   </Default>
  </LiteralOutput>
</DataOutputs>
```

- Fichero de implementación del servicio. Este puede realizarse en cualquier lenguaje. La documentación para el desarrollo en cualquiera de ellos, así como el acceso a variables de configuración, entradas o escritura de salidas está disponble también. El archivo de configuración que se ha presentado, se corresponde con este código en Python:

```
# -*- coding: utf-8 -*-
#import pdb
import zoo
import time
import random
import os

def DelayPy(conf,inputs,outputs):
   sleeptime=inputs["a"]["value"]
   sleep(sleeptime)
   outputs["result"]["value"]=conf["lenv"]["usid"]
    return zoo.SERVICE_SUCCEEDED
```

A.2.2. Pruebas a realizar

Para verificar el funcionamiento de la interfaz WPS ZOO, se ejecutarán peticiones web que cubran los servicios soportados. Para ello se puede utilizar la librería auxiliar *curl* como se muestra a continuación. Opcionalmente, todas las peticiones de tipo GET (es decir, todas menos *Execute*) pueden realizarse mandando una petición con los parámetros mediante un navegador web cualquiera.

```
#Peticion del listado de procesos disponibles y otros metadatos.
    Debe devolver el servicio simple creado en el apartado
   anterior
curl -X GET http://localhost/cgi-bin/zoo_loader.cgi?Service=WPS
   \&Request=GetCapabilities\&Version=2.0.0

# Describe las entradas y salidas del proceso
curl -X GET http://localhost/cgi-bin/zoo_loader.cgi?Service=WPS
   \&Request=DescribeProcess\&Version=2.0.0\&Identifier=DelayPy

#Prueba de peticion Execute asincrona. Devuelve un jobID (
   identificador de proceso)
```

```
  curl -X POST -d @"Request.xml" http://localhost/cgi-bin/
     zoo_loader.cgi

  #Peticion de estado usando el jobID
  curl -X GET http://localhost/cgi-bin/zoo_loader.cgi?Service=WPS&
     Request=GetStatus\&JobID=XXXXXXX\&Version=2.0.0

  #Peticion de resultado usando el jobID
  curl -X GET http://localhost/cgi-bin/zoo_loader.cgi?Service=WPS&
     Request=GetResult\&JobID=XXXXXXX\&Version=2.0.0

  #Peticion de descarte del proceso y sus outputs usando el jobID
  curl -X GET http://localhost/cgi-bin/zoo_loader.cgi?Service=WPS&
     Request=Dismiss\&JobID=XXXXXXX\&Version=2.0.0
```

EL archivo XML utilizado para las peticiones *Execute* es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<wps:Execute
  xmlns:wps="http://www.opengis.net/wps/2.0"
  xmlns:ows="http://www.opengis.net/ows/2.0"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/wps/2.0 ../wps.xsd"

  service="WPS"
  version="2.0.0"
  status="true"
  store="true"
  mode="async">

  <ows:Identifier>DelayPy</ows:Identifier>
  <wps:Input id="a">
           <wps:Data>1</wps:Data>
  </wps:Input>
    <wps:Output id="result" status="true"/>
</wps:Execute>
```

B. Anexo II: Integración con herramientas de visualización

B.1. SERVICIO REST AUXILIAR

Para permitir el acceso y modificación de archivos del sistema, ya sea para configurar el funcionamiento de éste o para visualizar los archivos de *Logging*, es necesario desarrollar un servicio REST reducido. Este servicio proveerá de dos principales comandos: GET y POST.

B.1.1. Subida o actualización de archivos

Esta funcionalidad se debe implementar añadiendo la capacidad de recibir conjuntos de datos binarios y de reconstruirlos para su posterior almacenamiento en el sistema de ficheros local. Esto permite actualizar, por ejemplo, archivos de configuración que definan los parámetros de funcionamiento del sistema.

A continuación se muestra el código PHP simple que implementa la funcionalidad:

```
<?php
echo "Starting PHP script\n";
$fname = $_POST['fname'];
print_r($_FILES);
print_r($_POST);
if(isset($_FILES['file'])){
    //The error validation could be done on the javascript client
       side.
    $errors= array();
    echo "File received\n";
    $file_name = $_FILES['file']['name'];
    $file_size =$_FILES['file']['size'];
    $file_tmp =$_FILES['file']['tmp_name'];
    $file_type=$_FILES['file']['type'];
    $file_ext = strtolower(pathinfo($file_name, PATHINFO_EXTENSION))
       ;
    $extensions = array("xml","");
    if(in_array($file_ext,$extensions )=== false){
     $errors[]="image extension not allowed, please choose a XML or
        log file.";
    }
```

```
    if($file_size > 2097152){
    $errors[]='File size cannot exceed 2 MB';
    }
    if(empty($errors)==true){
   echo "Moving file from ".$file_tmp." to /var/www/html/data/".
       $file_name."\n";
        move_uploaded_file($file_tmp,"/var/www/html/data/".$file_name
            );
        echo "File uploaded \n";
    }else{
        print_r($errors);
    }
}
?>
```

PHP accede al formulario POST recibido, en concreto a una variable "FILES" que se genera por defecto. A continuación genera el archivo, que debe tener como máximo 2 Mb de tamaño. En el caso de superar este límite, sería necesario implementar una técnica de segmentación de ficheros, o recurrir a otras tecnologías más sofisticadas que implementen servicios web ligeros.

B.1.2. Descarga de archivos

Este es el servicio de mayor importancia, puesto que en el caso de una arquitectura distribuida es de vital importancia tener un visor web externo que, sin importar que máquina en la red ha generado el log, pueda acceder a este para realizar consultas.

```
<?php
echo "Starting PHP script\n";

$fname = $_GET['fname'];
if (isset($fname)) {

   $path = "/var/www/html/data/".$fname;
   $gestor = fopen($path, "r");

   $prev = "";

   $max = $_GET['max_size'];
```

```
  if (!isset($max)){
    $max = 1024*1024;
  }
  $results = "";

  while(($buf = fgets($gestor,$max)) !== false){
    $result = $result.$prev;
    $max = $max - strlen($buf);
    $prev = $buf;
  }
  echo "Size is ".strlen($result);
  echo "
    ----------------------------------------------------------------
    ";
  echo $result;
}

else {
  echo "Error: File does not exist";
}



?>
```

Es necesario pasar dos parámetros a este controlador GET: el nombre del fichero que se quiere obtener, y el tamaño deseado de la respuesta. Aunque no existe una limitación inicialmente, puede ser interesante (por motivos de eficiencia de la red) responder al cliente solamente con las últimas entradas del log.

Si bien la versión actual se centra principalmente en el tamaño del fichero devuelto, otros posibles parámetros podrían ser el número de entradas del log o un intervalo de tiempo. Sin embargo, resulta interesante dejar este tipo de filtros como tarea de una aplicación *front-end*, para evitar uso de la CPU innecesario (en caso de ficheros demasiado grandes).

B.1.3. Descubrimiento de recursos

Adicionalmente se ha desarrollado un script simple encargado de informar, mediante una lista, de los ficheros disponibles en el servidor.

```
<?php
$dir  = '/var/www/html/data';
$extensions = array("","xml");
$files = array_diff(scandir($dir), array('..', '.'));
foreach ($files as $file){
  $ext = pathinfo($file, PATHINFO_EXTENSION);
    if(in_array($ext,$extensions )=== true){
    echo $file."|";
  }
}
?>
```

B.2. CONFIGURACIÓN

Existen una serie de pasos que es necesario seguir para el correcto funcionamiento del servicio REST, a saber:

- Añadir reglas en el firewall del servidor que permitan el tráfico entrante a los puertos 80 y 8080 (HTTP) y saliente de éstos. En el caso de que la máquina sea de tipo Unix, estas normas pueden inyectarse mediante el servicio “iptables”.
- Habilitiar el CORS (Cross Origin Resource Sharing), que por defecto viene desactivado en cualquier servidor por motivos de seguridad. En el caso de Apache, es posible habilitar el acceso a recursos en directorios especificos (en los ejemplos del apartado anterior, el directorio es “/var/www/data/”.
- Registrar el directorio compartido por el servicio web en el sistema de *Logging* descrito en 3.6.

C. Anexo III: Planificación del proyecto

C.1. Diagramas de Gantt

A continuación se muestra el diagrama de Gantt que muestra la planificación del trabajo realizado (figura C.1) y las subtareas que forman las tareas más complejas (figura C.2).

1			Búsqueda de información	11 días	lun 04/07/16	lun 18/07/16	
5			Análisis de frameworks WPS	3 días	mié 20/07/16	vie 22/07/16	1
6			Implementación de interfaz WPS (ZOO)	18 días	lun 25/07/16	mié 17/08/16	5
10			Diseño del sistema distribuido	10 días	jue 28/07/16	mié 10/08/16	7
15			Desarrollo de la aplicación	19 días	jue 18/08/16	mar 13/09/16	6
23			Evaluación de resultados	8 días	vie 16/09/16	mar 27/09/16	15
27			Redacción de memoria del proyecto	8 días			

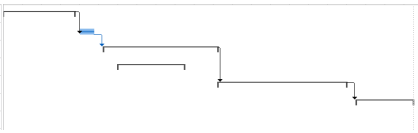

Figura C.1: Diagrama de Gantt simplificado.

1	✓		**Búsqueda de información**	**11 días**	**lun 04/07/16**	**lun 18/07/16**	
2	✓		Documentación sobre sistemas distribuidos	3 días	lun 04/07/16	mié 06/07/16	
3	✓		Estándares OGC	4 días	jue 07/07/16	mar 12/07/16	2
4	✓		Técnologías Web (Apache)	3 días	jue 14/07/16	lun 18/07/16	3
5	✓		**Análisis de frameworks WPS**	**3 días**	**mié 20/07/16**	**vie 22/07/16**	1
6	✓		**Implementación de interfaz WPS (ZOO)**	**18 días**	**lun 25/07/16**	**mié 17/08/16**	5
7	✓		Estudio de la documentación	3 días	lun 25/07/16	mié 27/07/16	
8	✓		Instalación	2 días	jue 11/08/16	vie 12/08/16	10
9	✓		Testing del framework	3 días	lun 15/08/16	mié 17/08/16	8
10	✓		**Diseño del sistema distribuido**	**10 días**	**jue 28/07/16**	**mié 10/08/16**	7
11	✓		Planteamiento de requerimientos	1 día	jue 28/07/16	jue 28/07/16	
12	✓		Modelado UML	4 días	vie 29/07/16	mié 03/08/16	11
13	✓		Diseño de red (Alta Disponibilidad) + UML	3 días	jue 04/08/16	lun 08/08/16	12
14	✓		Selección de tecnologías a utilizar	2 días	mar 09/08/16	mié 10/08/16	13
15	✓		**Desarrollo de la aplicación**	**19 días**	**jue 18/08/16**	**mar 13/09/16**	6
16	✓		Comunicación entre módulos	2 días	jue 18/08/16	vie 19/08/16	
17	✓		Servicio WPS	3 días	lun 22/08/16	mié 24/08/16	16
18	✓		Distribución de carga (Dispatcher)	4 días	jue 25/08/16	mar 30/08/16	17
19	✓		Propagación del estado de procesos	3 días	mié 31/08/16	vie 02/09/16	18
20	✓		Lanzador de procesos (ProcessControl)	4 días	lun 05/09/16	jue 08/09/16	19
21	✓		Zona compartida de datos	6 días	jue 25/08/16	jue 01/09/16	17
22	✓		Gestor de red (NetManager)	3 días	vie 09/09/16	mar 13/09/16	20;21
23	✓		**Evaluación de resultados**	**8 días**	**vie 16/09/16**	**mar 27/09/16**	15
24	✓		Desarrollo de entorno de pruebas	2 días	vie 16/09/16	lun 19/09/16	
25	✓		Corrección de errores	3 días	mar 20/09/16	jue 22/09/16	24
26	✓		Determinar requerimientos operacionales	2 días	lun 26/09/16	mar 27/09/16	25
27			**Redacción de memoria del proyecto**	**8 días**			

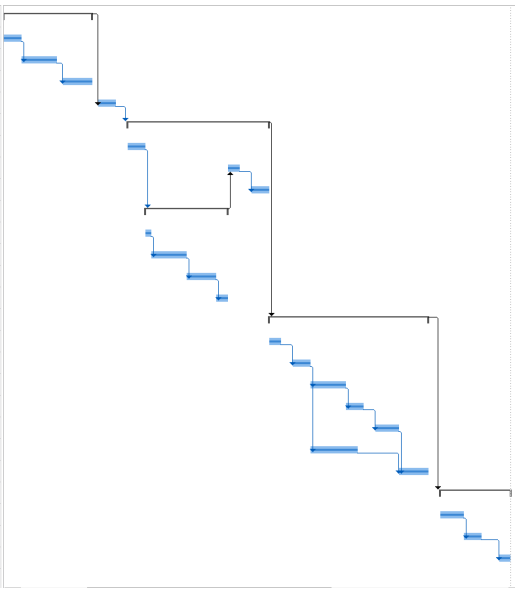

Figura C.2: Tareas realizadas, incluyendo las subtareas.

Cabe destacar que el driagrama refleja una estimación de días dedicados a cada tarea. Cada día consiste de entre 5 y 6 horas hábiles, equivalencia que se ha aplicado en el cálculo estimado del tiempo dedicado a la preparación de la documentación a entregar.

C.2. Presupuesto

A continuación se adjunta un presupuesto para el proyecto, teniendo en cuenta las horas de dedicación al desarrollo completo de éste, así como las horas dedicadas por tutores en la

asesoría para su diseño y documentación.

Tabla C.1: Presupuesto

Descripción	Precio/Unidad	Cantidad	Importe
Coste personal			
Ingeniero técnico (€/h)	16	465 h	7440 €
Material			
Depreciación o uso informático de ordenador personal (€/h)	0.5	465 h	232.5 €
Material fungible (papelería, etc.) (€)	1	20 €	20 €
Asesoría			
Tutorías (Ingeniero) (€/h)	60	20 h	1200 €

IMPORTE	8892.5 €
Gastos generales (15 %)	1333.88 €
Beneficio industrial (6 %)	711.4 €
BASE IMPONIBLE	10937.78 €
IVA (21 %)	2996.93 €
TOTAL	13234.71 €