Universitat Autònoma de Barcelona

Escola d Enginyeria
Departament d Arquitectura de Computadors
i Sistemes Operatius

TDP-Shell: Entorno para acoplar gestores de colas y herramientas de monitorizacion.

Tesis doctoral presentada por **Vicente-Jose Ivars Camanez** para optar al grado de Doctor por la Universitat Autonoma de Barcelona, bajo la direccion de los Drs. Miquel Angel Senar i Rosell y Elisa Ruth Heymann Pignolo.

Bellaterra, 22 de junio de 2012

TDP-Shell: Entorno para acoplar gestores de colas y herramientas de monitorización.

Tesis doctoral presentada por Vicente-José Ivars Camañez para optar al grado de Doctor por la Universitat Autònoma de Barcelona. Trabajo realizado en el Departament d'Arquitectura de Computadors i Sistemes Operatius de la Escola d'Enginyeria de la Universitat Autònoma de Barcelona, dentro del programa de Doctorado en Informática, bajo la dirección de los Drs. Miquel Angel Senar i Rosell y Elisa Ruth Heymann Pignolo.

Bellaterra, 22 de junio de 2012

Dr. Miquel Angel Senar i Rosell

Dra. Elisa Ruth Heymann Pignolo

Vicente-José Ivars Camañez

Resumen

Hoy en día la mayoría de aplicaciones distribuidas se ejecutan en clusters de ordenadores gestionados por un gestor de colas. Por otro lado, los usuarios pueden utilizar las herramientas de monitorización actuales para detectar los problemas en sus aplicaciones distribuidas. Pero para estos usuarios, es un problema utilizar estas herramientas de monitorización cuando el cluster está controlado por un gestor de colas.

Este problema se debe al hecho de que los gestores de colas y las herramientas de monitorización, no gestionan adecuadamente los recursos que deben compartir al ejecutar y operar con aplicaciones distribuidas. A este problema le denominamos "falta de interoperabilidad" y para resolverlo se ha desarrollado un entorno de trabajo llamado TDP-Shell. Este entorno soporta, sin alterar sus códigos fuentes, diferentes gestores de colas, como Cóndor o SGE y diferentes herramientas de monitorización, como Paradyn, Gdb y Totalview.

En este trabajo se describe el desarrollo del entorno de trabajo TDP-Shell, el cual permite la monitorización de aplicaciones secuenciales y distribuidas en un cluster controlado por un gestor de colas, así como un nuevo tipo de monitorización denominada "retardada".

Agradecimientos

En primer lugar, quiero agradecer a mis directores de tesis, Miquel y Elisa, la orientación y el apoyo recibido durante la realización de este trabajo tesis. Su ayuda, consejos y aportaciones han sido imprescindibles para poder finalizar con éxito este trabajo de tesis.

También quiero agradecer a los profesores del departamento (entre ellos Emilio, Lola, Tomàs, Remo, Anna y Porfi) , a mis compañeros doctorandos y al personal técnico del departamento (Dani y Javier), el apoyo recibido en los momentos difíciles que la realización de toda tesis conlleva.

En un contexto más personal, agradezco a mi mujer el apoyo que me ha dado de forma incondicional durante la realización de mi tesis, sobretodo porque ella también ha estado realizando su tesis. Gracias por estar siempre allí, sobretodo en los peores momentos.

No puedo dejar de agradecer la paciencia de mi hija Judith (que en los niños suele ser una virtud escasa) por las veces que no ha podido jugar, salir a pasear y en general disfrutar de la vida conmigo, sobretodo durante estos últimos meses de redacción de la memoria de tesis.

Para finalizar también quiero agradecer a mis padres y a mi hermano el apoyo constante y desinteresado que me han dado durante toda mi trayectoria universitaria, desde el comienzo de mi carrera universitaria hasta la finalización de este trabajo de tesis. Posiblemente sin este apoyo, yo no estaría en este momento redactando estos agradecimientos.

A todos vosotros, muchísimas gracias

Índice general

Índice de figuras

1

Introducción

El aumento de prestaciones y el abaratamiento de los diferentes recursos informáticos, como las CPU's, las memorias o los discos, ha permitido un importante incremento del número de clusters de ordenadores en los últimos años, tanto a nivel de investigación como empresarial. Este incremento de clusters de ordenadores ha implicado, cada vez más, la necesidad de crear sistemas que se encarguen de la correcta gestión de sus recursos. Persiguiendo este objetivo, en los últimos años se han desarrollado un conjunto de estos sistemas de gestión de recursos (también conocidos como *gestores de trabajos*) basados en colas (a partir de este punto *gestores de colas*), como Condor [1], SGE (Sun Grid Engine) [1], PBS (Portable Batch System) [4] o Torke [5].

Estos gestores de colas se encargan de ejecutar los trabajos de los usuarios en su cluster. Estos trabajos tienen asociados unos *entornos de ejecución* (variables de entorno, ficheros de configuración, dirección y nombre de los ejecutables, etc) que permiten poder ejecutar correctamente, en un cluster, las diferentes aplicaciones de los usuarios. Para gestionar la ejecución de los trabajos, los gestores de colas siguen los siguientes pasos importantes:

1) Esperar la llegada de trabajos de los usuarios, los cuales definen las aplicaciones que estos quieren ejecutar en el cluster. Para poder enviarle trabajos, los gestores de colas ofrecen unas interfaces especiales (suelen ser gráficas), que se ejecutan en las máquinas

[1] A partir de principios del año 2011, la continuación de SGE se ha dividido en dos caminos, el comercial con Oracle Grid Engine [2] y el open source, con Open Grid Schedule [3], pese a esta división y debido a la similitud entre estos dos nuevos gestores de colas, en este trabajo de tesis se seguirá haciendo referencia a SGE

locales de los usuarios que desean realizar este envío de trabajos.

2) Situar estos trabajos en una cola determinada, dependiendo de factores como la disponibilidad de recursos que necesita el trabajo en ese momento o la prioridad de éste (puede definirla el usuario).

3) Los trabajos esperan en su cola hasta que los recursos del cluster que necesitan están disponibles. Una vez ocurre este este hecho, son puestos en ejecución.

Junto a esta gestión de los trabajos, la mayoría de gestores de colas ofrecen unas características para adaptarse a la mayoría de arquitecturas de clusters, así como permitir a los usuarios poder gestionar sus trabajos. Estas características son:

a) Permiten la monitorización de los trabajos, para que el usuario pueda conocer, en cualquier instante de tiempo, en que estado se encuentran dichos trabajos (todavía están en la cola o ya están en ejecución).

b) Son tolerantes a fallos, esto implica que el gestor de colas se adapta a las posibles pérdidas de recursos en el cluster (p.e. la caída de una o varias de sus máquinas o nodos de comunicación). Si esto sucede, vuelve a planificar la ejecución de los trabajos existentes en sus colas, para aprovechar los recursos operativos del cluster existentes en un momento determinado.

c) Debido a la naturaleza heterogénea de los recursos de un cluster (diferentes tipos de máquinas, sistemas operativos y redes de interconexión) la mayoría de los gestores de colas son multiplataforma.

d) Permiten y facilitan la copia remota de archivos, tanto desde la maquina local del usuario a las del cluster como de estas últimas a la máquina local del usuario. Para realizar esta copia remota, los gestores de colas aprovechan los sistemas de archivos por red, como NFS. Si estos no están disponibles en el cluster, entonces realizan la copia directa de los archivos a los directorios de las máquinas del cluster que los necesitan.

e) Permiten definir a los usuarios donde el gestor de colas debe situar la información sobre la ocurrencia de errores durante la ejecución del trabajo, así como la de

finalización correcta del mismo. Esta información normalmente es situada en unos archivos especiales que son copiados en la máquina local del usuario.

El incremento de clusters de ordenadores también ha permitido el desarrollo de aplicaciones distribuidas (formadas por múltiples procesos), cuyas necesidades de capacidad de computo elevado se ajustan a los recursos que ofrecen estos clusters. Este nuevo entorno de ejecución distribuido, ha implicado que adquiera una especial relevancia la detección de cierto tipo de errores y problemas de rendimiento, como el incorrecto balanceo de carga entre máquinas o cuellos de botella en las comunicaciones entre diferentes procesos. Por este motivo, muchas *herramientas de depuración y monitorización (a partir de ahora herramientas de monitorización)* que monitorizan aplicaciones serie (formadas por un solo proceso), como GDB [6], Totalview [7] o Paradyn [8], se han adaptado para dar soporte a las nuevas necesidades de depuración y monitorización que requieren las aplicaciones distribuidas. Ejemplo de esto es la posibilidad que ofrecen Paradyn o Totalview, de mostrar gráficamente y en tiempo de ejecución, en que canales de comunicación (normalmente TCP/IP o UDP/IP) entre los diferentes procesos de la aplicación se producen cuellos de botella (normalmente debido a un alto intercambio de mensajes que satura la red de interconexión).

También se han creado herramientas de monitorización dedicadas exclusivamente a monitorizar aplicaciones distribuidas, como es el caso de STAT [9]. Esta herramienta reúne y fusiona (en un formato compacto e intuitivo basado en grafos) un conjunto de trazas con la información de las pilas de ejecución de los procesos de la aplicación distribuida (basada en el protocolo MPI [10] [11]). Estas trazas pueden ser analizadas utilizando la interfaz gráfica (STAT GUI) que ofrece la propia herramienta, ofreciendo además la posibilidad de utilizar otras herramientas, como Totalview, para realizar un análisis en mayor profundidad.

Relacionado con el tema de las herramientas de monitorización para aplicaciones distribuidas, recientemente se han desarrollado un conjunto de componentes para favorecer el diseño y funcionamiento de estas herramientas. Ejemplos de esto son el proyecto *LaunchMON* [12], cuyo objetivo es ayudar a la escalabilidad de las herramientas de monitorización que se ejecutan en un cluster, o la interfaz *MPIR* [13], cuyo objetivo es ofrecer un estándar para el diseño de herramientas de monitorización que trabajen con aplicaciones distribuidas basadas en el protocolo MPI.

Como la mayoría de clusters están controlados por un gestor de colas y las

herramientas de monitorización permiten operar con una gran diversidad de tipos de aplicaciones (tanto serie, como distribuidas), sería útil para los usuarios poder utilizar estas herramientas, en su forma habitual, para poder monitorizar sus aplicaciones que se ejecutan en estos clusters controlados por un gestor de colas. Por ejemplo, pudiendo iniciar la monitorización de un proceso desde la interfaz de usuario de la herramienta, como permiten Paradyn o Totalview. Pero actualmente este hecho está lejos de poder ser realizado, el motivo principal es que las herramientas de monitorización y los gestores de colas no están diseñados para compartir, la información y los recursos necesarios que les permitan realizar conjuntamente el proceso de monitorización. Este hecho provoca el problema que denominamos *falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización.*

En la literatura se puede encontrar otra referencia al problema de interoperabilidad relacionado con las herramientas de monitorización, el cual se produce cuando es difícil (o incluso imposible) utilizar simultáneamente diversas de estas herramientas para monitorizar la misma aplicación [14] [15] [16]. La idea de esta propuesta es utilizar las mejores opciones que ofrece cada herramienta para conseguir una monitorización más completa de la aplicación. Por ejemplo, utilizar la buena visualización de las variables que ofrece la herramienta A con la de los registros del procesador que ofrece la herramienta B y de esta manera poder detectar más rápidamente los errores de desbordamiento de segmento que se producen en una aplicación. Mientras que en este caso de falta interoperabilidad entre herramientas, el problema es debido a la falta de compartición de recursos entre estas herramientas que han de colaborar, el problema de interoperabilidad tratado en este trabajo de tesis, es debido a la dificultad de utilizar las herramientas de monitorización en clusters controlados por gestores de colas.

Por lo tanto, las contribuciones principales de este trabajo de tesis son:

- Explicar las causas que originan al problema de la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización.

- Explicar los pasos realizados para obtener una solución a dicho problema. Esta solución ha consistido en el desarrollo de un *entorno de trabajo denominado TDP-Shell* que es flexible para adaptarse a un gran conjunto de gestores de colas y herramientas de monitorización, evitando la modificación de sus códigos ejecutables y con un uso sencillo para sus usuarios.

En el resto de capítulos de este trabajo se explicará más detalladamente como se ha obtenido esta solución de la siguiente manera:

Capítulo 2) En este capítulo se explica el estudio del entorno inicial que rodea el problema de la falta de interoperabilidad entre los gestores de colas y herramientas de monitorización. Este estudio se ha dividido en dos puntos importantes : El primero ha consistido en el estudio de la arquitectura y los métodos de intercambio de información que ofrecen estos gestores de colas y las herramientas de monitorización. Por su parte, el segundo punto ha consistido en el estudio de las propuestas existentes para solucionar el problema objeto de este trabajo de tesis, para averiguar sus posibles limitaciones así como las aportaciones que pueden realizar al diseño del entorno TDP-Shell.

Capítulo 3) En este capítulo se explican las causas que originan el problema de la falta de interoperabilidad entre gestores de colas y herramientas de monitorización, así como la complejidad y el alto coste de tiempo que le puede implicar a un usuario encontrar y desarrollar una solución al problema.

Capítulo 4) Partiendo de lo explicado en los dos capítulos anteriores, en este capítulo se explica el diseño de la arquitectura base del entorno TDP-Shell, el cual soluciona el problema de falta de interoperabilidad entre las herramientas de monitorización y los gestores de colas en el caso de la monitorización de aplicaciones serie (formadas por un solo proceso).

Capítulo 5) En este capítulo se muestran las modificaciones que se han realizado en la arquitectura base del entorno TDP-Shell para solucionar el problema de la falta de interoperabilidad de las herramientas que monitorizan aplicaciones distribuidas y que se ejecutan en un cluster controlado por un gestor de colas.

Capítulo 6) En este capítulo se explica una modificación menor realizada en el diseño del entorno TDP-Shell, la cual permite que una herramienta de monitorización se adjunte a una aplicación que ya esta en ejecución en un cluster controlado

por un gestor de colas. A este caso especial se le denomina *Monitorización retardada*.

Capítulo 7) En este capítulo se muestran unos casos prácticos de como utilizar los diferentes componentes del entorno TDP-Shell para solucionar el problema de la falta interoperabilidad entre un conjunto de gestores de colas y de herramientas de monitorización. También se muestran algunos resultados obtenidos al evaluar la ejecución del entorno de trabajo TDP-Shell en clusters controlados por un gestor de colas.

Capítulo 8) En este último capítulo se explican las conclusiones extraídas de la realización de esta tesis doctoral, así como las lineas abiertas que ésta deja.

2

Estado del arte

Como se ha explicado en el capitulo 1 de introducción, en la actualidad la mayoría de clusters son controlados por un gestor de colas como Condor, SGE, PBS o Torke. También se ha explicado en este capitulo de introducción, que las herramientas de monitorización mas utilizadas actualmente, como Gdb, Totalview o Paradyn, se han adaptado para poder monitorizar aplicaciones distribuidas que se ejecutan en clusters de ordenadores. Por lo tanto seria útil que los usuarios pudieran utilizar estas herramientas de monitorización en clusters controlados por un gestor de colas, pero esto actualmente es muy difícil de conseguir, generando el *problema de la falta interoperabilidad entre los gestores de colas y las herramientas de monitorización*, la solución del cual es el objetivo de este trabajo de tesis.

En este capitulo se explicarán los dos puntos de partida que han permitido desarrollar el *entorno* `TDP-Shell` como una solución al este problema de falta de interoperabilidad. El primer punto ha consistido en el estudio de la arquitectura y de los métodos de intercambio de información que ofrecen los gestores de colas y las herramientas de monitorización para descubrir las causas que generan sus problemas de interacción. El segundo punto ha consistido en el estudio de las propuestas existentes para solucionar el problema de la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización, en este caso el *protocolo TDP* [17]. El estudio de estas propuestas puede permitir averiguar sus posibles problemas o limitaciones, así como extraer las ideas que ayuden al correcto planteamiento del diseño del entorno `TDP-Shell`. Para mostrar todas estas ideas, este capitulo se ha dividido en los siguientes apartados: En el primer apartado

se explicarán los motivos por los que se han escogido los gestores de colas Condor y SGE así como las herramientas de monitorización Gdb, Totalview y Paradyn. En el siguiente apartado 2.2 se explicará el formato de los ficheros de descripción de trabajos de Condor y SGE. En los siguientes 2 apartados se explicarán las características mas importantes de la arquitectura de los gestores de colas (apartado 2.3) y herramientas de monitorización (apartado 2.4) escogidos. En el apartado 2.5 se hará una introducción al protocolo TDP y en el último apartado se mostrarán las conclusiones extraídas de este capitulo.

2.1. Gestores de colas y herramientas de monitorización escogidas

Dependiendo del formato del fichero de descripción de trabajos que utilizan, los gestores de colas más utilizados se pueden clasificar en dos grandes grupos. El primer grupo está formado por los que utilizan *shell-scripts* para describir los trabajos que han de ejecutar en su cluster, a este grupo pertenecen SGE, PBS o Torke. El segundo grupo esta formado por los gestores de colas que utilizan un *formato propio de fichero para definir los trabajos* a ejecutar en su cluster, a este tipo pertenece Condor, el cual utiliza un *fichero de descripción de trabajos* basado en comandos. Debido a esta clasificación, para la realización de este trabajo de tesis se han escogido *Condor y SGE* como representantes del conjunto de gestores de colas.

Respecto a las herramientas de monitorización, se han escogido *Gdb, Totalview o Paradyn*, para la realización de este trabajo de tesis por dos motivos principales: El primero es que son las herramientas con una utilización más amplia entre los usuarios (tanto a nivel personal, académico y profesional). El segundo motivo es que dan soporte a una gran variedad de tipos de aplicaciones, esto es, monitorizan tanto aplicaciones serie (formadas por un solo proceso), como distribuidas (formadas por varios procesos ejecutándose en diferentes máquinas). En el capitulo de introducción se ha mencionado la herramienta STAT, la cual no ha sido escogida debido a que, de momento, está dedicada solo a aplicaciones distribuidas basadas en el protocolo MPI.

2.2. Ficheros de descripción de trabajos de Condor y SGE

Los *ficheros de descripción de trabajos* son el mecanismo que ofrecen los gestores de colas para que se les describa el trabajo que han de ejecutar en el cluster que gestionan. El objetivo de esta descripción, es informar al gestor de colas del *entorno de ejecución* necesario para que pueda ejecutar correctamente, en las diferentes máquinas del cluster, el o los procesos que se definen en dicho trabajo. Este entorno de ejecución esta formado normalmente por:

- Variables de entorno que definen cierta información necesaria para la ejecución de los procesos definidos en el trabajo (por ejemplo situación de ciertos de sus ficheros de configuración).

- Copia remota de los ficheros necesarios para la ejecución del trabajo, desde la maquina donde se realiza la petición al gestor de colas (normalmente la local del usuario) a las maquinas del cluster. Esta copia remota puede aprovechar la existencia de un sistema distribuido de ficheros como NFS en el cluster.

- Las rutas y los nombres de los diferentes ejecutables de los procesos que generará el trabajo.

- Los posibles argumentos de estos ejecutables.

- Donde situar la información que pueden necesitar ciertos procesos y que se les introduce a través de la entrada estándar, la cual normalmente está asociada al teclado. Habitualmente esta información es situada en un fichero especial (con todas las posibles entradas del teclado) y es copiado a las máquinas remotas para que sea accesible al proceso que lo necesite (a través de mecanismos como la redirección de la entrada estándar desde un fichero).

- El nombre y la ruta, normalmente en la máquina local del usuario, de los ficheros donde situar la información de salida o de error producidos durante la ejecución del trabajo.

- Información de gestión, como el rango de máquinas del cluster donde ejecutar los trabajos, la fecha u horario cuando ha de comenzar un trabajo o el nombre que identificará a dicho trabajo.

En los siguientes subapartados se explicarán las características más relevantes de los ficheros de descripción de trabajos de Condor y SGE.

2.2.1. Ficheros de descripción de trabajos de Condor

Condor utiliza un formato propio de ficheros de descripción de trabajos, el cual esta formado por un conjunto de comandos cuyo formato es *Identificador = Valores*. El campo *Identificador* informa al gestor de colas del tipo de acción que debe realizar y el campo *Valores* define el comportamiento de esta acción. Por ejemplo, para definir el ejecutable principal del trabajo (cuyo nombre es *user_exec*) se utilizaría el siguiente comando: *Executable = user_exec*.
Para poder describir la mayoría de escenarios de ejecución que puede encontrarse, el gestor de colas Condor ofrece de una gran variedad de comandos [18]. A través de ellos se pueden definir las siguientes informaciones importantes:

- Definición del ejecutable del trabajo: Con el comando *Executable* se informa de la dirección y el nombre del ejecutable del proceso principal del trabajo y con el comando *Arguments* de sus posibles argumentos (separados por espacios en blanco). El gestor de colas Condor copia automáticamente el ejecutable de la máquina local desde donde se somete el trabajo, al directorio de trabajo especial de las maquinas del cluster que Condor utiliza para ejecutar los trabajos en ellas (en sistemas Linux suele ser /home/condor). Para evitar que se realice esta copia automáticamente, se puede utilizar el comando *Transfer_executable* con el valor NO.

- Declaración de variables de entorno: El comando *Environment* contiene las posibles variables de entorno.

- Ficheros especiales con información de salida de los trabajos: Los Comandos *Output*, *Error* y *Log*, informan de la dirección y el nombre (en la máquina local del usuario) de los ficheros donde situar, respectivamente, la información producida por la salida estándar de los diferentes procesos generados por el trabajo, de los

errores durante su ejecución y de la gestión de Condor durante ejecución del trabajo (fecha de envío del trabajo, como ha finalizado, maquinas donde se ha ejecutado, etc).

- Fichero con la entrada estándar: El comando *Input* contiene el fichero donde está situada la información que se ha de suministrar por la entrada estándar al proceso principal del trabajo (en sistemas Unix/Linux es la entrada *stdin* relacionada con teclado).

- Definición de los posibles entornos de ejecución: El comando *Universe* permite especificar diferentes entornos de ejecución para los trabajos. Estos entornos de ejecución o universos pueden ser: *vanilla, standard, scheduler, local, grid, parallel, java y vm*. Cada uno de ellos ofrece sus propias características, por ejemplo, el universo *standard* permite que en el trabajo se puedan instalar puntos de chequeo y realizar llamadas remotas al sistema (enlazando el ejecutable del trabajo con el programa Condor_compile y las librerías de Condor). El *universo vanilla* (que es el por defecto) permite ejecutar los trabajos que no requieran o necesiten utilizar el *universo standard* (por ese motivo también se les denominan "trabajos normales") y el *universo parallel* está preparado para la ejecución de trabajos distribuidos como los que utilicen el protocolo MPI.

- Control de la transferencia de ficheros: Condor ofrece un conjunto de comandos para gestionar el cuando y como se han de transferir los ficheros entre la máquina que envía el trabajo y las del cluster, estos comandos son:

  - Comando *Should_transfer_files*: Informa si se deben o no transferir los ficheros a/o desde las máquinas remotas. Si es afirmativo (valor YES) se realizan las transferencias a/desde todas las máquinas del cluster, si es negativo (valor NO) , no se realiza la transferencia ya que se presupone la existencia de un sistema distribuido de ficheros en el cluster (tipo NFS o AFS, configurado en Condor previamente). Existe un valor especial de este comando, denominado IF_NEEDED, el cual indica que se realice o no la transferencia a/desde las máquinas del cluster dependiendo si existe o no uno de estos sistemas de archivos compartido en el cluster.

- Comando *When_to_transfer_output*: Informa del momento en el que se han de transferir, a la máquina que ha enviado el trabajo, los ficheros de salida de este. Si el valor es ON_EXIT, entonces esta transferencia es realizada al finalizar el trabajo, si es ON_EXIT_OR_EVICT, entonces es realizada cuando termina o si por algún motivo, el trabajo es desalojado (deja de estar en ejecución).

- Comando *Transfer_input_files*: Informa de la lista de ficheros (separados por comas) que se han de transmitir de la máquina que ha solicitado el trabajo a las remotas del cluster.

- Comando *Transfer_output_files*: Informa de los ficheros a transmitir (también separados por comas) de la o las máquinas remotas a la local (los ficheros de log, salida estándar y error son transmitidos automáticamente).

- Administración del trabajo: Los dos comandos que se explicarán a continuación *Requirements* y *Rank* son, con diferencia, los más complejos y a la vez los proporcionan más posibilidades dentro del repertorio que ofrece Condor. Por este motivo se les dedicará una explicación más detallada en este punto.
  Con el comando *Requirements* se informa la gestor de colas Condor de los requerimientos que han de cumplir las máquinas del cluster para que puedan ejecutar el trabajo. Estos requerimientos son expresados como conjunto de expresiones condicionales (A $>$, $<$, $==$, $\neq$, $\geq$ o $\leq$ B) unidas por el operador lógico && (And). Si al evaluar todas las expresiones sobre una máquina concreta dan el valor de *cierto (o True)*, entonces se dice que dicha maquina cumple con los requerimientos. Por ejemplo si el valor de los requerimientos es: *Memory* $\geq$ *64 && OpSys == "LINUX"*, todas las máquinas del cluster con mas de 64 Mbytes y con Sistema Operativo Linux cumplirían con estos requerimientos.
  Con el comando *Rank* se informa del rango (o conjunto) de máquinas a escoger de entre las que cumplen los requerimientos del trabajo, pudiendo expresar de de esta manera preferencias. Para definir este rango de máquinas se utilizan un conjunto expresiones en punto flotante, unidas por el operador || (Or lógico). Estas expresiones, al ser evaluadas en cada maquina que cumple los requerimientos del

trabajo, producen unos valores en punto flotante, siendo las maquinas que obtengan los valores del rango más grandes las seleccionadas. Por ejemplo, si entre las máquinas que cumplen los requerimientos del ejemplo del comando *Requeriments*, se prefiere que el trabajo se ejecute en las siguientes: *cluster_machine_one* y *cluster_machine_two*, entonces la expresión (o valor) que se debe situar en el comando Rank es: *(machine == "cluster_machine_one") || (machine == "cluster_machine_two")*. La evaluación de esta expresión funciona de la siguiente manera: Del total de máquinas que cumplen con los requerimientos de tener una memoria principal mayor de 64 MBytes y que su Sistema Operativo sea Linux, las que coincidan con las situadas en la expresión del comando *Rank* reciben una puntuación de *1.0* y el resto de *0.0* (las expresiones condicionales se evalúan: *1.0 si son verdaderas, 0.0 en caso contrario*). Por lo tanto al recibir una puntuación más alta, estas son las máquinas escogidas. Otros comandos de administración son el comando *Queue* que sitúa una o varias copias del trabajo en la cola o el comando *Priority* que permite definir la prioridad de un trabajo, con lo cual los trabajos del mismo usuario con prioridad más alta, se ejecutarán antes.

A continuación se muestra un sencillo ejemplo (FDT 2.1) que ilustra la edición de un fichero de descripción de trabajos para Condor.

FDT 2.1 Fichero de descripción de trabajos para Condor

```
1: Executable = user_exec
2: Universe = vanilla
3: Input = keyboard.data
4: Output = user_exec.out
5: Error = user_exec.error
6: Log = user_exec.log
7: Requirements = Memory ≥ 32 && OpSys == "LINUX"&&Arch =="INTEL"
8: Rank = (machine == "cluster_machine_one") ||
   (machine == "cluster_machine_two") ||
   (machine == "cluster_machine_tree")
9: Queue
```

En este ejemplo se puede observar como se declaran algunos de los comandos explicados en el punto anterior.

- En la linea 1 se declara el nombre ejecutable del proceso principal del trabajo.

- En la línea 2 se informa que el entorno de ejecución será del tipo *vanilla*
- En la línea 3 se declara el nombre del archivo que contiene la entrada estándar (teclado) para el proceso principal.
- En las líneas 4, 5 y 6 se informa del nombre los ficheros para la salida estándar, el de los posibles errores y el de la información de la gestión del trabajo.
- El la línea 7 se informa de los requerimientos ha de cumplir la máquina del cluster que sea escogida para ejecutar el trabajo. En este ejemplo son: Que tenga mas de *32 MBytes de memoria principal*, que su *Sistema operativo sea Linux* y que su *arquitectura este basada en la de Intel (IA-32, de 32 bits).*
- En la línea 8 se informa de las máquinas, que cumplen con los requerimientos anteriores, donde se prefiere que se ejecute el trabajo, estas son: *cluster_machine_one, cluster_machine_two y cluster_machine_tree.*

Para enviarle un trabajo, el gestor de colas Condor ofrece el comando de línea *condor_submit* [18] (se ejecuta desde cualquier consola de comandos), cuyo argumento principal es el fichero de descripción de trabajos que se quiere enviar. Además, Condor también ofrece un conjunto de comandos [19] para gestionar e informar sobre los trabajos que se van a ejecutar o ya se están ejecutando, los más importantes son:

- *condor_status*: Informa del estado de los recursos del cluster que gestiona Condor. Entre la información que devuelve se puede encontrar: El nombre de las maquinas, su arquitectura, su Sistema Operativo, su actividad (si están o no ocupadas ejecutando algún trabajo) o la cantidad de memoria que poseen.
- *condor_q*: Informa del estado actual de los trabajos enviados a Condor. A través de este comando se puede averiguar si un trabajo esta en ejecución, su identificador, su propietario o la hora en que fue enviado a su cola. Con el argumento *-analyze*, este comando ofrece la información de los posibles motivos por los cuales un trabajo no se ha podido ejecutar, normalmente porque no se han encontrado las máquinas que cumplan con los requerimientos del trabajo.
- *condor_rm*: Elimina un trabajo de la cola de ejecución en cualquier momento, incluso si ya se está ejecutando.

2.2.2. Ficheros de descripción de trabajos de SGE

El gestor de colas SGE, utiliza el formato de los ficheros *shell-scrips* para que los usuarios (o aplicaciones) puedan describir sus trabajos. Este formato tiene la ventaja, comparado con el de Condor, que es familiar para muchos usuarios, sobretodo los que trabajan con ordenadores cuyo Sistema Operativo es Unix/Linux (donde se hace un uso extensivo de este tipo de ficheros). En estos ficheros shell-script de descripción de trabajos, SGE permite que se sitúen unas opciones o directivas propias de este gestor de colas, cuyo principal objetivo es que los usuarios puedan suministrarle cierta información de gestión que le permita realizar una ejecución correcta de los trabajos. Para situarlas en los ficheros shell-script se utiliza el siguiente formato: *#$ opción*, el cual coincide con el de los comentarios de los shell-scripts (comienzan con el carater #). Esto es hecho de esta manera para evitar que estas opciones especiales de SGE afecten a la ejecución del fichero shell-scrip por parte de un interprete de comandos o shell externo a SGE. Las directivas especiales más comunes que ofrece SGE son [20]:

- *-e*: Define la dirección (puede ser en una maquina diferente de donde el usuario a enviado el trabajo) y el nombre del fichero que contiene la salida de los posibles errores sucedidos durante la ejecución del trabajo.
- *-i*: Define la dirección y el nombre del fichero que contiene la entrada estándar que necesita algún proceso definido en el trabajo.
- *-N*: Indica el nombre que identificará el trabajo.
- *-o*: Define la dirección y el nombre del fichero que contiene la salida estándar de los procesos ejecutados por el trabajo.
- *-p*: Indica la prioridad que se le asocia al trabajo.
- *-pe*: Define el tipo de entorno de paralelo que se ha de utilizar para la correcta ejecución del trabajo. Estos entornos paralelos definen los entornos de ejecución que necesitan ciertas aplicaciones distribuidas, como las basadas en MPI o PVM, para su correcto funcionamiento y ejecución.
- *-S*: Indica la dirección del interprete de comandos (también conocido como shell) que interpretará el fichero shell-script que define el trabajo.

- *-v*: Define las variables de entorno que serán exportadas al contexto de ejecución del trabajo. con la opción *-V* se especifica que todas la variables de entorno activas en el contexto desde donde se realiza el envío del trabajo (normalmente un terminal o consola), también estén activas en el contexto de ejecución remoto (situado en la máquina del cluster) del trabajo.

A continuación se muestra un ejemplo (FDT 2.2) de un fichero de descripción de trabajos (shell-script para bourne shell) para SGE:

FDT 2.2 Fichero de descripción de trabajos para SGE

```
1: #!/bin/sh
2: #$ -N user_exec
3: #$ -o user_exec_out
4: #$ -e user_exec_error
5: scp user@user_localhost:/home/user/user_exec user_exec
6: chmod u+x user_exec
7: ./user_exec
8: rm user_exec
```

En este ejemplo se puede observar los siguientes puntos interesantes:

- En la línea 2 se utiliza la opción especial de SGE que le indica cual es el nombre del trabajo (user_exec).

- En las líneas 3 y 4 se utilizan las opciones especiales de SGE para indicarle el nombre de los ficheros con la salida estándar (-o) y de los posibles errores (-e) producidos durante la ejecución del trabajo.

- El resto de líneas de este fichero (shell-script) de descripción de trabajos contienen los comandos de shell que realizan: La copia remota del fichero ejecutable del usuario (línea 5) a un directorio compartido, */home/user/user_exec*, de las máquinas del cluster (accesible por todas sus máquinas a través de un sistema de ficheros distribuido). El siguiente comando (línea 6) da permisos de ejecución (x en Unix/Linux) a este fichero ejecutable para que pueda ser ejecutado a continuación (línea 7). El último comando shell (línea 8) elimina, del directorio de la máquina remota, el fichero ejecutable copiado remotamente por el comando de la línea 5.

Como en el caso del gestor de Condor, SGE también ofrece un conjunto de comandos en línea [21] para poder operar y enviarle trabajos, los más importantes son:

- *qsub*: Envía un trabajo al gestor de colas SGE. Uno de sus argumentos más importantes es el nombre del fichero de descripción de trabajos (shell-script) que define el trabajo. Este comando permite que las opciones o directivas especiales de gestión para SGE (explicadas anteriormente), se le suministren directamente como argumentos en lugar de estar situadas en el fichero de descripción de trabajos. Por ejemplo, para definir el nombre del trabajo se podría utilizar: *qsub -N job_name job_script.sh*. Hay que destacar que está opción de pasar las directivas como argumentos no es la recomendada, ya que si su cantidad es elevada, la escritura del comando *qsub* puede ser muy extensa y tediosa, facilitando la posibilidad de introducir errores.

- *qdel*: Elimina un trabajo, tanto si está esperando ser ejecutado (en la cola) como si está en ejecución.

- *qstat*: Devuelve la información asociada a un trabajo o trabajos, así como el estado de las diferentes colas de SGE.

- *qsh, qlogin, qrsh*: Estos comandos en línea del gestor de colas SGE permiten definir trabajos que abren sesiones interactivas en las máquinas del cluster. Es importante destacar que, al contrario que *qsub*, estos comandos no soportan que se les pasen, como argumentos, todas las opciones de gestión que ofrece SGE. El comando *qsh* envía un trabajo a SGE que se encarga de ejecutar un terminal X (xterm) en la maquina del cluster, redireccionando la salida de este terminal al display del servidor X situado en la máquina que ha ejecutado este comando (implicando la necesidad de servidores X en ambas máquinas). El comando *qlogin* envía un trabajo que utiliza la herramienta de conexión remota *telnet* (basada en el protocolo del mismo nombre) para abrir una conexión remota en las máquinas del cluster. El comando *qrsh* realiza la mismo función que *qlogin* pero su trabajo usa la herramienta de conexión remota *rsh* para abrir la conexión remota. Debido a que estas dos herramientas de conexión remota son inseguras, SGE permite configurar *qlogin, qrsh* para que utilicen herramientas de conexión basadas en el protocolo seguro ssh.

Estos comandos, al abrir sesiones remotas (para la interactividad), pueden permitir a los usuarios (dependiendo de sus privilegios) acceder directamente a las máquinas del cluster que el gestor de colas SGE controla, pudiendo implicar problemas en su gestión. Este hecho provoca que en muchos de estos clusters controlados por SGE, no se permita la utilización de estos comandos *qsh, qlogin, qrsh* a los usuarios (o solo alguno de ellos, con privilegios muy restrictivos).

- *qmon*: Este comando pone en ejecución una interfaz gráfica de usuario, desde donde un usuario puede realizar la mayoría de tareas administrativas tanto con sus trabajos como para, dependiendo de su nivel privilegio, configurar ciertos componentes del gestor de colas. Algunas de estas tareas son:
  - Enviar trabajos.
  - Controlar y visualizar la evolución de sus trabajos.
  - Gestionar y obtener información sobre las diferentes colas del gestor de colas.
  - Configurar los usuarios.
  - Configurar la maquina local desde donde se ejecuta este comando y las máquinas del cluster.
  - Configurar los entornos de ejecución paralelos.

2.3. Arquitectura básica de los gestores de colas Condor y SGE

En este apartado se van a explicar las arquitecturas de los gestores de colas Condor y SGE, para mostrar que para realizar sus funciones, estas arquitecturas se basan en la utilización de dos componentes principales, los cuales también son utilizados por la arquitectura de las herramientas de monitorización (como se verá en el apartado de la arquitectura de dichas herramientas). Estos componentes son:

- *El Componente local*: Normalmente está situado en la máquina local del usuario, se encarga de recoger las peticiones del usuario, enviarlas al componente remoto y mostrar las respuestas o resultados obtenidos de estas peticiones.

- *Componente remoto*: Normalmente se ejecuta en las máquinas del cluster, recibe del *componente local* las peticiones que debe realizar, realiza las acciones necesarias para llevarlas a cavo y envía los resultados a dicho componente local.

En el caso de los gestores de colas, las peticiones que recibe el componente local y acciones que debe realizar el componente remoto están orientadas al envío y posterior ejecución de los trabajos de los usuarios. En este apartado también se mostrará que la arquitectura de los gestores de colas define un tercer componente importante, el *gestor central*, el cual está dedicado a controlar y gestionar a los otros dos componentes, así como los recursos del cluster (este componente no suele ser visible a los usuarios).

2.3.1. Arquitectura del gestor de colas Condor

Dependiendo de las funciones de máquinas que gestiona, la arquitectura del gestor de colas Condor clasifica las máquinas en diversos tipos [22], los cuales son:

- **Gestor central (Central Manager):** Solo puede haber un Gestor central en el cluster que controla Condor. Este es el gestor global del sistema y su función es controlar el conjunto de servicios (envío de trabajos, colas, ejecución de trabajos, etc) que forman parte de Condor. Si el Gestor Central cae, el gestor de colas Condor deja de funcionar, por lo tanto este gestor siempre ha de estar en ejecución.

- **Ejecución de trabajos (Execute):** Son las máquinas dedicadas a ejecutar los trabajos enviados por los usuarios. Para escogerlas correctamente, hay que tener en cuenta los posibles recursos que van a necesitar los diversos trabajos que se van a ejecutar en el cluster y escoger las máquinas que los cumplan (si puede ser lo mas sobradamente posible). Por ejemplo, si una importante parte de los trabajos que se van a ejecutar en el cluster de Condor, implican un intensivo calculo en punto flotante y cantidad de memoria, entonces las máquinas con una o unas CPU's (con N cores) con unas prestaciones en MFLOPS altas, con una mayor cantidad de memoria y con una zona de intercambio o swap importante (en disco duro, por la posible paginación de la memoria virtual), serán las candidatas a ser

declaradas como Ejecutoras.

- **Envío de trabajos (Submit):** Estas son las máquinas desde donde el usuario puede interactuar con el gestor de colas Condor, esto es: Enviarle los trabajos y recibir información del estado de ejecución de los mismos.

Es importante destacar que una máquina puede pertenecer a más de un tipo. Por ejemplo, puede permitir enviar trabajos (pertenecer al tipo *Envío de trabajos*) y también ejecutarlos (pertenecer al tipo *Ejecución de trabajos*). Para que cada tipo de máquina pueda realizar sus funciones, la arquitectura de Condor define unos *procesos demonios* que se encargan de realizarlas y que serán ejecutados por estas máquinas. Los demonios más importantes son:

- *condor_master:* Se encarga de poner y mantener en ejecución el resto de demonios de Condor. En caso de que uno de los demonios que controla *condor_master* deje de funcionar y no pueda volver a ponerlo en ejecución (después de n intentos), envía un mail al administrador de la máquina para que realice este proceso manualmente. Tiene que haber un demonio *condor_master* en ejecución en cada maquina del cluster del gestor de colas Condor.

- *condor_startd:* Se encarga de gestionar la ejecución de los trabajos en la máquina donde se ejecuta. Para ello, controla si los recursos disponibles de la máquina permiten ejecutar algún trabajo, así como si es necesario cambiar el estado de ejecución de algún trabajo (suspender, acabar,etc) al cambiar la disponibilidad de estos recursos.

- *condor_schedd:* Gestiona las peticiones de envío de trabajos al gestor de colas Condor. Cuando se envía un trabajo a *condor_schedd:* (vía *condor_submit*), este es puesto en la *cola de trabajos* que este demonio gestiona. Los comandos de línea *condor_q* o *condor_rm* se comunican con este demonio para realizar sus funciones. Este demonio también se encarga de solicitar los recursos necesarios para poder ejecutar cada uno de los trabajos situados en su cola.

- *condor_collector:* Este demonio de encarga de recoger la información de todas las máquinas que gestiona Condor. Esta información contiene el estado de los demonios de Condor, los recursos de cada máquina y los trabajos enviados a cada demonio *condor_schedd:*. El comando *condor_status* se comunica con este demonio para obtener la información que muestra al usuario.

- *condor_negotiator:* Este demonio es el encargado de la toma de decisiones dentro de Condor. Cada *ciclo de negociación*, pide el estado de los recursos del sistema al demonio condor_collector y se comunica con los diferentes demonios *condor_schedd:*, para ver si los recursos que solicitan los trabajos situados en sus *colas de trabajos* ya están disponibles. También se encarga de gestionar la prioridad de los trabajos, para asignarles los recursos necesarios y ejecutarlos en el orden correcto.

En la figura 2.1 se muestra la arquitectura del gestor de colas Condor, en ella puede apreciarse la relación entre tipo de máquina y los demonios que ha de ejecutar. La máquina escogida para ser el gestor central de Condor, deberá tener en ejecución los demonios: *condor_master*, *condor_collector* y *condor_negotiator*. Las máquinas que solo han de ejecutar trabajos, deberán ejecutar los demonios *condor_master* y *condor_startd* y las máquinas que solo se han de utilizar para enviar trabajos, deberán ejecutar los demonios *condor_master* y *condor_schedd*.
Si algunas máquinas utilizan tanto para enviar como para ejecutar trabajos, entonces deberán ejecutar los demonios: *condor_master*, *condor_schedd* y *condor_startd*.

Una vez vista la arquitectura del gestor de colas Condor, se puede observar la existencia del *componente local*, situado en las máquinas con la funcionalidad de envío de trabajos y la de los *componentes remotos*, situados en las máquinas dedicadas a la ejecución de trabajos.

2.3.2. Arquitectura del gestor de colas SGE

La arquitectura del gestor de colas SGE [23] [24] comparte muchas características con la del gestor de colas Condor. Define una máquina *gestor central*, que se encarga de recoger los envíos de trabajos de los usuarios y de procesar todas las peticiones administrativas. También define unas máquinas para la *ejecución de trabajos*, los

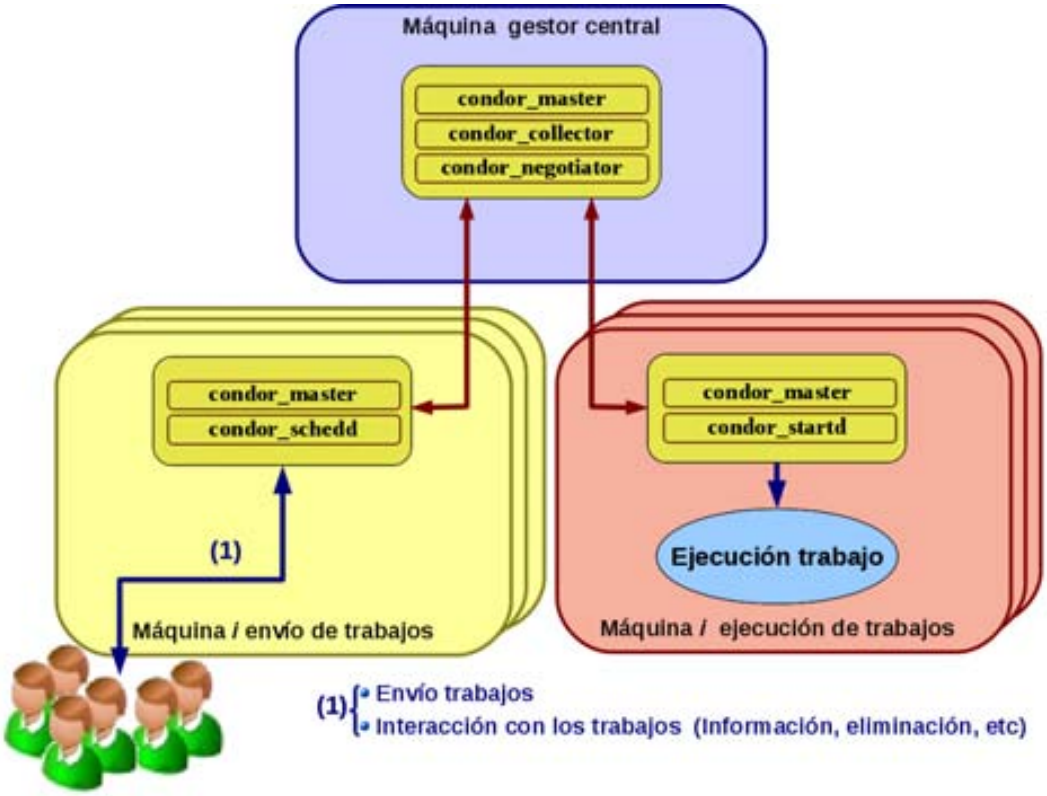


Figura 2.1: Arquitectura del gestor de colas Condor

cuales son enviados desde la máquina gestora central. Como en el caso de Condor, la arquitectura de SGE define un conjunto de procesos demonio que se encargarán de realizar las funciones designadas a cada tipo de maquina. Estos demonios son:

- *sge_qmaster:* Demonio central que se encarga de la gestión de todos los componentes del gestor de colas SGE, entre ellos las maquinas del cluster, las colas, la carga del sistema o los permisos de los usuarios. El demonio *sge_qmaster* procesa las decisiones de planificación que recibe del demonio *sge_schedd* y la información que recibe de los diferentes demonios *sge_execd*, para realizar las acciones necesarias destinadas a enviar los diferentes trabajos a ejecutar.

- *sge_schedd:* Este demonio se encarga de mantener actualizada la información del cluster que y de tomar las siguientes decisiones de planificación: A que cola se envía cada trabajos y como reordenar estos trabajos para mantener su prioridad, su tiempo de ejecución o sus recursos. Una vez tomadas estas decisiones de

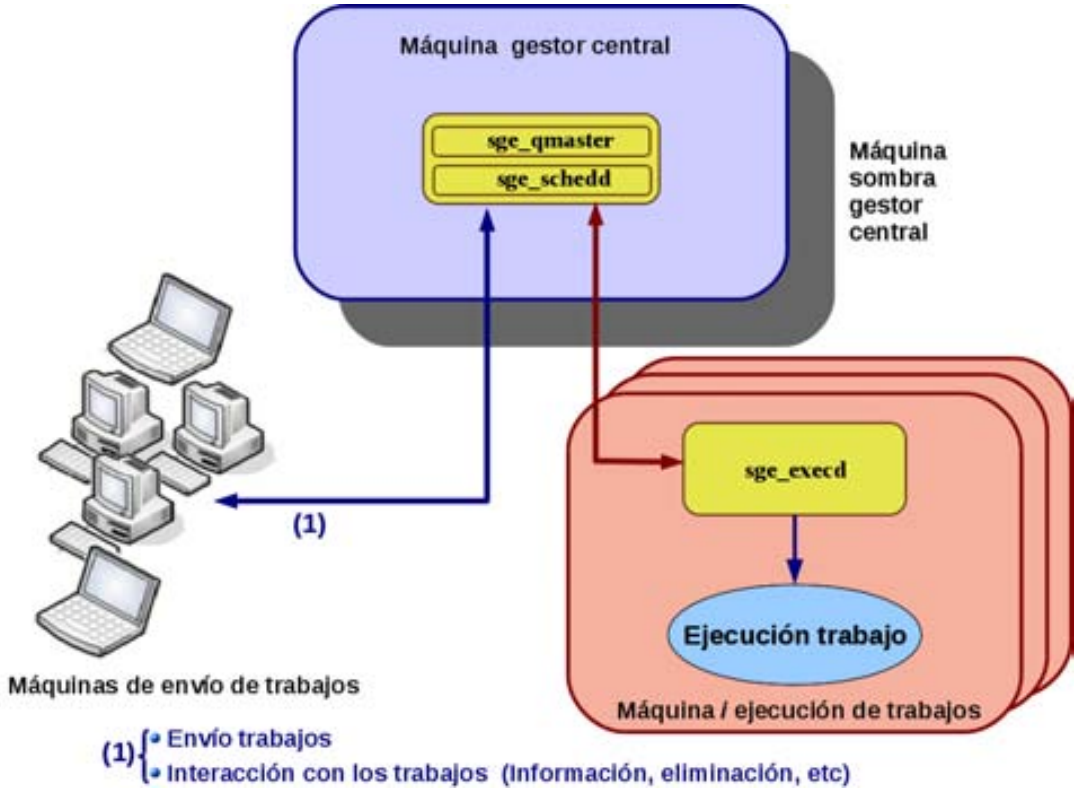


Figura 2.2: Arquitectura del gestor de colas SGE

planificación, se las envía al demonio *sge_qmaster*.

- *sge_execd:* Este demonio es el responsable de ejecutar los trabajos, enviando a cada intervalo de tiempo fijado *(t_execd_interval)*, información de los mismos al demonio *sge_qmaster*. Para definir la capacidad de ejecución de una máquina especifica, SGE define el concepto de *slot* y aunque no hay limite fijado por defecto, normalmente el número de slots (o unidades de ejecución) que ofrece una máquina es el del número de los *cores* de su CPU.

Si el demonio *sge_qmaster* no recibe información de uno de sus demonios *sge_execd* después de haber sucedido un número de intervalos de tiempo *(t_execd_interval)*, la maquina donde reside este demonio es marcada como *fuera de servicio* y sus recursos son eliminados de las listas de recursos disponibles para la ejecución de trabajos.
En la figura 2.2 se puede observar la relación entre los tipos de máquinas y sus demonios. La maquina *gestor central* debe ejecutar los demonios *sge_schedd* y *sge_qmaster*, mientras

que las *máquinas dedicadas a la ejecución de trabajos*, deben ejecutar el demonio *sge_execd*.
Como puede observarse, la arquitectura de SGE no utiliza un demonio que se encarga de recibir las peticiones de los usuarios (como utiliza Condor) y enviarlas al demonio correspondiente (en este caso *sge_qmaster*). Las máquinas desde donde se pueden enviar trabajos (a través de qsub o qmon) se comunican directamente con el demonio (*sge_qmaster*) de la máquina *gestor central*.
Debido a que si la maquina *gestor central* cae (y por lo tanto sus demonios dejan de ejecutarse) el sistema deja de funcionar, SGE define unas máquinas especiales, denominadas *sombra del gestor central (shadow master)*, las cuales pueden adoptar el papel de este gestor central, en caso de que el actual haya caído.
Para finalizar y como se ha observado también en el gestor de colas Condor, la arquitectura de SGE define un *componente local*, dividido entre las maquinas que envían trabajos y la máquina *gestor central* (donde se envían los trabajos) y unos *componentes remotos* dedicados a la ejecución de estos trabajos.

2.4. Arquitectura básica de las herramientas de monitorización

En este apartado, igual que se ha hecho con el de los gestores de colas, se explicarán las características más importantes de las arquitecturas de las herramientas de monitorización escogidas para la realización de este trabajo de tesis. Se va ha prestar especial atención a las posibilidades de *monitorización remota* que ofrecen para operar con procesos situados en una máquina diferente a la local. Esto hecho es importante, ya que es el tipo entorno de ejecución que se van a encontrar las herramientas de monitorización cuando se necesite utilizarlas en un cluster controlado por un gestor de colas (los procesos a monitorizar se ejecutarán en las máquinas de dicho cluster).

Como en el caso de los gestores de colas, en este apartado también se mostrará que para la monitorización remota, las arquitecturas de las herramientas de monitorización también utilizan dos componentes principales: El *local*, que se encarga de recibir las peticiones del usuario y mostrarle las respuestas a dichas peticiones y el *remoto* que

se encarga de ejecutar las peticiones que recibe del *componente local*. Las peticiones y acciones con las que trabajaran estos dos componentes están orientadas a monitorizar la aplicación del usuario. A parte de la monitorización remota, las herramientas de monitorización también permiten la manera más habitual (y conocida) de monitorizar procesos: *La monitorización local*, la cual consiste en que el usuario ejecute, en su máquina local, el o los procesos de la herramienta para que primero pongan en ejecución el proceso a estudiar y posteriormente, dependiendo de sus peticiones, realicen las acciones de monitorización necesarias sobre dicho proceso. Estas acciones de monitorización normalmente implican suspender el proceso, para poder examinar el estado de los registros de la CPU, de la memoria que utiliza o de sus archivos de Entrada/Salida. Si el proceso a monitorizar ya está en marcha, las herramientas estudiadas en este apartado también permiten que se le adjunten para monitorizarlo a partir de este momento (En La ejecución remota también).
Como puede observarse, en este proceso de monitorización que se realiza en la máquina local del usuario, un solo componente de la herramienta (el proceso o procesos que ejecuta localmente el usuario) se encarga de de recoger las peticiones del usuario (función del componente local) y realizar las acciones necesarias para llevarlas a cavo (función del componente remoto).

2.4.1. Arquitectura de la herramienta GDB

GDB es una herramienta de monitorización que se puede obtener y utilizar libremente (software libre). A parte de la monitorización local, esta herramienta también ofrece la monitorización remota, la cual está pensada para los casos en que se necesite monitorizar un programa situado en una máquina remota y no se puede utilizar directamente la herramienta GDB como se haría en el caso local. Esta circunstancia puede ser debida a que en esta máquina remota no se permite la interacción desde la maquina local del usuario (por ejemplo no se pueden abrir terminales remotos, como ssh) o no posee suficientes recursos (por ejemplo, poca memoria) para ejecutarla.
Para poder realizar la ejecución remota, la arquitectura de GDB se basa en la utilización de dos procesos: Un proceso, denominado *gdb*, que se ejecuta en la maquina local del usuario y un proceso demonio, llamado *gdbserver*, que se ejecuta en la máquina remota donde se va ha monitorizar el proceso del usuario. El proceso *gdb* recoge las peticiones

del usuario, a través de unos comandos propios que se introducen por medio de una consola de texto, las envía al demonio *gdbserver* y recoge sus repuestas para mostrarlas al usuario. Por su parte el demonio *gdbserver*, recibe las peticiones provenientes del proceso *gdb*, realiza las acciones de monitorización sobre el proceso a monitorizar y envía el resultado de estas acciones a dicho proceso *gdb*. Por lo tanto, en la arquitectura de GDB, el *demonio gdbserver* es su *componente remoto* y el proceso *gdb* su *componente local*.
Para ejecutar correctamente estos dos componentes de la herramienta GDB [25] se han de realizar los siguientes pasos:

1. *Ejecutar el demonio gdbserver en la máquina remota*: Para ello se utiliza el comando cuya sintaxis es la siguiente:

   gdbserver comunicación dir_nombre_ejecutable [argumentos_ejecutable]

   Donde:

   - *comunicación*: Argumento que define el tipo de conexión con el proceso *gdb*. Se permiten diversos tipos de conexión, entre las cuales destacan el *protocolo serie (RS-232)* o el TCP/IP (Internet). Para este ultimo protocolo (que es el más habitual) se utiliza el siguiente formato *host:puerto*, donde *host* define el nombre de la máquina donde se ejecuta el demonio *gdbserver* y *port* define un puerto de comunicaciones libre en dicha máquina (y que utilizará dicho demonio). Como el host siempre es la máquina donde se ejecuta el propio *gdbserver*, esta parte del argumento se puede obviar y solo indicar la parte del puerto de comunicaciones, en este caso, este argumento tendría el formato:" *:port*".
   - *dir_nombre_ejecutable*: Argumento que indica la dirección y el nombre del ejecutable del proceso a monitorizar en la máquina remota. Como puede observarse, este proceso tiene que estar situado en la máquina remota antes de que puede ser monitorizado por el demonio *gdbeserver*.
   - *argumentos_ejecutable*: Define los posibles argumentos del proceso a monitorizar.

GDB también permite que el demonio *gdbserver* se adjunte a un proceso que ya esta en ejecución en la máquina remota. Para ello se utiliza el siguiente comando: *gdbserver –attach comunicación pid*, donde *pid* es el identificador del proceso que se quiere monitorizar y que es dado por el Sistema Operativo. Por lo tanto, antes de poder realizar la operativa de adjuntarse a un proceso, se ha de obtener este *pid*. La manera mas habitual de hacerlo es conectarse a la maquina remota, mediante un terminal remoto y efectuar la o las llamadas al Sistema operativo que le proporcionen dicho *pid* del proceso. Por ejemplo, si el sistema operativo de la máquina remota es Linux, se puede ejecutar un terminal remoto en ella (con ssh) y ejecutar el comando *ps*. También se puede usar la llamada al sistema *pidof*, la cual devuelve el *pid* del nombre del proceso pasado como su argumento. Para este ultimo método *gdbserver* se ejecutaría de la siguiente forma: *gdbserver - -attach comunicación 'pidof nombre_proceso_monitorizar'*.

2. *Conectar el componente local de GDB al demonio gdbserver*: Para ello se ejecuta, en la maquina local, el proceso *gdb*, pasándole como argumento el ejecutable del proceso que se quiere monitorizar (igual que para una monitorización local). Una vez en la consola del proceso *gdb*, se ejecuta el siguiente comando propio de la herramienta: *target remote comunicación*. En el caso de una conexión TCP/IP, este argumento *comunicación* tendrá el siguiente formato *host_gdbserver:port_gdbserver*, donde *host_gdbserver* es el nombre de la máquina remota donde se ejecuta el demonio *gdbserver* y *port_gdbserver* es el puerto de comunicaciones que utiliza dicho demonio.
   Para finalizar una monitorización remota se pueden utilizar los siguientes comandos de la herramienta: *detach*, se desconecta del demonio *gdbserver* y finaliza la ejecución de este y *disconnect* realiza la misma función que *detach*, pero sin finalizar la ejecución del demonio *gdbserver*

A continuación se muestra en ejemplo de los pasos explicados anteriormente:
Se desea monitorizar el proceso *user_exec* situado en el directorio */home/user/* de la maquina *host1* con la herramienta GDB. Para ello, primero se escoge un puerto de comunicaciones libre en esta máquina, para este caso el 2000 y a continuación se ejecuta en ella el demonio *gdbserver*. Si se utiliza la herramienta de conexión remota *ssh* para ejecutarlo desde la máquina local del usuario, el comando que lo realizaría seria:

maquina_local_user > ssh user@host1 gdbserver :2000 /home/user/user_exec

Una vez el demonio *gdbeserver* está en ejecución en la máquina remota, el proceso *gdb* ya se puede conectar a el. Para ello, en la consola de este proceso se introducirá el siguiente comando:

consola_gdb> target remote host1:2000

Como ultimo apunte a esta herramienta se puede comentar que el ejecutable que se le pasa al demonio *gdbserver* no es necesario que contenga la tabla de símbolos (información de monitorización), mientras que el que se le pasa al proceso *gdb* si que la he de tener. Esto implica que el ejecutable que monitorizará gdbserver ocupará menos espacio (consumirá menos recursos) que el del GDB local.

2.4.2. Arquitectura de la herramienta Paradyn

Como GDB, Paradyn también es una herramienta de software libre (se puede obtener y utilizar libremente) que ofrece la posibilidad de la monitorización remota. Para ello su arquitectura define dos componentes: El local, formado por el proceso *paradyn* [1] y el remoto formado por el proceso demonio denominado *paradynd* [27]. El proceso *paradyn*, que se ejecuta en la maquina local del usuario, utiliza una interfaz gráfica (funciona en sistemas UNIX/Linux con X-Windows y MS-Windows) que, como en el resto de componentes locales vistos en este capitulo, recoge las peticiones de usuario, las envía al demonio *paradynd* y recoge las respuestas de este para mostrarlas al usuario. Por su lado, el demonio *paradynd* (aplicación tipo consola, no gráfica), que se ejecuta en la máquina remota, recoge las peticiones del proceso *paradyn*, realiza las acciones de monitorización sobre el proceso a monitorizar y envía las respuestas de vuelta al componente local de la herramienta. Para realizar sus acciones de monitorización, el demonio *paradynd* utiliza la librería *Dyninst* [28], la cual permite la inserción de código (el de monitorización en el caso de esta herramienta) en un programa, tanto si está en ejecución (como proceso) como si está situado en el disco.
Para la realización de la monitorización remota, esta herramienta ofrece dos métodos [26]:

[1] El proceso paradyn crea otros procesos (como el termWin, consola de información) que le ayudan a realizar sus funciones [26]. Pero como es el proceso principal del componente local de esta herramienta, siempre se hará referencia a el como dicho componente local

1. *Automático*: Este es el método recomendado por la herramienta para realizar la monitorización remota. Con este método, el componente local de Paradyn (el proceso *paradyn*) pone automáticamente en ejecución, en la máquina remota, el demonio *paradynd* para que este inicie el proceso de monitorización. Para ello se necesitan dos condiciones principales:

   - El usuario debe suministrar al proceso *paradyn* ciertos datos (a través de sus diferentes ventanas) que necesita para poner en ejecución el demonio *paradynd*. Estos datos son: La *máquina remota* donde este demonio va a monitorizar el proceso, *el usuario*, el *nombre del ejecutable* del proceso con sus posibles argumentos y un campo especial que define el *tipo de demonio paradynd* a utilizar según el programa que se va a monitorizar. Los tipos habituales de este demonio son: *defd* para la mayoría de aplicaciones UNIX/Linux, *winntd* para las aplicaciones MS-Windows, *mpid* para las aplicaciones distribuidas basadas en el protocolo MPI y para las aplicaciones multi-thread de Solaris o AIX, el tipo es *mt_d*. La monitorizan de aplicaciones distribuidas (como las MPI), implica la ejecución de diversos demonios *paradynd* en cada máquina donde se ejecuta un proceso de la aplicación distribuida a monitorizar.
   - El proceso *paradyn* utiliza una herramienta de conexión remota, normalmente ssh, para ejecutar el demonio *paradynd* en la maquina remota donde se desea monitorizar un proceso. Por lo tanto es importante destacar que esta máquina remota debe soportar este tipo de conexión remota.

2. *Manual*: Como en el caso de GDB, si el componente local de Paradyn no puede ejecutar automáticamente su demonio *paradynd*, esta herramienta ofrece la posibilidad de que un usuario o una aplicación puedan ejecutar, en una máquina remota, este demonio. Para ello, el proceso *paradyn* ofrece un argumento especial *-x fichero_información_paradynd*, el cual sitúa la información necesaria que se necesita para poder ejecutar el demonio *paradynd* en el fichero *fichero_información_paradynd*. El formato de este fichero es el siguiente:
   paradynd -z<tipo> -l<n> -m<nombre_maquina> -p0 -P<puerto_comunicaciones>
   Donde:

   - *-z<tipo>*: Indica el *tipo* del demonio *paradynd* (defd, winntd, mpid o mt_d).

- *-l<n>*: Define quien ejecuta el demonio: Si *n* vale 0, el demonio es ejecutado automáticamente por el proceso *paradyn*, si vale 1, el demonio es ejecutado manualmente por un usuario o una aplicación. Normalmente se deja el valor devuelto por el proceso *paradyn*.
- *-m<nombre_máquina>*: Indica el nombre de la máquina donde se ejecuta el proceso *paradyn*.
- *-p0 -P<puerto_comunicaciones>*: Informa de los puertos de conexión (TCP/IP) del componente local de Paradyn por donde se comunica con sus demonios *paradynd*.

Aparte de estos argumentos, el demonio paradynd también admite los siguientes argumentos importantes:

- *-runme <nombre_ejecutable>*: Indica la dirección y el nombre del ejecutable, en la máquina remota, del proceso a monitorizar.
- *-apid <pid>*: Como en la herramienta GDB, Paradyn también permite que el demonio *paradynd* se adjunte a un proceso que ya se está ejecutando en la máquina remota. El valor *<pid>* es el identificador del proceso que da el Sistema Operativo y como sucedía con el demonio gdbserver (de GDB) el usuario (o la aplicación) debe suministrar este pid.

Con los mismos datos (máquina y ejecutable) utilizados en el ejemplo de GDB, para ejecutar el demonio *paradynd*, primero se ejecuta, en la máquina local del usuario, el proceso *paradyn* con el siguiente comando:

maquina_local_user> paradyn -x fichero_paradynd

El fichero *fichero_paradynd* contiene la información necesaria para la ejecución manual del demonio *paradynd*, esta información tiene el siguiente formato:

paradynd -z<flavor> -l1 -p0 -P2000

Donde el valor *<flavor>* hay que sustituirlo por el tipo de demonio *paradynd* a utilizar, si la aplicación es estándar (serie y el Sistema operativo de la maquina es Unix/Linux) el valor de este argumento será *defd*. El resto de valores del archivo indican que el lanzamiento del demonio será manual (-l1) e informan los puertos de comunicación (-p0

-P2000) que utiliza el proceso *paradyn* para aceptar las conexiones de sus demonios. Una vez está en ejecución el proceso *paradyn*, se accede a la máquina remota host1 y se ejecuta el el siguiente comando :

host1> paradynd -zdefd -l1 -p0 -P2000 -runme /home/user/user_exec

Se puede observar que, a diferencia de la herramienta GDB, se ha supuesto que el usuario no puede acceder a la máquina host1 desde su máquina local (utilizando ssh) y si que puede acceder directamente (personalmente) a esta maquina para ejecutar en ella el demonio *paradynd*. Esto se ha hecho de esta manera, debido a que si se puede usar la herramienta de conexión remota *ssh* para acceder a la maquina remota, entonces la herramienta *Paradyn* pude usar su método automático (y recomendado) para monitorizar remotamente el proceso.

2.4.3. Arquitectura de la herramienta Totalview

La herramienta Totalview, a diferencia de la dos anteriores, es una aplicación comercial (aunque ofrece una versión de prueba gratuita durante un periodo de tiempo [29]) que también permite la monitorización remota. Para ello, su arquitectura define dos componentes: El local, formado por el proceso *totlaview* y el remoto, formado por el proceso demonio *tvdsvr* [30]. Las funciones de estos dos componentes son las mismas que las explicadas para los de las herramientas GDB y Paradyn. Para interactuar con el usuario, el proceso *totlaview* utiliza una interfaz gráfica, la cual posee versiones para sistemas basados en X-Windows (Unix/Linux), MS-Windows y Apple Mac OS. Como se ha visto en la herramienta Paradyn, para la monitorización remota la herramienta Totalview también ofrece dos métodos principales:

1. *Automático*: Este es el *método estándar de monitorización remota* y el recomendado por la herramienta. Este método se basa en que el proceso *totlaview* se encarga de poner en ejecución, en la máquina remota, al demonio *tvdsvr*, indicándole las acciones que ha de realizar para que se pueda realizar correctamente el proceso de monitorización. Para que el componente local de Totalview pueda realizar esta función, el usuario debe proporcionarle cierta información (a través de sus diferentes ventanas) como el nombre del ejecutable del proceso a monitorizar (y sus posibles argumentos) o la máquina remota donde este se va a ejecutar.

La herramienta Totalview, también permite al usuario poder definir o configurar ciertos parámetros respecto a la forma en que el proceso *totlaview* ha de poner en ejecución el demonio *tvdsvr*, como la herramienta de conexión remota que se va utilizar (rsh, ssh o remsh, esta último para máquinas HP) o el directorio donde se encuentra el demonio *tvdsvr* en la máquina remota.

2. *Manual*: Como en el caso de las herramientas explicadas en los puntos anteriores, si el proceso *totlaview* no puede ejecutar remotamente su demonio *tvdsvr*, el diseño de la arquitectura de esta herramienta permite que un usuario o una aplicación puedan ejecutar, en una máquina remota, este demonio. Para ello, se han de seguir los siguientes pasos:

   a) Ejecutar en la máquina remota el demonio *tvdsvr* con el comando mostrado a continuación, en el se muestran los argumentos más importantes de este demonio [31]:

   tvdsvr -server {*-search_port* | *-port puerto_comunicaciones*} *-set_pw password*

   Donde:

   - *-server*: Indica que el proceso *tvdsvr* será ejecutado en modo demonio y que, si no se indica lo contrario, aceptará peticiones por el puerto de conexión por defecto, el 4142.
   - *-search_port o -port*: En el caso de que no se pueda usar el puerto de conexión por defecto, con estos argumentos se puede definir el nuevo puerto de conexión del demonio *tvdsvr*. Con *-search_port* se obtiene un puerto de conexión automáticamente y con *-port* se le asigna un puerto escogido externamente (por un usuario o una aplicación).
   - *-set_pw nuevo_passw*: Por defecto y como medida de seguridad (para evitar intrusiones de otros usuarios), el demonio *tvdsvr* escoge aleatoriamente un password, cuyo formato es *número_hex_alto:número_hex_bajo* (2 números de 32 bits en formato hexadecimal) y por el cual preguntará el proceso *totlaview* cuando se conecte con este demonio. Para que el usuario conozca este password y se lo pueda introducir al componente local de esta herramienta cuando se lo solicite, el demonio *tvdsvr* lo muestra por su salida estándar a través del siguiente mensaje: *pw*

= *número_hex_alto:número_hex_bajo*. Para que este sistema funcione, el usuario ha de tener acceso a la salida estándar de los procesos que se ejecutan en la máquina remota.
Si se desea que el demonio *tvdsvr* no escoja aleatoriamente el password, se debe utilizar este argumento *-set_pw* con el nuevo password, *nuevo_passw* (formato *número_hex_alto:número_hex_bajo*), como su argumento. Con esto se informa a los dos componentes de la herramienta que *nuevo_passw* será el password que han de utilizar. Si el valor de *nuevo_passw* es *0:0*, entonces no se utilizará está política de seguridad y no se preguntará por ningún password, implicando posibles riesgos en la seguridad.

b) Una vez está en ejecución el demonio *tvdsvr* en la máquina remota, se puede ejecutar, en la máquina local del usuario, el proceso *totlaview*. Para ello se utiliza el siguiente comando (donde se muestran sus argumentos más importantes): *totalview direccion_nombre_ejecutable [-a argumentos] [-pid pid_proceso] -remote maquina_remota:puerto_comunicaciones* Donde:

- *direccion_nombre_ejecutable*: Indica la dirección y el nombre, en la maquina remota, del ejecutable del proceso a monitorizar.
- *-a argumentos*: Se informa de los posibles argumentos del proceso a monitorizar.
- *–pid pid_proceso*: En caso de que el demonio *tvdsvr* se haya de adjuntar a un proceso, informa de su identificador (*pid_proceso*) dado por el S.O. de la máquina remota. Es importante destacar que también hay que pasar la dirección y el nombre del ejecutable (en la máquina remota) del proceso al que se va a adjuntar.
- *-remote maquina_remota:puerto_comunicaciones*: Informa del nombre de la maquina remota *maquina_remota* donde se ejecuta el demonio *tvdsvr* y de su puerto de conexión *puerto_comunicaciones* por donde se comunicará con el proceso *totlaview* .

Aprovechando los datos utilizados en los ejemplos de las dos herramientas de monitorización anteriores, para ejecutar el demonio *tvdsvr* en la máquina remota, se utilizaría el siguiente comando:

host1> tvdsvr -server -port 2000 -set_pw 0:0

Donde el argumento *-port 2000* define el puerto de conexión (el 2000) que utilizará el demonio *tvdsvr* y el argumento -set_pw 0:0 que no se preguntará por ningún password (opción insegura).
Como en la herramienta *Paradynd*, se ha supuesto que el usuario no tiene acceso desde su máquina local a la máquina remota *host1* (pero si que puede acceder personalmente), ya que si este hecho sucediera, la herramienta *Totlaview* podría usar su método automático de monitorización remota. Una vez el demonio *tvdsvr* está en ejecución en la máquina remota, el proceso *totlaview* se puede ejecutar desde la maquina local del usuario y conectarse a el con el siguiente comando:

maquina_local_user> totlaview /home/user/user_exec -remote host1:2000

2.4.4. Esquema común de la arquitectura de las herramientas de monitorización

En la figura 2.3 se muestra un esquema de la arquitectura que comparten las herramientas de monitorización explicadas en este apartado. En este esquema se puede observar la utilización de un componente local, situado en la máquina local del usuario, para interactuar con dicho usuario y enviar sus peticiones al componente remoto, así como la utilización de este último para realizar las acciones de monitorización (en la máquina remota) necesarias que conduzcan a la resolución de las peticiones de este usuario.

2.5. Protocolo TDP (Tool Daemon Protocol)

Basándose en la arquitectura de dos componentes, el local y el remoto, de los gestores de colas y las herramientas de monitorización, el protocolo TDP ha sido la primera aproximación para solucionar su problema de la falta de interoperabilidad. Para ello, define una interfaz estándar que encapsule los servicios necesarios que permitan a los componentes de las herramientas de monitorización, interactuar con los de los gestores de colas y monitorizar aplicaciones en un cluster controlado por estos últimos. De la implementación de estos servicios se encargan un conjunto de primitivas (o funciones) que se pueden clasificar en los siguientes tres conjuntos:

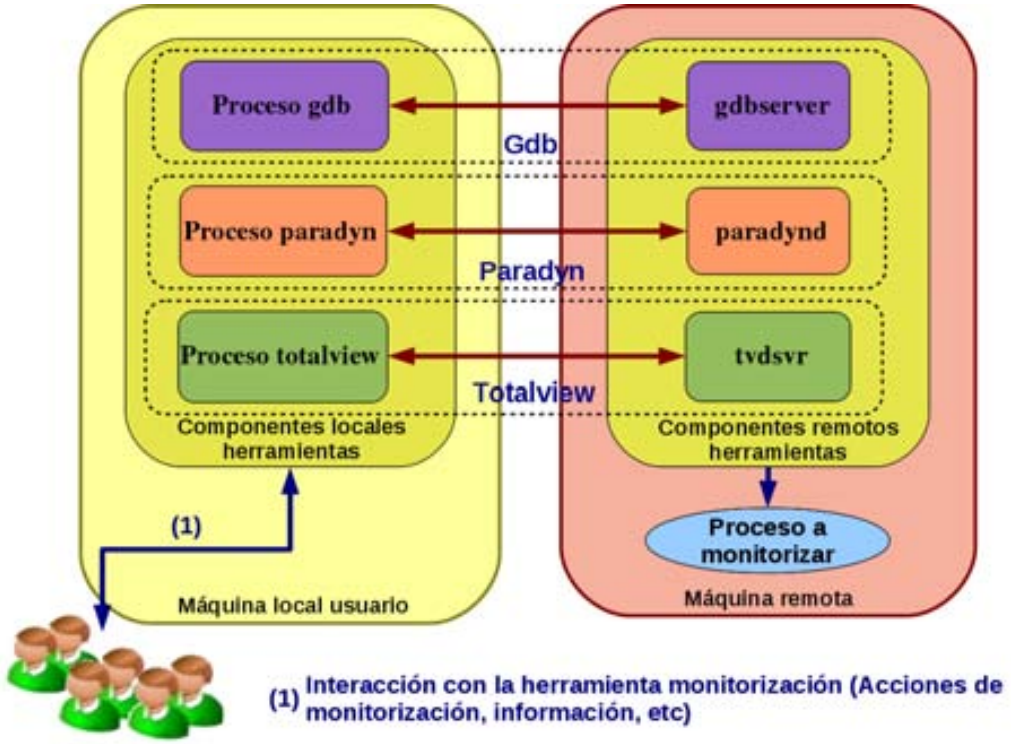


Figura 2.3: Esquema general arquitectura de las herramientas de monitorización GDB, Paradyn y Totalview

1) **Intercambio de información**: Realizan el intercambio de información que necesitan los diferentes componentes de los gestores de colas y herramientas de monitorización para su correcta interacción.

2) **Gestión de eventos**: Gestionan los sucesos que no son predecibles en el tiempo (por ejemplo, avisar de los posibles errores).

3) **Control de procesos**: Gestionan y controlan los procesos que se van a ejecutar en la máquina local del usuario (componente local de la herramienta) y en las máquinas del cluster (componentes remotos de la herramienta y de la aplicación a monitorizar).

En el capitulo 4 se explicará más detalladamente cada uno de estos grupos de primitivas, así como la principal problemática de implementar (en formato de librería) este protocolo. Es importante destacar que el entorno de trabajo `TDP-Shell` no es una implementación del protocolo TDP, sus diseños de la solución para el problema de la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización son

bastante diferentes. Lo que si que es cierto es que las ideas propuestas por el protocolo TDP han servido como punto de partida para del diseño e implementación del entorno de trabajo `TDP-Shell`.

2.6. Conclusiones

Los *ficheros de descripción de trabajos* describen a los gestores de colas el trabajo que han de ejecutar en el cluster que gestionan. Con ello se informa al gestor de colas del *entorno de ejecución* necesario (variables de entorno, ejecutables, copia remota de ficheros) para que pueda ejecutar, en las máquinas del cluster, las aplicaciones que se definen en dicho trabajo. Dependiendo del formato de estos ficheros de descripción de trabajos, los gestores de colas se pueden clasificar en dos grandes grupos. Los que utilizan el formato de *shell-scripts* como SGE o Torke y los que utilizan un *formato propio de fichero de descripción de trabajos* como Condor, el cual utiliza un fichero de descripción de trabajos basado en comandos. Debido a esta clasificación, se han escogido a *Condor y SGE* como representantes de los gestores de colas. Tanto Condor como SGE ofrecen un conjunto de comandos para poder enviarles los trabajos (*condor_submit*, en Condor y *qsub* en SGE), obtener información de ellos (*condor_q*, en Condor y *qstat* en SGE) y finalizar su ejecución (*condor_rm*, en Condor y *qdel* en SGE). Por su parte, SGE también ofrece una interfaz gráfica que permite realizar estas acciones sobre los trabajos, así como la configuración y gestión (dependiendo de los derechos de acceso) de ciertas partes del gestor de colas.
Para la realización de este trabajo de tesis se han escogido las herramientas de monitorización *Gdb, Totalview y Paradyn*, porque son unas de las herramientas mas utilizadas y dan soporte a una gran variedad de tipos de aplicaciones (monitorizan tanto aplicaciones serie como distribuidas). Las arquitecturas de los gestores de colas y herramientas de monitorización, en su modo de monitorización remota (el proceso no está en la misma máquina local del usuario), comparten una característica en común muy importante: Su arquitectura se basa en la utilización de dos componentes, los cuales son:

- *Componente local*: Normalmente está situado en la máquina local del usuario. Este componente se encarga de recoger las peticiones del usuario, comunicarlas

al *componente remoto* y mostrar las respuestas o resultados obtenidos de estas peticiones y devueltos por este *componente remoto*.

- *Componente remoto*: Normalmente se ejecuta en las máquinas del cluster, recibe del *componente local* las peticiones que debe realizar, realiza las acciones necesarias para llevarlas a cavo y envía los resultados a dicho componente local. En el caso de los gestores de colas, las peticiones y acciones que deberá realizar este componente estarán orientadas a la gestión de procesos y recursos de su máquina. Por su parte, en el caso de las herramientas de monitorización, las acciones que realizará este componente estarán orientadas al control de los procesos para poder realizar su monitorización

La arquitectura de los gestores de colas también utiliza un tercer tipo de componente, denominado *gestor central*, el cual esta dedicado a la gestión y control de los componentes del gestor de colas, entre ellos sus componentes locales y remotos. Para implementar la funcionalidad de estos componentes, los gestores de colas utilizan un conjunto de procesos demonio que se ejecutan en las máquinas del cluster dependiendo de la función que estas han de desempeñar. En Condor, estos demonios son los siguientes:

- Las máquinas que solo han de ejecutar trabajos, actúan son como componentes remotos, deben ejecutar los demonios *condor_master* y *condor_startd*.
- Las máquinas que solo se han de utilizar para interactuar con el gestor de colas, realizan la función de componente local, deben ejecutar los demonios *condor_master* y *condor_schedd*.
- La máquina escogida para ser el gestor central de Condor, debe tener en ejecución los demonios *condor_master*, *condor_collector* y *condor_negotiator*.

Por su parte el gestor de colas SGE utiliza los siguientes demonios:

- Las máquinas con la funcionalidad de componentes remotos (ejecutan trabajos), deben ejecutar el demonio *sge_execd*.
- Las máquinas que realizan la función de componente local, se conectan directamente a la máquina que actúa como gestor central, sin necesidad de utilizar ningún demonio.

- La máquina escogida para ser el gestor central de SGE debe tener en ejecución los demonios *sge_schedd* y *sge_qmaster*.

En el caso de las herramientas de monitorización, la utilización de los componentes local y remoto es todavía más evidente que en los gestores de colas. Para realizar sus actividades de monitorización remota, la arquitectura de estas herramientas utiliza:

- Un proceso (o conjunto de ellos), que se ejecuta en la máquina local del usuario y realiza la función de interfaz de usuario o componente local de la herramienta. En la herramienta GDB es el proceso *gdb*, en Paradyn es el proceso *paradyn* y en Totalview, el proceso *totalview*.

- Un proceso demonio, el cual se ejecuta en la maquina remota donde se quiere monitorizar y actúa como componente remoto de la herramienta. Para la herramienta GDB, su demonio es *gdbserver*, para Paradyn es *paradynd* y para Totalview *tvdsrv*.

El protocolo TDP ha sido la primera aproximación para implementar una solución al problema de la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización.Este protocolo propone una interfaz común a ambos que encapsule los servicios necesarios que permitan a estos gestores de colas y herramientas de monitorización interoperar adecuadamente.

3

Problema de Interoperabilidad

En el capitulo de introducción se definió el problema objeto de estudio y solución de este trabajo, el cual consiste en *la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización*. Este problema provoca que un usuario no pueda utilizar, en su forma estándar, estas herramientas de monitorización en clusters controlados por uno de estos gestores de colas. Por otro lado, en el capitulo 2 se han mostrado las arquitecturas de las herramientas de monitorización y de los gestores de colas. Como se ha visto, el diseño de estas arquitecturas se basa en la existencia de dos componentes [1]: El *componente local*, normalmente situado en la maquina local del usuario y los *componentes remotos*, situados en las máquinas del cluster (las cuales denominaremos *máquinas remotas*) donde se van a ejecutar y monitorizar (en el caso de las herramientas) los diferentes procesos de la aplicación del usuarios. También se han definido las funciones principales de cada uno de estos componentes. Las del componente local son interactuar con el usuario, aceptando sus peticiones para enviarlas a los componentes remotos y recogiendo los resultados obtenidos de estas peticiones para mostrarlos al usuario. Por su parte, las funciones de los componentes remotos son encargarse de la ejecución de las peticiones del usuario en las máquinas remotas y de comunicar al componente local los resultados obtenidos. En las herramientas de monitorización, esta peticiones están orientadas al control y a la obtención de información del estado de los procesos (p.e.

[1] En los gestores de colas se ha explicado un tercer componente, *el gestor central*, cuya función es controlar los recursos cluster y a los componentes del propio gestor. A partir de este capitulo se mostrará al arquitectura de los gestores de colas compuesta por los componentes locales y remotos, pero aunque no se muestre, siempre se considerará la existencia de este tercer componente

parar un proceso para examinar los registros de la CPU). Por otro lado, las peticiones para los gestores de colas estarían orientadas a informar de los entornos de ejecución de las aplicaciones (p.e. nombre del ejecutable, ficheros de estrada/salida, variables de entorno, etc) que permitan a estos gestores ejecutarlas correctamente en el cluster que controlan.

Partiendo de la definición del problema y de la arquitectura de los gestores de colas y las herramientas de monitorización, en este capítulo se explicarán detalladamente las causas que generan el problema de interoperabilidad y el coste que representaría para el usuario implementar una solución personalmente.

3.1. Causas del problema de Interoperabilidad

En los siguientes puntos se explican las principales causas que provocan el problema de la la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización, los cuales se observan en la figura 3.1.

a) Las herramientas de monitorización utilizan, como metodología estándar, herramientas de conexión remota, como puede ser ssh, para acceder a las maquinas del cluster y ejecutar en ellas sus componentes remotos. Por otro lado y para evitar los posibles problemas que aparecerían al compartir la gestión de los recursos del cluster, por ejemplo entre una herramienta de conexión remota y el gestor de colas, este último suele ser su única vía de acceso permitida. Este hecho provoca que las herramientas de monitorización no puedan utilizar las herramientas de conexión remota para ejecutar sus componentes remotos en un cluster controlado por un gestor de colas, impidiendo al usuario utilizar estas herramientas en su forma habitual.

b) Para poder solucionar el problema explicado en el punto anterior, las herramientas de monitorización deberían poder solicitar al gestor de colas que gestione la ejecución de sus componentes remotos en las máquinas del cluster. Por su parte, el gestor de colas debería poder informar a la herramienta de monitorización de los posibles errores o cambios de estado sufridos durante la ejecución de sus componentes remotos, así como de suministrar cierta información de los procesos que el ha creado y que podría ser necesaria para que la herramienta de monitorización pueda realizar algunas de sus funciones, como el caso de adjuntarse a alguno de estos procesos en ejecución (necesita

el identificador de proceso dado por el sistema operativo de la máquina). Para poder realizar todas las acciones expresadas anteriormente, es necesaria la existencia de mecanismos de intercambio de información entre los diferentes componentes, tanto locales como remotos, de las herramientas y de los gestores de colas. Pero en la actualidad, la mayoría de estos gestores de colas y herramientas de monitorización no ofrecen dichos mecanismos de comunicación y tampoco aprovechan los limitados mecanismos de intercambio de información que ofrecen y que podrían ayudar a solucionar el problema de su falta de interoperabilidad. Por ejemplo, las herramientas de monitorización no suelen ofrecer la posibilidad de informar directamente a los gestores de colas (normalmente a través de ficheros de descripción de trabajos), del entorno de ejecución que necesitan sus componentes remotos para ejecutarlos correctamente. Por su parte, los gestores de colas solo suelen ofrecer la posibilidad de situar en un fichero de texto, los resultados o los errores de la ejecución de sus trabajos. Este hecho no es aprovechado por las herramientas de monitorización ya que no están preparadas para extraer la información de estos ficheros, con el problema añadido de que al generarse al final de la ejecución del trabajo, no permiten el intercambio de información interactivamente durante la ejecución del mismo (muy necesaria en un proceso de monitorización).

c) Una acción importante que realizan las herramientas de monitorización es la sincronización temporal de la ejecución de sus componentes locales y remotos. Esta sincronización significa que para la correcta ejecución de uno de los componentes, es necesario que el otro componente le proporcione cierta información, implicando que este último ha de ejecutarse previamente al que necesita dicha información. Por ejemplo, en el caso de Paradyn, sus componentes remotos necesitan conocer los puertos de comunicación de su componente local, para poder comunicarse con el. Por lo tanto, primero es necesario ejecutar el componente local de esta herramienta (normalmente en la maquina local del usuario) para que cree los puertos de comunicación y a continuación, crear los componentes remotos de esta herramienta (en las máquinas del cluster), para que tengan acceso a los puertos de comunicación de su componente local. En un cluster controlado por un gestor de colas, la falta de mecanismos de intercambio de información entre este gestor y las herramientas de monitorización, conlleva la dificultad de realizar la sincronización temporal de la

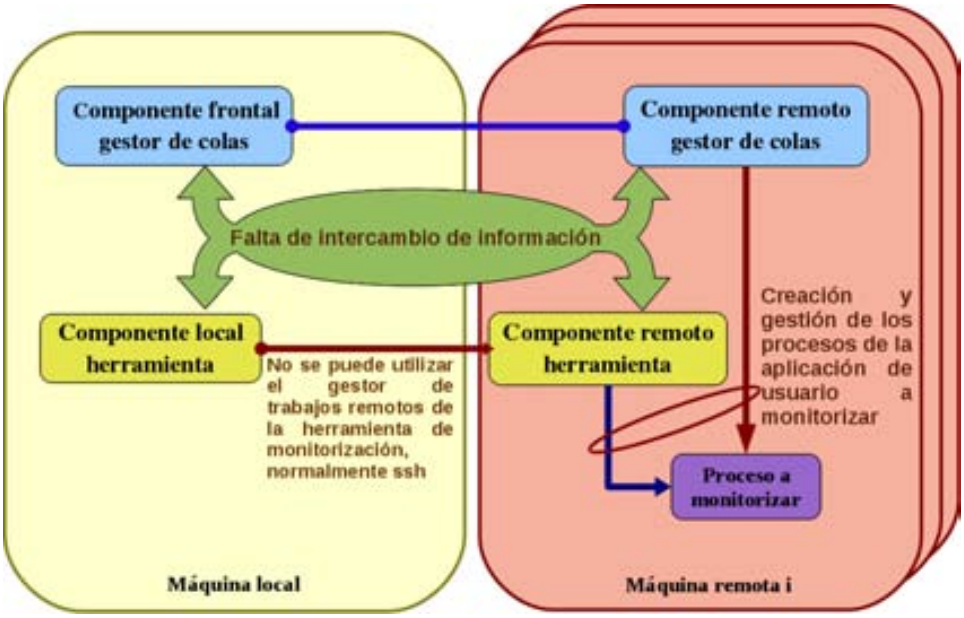


Figura 3.1: Problemas de la falta de interoperabilidad entre los gestores de colas y herramientas de monitorización.

ejecución de los componentes de esta herramienta. Esto es debido a que sucede uno de los siguientes casos:

- El componente local de la herramienta, no puede suministrar directamente al gestor de colas la información que necesita para ejecutar los componentes remotos de esta herramienta. Al desconocer esta información, el gestor de colas no sabe como y cuando puede los puede ejecutar.
- El gestor de colas, después de ejecutar los componentes remotos de la herramienta, no puede informar del momento a partir del cual ya se puede ejecutar el componente local de dicha herramienta.

En el ejemplo de Paradyn, al no estar preparado su componente local para intercambiar automáticamente información con el gestor de colas, no le puede informar de los puertos de comunicación para sus componentes remotos, imposibilitando que dicho gestor de colas sepa el momento a partir del cual puede ejecutar estos componentes remotos de Paradyn en su cluster.

d) Un último problema importante aparece cuando se intenta utilizar una herramienta

de monitorización en un cluster controlado por un gestor de colas, es quién de ellos crea y gestiona los procesos que se van a monitorizar. En su funcionamiento estándar, tanto el gestor de colas como la herramienta de monitorización controlan los procesos que ellos crean. Por ejemplo, la herramienta puede pausar los procesos que monitoriza para poder obtener información de estos (el estado de la memoria o de los registros de la CPU). Debido a esto, la gestión que realiza la herramienta sobre los procesos que monitoriza, puede entrar en conflicto con la del gestor de colas y por el otro lado, cualquier cambio de estado, producido por el gestor de colas sobre los procesos que la herramienta está monitorizando, puede afectarla negativamente. Otro problema asociado con el control de procesos se produce cuando la herramienta necesita solicitar cierta acción sobre un proceso que ha creado el gestor de colas y que este no puede realizar. Por ejemplo, la herramienta puede necesitar que el gestor de colas cree el proceso a monitorizar pausado (se carga el proceso en memoria, pero no empieza a ejecutarse) para poder situarle los datos y los códigos especiales de monitorización. El problema de esta petición es que la mayoría de gestores de colas (como Condor y SGE) no soportan esta creación pausada de procesos.

3.2. Estudio de las posibles soluciones al problema de interoperabilidad

Partiendo de los problemas los cuales generan la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización, enumerados en el apartado anterior, la solución a este problema debe contemplar los siguientes puntos:

a) Creación del entorno de ejecución remoto en las máquinas del cluster: Debido a que el control de acceso y ejecución en las maquinas del cluster está controlado por un gestor de colas, la solución al problema de interoperabilidad entre estos últimos y las herramientas de monitorización debe proporcionar mecanismos que permitan a los componentes de la herramienta (normalmente su componente local), informar a los componentes de los gestores de colas (normalmente sus componentes remotos), del entorno de ejecución (variables de entorno, ejecutables, ficheros de configuración) que les permita ejecutar correctamente, en las máquinas del cluster que gestionan, los componentes remotos de la herramienta de monitorización y en caso necesario, los

procesos de la aplicación de usuario que van a ser monitorizados.

b) Intercambio de información y sincronización en la ejecución: La solución al problema de la interoperabilidad entre herramientas de monitorización y gestores de colas, debe proporcionar mecanismos que se encarguen de la gestión del intercambio de la información entre sus componentes (locales y remotos) que permitan:

   - Realizar correctamente la ejecución sincronizada de los componentes locales y remotos de la herramienta.
   - Averiguar los cambios de estado producidos por sus acciones de control sobre los componentes remotos (realizadas por el gestor de colas) o los procesos que se monitorizan (realizadas por ambos).

c) Operativas especiales de las herramientas de monitorización: Ciertas operaciones (sobre los procesos a monitorizar) solicitadas por las herramientas de monitorización a los gestores de colas, no pueden ser realizadas directamente por estos últimos (por ejemplo, la herramienta solicita al gestor de colas la creación de un proceso pausado o que pause un proceso). Por ello, la solución al problema de interoperabilidad tratado en este trabajo de tesis, debe proporcionar los mecanismos necesarios que se encarguen de realizar estas operaciones especiales.

3.2.1. Estudio de los posibles diseños para la solución del problema de interoperabilidad

Para poder obtener una solución al problema de la interoperabilidad entre los gestores de colas y las herramientas de monitorización y que cubra todos los puntos mencionados anteriormente, se puede optar principalmente por dos diseños:

a) Integrar la solución en sus respectivos códigos ejecutables:

Este diseño de la solución (figura 3.2) se basa en desarrollar las estructuras de datos y operativas (funciones, objetos, etc) que solucionan el problema de interoperabilidad entre los gestores de colas y herramientas de monitorización para ser integradas dentro de sus respectivos códigos ejecutables. Una buena opción para implementar este diseño, pensando en la modularización y reutilización del código, es encapsular estas

estructuras de datos y operativas en una librería o conjunto de librerías especiales, las cuales se incorporaran (estática o dinámicamente) en los códigos ejecutables de los gestores de colas y las herramientas de monitorización. De esta forma, cada acción o intercambio de información que deban realizar sus componentes, se traducirá en un conjunto de llamadas a las funciones definidas en estas librerías especiales. Esta opción implica disponer de los códigos fuentes (mayoritariamente escritos C,C++ o java) de los gestores de colas y de las herramientas de monitorización, para poder situar en ellos las llamadas a las funciones de estas librerías especiales y generar sus nuevos códigos ejecutables. Estos códigos también contendrán el código ejecutable de las librerías, en caso que estas sean estáticas o la información para el enlazador dinámico del sistema operativo (ld en Linux) en caso de que sean dinámicas y se deban enlazar en tiempo de ejecución. Respecto al formato de estas librerías especiales, el recomendado es el dinámico por dos motivos, el primero es que permiten que múltiples programas puedan usar una copia de esta librería al mismo tiempo, permitiendo generar ficheros ejecutables más pequeños y el segundo es que al enlazarse dinámicamente en tiempo de ejecución, estas librerías permiten modificar su código sin tener que recompilar los programas fuente para incluir su código en ellos (como las estáticas).

Sin contar con el esfuerzo en generar la parte de la solución encapsulada en la librería, este diseño tiene diversos inconvenientes importantes:

- La complicación o incluso la imposibilidad de obtener el código fuente de ciertas herramientas de monitorización y gestores de colas. Esto es debido a que algunos de ellos son productos comerciales bajo licencia (por ejemplo Totalview) o únicamente se distribuyen sus códigos binarios (como Condor).
- La necesidad de poseer conocimientos avanzados de ingeniería del software y de lenguajes de programación, debido a que la mayoría de los códigos fuente de los gestores de colas y herramientas de programación están escritos en lenguajes de alto nivel como C, C++ o Java. Además, para controlar y gestionar la ejecución de los procesos con los que trabajan, estos gestores de colas y herramientas de monitorización utilizan llamadas al sistema operativo (en sistemas UNIX/Linux: signals, ptrace o kill) o a funciones de librerías especiales (Paradyn utiliza la librería Dyninst para insertar código en los procesos que monitoriza), lo cual obliga al diseñador de este tipo de solución a poseer también amplios

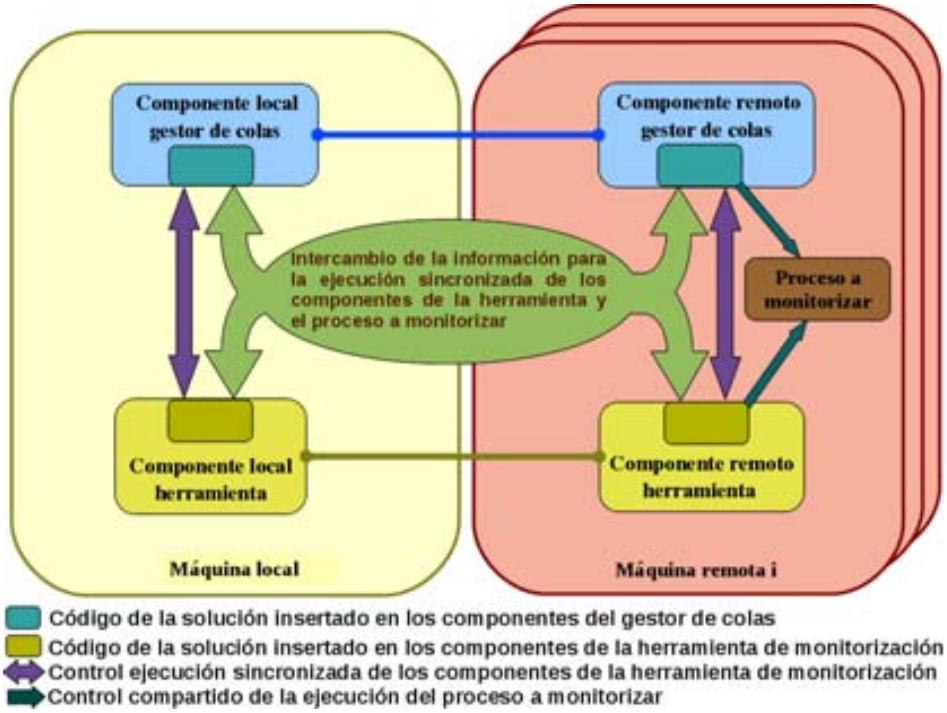


Figura 3.2: Esquema diseño de la solución integrada en el código fuente de los gestores de colas y herramientas de monitorización.

conocimientos sobre ellos.

- El código fuente de los gestores de colas y las herramientas de monitorización, como la mayoría de grandes aplicaciones informáticas, suele estar distribuido entre diversos programas fuente (.h, .cpp o .java), cuyo tamaño puede oscilar entre unas decenas o centenares de miles (o incluso millones) de líneas de código. A parte de estas características, estos códigos fuente no suelen incluir demasiada (o incluso ninguna) información sobre la codificación realizada en ellos, ya sea utilizando comentarios en el mismo código o en manuales externos creados por los propios programadores. Todo ello implica dedicar una considerable cantidad de tiempo a comprender la codificación realizada en estos códigos fuente, tanto para poder averiguar en que puntos se puede obtener la información necesaria, como donde se deben modificar para incluir el nuevo código de la solución al problema.

b) Entorno de trabajo intermedio:

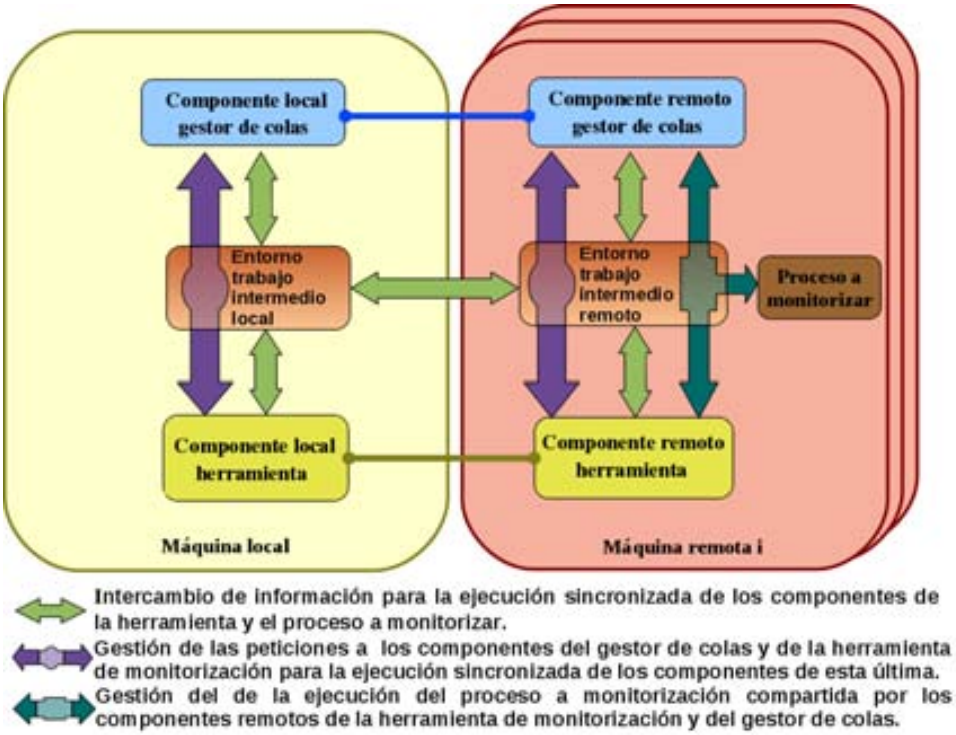


Figura 3.3: Esquema del diseño de la solución formada por un entorno intermedio de trabajo.

El diseño de esta segunda solución para problema de interoperabilidad entre gestores de colas y herramientas de monitorización (como se puede ver en figura 3.3) consiste en desarrollar un entorno de trabajo intermedio. Este entorno estará formado por un conjunto de componentes, los cuales podrán ser procesos, ficheros especiales o shell-scripts, que se encargarán de interactuar con los gestores de colas y herramientas de monitorización, para poner en ejecución, de una forma sincronizada, los componentes locales (en la máquina local del usuario) y remotos (en las máquinas del cluster) de estas últimas. Debido a que esta solución no está integrada en el código ejecutable de estos gestores de colas y las herramientas de monitorización, no será necesario dedicar una importante cantidad de tiempo a comprender sus códigos fuentes, permitiendo centrar el tiempo en el diseño de la solución al problema.

Los componentes del entorno de trabajo intermedio deberán realizar las siguientes funciones importantes:

1) Informar a los gestores de colas, normalmente a través de sus archivos de

descripción de trabajos, del entorno de ejecución remoto que han de crear, el cual les informa de las acciones a realizar para que se puedan ejecutar correctamente, en las máquinas de su cluster, los posibles componentes remotos que intervienen en el proceso de monitorización. Estos componentes pueden ser: Los procesos de la aplicación, los componentes remotos de la herramienta y si es necesario, algunos componentes de este entorno intermedio que realicen ciertas acciones en las máquinas del cluster.
Un ejemplo de las acciones que realizan estos entornos de ejecución remotos, creados por los gestores de colas, podría ser la creación de la variable de entorno que necesita uno de los componentes del entorno intermedio, la cual le informa de la dirección del ejecutable del componente remoto de la herramienta que este ha de ejecutar.

2) En caso de su existencia, aprovechar los mecanismos que ofrecen las herramientas de monitorización, para obtener la información necesaria de como ejecutar sus componentes remotos. En el caso de Paradyn, se puede solicitar a su componente local que devuelva, en un fichero, la información de los argumentos que se le han de pasar a sus componentes remotos. La idea de esta manera de proceder es que estos componentes se puedan ejecutar manualmente, normalmente utilizando terminales remotos abiertos en las máquinas del cluster. Por lo tanto, procesando la información situada en este fichero por el componente local de Paradyn, los componentes de la solución intermedia pueden obtener los argumentos que necesitan los componentes remotos de esta herramienta de monitorización para su correcta ejecución.

3) Los componentes del entorno de trabajo intermedio deben poseer mecanismos comunicación con los componentes de los gestores de colas y de las herramientas, así como entre ellos, que les permitan enviar y recibir, en el momento temporal adecuado, la información que necesita cada componente de las herramientas de monitorización para su correcta ejecución sincronizada. Por ejemplo, la herramienta Gdb necesita conocer el puerto de comunicación de su componente remoto (gdbserver), al cual se le pasa como argumento. Por lo tanto, los componentes de la solución intermedia que se encargan de la ejecución de la parte remota de esta herramienta, deben primero obtener un puerto de

comunicaciones libre (usando funciones del sistema operativo o de librerías dedicadas), después ejecutar el componente remoto de Gdb (pasándole el puerto de comunicaciones como argumento) y finalmente, pasar la información de este puerto de comunicaciones a los componentes de la solución intermedia que se encargan de la parte local de la herramienta. Una vez tienen esta información, ya pueden ejecutar el componente local de la herramienta GDB, informándole del puerto de comunicaciones donde escucha su componente remoto.

A parte de estos dos diseños, se podía pensar en un tercer diseño híbrido de los dos anteriores para la solución del problema de la falta de inteoperabilidad entre los gestores de colas y las herramientas de monitorización. Este diseño consistiría en diseñar una parte de la solución integrada en los códigos ejecutables y el resto de la solución mediante un entorno de trabajo intermedio. Al tener una parte de la solución integrada, este tipo de solución puede tener los mismos problemas que aparecen en la solución integrada pura, los cuales son, la necesidad de estudiar los códigos fuentes del las herramientas de monitorización y los gestores de colas y la imposibilidad de conseguir algunos de estos códigos fuentes.
Debido a que el diseño de la solución integrada en el código ofrece importantes inconvenientes (estudio de códigos fuentes y posible imposibilidad de obtenerlos), el diseño del entorno `TDP-Shell` se ha basado en el tipo de solución de *entorno de trabajo intermedio*

3.2.2. Implicaciones del número de posibles soluciones

Un factor importante que no ayuda a simplificar el diseño (ya sea del tipo integrado, entorno intermedio o híbrido) de la solución final al problema tratado en este trabajo de Tesis, es que una solución obtenida para una pareja *(gestor de colas, herramienta de monitorización)* normalmente no funciona con otra pareja de estas dos aplicaciones. Los principales motivos de que esto suceda son:

a) Cada gestor de colas utiliza un formato diferente de fichero de descripción de trabajos, incompatibles entre si. Como se ha visto en el capitulo 2, SGE utiliza shell-scripts y Condor un fichero especial de descripción de trabajos basado en comandos, los cuales no tienen prácticamente ninguna similitud entre si.

b) Cada herramienta de monitorización ofrece su propia metodología para iniciar el proceso de monitorización, implicando diferentes pasos para ejecutar, utilizando las herramientas de conexión remota, sus componentes remotos en las máquinas de un cluster. Algunas, como Paradyn y Totalview permiten tanto que esta ejecución se realice automáticamente desde sus componentes locales (pero la manera y la información que solicitan para realizarlo no es la misma para las dos), como que dicha ejecución se pueda realizar manualmente, por ejemplo ejecutando sus componentes remotos a través de terminales remotos abiertos en las máquinas del cluster. Para este ultimo caso, las herramientas ofrecen métodos para informar al usuario (o a las aplicaciones) del entorno de ejecución (principalmente los argumentos) que necesitan sus componentes remotos para su ejecución.
   Por el contrario, existen otras herramientas que solo permiten una forma de ejecutar de sus componentes remotos. Por ejemplo GDB solo ofrece la posibilidad que su componente remoto sea ejecutado manualmente, normalmente usando un terminal remoto abierto en la máquina donde se quiere ejecutar dicho componente remoto.

c) La información que necesitan los diferentes componentes remotos de cada herramienta de monitorización para su ejecución, difiere en cuanto al tipo y la cardinalidad de esta. Por ejemplo, el componente local de la herramienta Paradyn obtiene automáticamente los dos puertos de comunicación que necesitan sus componentes remotos, los cuales les son pasados como argumentos. Por el contrario, para la herramienta GDB, primero se ha de obtener un puerto de comunicación libre en la máquina remota donde se pretende ejecutar su componente remoto (normalmente lo hace el usuario), para pasárselo como argumento. Cuando este componente remoto de GDB esté en ejecución, se podrá ejecutar su componente local y pasarle la información del puerto de comunicación de este componente remoto, para que se le pueda conectar.
   Como puede observarse, tanto el número de puertos de comunicación, como que componente los obtiene, difiere de la herramienta Paradyn a la herramienta GDB.

Optar por crear una solución específica para cada pareja de gestor de colas, herramienta de monitorización, implica obtener tantas soluciones como el conjunto total de estas parejas pueda obtener. Este número de parejas viene determinado por la cardinalidad, *card()* , del conjunto obtenido al realizar el producto cartesiano del conjunto de gestores de colas (GC) por el de herramientas de monitorización (HM), el

cual se denota como GC x HM = { (gc,hm) | gc ϵ GC y hm ϵ HM }. Si el número de elementos de GC es n y el de HM es m , entonces *card (GC x HM) es m x n.* Esta manera de proceder tiene diversos inconvenientes, el primero es el gran número de soluciones que se podrán generar sobretodo si m y n son altos. El segundo problema es que la incorporación de nuevas herramientas y gestores de colas implica generar nuevas soluciones. Por ejemplo, la incorporación de un gestor de colas nuevo (gcn) conlleva generar m soluciones nuevas ya que *card ({ (gcn,hm) | gcn es el gestor de colas nuevo y hm ϵ HM })* es m, la incorporación de una nueva herramienta de monitorización implica n soluciones y la de un gestor nuevo (gcn) más una herramienta nueva (hmn) implica *(n+1)+m* soluciones nuevas (no son m+1 debido a que la pareja (gcn, hmn) ya está incluida en los n+1).

3.3. Implicaciones para los usuarios que deseen solucionar el problema de interoperabilidad

Del apartado anterior (3.2) se puede observar que, para obtener una la solución al problema de la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización, un usuario debe:

a) Poseer amplios conocimientos de ingeniería del software (análisis, diseño e programación de la solución) y de las llamadas al sistema operativo o a funciones de librerías especiales que gestionan las operaciones sobre los procesos. Además, si escoge la solución integrada en el código, también debe poseer amplios conocimientos de análisis de código fuente.

b) Si no tiene los conocimientos indicados en el punto anterior, el usuario debe dedicar una considerable cantidad de tiempo a adquirirlos. Una vez posee estos conocimientos, el usuario puede dedicarse a implementar la solución propiamente dicha. Esta solución le implica desarrollar un conjunto de componentes (integrados en el código o formando un entorno intermedio), los cuales gestionan la ejecución sincronizada de los componentes de la herramienta y el intercambio de información entre los componentes de esta última y los de los gestores de colas. El desarrollo de estos componentes también le implica al usuario dedicarles una considerable cantidad de tiempo.

Estos hechos enumerados en los dos puntos anteriores tienen como resultado negativo el apartar al usuario de su objetivo principal, el cual es *monitorizar su aplicación para descubrir y solucionar tanto errores como problemas de rendimiento.*

En los capítulos siguientes de este trabajo se explicará, tanto la arquitectura como la manera de utilizar el entorno desarrollado, al cual hemos denominado *Tool Daemon Protocol-Shell o* `TDP-Shell`, cuyo principal objetivo es solucionar el problema de la falta de interoperabilidad entre gestores de colas y herramientas de monitorización, sin que su aprendizaje implique un gran coste temporal para el usuario.

3.4. Conclusiones

Las causas que originan el problema de la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización, son las siguientes:

1) Las herramientas de monitorización no pueden utilizar sus herramientas de conexión remota para ejecutar sus componentes remotos en las máquinas del cluster controlado por un gestor de colas.

2) Falta de mecanismos de intercambio de información entre los componentes de los gestores de colas y de las herramientas de monitorización que permitan la ejecución sincronizada de los componentes de esta ultima.

3) Los problemas en el intercambio de información provocan que tanto los gestores de colas como las herramientas de monitorización, no puedan informarse mutuamente de los cambios en los estados de los procesos que gestionan y que puedan afectar negativamente a sus actividades.

4) Imposibilidad de que ciertas operaciones sobre los procesos monitorizados por la herramienta y que deban ser realizadas por el gestor de colas, puedan ser llevadas a cavo por este (por ejemplo crear un proceso pausado).

Para solucionar el problema de la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización se puede optar por dos diseños:

a) *Integrar la solución en sus códigos ejecutables*: Este diseño consiste en crear las estructuras de datos y operativas que solucionan la falta de interoperabilidad entre

los gestores de colas y las herramientas de monitorización, para ser integradas en sus respectivos códigos fuentes. Este diseño tiene dos problemas importantes, el primero es la gran cantidad de tiempo que se ha de invertir estudiando los ficheros fuentes de estos ejecutables, para averiguar como situar en ellos la solución. Esto es debido porque suelen estar realizados en lenguajes de alto nivel (como C, C++ o Java) y estar compuestos por decenas o incluso millones de líneas de código con poca información de la codificación realizada en ellos. El segundo problema de este diseño es la imposibilidad de obtener alguno de los códigos fuentes, por ser productos comerciales bajo licencia (como Totalview).

b) *Entorno de trabajo intermedio*: Esta solución se basa en crear unos componentes intermedios que interactúen con los gestores de colas y las herramientas de monitorización para solucionar el problema de su falta de interoperabilidad. Esta solución tiene la ventaja de que no es necesario el estudio de los códigos fuentes de los gestores de colas y herramientas de monitorización, pudiendo dedicar el tiempo al diseño de la solución del problema propiamente dicho. Debido ha estas ventajas, el diseño de la solución adoptado por el entorno `TDP-Shell` es de este tipo.

Independientemente del diseño de la solución al problema de interoperabilidad escogido, existe un factor que no ayuda a la obtención final de esta solución. Este factor consiste en que la solución encontrada para un par *(gestor de colas, herramienta de monitorización)*, normalmente no funciona para otro par de estas aplicaciones. Esto es debido a que:

a) Cada gestor de colas utiliza sus propios ficheros de descripción de trabajos, con pocas similitudes entre ellos. Condor usa un fichero basado en comandos y SGE shell-scripts.

b) Cada herramienta ofrece su propia metodología para indicar como poner en ejecución sus componentes remotos. Paradyn y Totalview ofrecen la posibilidad de hacerlo desde su componente local (pero cada uno de manera diferente) y GDB solo ofrece la posibilidad de que el usuario lo haga manualmente (a través de terminales remotos abiertos en las máquinas del cluster).

c) La información que necesitan los diferentes componentes remotos de las herramientas difiere respecto al tipo y número. Por ejemplo, el componente local de Paradyn escoge dos puertos de comunicación libres y se los pasa a sus componentes remotos, mientras

que para GDB es el usuario quien ha de obtener un puerto libre en la máquina remota y pasarlo como argumento al componente remoto de esta herramienta.

Para obtener una solución global al problema de la interoperabilidad para un conjunto de m gestores de colas y otro de n herramientas de monitorización, se debe desarrollar una solución particular para cada pareja de de elementos de estos dos conjuntos, generando un total de mxn de estas soluciones particulares.
Todos estos factores implican, que para obtener una solución al problema de interoperabilidad entre los gestores de colas y herramientas de monitorización, un usuario debe dedicar una considerable cantidad de tiempo, apartándole de su objetivo principal, el cual es monitorizar su aplicación.

4

Arquitectura base del entorno de trabajo TDP-Shell

En el capítulo anterior se han mostrado las causas que dificultan (o imposibilitan) la utilización de un conjunto de herramientas de monitorización en clusters controlados por diferentes gestores de colas, dando lugar al problema que denominamos *la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización*. También se han explicado las dificultades que puede encontrar un usuario que quiera solucionar este problema, implicándole una gran cantidad de tiempo dedicado a este propósito y apartándole de su objetivo principal, que es monitorizar su aplicación para solucionar sus posibles errores y problemas de rendimiento.

Para solucionar este problema de la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización se ha propuesto y desarrollado el entorno de trabajo `TDP-Shell`. En este capítulo se explicará la *arquitectura base de este entorno* [32], la cual soluciona el problema de la falta de interoperabilidad entre gestores de colas y herramientas de monitorización cuando estas ultimas monitorizan aplicaciones tipo serie, esto es, formadas por un solo proceso. Esta arquitectura base es el punto de partida para el diseño de las nuevas versiones del entorno `TDP-Shell`, las cuales solucionan el problema de la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización, cuando estas últimas monitorizan aplicaciones distribuidas (formadas por más de un proceso) como las basadas en el protocolo MPI.

Para cumplir con los objetivos de este capitulo, en su primer apartado se explica, con mayor profundidad el *protocolo TDP* el cual se introdujo en el capitulo 2 y dio origen al entrono de trabajo `TDP-Shell` (de ahí su prefijo TDP). En los siguientes apartados se muestran los requerimientos generales de este entorno, para finalizar con la explicación del diseño y las funciones que realiza cada componente de la arquitectura base del entorno `TDP-Shell`.

4.1. Protocolo TDP (Tool Daemon Protocol)

Como se ha explicado en el capitulo 2, el protocolo TDP es una primera aproximación para solucionar el problema de la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización. Para ello propone una interfaz estándar para ambos que encapsule los servicios necesarios para la correcta interacción entre los diferentes componentes de las herramientas de monitorización y los gestores de colas. Estos servicios se agrupan en tres conjuntos, los cuales son:

1) **Intercambio de información**: Para poder realizar el intercambio de información que necesitan los diferentes componentes de los gestores de colas y herramientas de monitorización para su correcta interoperabilidad, el protocolo TDP define el *Espacio de atributos*. En este espacio, la información se almacena como un conjunto de *tuplas* de la forma *(atributo, valor)*, donde *atributo* es el identificador de la información y *valor* es propiamente el valor de dicha información. Por ejemplo, la tupla *(PORT, 2014)* identificaría el puerto de comunicaciones donde escucha peticiones uno de los componentes de la herramienta. Estas características hacen que este Espacio de atributos entorno guarde ciertas similitudes con Linda [33]. Para el situar y extraer información de este Espacio de atributos, el protocolo TDP ofrece dos primitivas básicas:

   - *tdp_get (atributo):valor*: Obtener del Espacio de atributos el *valor* de una tupla identificada por un *Atributo*.
   - *tdp_put (atributo, valor)*: Insertar en el Espacio de atributos la tupla *(atributo, valor)*.

   Estas primitivas son bloqueantes (*o síncronas*), esto significa que el proceso que las

llama suspende su ejecución hasta la finalización de la operación de comunicación asociada a dicha primitiva. Un *tdp_get* bloquea el proceso que lo llama hasta recibir, del Espacio de atributos, el valor asociado a la tupla que ha solicitado. Un *tdp_put* bloquea el proceso que lo llama hasta recibir la confirmación del Espacio de atributos que la tupla ha sido correctamente insertada en el.

2) **Gestión de eventos:** El protocolo TDP también ofrece la gestión de sucesos que no son predecibles en el tiempo (son asíncronos), a las cuales se les denomina *eventos*. La gestión de estos eventos se realiza mediante las versiones no bloqueantes (*o asíncronas*) de las primitivas vistas en el punto anterior, las cuales son:

   - *tdp_async_get (atributo, nombre_funcion:input)*: Versión asíncrona de *tdp_get*
   - *tdp_async_put (atributo, valor, nombre_funcion:input)*: Versión asíncrona *tdp_put*

   Estas primitivas no bloquean al proceso que las invoca, informan al Espacio de atributos de la operación de comunicación que quieren realizar (situar o recibir tuplas), permitiendo al proceso continuar su ejecución. Cuando se inserta en el Espacio de atributos la tupla que ha solicitado un *tdp_async_get* o la tupla que ha enviado un *tdp_async_put* se ha insertado correctamente en este Espacio de tuplas, la operación de comunicación asociada a estas primitivas asíncronas está lista para su finalización (ha sucedido el evento que se quiere controlar). Para ello, el Espacio de atributos informa de este hecho al proceso que las invocó para que este suspenda su ejecución normal y llame a la función especial (*nombre_función*) asociada a la primitiva asíncrona, la cual se encarga de ejecutar las acciones que finalizaran la operación comunicación asociada a dicha primitiva (este proceso es similar al tratamiento de una interrupción que realizan los sistema operativo). Por ejemplo, los componentes locales de la herramienta pueden utilizar una primitiva asíncrona *tdp_async_get* sobre un una tupla especifica, para gestionar la posible ocurrencia de errores durante la ejecución de alguno de sus componente remotos (este evento es asíncrono, ya que no se sabe si sucederá y si es el caso, cuando). Si alguno de estos errores llega a producirse, entonces la tupla especifica que lo identifica será situada en el Espacio de atributos (con un *tdp_put* o *tdp_async_put*), provocando que el componente local de la herramienta llame a la función especial asociada a la primitiva *tdp_async_get* que gestiona el error, para que esta realice las acciones necesarias relacionadas con la ocurrencia de dicho error.

3) **Control de procesos**: Por último el protocolo TDP también define un conjunto de primitivas para la gestión y control de los procesos locales que se van a ejecutar en las máquinas del cluster (sobre un sistema operativo específico). Estas primitivas pueden ser usadas por las herramientas de monitorización o gestores de colas en el caso de que no posean las funcionalidades que estas realizan. Las más importantes son:

   - *tdp_create*: Crea un proceso.
   - *tdp_create_paused*: Crea un proceso e inmediatamente lo pausa.
   - *tdp_kill*: Finaliza un proceso creado previamente con *tdp_create*
   - *tdp_paused*: Pausa un proceso que esta en ejecución.
   - *tdp_continue*: Continúa un proceso que ha sido pausado con *tdp_paused*

   Por ejemplo, si el componente remoto del gestor de colas no puede crear un proceso en estado pausado (se sitúa en memoria pero no se empieza su ejecución). Entonces puede utilizar la primitiva *tdp_create_paused* para realizar esta funcionalidad.

Como se ha visto, el protocolo TDP propone una interfaz pensada para que su implementación (normalmente en forma de librería) sea incorporada en el código de los diferentes gestores de colas y herramientas de monitorización, con los problemas que esta característica supone (vistos en el apartado 3.2.1). Este hecho no implica que las ideas aportadas por el protocolo TDP no puedan ser aprovechadas en el desarrollo del entorno `TDP-Shell` (que evita la modificación del código fuente) y en los siguientes puntos de este capítulo se mostraran las maneras en que esto se ha realizado, convirtiendo al protocolo TDP en el punto de partida del diseño del entorno de trabajo `TDP-Shell`.

4.2. requerimientos generales del entorno de trabajo TDP-Shell

Para que el uso del entorno `TDP-Shell` sea interesante a sus usuarios además de facilitar su diseño y posterior implementación, se ha propuesto que este entorno cumpla los siguientes requerimientos generales (para todas sus versiones):

a) Facilidad de uso, para evitar que sus usuarios abandonen su utilización debido a que necesiten dedicar una gran cantidad de tiempo a la comprensión de su utilización.

b) Flexibilidad para adaptarse a diferentes gestores de colas y herramientas de monitorización.

c) Su diseño no ha de implicar (o evitar al máximo posible) modificar el código ejecutable de las herramientas de monitorización y de los gestores de colas. Con ello se evita la gran cantidad tiempo que se ha de invertir estudiando sus códigos fuentes, normalmente compuestos de centenares de miles o de millones de líneas de código, repartidas en diversos ficheros fuente y sin mucha información que ayude a interpretar su funcionalidad. Otro factor que afecta en la decisión de adoptar este requerimiento del entorno `TDP-Shell`, es la imposibilidad (o gran dificultad) de acceder a ciertos códigos fuentes, por ser productos comerciales bajo licencia (por ejemplo la herramienta de monitorización Totalview).

4.3. Núcleo de la arquitectura base del entorno de trabajo TDP-Shell

Al proponer, como requisito general, que el entorno `TDP-Shell` evité modificar el código ejecutable de los gestores de colas y de las herramientas de monitorización, uno de los requisitos funcionales más importantes que ha de cumplir la arquitectura de este entorno ha de ser que sea del tipo *entorno de trabajo intermedio* (explicada en el capítulo 3.2.1). Esto implica el diseño de un conjunto de componentes (procesos, ficheros de configuración, shell-scripts, etc) que interactúen con los de las herramientas de monitorización y de los gestores de colas, para conseguir su correcta colaboración en el proceso de monitorización de una aplicación serie. En el apartado 3.1 se ha explicado que uno de los principales problemas que causa la falta de interoperabilidad entre los gestores de colas y las herramientas de monitorización, es el hecho de que estas últimas no pueden utilizar sus herramientas de conexión remota (como ssh), para ejecutar sus componentes remotos en el cluster controlado por alguno de estos gestores de colas. También se ha explicado que la ejecución de los componentes locales y remotos de las herramientas de monitorización se ha de realizar de una forma sincronizada en el tiempo. Por lo tanto, la solución al problema de interoperabilidad entre gestores de colas y herramientas de monitorización se puede dividir en dos partes: la que se encarga de la *gestión de la parte local del problema* y la que se encarga de *la parte remota del*

mismo. La principal función de la parte local es la creación del componente local de la herramienta y la de la parte remota de la creación del componente remoto y si es necesario del proceso al monitorizar. Para realizar estas funciones, ambas partes de la solución se deben intercambiar la información necesaria que les permita sincronizar la ejecución de ambos componentes de la herramienta. Para implementar las funciones de estas dos partes de la solución, la arquitectura base del entorno `TDP-Shell` define tres componentes principales (mostrados en la figura 4.1), los cuales son:

1) **Componente local del entorno TDP-Shell**: Normalmente se ejecuta en la máquina local del usuario y se encarga de crear el componente local de de la herramienta. A este componente se le denominará a partir de este punto `tdp_console`.

2) **Componente remoto del entorno TDP-Shell**: Se ejecuta en la máquina remota (creado por el gestor de colas), encargándose de la creación del componente remoto de la herramienta de monitorización y en caso necesario, del proceso del usuario a monitorizar. A este componente se denominará a partir de este punto `tdp_agent`.

3) **Espacio de atributos tdp**: Igual que el espacio de atributos propuesto por el protocolo TDP (es una aportación de este protocolo), su función es almacenar la información, en formato de tuplas, que necesitan intercambiar `tdp_console` y `tdp_agent`, para que puedan llevar a cavo la ejecución sincronizada de los componentes de la herramienta de monitorización. Igual que en el protocolo TDP, este espacio de atributos tdp ofrece unas primitivas síncronas y asíncronas (get y put) para situar y extraer las tuplas que contienen la información ubicada en él.

Como el encargado de la ejecución del componente `tdp_agent` en una máquina del cluster solo puede ser el gestor de colas, el componente `tdp_console`, aparte de la función explicada anteriormente, también se encarga de realizar la petición al gestor de colas para que ejecute el componente remoto del entorno `TDP-Shell`.

4.4. Descripción general de la arquitectura base de TDP-Shell

Antes de comenzar a explicar el diseño y funciones de los componentes del entorno `TDP-Shell`, explicados en el apartado anterior, es necesario conocer la información que

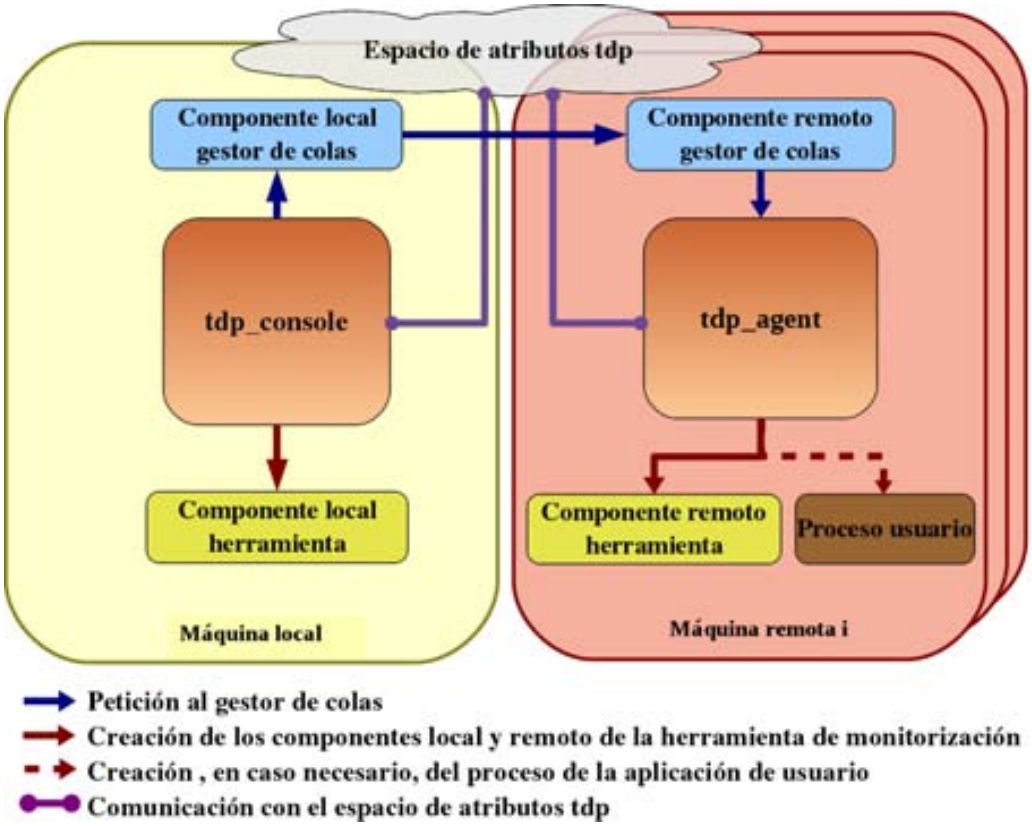


Figura 4.1: Componentes principales de la arquitectura del entorno de trabajo TDP-Shell

estos necesitan, tanto de los gestores de colas como de las herramientas de monitorización, para que puedan realizar correctamente sus funciones. Esta información se puede resumir en los siguientes puntos:

a) Entorno de ejecución del componente remoto de la herramienta y del proceso del usuario que va ha ser monitorizado. Este entorno de ejecución define los requerimientos que pueden necesitar estos componentes para su correcta ejecución. Algunos de los más importantes son:

   - Situación (directorio) y nombre, en las máquinas del cluster, de sus ejecutables.
   - Los posibles argumentos de estos ejecutables.
   - Ficheros de entrada o de salida que puedan requerir. Estos ficheros pueden estar situados en la máquina local del usuario y ser necesaria su copia a las maquinas del cluster. Para realizar esta copia se pueden utilizar los mecanismos de copia

remota (máquina local usuario <-> máquina cluster) que ofrecen los gestores de colas.

- Definición de las variables de entorno que ambos componentes puedan necesitar.

b) Entorno de ejecución del componente local de la herramienta (ejecutable, argumentos o ficheros), normalmente situado en la máquina local del usuario.

c) Información de los puertos de comunicación por donde se intercambian la información los componentes de la herramienta de monitorización. Normalmente se utiliza el protocolo TCP (o UDP)/IP, por lo tanto, esta información está formada por un conjunto de parejas: *máquina - puerto o puertos de comunicación*, por donde escucha el componente local de la herramienta o los componentes remotos de esta.

d) Como se ha visto es el componente `tdp_console` el que informa al gestor de colas de cómo se debe crear, en la máquina del cluster que este gestiona, el componente `tdp_agent`. Por lo tanto y como en el caso de los componentes remotos de la herramienta, puede ser necesario informar del entorno de ejecución, en las máquinas del cluster, para el componente `tdp_agent`.

Debido a que el entorno de trabajo `TDP-Shell` no debe modificar el código fuente de los gestores de colas ni el de las herramientas de monitorización, sus componentes no pueden obtener la información directamente del código fuente de estos. Para ello, deben utilizar otras fuentes de información, las principales son:

- **Ficheros de descripción de trabajos**: A través de estos ficheros, propios del gestor de colas, se describe el entorno de ejecución del trabajo (ejecutables de sus procesos, ficheros, copia remota de estos, variables de entorno, etc) que el usuario desea que se ejecute en el cluster que controla el gestor de colas. Antes de la utilización del entorno de trabajo `TDP-Shell`, el usuario ya tiene creado el fichero de descripción de trabajos para la aplicación que desea monitorizar y es el que utiliza para que el gestor de colas la ejecute en las máquinas de su cluster. A parte de la aplicación de usuario, los otros componentes remotos del Entorno `TDP-Shell`, el de la herramienta de monitorización y el del entorno `TDP-Shell` (el `tdp_agent`), también se ejecutan en una máquina del cluster. Por lo tanto y si es necesario, para estos componentes también se pueden definir los ficheros de descripción de

trabajos que informen de sus respectivos entornos de ejecución al gestor de colas. Procesando la información contenida en estos ficheros de descripción de trabajos, el entorno `TDP-Shell` puede obtener un nuevo *fichero de descripción de trabajos global* el cual describe el *entorno de ejecución común o global* que permite al gestor de colas realizar, en alguna de las máquinas de su cluster, las acciones necesarias para que se puedan ejecutar, directamente por el o a través de otro componente, los componentes remotos del entorno `TDP-Shell`. Por ejemplo, el proceso de la aplicación del usuario necesita que el archivo A sea accesible en la máquina del cluster y el componente remoto de la herramienta de monitorización necesita que lo sea fichero B. Si ambos ficheros están en la máquina del usuario, el nuevo fichero de descripción de trabajos global definirá, entre otras acciones, la copia de los ficheros A y B de la máquina local del usuario a las máquinas del cluster (pudiendo aprovechar los mecanismos de copia remota del gestor de colas). De esta forma, el componente `tdp_agent` ya tendrá disponible estos dos ficheros en la máquina del cluster y podrá ejecutar tanto el componente remoto de la herramienta como el del proceso de la aplicación de usuario.

El encargado de procesar los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell`, para obtener fichero de descripción de trabajos global, es el componente `tdp_console` (figura 4.2, puntos 2 y 3a). Una vez creado este fichero, es pasado al gestor de colas (figura 4.2, punto 3b) para que pueda crear el entorno de ejecución de los componentes remotos del entorno `TDP-Shell`.
Como se ha explicado en el apartado anterior (4.4, el componente `tdp_console` es el encargado de informar al gestor de colas de que ejecute el componente `tdp_agent`. Por lo tanto, el único fichero de descripción de trabajos que se le debe pasar obligatoriamente a `tdp_console`, es el del `tdp_agent` (contiene la información que permite al gestor de colas ejecutar este componente). Los ficheros de descripción de trabajos del resto de los componentes remotos de `TDP-Shell`, se le suministrarán a `tdp_console` en el caso de sea necesario que el gestor de colas realice algunas acciones para su correcta ejecución.

De este procesamiento de los ficheros de descripción de trabajos, `tdp_console` también obtiene cierta información que puede ser interesantes para las acciones

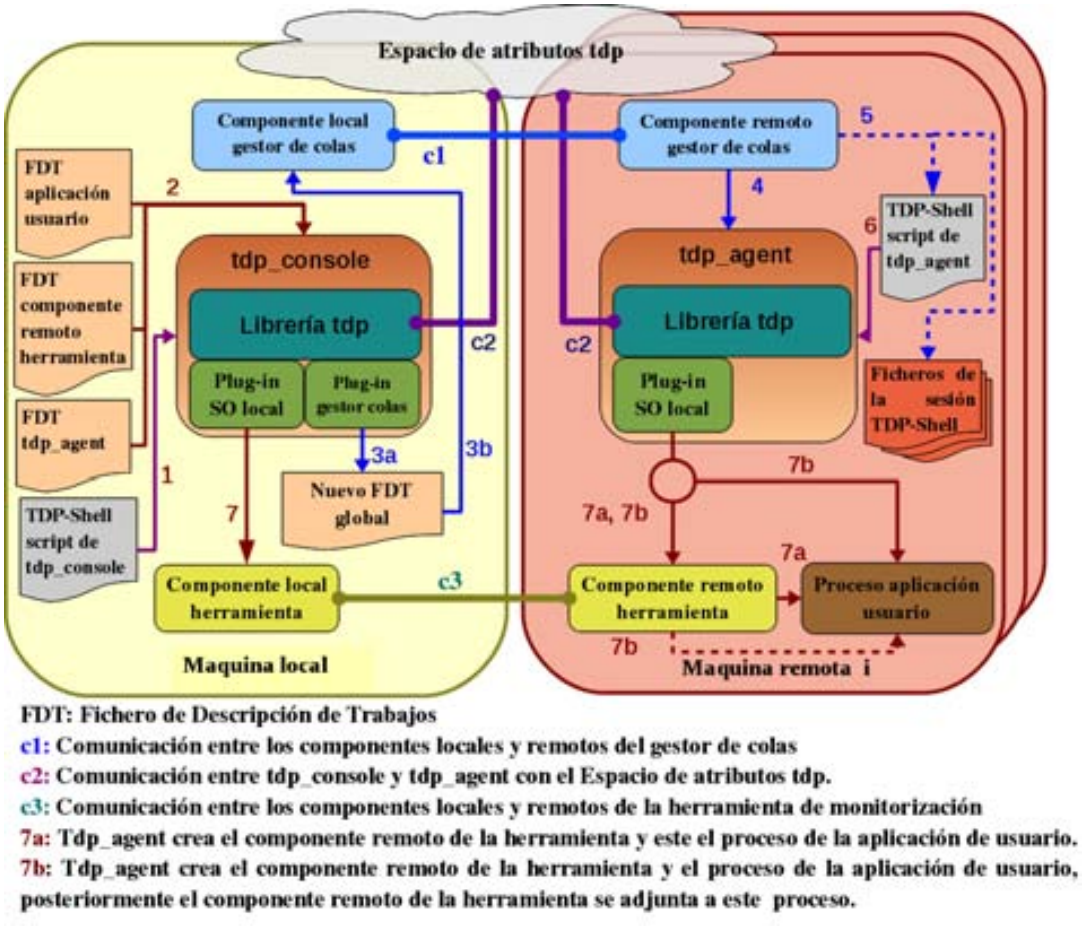


Figura 4.2: Esquema de la arquitectura base del entorno de trabajo TDP-Shell

que deberá llevar a cavo el `tdp_agent` en la máquina remota. Algunos de estas informaciones son:

- La dirección, el nombre del ejecutable y los posibles argumentos del proceso de la aplicación de usuario
- Los mismos datos pero para el de componente remoto de la herramienta de monitorización.

Para que estos datos sean accesibles al `tdp_agent`, el componente `tdp_console` los sitúa en el espacio de atributos tdp a través de unas tuplas especiales. Por ejemplo, *la tupla con el atributo TDP_USER_EXEC* contiene la información del ejecutable del proceso de usuario en las máquinas remotas.

- **Suministrada por la propia herramienta**: La otra vía para obtener la información que necesita el entorno `TDP-Shell` para la ejecución sincronizada de los componentes de las herramientas de monitorización, la suministran estas propias herramientas. Algunas ofrecen utilidades u opciones que informan de como ejecutar sus componentes remotos manualmente (normalmente a través de terminales remotos abiertos en las máquinas del cluster). Por ejemplo, el componente local de la herramienta Paradyn, a través de su argumento *-x nombre_fichero* ofrece la posibilidad de informar, a través del fichero *nombre_fichero*, de los argumentos que necesita su componente remoto (entre ellos los puertos de comunicaciones) para su ejecución manual. Por lo tanto, procesando este fichero, el entorno `TDP-Shell` puede obtener los argumentos que necesita para ejecutar este componente remoto de Paradyn.

  Otras herramientas, como Gdb o Totalview (Paradyn también la ofrece), ofrecen una documentación (normalmente el *manual de usuario*) donde explican la información necesaria para poder ejecutar sus componentes remotos manualmente. Una vez obtenida esta información, no es un buen método incorporarla directamente en el código de los componentes del entorno de trabajo `TDP-Shell`, sino que es más aconsejable incorporarla a través de métodos dinámicos, como ficheros o variables de entorno, ya que estos permitan variar o actualizar la información que necesitan las herramientas de monitorización (por ejemplo cuando hay una actualización de estas), sin necesidad de volver a generar el código de los componentes del entorno de trabajo `TDP-Shell`.

A parte de la obtención de la información, los componentes del entorno `TDP-Shell` también deben encargarse del correcto intercambio de dicha información para sincronizar la ejecución de los componentes locales y remotos de la herramienta de monitorización y en caso necesario, del proceso de la aplicación de usuario (figura 4.2, puntos 7, 7a y 7b). Para ello es importante poder definir el orden temporal de ejecución de estos componentes, el cual vendrá definido por la obtención de cierta información por parte de uno de los componentes de la herramienta, por ejemplo su puerto de comunicación donde acepta peticiones, la cual es necesitada por el resto de componentes de la misma, para comenzar o continuar su ejecución. Por lo tanto, en el proceso de sincronización es importante tanto el "que" como "el cuando" se envía la información. Para definir este

proceso de sincronización e intercambio de información, el entorno `TDP-Shell` utiliza los siguientes archivos especiales:

Archivos TDP-Shell script: Estos dos archivos especiales, uno interpretado por el `tdp_console` (figura 4.2, punto 1) y el otro por el `tdp_agent` (figura 4.2, punto 6), definen el orden temporal de envió y recepción de la información necesaria para que ambos componentes del entrono `TDP-Shell` puedan ejecutar, de una forma sincronizada, el componente local y remoto de la herramienta de monitorización. Para estos archivos se ha escogido un formato parecido a los ficheros shell-script (utilizados por los interpretes de comandos como bash shell o C-shell) debido a que son ampliamente utilizados y por lo tanto, su sintaxis y semántica es conocida. Aprovechando este formato, estos ficheros `TDP-Shell` script definen el intercambio de información a través de:

- *Comandos tdp de comunicación con el espacio de atributos tdp*: Basados en las primitivas de comunicación bloqueantes y no bloqueantes definidas por el protocolo TDP, son los encargados de realizar el intercambio de información (situar y extraer) con el espacio de atributos tdp.

- *Comandos tdp creación y control de procesos locales*: Igual que los anteriores, también están basados en las primitivas de control de procesos definidas por el protocolo TDP. Son los encargados de crear, operar (pausar, continuar ejecución) y obtener la información del estado (en ejecución, pausado o error) de los procesos que se van a ejecutar en una maquina determinada (local del usuario o remota del cluster). Por ejemplo, a través de estos comandos (el de creación) `tdp_agent` puede ejecutar el componente remoto de la herramienta. Para realizar estas funciones principalmente utilizan los servicios del sistema operativo local de dicha máquina.

- *Comandos tdp de comunicación con el gestor de colas*: Propios del entorno de trabajo `TDP-Shell`, son utilizados por `tdp_console` para procesar los ficheros de descripción de trabajos de los componentes remotos del entorno `TDP-Shell` y obtener el fichero de descripción de trabajos global. Este fichero, como se ha mencionado anteriormente, informa al gestor de colas del entorno de ejecución común de estos componentes remotos, el cual, entre otras acciones, le informa como ha de ejecutar el componente `tdp_agent` en una maquina del cluster que controla.

- *Instrucciones especiales*: Se encargan de añadir funcionalidad a los ficheros `TDP-Shell` script. Pueden ser bucles (while), instrucciones condicionales (if, then, else), variables, asignaciones y declaraciones de funciones locales.

A partir de este punto, tanto a los diferentes tipos de comandos tdp como a las instrucciones especiales que forman parte de un fichero `TDP-Shell` script, se les denominará conjuntamente *código tdp*

El código de los diferentes comandos tdp, esta implementado en una librería especial que incorporan tanto `tdp_console` como `tdp_agent`, la cual es:

librería tdp :(figura 4.2, librería tdp) Librería (implementada de forma dinámica) que contiene el código de los de los comandos tdp de de comunicación bloqueantes y no bloqueantes, así como el de los comandos tdp encargados de la creación y gestión de procesos locales y los de comunicación con el gestor de colas. Parte de la funcionalidad de estos dos últimos grupos de comandos es dependiente del sistema operativo de las máquinas del cluster y del gestor de colas que lo gestiona. Para evitar diseñar una versión de la librería para cada uno de estos sistemas operativos y gestores de colas donde se vaya ha utilizar el entorno `TDP-Shell`, la librería tdp incorpora unos *plug-ins especiales*. Estos son unos ficheros con el formato de una librería dinámica, que contienen el código de ciertas funciones y que a diferencia del comportamiento estándar de las librerías dinámicas, que son gestionadas por el enlazador dinámico del sistema operativo, estos plug-ins son llamados directamente por la librería tdp (a través de unas llamadas específicas del S.O, como dlopen para UNIX/Linux). Los plug-ins del sistema operativo local (figura 4.2, plug-in sistema operativo local) contienen un conjunto de funciones que se encargan de la creación y gestión de procesos aprovechando las llamadas que ofrece el sistema operativo particular y los plug-ins del gestor de colas (figura 4.2, plug-in gestor colas) contienen el código de las funciones que se encargan de la comunicación con el gestor de colas, entre ellas las que generan el nuevo fichero de descripción de trabajos (tras procesar los tres de los componentes remotos de `TDP-Shell`). Con esta implementación, los comandos tdp de creación de procesos locales y de comunicación con el gestor de colas de la librería tdp, llaman a las funciones específicas de los plug-ins para realizar sus acciones. Por ejemplo, para un cluster controlado por el gestor de colas Condor donde el sistema operativo de las máquinas que gestiona es Linux, la librería tdp incorporaría un plug-in específico para Linux y otro para Condor.

La utilización de los plug-ins ofrece ciertas ventajas, en cuanto al diseño del entorno `TDP-Shell`, las cuales son:

- Al evitar el diseño de múltiples versiones de la librería tdp, también evita la necesidad de la compilación y generación de código de estas nuevas versiones de la librería, así como de los componentes `tdp_console` y `tdp_agent` que las incorporan. Por ejemplo, para los gestores de colas Condor y SGE, se deberían crear dos versiones de librería tdp (una para cada uno de ellos) y a su vez, dos versiones de tdp_consle y otras dos de `tdp_agent` (uno para cada versión librería tdp). Este hecho implica una dedicación de tiempo y un incremento del espacio en disco que ocupa cada versión de los ejecutables de estos componentes del entorno `TDP-Shell` (sobretodo si la versión de la librería tdp es estática).

- Permite un diseño mas modular de la solución, al repartir las funcionalidades de la librería tdp en diferentes módulos. Cada uno de estos módulos puede desarrollarse por separado (teniendo en cuenta su interfaz común de intercambio de información), permitiendo centrarse mejor en la optimización de su diseño y funcionalidades propias.

Como puede observarse, la arquitectura base del entorno `TDP-Shell` divide la solución del problema de la falta de interoperabilidad entre los gestores de colas y herramientas de monitorización en dos puntos importantes:

1. La creación del entorno de ejecución global de los diferentes componentes remotos de `TDP-Shell` para el gestor de colas (utilizando el fichero de descripción de trabajos global).

2. El intercambio de información y la ejecución sincronizada de los componentes de la herramienta, mediante los ficheros `TDP-Shell` script para `tdp_console` y `tdp_agent`.

Esta división se ha escogido por dos motivos importantes:

1. Separa la puesta en marcha del entorno remoto de monitorización, cuyo encargado es el gestor de Colas, de la gestión del procedimiento de monitorización, cuyos encargados son `tdp_console` y `tdp_agent` utilizando sus ficheros `TDP-Shell` script.

2. Permite simplificar la complejidad de la solución al problema de la interoperabilidad (explicado en el apartado 3.2.2) implementada por el entorno `TDP-Shell`. Con esta esta división de la solución se consigue:

   - Si el cluster cambia de gestor de colas, en la mayoría de casos solo será necesario definir los ficheros de descripción de trabajo de los componentes remotos de `TDP-Shell` para el nuevo gestor de colas. No será necesario modificar (ni crear de nuevo) los ficheros `TDP-Shell` script de las diferentes herramientas. Estos ficheros definen la información que deben intercambiarse los componentes de la herramienta de monitorización, así como la manera en que deben ejecutarse sincronizadamente, acciones que son independientes del gestor de colas que controla el cluster.
   - Si se introduce una herramienta nueva, entonces se deberán definir sus ficheros `TDP-Shell` script y el archivo de descripción de trabajos de su componente remoto. Los archivos de descripción de trabajos de la aplicación de usuario y del `tdp_agent` se podrán seguir utilizando, ya que no se ha cambiado de gestor de colas.

En los apartados siguientes de este capitulo se profundizará en la explicación de como se procesan los archivos de descripción de trabajos dependiendo del gestor de colas utilizado, así como de la explicación de los diferentes grupos de comandos tdp que se pueden utilizaren los archivos `TDP-Shell` script.

4.5. Procesamiento de los ficheros de descripción de trabajos por tdp_console

Como se ha visto en el capitulo 2, los gestores de colas Condor y SGE utilizan ficheros de descripción de trabajos con formatos muy diferentes. Esto implica que, dependiendo del gestor de colas, `tdp_console` deberá procesar de manera diferente los archivos de descripción de trabajos de los componentes remotos de `TDP-Shell` para obtener el nuevo archivo de descripción de trabajos global [1]. En los siguientes puntos de este apartado

[1] Como se ha indicado en el apartado4.4, la parte principal de este procesamiento la realiza el plug-in del gestor de colas que se incorpora a `tdp_console` a través de la librería tdp. Por este motivo, siempre

se explicará el procedimiento que sigue `tdp_console` para obtener este nuevo archivo de descripción de trabajos global dependiendo si el gestor de colas es SGE o Condor.

4.5.1. SGE

El gestor de colas SGE utiliza ficheros shell-scripts para definir el entorno de ejecución de los trabajos que ha de ejecutar en las máquinas del cluster que controla. Estos ficheros tienen las siguientes características importantes respecto a la manera de localizar la información en ellos:

- No existe ninguna estructuración formal que permita identificar o localizar, clara e inequívocamente, la información contenida en los ficheros shell-script. Por ejemplo, no ofrecen directivas o instrucciones especiales (como Condor) que permitan averiguar, entre sus muchas líneas de comandos, cual es la del ejecutable (o ejecutables) de su proceso principal (los comentarios tampoco sirven, ya que son personales).
- Diversos códigos shell-script pueden realizar la misma tarea (sobretodo si son programados por diferentes usuarios). Por lo tanto no existe una forma única de escribir código para este tipo de ficheros. Por ejemplo, si un shell-script sitúa, al principio de su fichero, su ejecutable principal, no se puede garantizar que otro shell-script, que realice funciones similares (o incluso las mismas), también lo sitúe al principio.

Esta falta de mecanismos para identificar la información en los ficheros shell-script (incluidos los de los componentes remotos del entorno `TDP-Shell`), dificulta enormemente que `tdp_console` pueda localizar la información que de ellos necesita. Para solucionar este problema es necesario definir una estructuración formal para estos ficheros shell-script, a través de la cual puedan indicarle a `tdp_console` donde tienen situada la información que necesita. Esta estructuración ha de ser simple, ya que tendrá que ser utilizada por los usuarios (o administradores) del entorno `TDP-Shell`, debido a que ellos pueden crear algunos de estos ficheros shell-script de descripción de trabajos que utiliza `tdp_console` (por ejemplo el de la aplicación de usuario). Para este propósito se aprovechará la

que se haga referencia a que tdp_conosle procesa un archivo de descripción de trabajos, esta división de funciones siempre estará implícita

estructuración que utilizan la mayoría de programadores de shell-scripts de una forma automática e intuitiva, la cual consiste en dividir el fichero shell-script en las siguientes tres secciones (a partir de ahora *secciones de* `TDP-Shell`) correlativas:

1) *Sección pre-ejecutable*: En esta sección es donde se sitúan los comandos que se han de ejecutar antes del ejecutable de su proceso principal. Por ejemplo, las declaraciones de variables de entorno o la copia de ficheros de configuración que necesita dicho ejecutable principal.

2) *Sección ejecutable*: A continuación de la sección anterior, se sitúa el ejecutable del proceso principal y sus posibles argumentos.

3) *Sección post-ejecutable*: Esta es la zona donde se colocan los comandos que se han de ejecutar después del ejecutable principal. Por ejemplo, los comandos encargados de borrar los ficheros que ya no son necesarios (y que normalmente se han creado o copiado en la zona pre-ejecutable).

Hay que observar que las secciones pre/pos-ejecutables no son obligatorias, pude suceder que los procesos principales del shell-script no necesiten ninguna acción previa ni posterior a su ejecución. Utilizando esta división de los ficheros shell-script, `tdp_console` puede averiguar que acciones a de realizar el gestor de colas antes y después de la ejecución del proceso principal, así como la dirección, nombre y argumentos de este. Para delimitar al alcance de cada una de estas tres secciones especiales, el entorno `TDP-Shell` define unas primitivas especiales, las cuales son:

- *#TDP_begin_pre_block*: Define el comienzo de la zona pre-ejecutable del shell-script (si existe).
- *#TDP_exec*: La línea situada a continuación de esta primitiva contiene la dirección, el nombre y los posibles argumentos del ejecutable principal.
- *#TDP_begin_post_block*: Define el comienzo de la zona post-ejecutable del shell-script (en caso de su existencia).
- *#TDP_end_block:* Define el final de las zonas pre y post-ejecutable (si alguna de ellas existe).

Puede observarse que para que SGE pueda utilizar los ficheros shell-scripts que contengan estas tres secciones, las primitivas especiales del entorno de trabajo `TDP-Shell` se han definido como comentarios para los shell-script(símbolo # al iniciar la línea). Estos comentarios no son procesadas por SGE (por ser comentarios) y solo tienen significado para `tdp_console`.

Una vez situadas las primitivas especiales de `TDP-Shell` en los ficheros shell-script, `tdp_console` debe procesar las líneas de comandos shell, situadas entre estas primitivas, para obtener la información que necesita de estos ficheros. Estas líneas están formadas por una sucesión de cadenas de caracteres, normalmente separadas por espacios en blanco (también se pueden utilizar otros separadores como tabuladores o comas), donde cada una de estas cadenas puede representar, entre otros, el inicio de un comentario, un comando shell o el ejecutable de un proceso. Para realizar el procesamiento de estas cadenas de caracteres, `tdp_console` utiliza un conjunto de primitivas sobre estas, las principales de las cuales son:

- Suponiendo que la posición del primer carácter de una cadena de caracteres es la 0 y la del último es la longitud de la cadena (número de caracteres) menos 1.

- *Concatenar las cadenas A y B* : Devuelve la unión de las cadenas de caracteres A con B. Por ejemplo, la primitiva *Concatenar las cadenas "ABC" y "CD"* devuelve *"ABCD"*

- *Buscar la cadena A dentro de la cadena B a partir de la posición i* : Si la cadena A esta incluida en (o forma parte de) la cadena B a partir de la posición i de esta, entonces devuelve la posición j de la cadena B donde está situado el primer carácter de la cadena A. Si es este el caso se dice que A es una subcadena de B. En caso contrario devuelve el tamaño de la cadena B. Si omite la posición inicial i, entonces la búsqueda comienza desde el primer elemento de la cadena B (posición 0) de Por ejemplo, la primitiva *Buscar la cadena "CD" dentro de la cadena "ABCD" a partir de 0* devuelve *2*.

- *Extraer de la cadena A una subcadena desde la posición i a la j*: Devuelve una subcadena de caracteres de A comprendida entre las posiciones i y j. Si *long_A* es la longitud de la cadena de caracteres A, entonces las posiciones i *y* j han de cumplir las siguientes condiciones: $i,j \geq 0$ *&* $i,j \leq$ *long_A - 1 &* $i \leq j$. En caso que

estas condiciones no se cumplan, esta primitiva devuelve la cadena de caracteres vacía (cuyo tamaño es 0). Por ejemplo, la primitiva *Extraer una subcadena de "ABCDEF" comprendida entre las posiciones 1 y 3* devuelve *"BCD"*.

- *Extraer de la cadena A una subcadena desde la posición i que este comprendida entre las cadenas ini y fin*: Igual que la primitiva anterior, devuelve la primera subcadena de caracteres a partir de la posición *i* de la cadena *A*, comprendida entre el carácter posterior de la cadena *ini* y del carácter anterior a la cadena *fin*. En caso de no encontrar la subcadena, devuelve la cadena de caracteres vacía. Por ejemplo, la primitiva *Extraer de la cadena "ABCDEFGHI" una subcadena desde la posición 1 y que esté comprendida entre "AB","F"* devuelve *"CDE"*, ya que es la primera subcadena de caracteres que empieza a continuación de la cadena *"AB"* y que finaliza antes de la cadena *"F"*. Existe otra versión de esta primitiva (la anterior es la *por defecto*) donde las cadenas *ini* y *fin* son incluidas en la subcadena resultante. Para este el caso, la cadena devuelta en el ejemplo anterior seria: "ABCDEF" ("AB" y "F" incluidas).

- *Comparar dos cadenas A y B*: Devuelve *verdadero* si la cadena de caracteres *A* es igual (la misma) que la cadena de caracteres *B*. En caso contrario devuelve *falso*. Por ejemplo, la primitiva *Comparar las cadenas "ABC" y "ABD"* devuelve *falso*.

4.5.1.1. Obtención del fichero de descripción de trabajos global

Para obtener el shell-script global de descripción de trabajos, `tdp_console` divide el procesamiento de los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell` en dos partes:

1) Procesar sus posibles directivas especiales de SGE (explicadas en el capitulo 2), para obtener la nuevas versiones de estas y situarlas en el fichero de descripción de trabajos global.

2) Procesar las tres secciones especiales de `TDP-Shell` situadas en estos ficheros de descripción de trabajos, para obtener su nueva versión que será situada en el fichero de descripción de trabajos global.

Procesamiento de las directivas de SGE

El gestor de colas SGE define un conjunto de directivas especiales, identificadas por el

prefijo *#$*, cuya función es informarle de ciertas acciones específicas que debe realizar con el trabajo que va a ejecutar. Como estas directivas afectan al entorno de ejecución de los trabajos que debe ejecutar SGE, `tdp_console` debe procesar las que se encuentran en los shell-script de los componentes remotos de `TDP-Shell`, para obtener la nueva version de estas directivas e insertarlas al inicio del shell-script de descripción de trabajos global obtenido al ejecutar algoritmo anterior.

El hecho de que SGE posea una importante variedad de directivas especiales, cada una de ellas con sus características propias, implica que diseñar e implementar un algoritmo que permita procesarlas correctamente, sea un trabajo laborioso que requiera una importante cantidad de tiempo y dedicación. Debido a que este trabajo de tesis tiene, además del procesamiento de las directivas de SGE, otros objetivos importantes, se ha decidido que el algoritmo de procesamiento de directivas de SGE que utilice `tdp_console` se centre en los siguientes casos:

1) **Directivas de SGE que se les efectuará algún tipo de procesamiento**

- **Directiva SGE que define el nombre del trabajo**: El formato de esta directiva es **#$ -N nombre_del_trabajo**. Independientemente de los nombres que se hayan dado a los trabajos de los componentes remotos del entorno `TDP-Shell`, para identificar claramente que se trata de un trabajo propio de este entorno, la nueva directiva que define su nombre y que se sitúa en el shell-script de descripción de trabajos global es: *#$ -N TDP_Shell_job*.
- **Directivas de intercambio información**: Informan a SGE donde situar la información que van a necesitar o devolver los procesos definidos en sus trabajos. Estas directivas y su resultado al ser procesadas por `tdp_console` se resume en los siguientes puntos:
  - *Directiva SGE que define el nombre del fichero de error*:A través de la directiva **#$ -e fichero_salida_error**, el usuario informa al gestor de colas, de la dirección y el nombre del fichero donde debe situar la información sobre los posibles errores ocurridos durante la ejecución del trabajo. Igual que en el caso de la directiva que define el nombre del trabajo, para identificar claramente que se trata del fichero donde situar la posible salida del error

de un trabajo del entorno `TDP-Shell`, el valor de la nueva directiva SGE que identifica esta salida del error es: **#$ -e TDP_Shell.err**.

- *Directiva SGE que define el nombre de los ficheros de la salida estándar*: A través de esta directiva **#$ -o fichero_salida_standard**, el usuario informa a SGE del nombre del fichero donde debe situar la información de la salida estándar que producen los procesos definidas en el trabajo. Igual que en los casos anteriores, para identificar claramente que se trata de la salida estándar de un trabajo del entorno `TDP-Shell`, la nueva directiva SGE que lo identifica es: **#$ -o TDP_Shell.out**.

2) **Directivas de SGE que no se les efectuará ningún tipo de procesamiento**: El resto de directivas de SGE no sufren ninguna modificación, no son procesadas y son copiadas en el nuevo fichero de descripción de trabajos global con los mismos valores que en sus ficheros shell-script de descripción de trabajos origen. Si durante este proceso, `tdp_console` encuentra la misma directiva SGE definida en dos o más de estos ficheros, solo sitúa una de ellas en el fichero de descripción de trabajos global (la primera que ha procesado) e informa de este hecho al usuario. Con este aviso se advierte a este usuario que la directiva de SGE ignorada puede contener información importante para el gestor de colas y provocar una ejecución incorrecta de alguno de los componentes remotos del entorno `TDP-Shell`. Si este es el caso, el usuario puede modificar los ficheros shell-script de estos componentes para poder solucionar el problema.

Una vez vista la división del procesamiento de las directivas de SGE, se pueden mostrar los principales pasos del algoritmo que realiza este procesamiento, los cuales son:

Estructuras de datos globales a todo el algoritmo:

- La lista: *lista_directivas_SGE_procesa_TDP-Shell* contiene las las directivas de SGE que `tdp_console` procesa. Cada uno de sus elementos tiene el formato "-<*id_directiva*>", donde (<*id_directiva*> es el carácter que identifica a la directiva.

1: **para** cada cadena de caracteres:*cadena_shell-script_remoto*, perteneciente a los ficheros shell-script de los componentes remotos de `TDP-Shell` **hacer**

2: Extraer las cadenas de caracteres comprendidas entre "#$" (inicio directiva) y "CR" (final directiva SGE). En este caso "#$" y "CR" se incluyen.
3: Almacenarlas en la lista:*lista_directivas_SGE_shell-script* (almacena todas las posibles directivas SGE de los shell-script de los componentes remotos de `TDP-Shell`)
4: **fin para**
5: **si** *lista_directivas_SGE_shell-script* no está vacía **entonces**
6: **para** toda cadena de caracteres:*directiva_SGE_procesa_TDP-Shell* perteneciente a *lista_directivas_SGE_procesa_TDP-Shell* **hacer**
7: **para** toda cadena de caracteres:*directiva_SGE_shell-script* perteneciente a *lista_directivas_SGE_shell-script* **hacer**
8: Buscar la cadena de caracteres *directiva_SGE_procesa_TDP-Shell* en la *directiva_SGE_shell-script*
9: **si** *directiva_SGE_procesa_TDP-Shell* existe (la *primitiva buscar* devuelve una posición menor que el tamaño de la cadena *directiva_SGE_shell-script*) **entonces**
10: Insertar la *directiva_SGE_shell-script* en la lista:*lista_misma_directiva_SGE* (Almacena las directivas de SGE de los shell-script de los componentes remotos de `TDP-Shell` que puede procesar `tdp_console`)
11: Eliminar de la *lista_directivas_SGE_shell-script* la cadena *directiva_SGE_shell-script* (Esta directiva de SGE no tendrá que ser procesada más)
12: **fin si**
13: **fin para**
14: **si** *lista_misma_directiva_SGE* no está vacía **entonces**
15: Procesar las cadenas de caracteres de *lista_misma_directiva_SGE* para generar la directiva de SGE (identificada por directiva_SGE_procesa_TDP-Shell) que será concatenada a la cadena de caracteres del fichero de descripción de trabajos global: *cadena_caracteres_shell-script_trabajos_global*
16: **fin si**
17: **fin para**
18: Concatenar las cadenas de caracteres pertenecientes a *lista_directivas_SGE_shell-script* en la *cadena_caracteres_shell-script_trabajos_global* (resto de directivas SGE

no procesadas por `tdp_console`)

19: **fin si**

Procesamiento de las secciones del entorno TDP-Shell

Una vez obtenidas las posibles directivas especiales de SGE y situarlas en la cadena de caracteres *cadena_caracteres_shell-script_trabajos_global*, `tdp_console` procesa las secciones de `TDP-Shell` situadas en los shell-scripts de los componentes remotos de `TDP-Shell`, utilizando el siguiente algoritmo:

1: **para** cada cadena de caracteres:*cadena_shell-script_remoto*, perteneciente a los ficheros shell-script de los componentes remotos de `TDP-Shell` **hacer**
2: Extraer la cadena de caracteres: *cadena_pre_ejecutable*, comprendida entre "#TDP_begin_pre_block" y "#TDP_end_block"
3: **si** *cadena_pre_ejecutable* no está vacía (se ha declarado la zona pre-ejecutable) **entonces**
4: Insertar la *cadena_pre_ejecutable* en la lista:*lista_secciones_pre-ejecutables* (Esta lista contiene las sección pre-ejecutable de todos los shell-script de los componentes remotos de `TDP-Shell`).
5: **fin si**
6: **fin para**
7: **si** *lista_secciones_pre-ejecutables* no está vacía (se ha declarado alguna sección pre-ejecutable) **entonces**
8: Concatenar las cadenas de caracteres pertenecientes a la lista_secciones_pre-ejecutables en la *cadena_caracteres_shell-script_trabajos_global* (Esta última cadena tiene ahora las posibles Directivas SGE y la posible sección pre-ejecutable global).
9: **fin si**
10: Extraer de la cadena de caracteres del shell-script del componente `tdp_agent`, la cadena de caracteres: *cadena_ejecutable_global* comprendida entre "#TDP_exec" y "CR" (final de línea).

Comentario: Esta cadena contiene el ejecutable (y sus posibles argumentos) del nuevo shell-script de descripción de trabajos global.Este ejecutable es el del fichero shell-script del componente `tdp_agent`, debido a que es el gestor de colas SGE (uno de sus componentes remotos) el encargado de ponerlo en ejecución (como se ha visto en

el apartado 4.3).

11: Para las *secciones post-ejecutables*, seguir el mismo proceder que en el caso de las pre-ejecutables: Obtener la nueva cadena de caracteres del *entorno de ejecución posterior a la ejecución de los componentes remotos de* ***TDP-Shell***, concatenando, en caso de su existencia, las cadenas de caracteres de cada sección post-ejecutable de los ficheros shell-script de dichos componentes (situadas entre *#TDP_begin_post_block* y *#TDP_end_block*). En caso de la obtención de esta cadena (existe alguna sección post-ejecutable), concatenarla con la cadena de caracteres *cadena_caracteres_shell-script_trabajos_global* y guardarla en el archivo de descripción de trabajos global.

4.5.1.2. Obtención de las tuplas especiales

Al procesar los los archivos de descripción de trabajos de los componentes remotos del entorno `TDP-Shell`, `tdp_console` también puede obtener cierta información que puede ser interesante para las funciones que ha de realizar el componente `tdp_agent`. Esta información es:

a) El ejecutable del proceso de la aplicación de usuario y sus posibles argumentos.

b) El ejecutable del componente remoto de la herramienta y sus posibles argumentos.

c) El nombre del archivo `TDP-Shell` script para el componente `tdp_agent`.

Esta información esta situada en las cadenas de caracteres (o líneas) de las secciones ejecutables de los shell-scripts de los componentes remotos del entorno `TDP-Shell`. Para obtenerla, `tdp_console` procesa, estas cadenas de caracteres de la siguiente manera:

1) Extrae la primera cadena de caracteres situada entre el principio de la línea y los caracteres *espacio en blanco (espacio)* o *tabulador*. Esta subcadena contiene la dirección y el nombre del ejecutable del proceso principal del archivo shell-script.

2) Extrae la cadena de caracteres comprendida entre el carácter espacio o tabulador posterior del ejecutable y el carácter 'CR' (final línea ejecutable). Esta cadena de caracteres contiene los argumentos del ejecutable principal.

3) El nombre del archivo `TDP-Shell` script del componente `tdp_agent`, normalmente es pasado como uno de los argumentos de su ejecutable, identificado por el prefijo *-tf*.

Para encontrarlos se busca, en la cadena de caracteres de los argumentos del ejecutable del `tdp_agent` (obtenida en el punto anterior), las posiciones de la cadena "-tf:" (principio argumento del `TDP-Shell` script) y posteriormente de los caracteres espacio, tabulador o CR (los cuales marcan el final del argumento). Una vez obtenidas, extraer la cadena de caracteres comprendida entre estas dos posiciones, la cual contiene el nombre del archivo `TDP-Shell` script para el componente `tdp_agent`.

Una vez extraídas estas cadenas de caracteres, `tdp_console` las sitúa, en el espacio de atributos tdp, para que sean accesibles por el componente `tdp_agent`. Para ello utiliza las siguientes tuplas especiales:

a) **TDP_USER_EXEC y TDP_USER_ARGS**: Los valores de estas tuplas contienen respectivamente, el ejecutable del proceso de la aplicación de usuario y sus posibles argumentos.

b) **TDP_TOOL_EXEC y TDP_TOOL_ARGS** :Sus valores contienen respectivamente, el ejecutable del componente remoto de la herramienta y sus posibles argumentos.

c) **TDP_SHELL_SCRIPT_AGENT**: El valor de esta tupla contiene el nombre del archivo `TDP-Shell` script para el componente `tdp_agent`.

Si `tdp_console` no consigue encontrar estas informaciones (por ejemplo, no se le ha pasado el shell-script o el ejecutable no tiene argumentos), entonces sitúa **NOTHING** como valor de la tupla especial que debería contener la información no encontrada.

4.5.1.3. Ejemplo básico

A continuación se muestra un sencillo ejemplo ilustrativo de como `tdp_console` procesa los tres archivos de descripción de trabajos de los componentes remotos de `TDP-Shell`. De este procesamiento obtiene el nuevo archivo global de descripción de trabajos y los valores de las tuplas especiales del entorno `TDP-Shell` para los componentes `tdp_agent`.

Archivos shell-script de los componentes remotos del entorno TDP-Shell

En los *ficheros FDT 4.1, FDT 4.2 y FDT 4.3* se muestran los tres shell-script de descripción de trabajos que describen el entorno de ejecución de cada componente remoto de `TDP-Shell`.

FDT 4.1 Fichero de descripción de trabajos de SGE para la aplicación de usuario

```
1: #!/bin/sh
2: # /home/user/user_aplication.sh
3: #$ -N user_application
4: #$ -o user.out
5: #$ -e user.err
6: #TDP_begin_pre_block
7: scp user@user_host:/home/user/bin/demo_prog demo_prog
8: chmod u+x demo_prog
9: #TDP_end_block
10: #TDP_exec
11: ./demo_prog
12: #TDP_begin_post_block
13: rm demo_prog
14: #TDP_end_block
```

FDT 4.2 Fichero de descripción de trabajos de SGE para el componente remoto de la herramienta

```
1: #!/bin/sh
2: # /home/user/paradynd.sh
3: #$ -N paradynd
4: #$ -o paradynd.out
5: #$ -e paradynd.err
6: #TDP_exec
7: /opt/paradyn/bin/paradynd
```

Como puede observarse, la *sección pre-ejecutable* del fichero shell-script de la aplicación de usuario, informa al gestor de colas SGE que copie su ejecutable, de la maquina local del usuario, al directorio compartido que este usuario posee en las máquinas del cluster (accesible a todas las máquinas, p.e vía NFS) y que a continuación lo declare como ejecutable. La *sección post-ejecutable* de este fichero shell-script informa a SGE que borre el ejecutable de la aplicación de usuario que previamente se ha copiado en las máquinas del cluster. El shell-script del componente remoto de la herramienta

FDT 4.3 Fichero de descripción de trabajos de SGE para el componente tdp_agent

```
1: #!/bin/sh
2: # /home/user/tdp_scripts/tdp_agent.sh
3: #$ -N tdp_agent
4: #$ -o tdp_agent.out
5: #$ -e tdp_agent.err
6: #TDP_begin_pre_block
7: scp user@user_host:/home/user/tdp_files/tdp_agent.tdp tdp_agent.tdp
8: #TDP_end_block
9: #TDP_exec
10: /opt/tdp/tdp_agent -cf:/opt/tdp/config/tdp_config.cfg -tf:tdp_agent.tdp
11: #TDP_begin_post_block
12: rm tdp_agent.tdp
13: #TDP_end_block
```

solo declara la *sección del ejecutable*, la cual supone que este ejecutable ya está instalado correctamente en las máquinas del cluster. Por lo tanto y a no estar definidas las secciones *pre ni post ejecutables*, el gestor de colas SGE, no debe realizar ninguna acción antes y después de la ejecución del componente remoto de la herramienta. En la *sección pre-ejecutable* del fichero shell-script del componente `tdp_agent` se informa al gestor de colas SGE que se copie el fichero `TDP-Shell` script que necesita `tdp_agent`, de la máquina local del usuario, al directorio compartido que este usuario posee las máquinas del cluster. En la *sección del ejecutable* de este shell-script, se puede observar que al componente `tdp_agent`, a parte de su archivo `TDP-Shell` script, también se le pasa como argumento, el archivo de configuración del entorno `TDP-Shell`. Este archivo contiene la información de conexión con *el servidor del espacio de atributos tdp* y de los *plug-ins*. Por ultimo, este archivo shell-script para el tdp_agent, también informa a SGE, en su *sección post-ejecutable*, que después de ejecutar el componente `tdp_agent`, borre su archivo `TDP-Shell` script del directorio del usuario situado en la máquina remota.

Shell-script global de descripción de trabajos

En la fichero 4.4 se muestra el shell-script generado por `tdp_console` después de haber procesado los tres ficheros de descripción de trabajos vistos en el punto anterior.

En el se pueden observar los siguiente puntos:

FDT 4.4 Fichero de descripción de trabajos global de SGE obtenido por tdp_console

```
 1: #!/bin/sh
 2: #$ -N TDP_Shell_job
 3: #$ -o TDP_Shell.out
 4: #$ -e TDP_Shell.err
 5: scp user@user_host:/home/user/bin/demo_prog demo_prog
 6: chmod u+x demo_prog
 7: scp user@user_host:/home/user/tdp_files/tdp_agent.tdp tdp_agent.tdp
 8: /opt/tdp/tdp_agent -cf:/opt/tdp/config/tdp_config.cfg -tf:tdp_agent.tdp
 9: rm demo_prog
10: rm tdp_agent.tdp
```

1) En su sección *sección pre-ejecutable* (líneas 5 a 7) se define el entorno de ejecución global previo que necesitan los componentes remotos del entorno `TDP-Shell` para su correcta ejecución. Este entorno de ejecución informa a SGE que copie, de la máquina local del usuario a su directorio compartido en las máquinas del cluster, el ejecutable del proceso de usuario y el fichero `TDP-Shell` script que necesita `tdp_agent`.

2) Su *ejecutable principal* (línea 8) es el mismo que el del shell-script del componente `tdp_agent` (fichero FDT 4.3, línea 7), de esta manera es el gestor de colas SGE quien pone este componente de `TDP-Shell` en ejecución en una máquina del cluster.

3) Su *sección post-ejecutable* (líneas 9 a 10) informa a SGE del entorno de ejecución global posterior de los componentes remotos del entorno `TDP-Shell`. En el se informa que cuando se termine la ejecución del componente `tdp_agent` (y en consecuencia el proceso de monitorización), se borren los archivos copiados en la sección pre-ejecutable, debido a que ya no son necesarios.

En el archivo global de descripción de trabajos de este ejemplo también pueden observarse los nuevos valores de las directivas específicas de SGE para el entorno `TDP-Shell`. En los tres archivos de descripción de trabajos de los componentes remotos de `TDP-Shell`, se han definido las directivas de SGE que identifican el nombre del trabajo (#$ -N), los nombres de los ficheros de la salida estándar (#$ -o) y de error (#$ -e). Como se ha explicado en el punto 4.5.1.1, `tdp_console` realiza un procesamiento especial con estas directivas, asignándoles unos valores predefinidos, independientemente

de los que tuviesen en sus ficheros shell-script originales. Estos nuevos valores son: *TDP-Shell_job* para el nombre del trabajo, *TDP-Shell_out* para la salida estándar y *TDP-Shell_err* para la salida de los posibles errores.

Tuplas especiales del entorno TDP-Shell

Como se ha explicado en el punto 4.5.1.2, del procesamiento de los archivos de descripción de trabajos de los componentes remotos de `TDP-Shell`, `tdp_console` también obtiene información útil para el componente `tdp_agent`, la cual sitúa en el espacio de atributos tdp a través de unas tuplas espaciales. Para este ejemplo, los valores de estas tuplas son:

a) **TDP_USER_EXEC**: *demo_prog*. Nombre del ejecutable de la aplicación de usuario

b) **TDP_USER_ARGS**: *NOTHING*. Posibles argumentos del ejecutable de la aplicación de usuario

c) **TDP_TOOL_EXEC**: */opt/paradyn/bin/paradynd*. Nombre del ejecutable del componente remoto de la herramienta.

d) **TDP_TOOL_ARGS**: *NOTHING*. Posibles argumentos del ejecutable del componente remoto de la herramienta.

e) **TDP_SHELL_SCRIPT_AGENT**: *-tf:tdp_agent.tdp*. Archivo `TDP-Shell` script para el componente `tdp_agent`.

4.5.2. Condor

El gestor de colas Condor utiliza un formato propio de ficheros de descripción de trabajos. Estos ficheros, igual que en el caso del gestor de colas SGE, están formados por un conjunto de cadenas de caracteres (también son ficheros de texto), pero a diferencia de las este último gestor de colas, estas cadenas describen unos comandos especiales de Condor cuyo formato es *identificador = valores*. A través del campo *identificador*, se informa a Condor del tipo de acción que debe realizar y mediante el campo *valores* se notifican los valores asociados a esta acción. Por ejemplo, si el ejecutable principal del trabajo es *user_exec*, el comando de Condor que lo identificaría

seria: *Executable = user_exec*. El formato de estos comandos especiales de Condor, favorece el procesamiento que realiza el `tdp_console` de sus ficheros de descripción de trabajos. Principalmente, `tdp_console` debe buscar, en estos ficheros, los diferentes valores que tiene un mismo comando de Condor, procesarlos y obtener el nuevo valor de este comando para el fichero de descripción de trabajos global. Esta búsqueda de los valores de los comandos de Condor, también le permite a `tdp_console` obtener el valor de las tuplas especiales con información útil para `tdp_agent`. Por lo tanto y a diferencia del gestor de colas SGE, en el caso de Condor no es necesario insertar en los ficheros de descripción de trabajos, ninguna directiva especial del entorno `TDP-Shell`.

Igual que se ha hecho con el gestor de colas SGE, en los siguientes puntos se explicará, mas detalladamente, como `tdp_console` obtiene el fichero de descripción de trabajos global para Condor y las tuplas especiales del entorno `TDP-Shell`.

4.5.2.1. Obtención del fichero de descripción de trabajos global

El procesamiento que realiza `tdp_console` de los ficheros de de descripción de trabajos de los componentes remotos de `TDP-Shell`, se basa en el siguiente algoritmo:
Previo al algoritmo:

- La lista: *lista_id_comandos_Condor_procesa_TDP-Shell* contiene los identificadores de los comandos de Condor que `tdp_console` procesa. El motivo por el que se han escogido estos comandos y como se procesan, se explicará a continuación de este algoritmo.

- Para simplificar y hacer más leíble el algoritmo a los *ficheros de descripción de trabajos de los componentes remotos del entorno* `TDP-Shell` se les denominará: *ficheros de descripción de trabajos remotos de* `TDP-Shell`.

- Debido a que el gestor de colas Condor puede interpretar sus comandos independientemente si están escritos en mayúsculas como en minúsculas, `tdp_console` primero pasa a minúsculas las cadenas de caracteres de los identificadores de los comandos de los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell`.

1: **para** cada comando de Condor:*id_comando_procesa_TDP-Shell* perteneciente a la *lista_comandos_Condor_procesa_TDP-Shell* **hacer**
2: **para** cada cadena de caracteres:*cadena_fichero_remoto*, perteneciente a los ficheros de descripción de trabajos remotos de `TDP-Shell` **hacer**
3: **para** cada línea:*linea_actual* de la *cadena_fichero_remoto Comentario:* Una línea esta comprendida entre el inicio de la cadena de caracteres y un 'CR' (1ª línea) o entre dos 'CR' o entre un 'CR' y el final de la cadena de caracteres (última línea) **hacer**
4: Extraer la cadena de caracteres: *id_comando_fichero_remoto* comprendida entre el principio de línea y "=" (separador de los campos de los comandos de Condor).
5: Buscar la cadena *id_comando_procesa_TDP-Shell* en *id_comando_fichero_remoto*
6: **si** la cadena *id_comando_procesa_TDP-Shell* existe **entonces**
7: Extraer la cadena:*valor_comando_fichero_remoto* comprendida entre las cadenas "=" y "CR" de la *linea_actual* (contiene el valor del comando de Condor)
8: Insertar la cadena *valor_comando_fichero_remoto* en la lista:*lista_valores_comandos_Condor* (contiene los valores de un mismo comando Condor)
9: SALIR DEL FOR (el de las líneas, no hace falta seguir buscando porque se ya ha encontrado el comando de Condor que se buscaba: el identificado por *id_comando_procesa_TDP-Shell*)
10: **fin si**
11: **fin para**
12: **fin para**
13: Procesar las cadenas de caracteres situadas en la *lista_valores_comandos_Condor* para obtener el nuevo valor:*valor_comando_procesa_TDP-Shell* del comando identificado por *id_comando_procesa_TDP-Shell*
14: Concatenar *id_comando_procesa_TDP-Shell* con "=" y con *valor_comando_procesa_TDP-Shell* para obtener la cadena de caracteres del nuevo comando de Condor: *comando_Condor_global*

15: Concatenar la cadena *comando_Condor_global* con la cadena: *cadena_caracteres_fichero_trabajos_global* (Contiene la cadena de caracteres del fichero de descripción de trabajos global)
16: **fin para**
17: Guardar la *cadena_caracteres_fichero_trabajos_global* en el fichero de descripción de trabajos global.

Igual que sucede con el procesamiento de las directivas de SGE, Condor también posee una gran variedad de comandos, esto hace que diseñar e implementar un algoritmo que permita a `tdp_console` procesarlos todos sea un trabajo que requiera una gran cantidad de tiempo y dedicación. Igual que en el caso de SGE (debido a que este trabajo de tesis tiene también otros objetivos), se ha decidido acotar el procesamiento de los comandos de Condor a los más utilizados, los cuales su muestran a continuación:

- **Executable:** Este comando indica a Condor el nombre del ejecutable principal del trabajo. Como en el entorno `TDP-Shell` es el gestor de colas el encargado de ejecutar los `tdp_agents` en las máquinas del cluster que el controla, `tdp_console` obtendrá el ejecutable del fichero de descripción de trabajos global leyendo el valor del *comando executable* situado en el fichero de descripción de trabajos del `tdp_agent`. Por defecto, Condor copia el fichero del ejecutable de la máquina local del usuario (que realiza la petición a Condor) al directorio de trabajo de este gestor de colas situado en la máquina del cluster. Por lo tanto y a menos que no se indique lo contrario (ver el siguiente comando de Condor), los ejecutables de los otros ficheros de descripción de trabajos de los componentes remotos del entorno `TDP-Shell` pasados a `tdp_console`, también se deberán copiar de la máquina local del usuario a la del cluster, utilizando el *comando transfer_input_files*, el cual será explicado en los siguientes puntos.

- **Transfer_executable:** A través de este comando se informa a Condor si debe copiar (valor YES) o no (valor NO) el ejecutable del trabajo desde la máquina que lo somete (normalmente la local del usuario) al directorio de trabajo de Condor de la máquina del cluster. Si en alguno de los ficheros de descripción de trabajo de los componentes remotos del entorno `TDP-Shell`, `tdp_console` encuentra que el valor de este comando es NO, entonces no realiza la copia remota

del ejecutable de este fichero explicada en el *comando Executable* (utilizando el *comando transfer_input_files*).

- **Arguments:** Este comando es utilizado para informar a Condor de la lista de los posibles argumentos del ejecutable del fichero de descripción de trabajos. Como en el entorno `TDP-Shell`, el ejecutable del fichero de descripción de trabajos global es el del componente `tdp_agent`, el valor del *comando Arguments* obtenido por `tdp_console` para este fichero global, contiene el mismo valor que el del *comando Arguments* situado en el fichero de descripción de trabajos del componente `tdp_agent`.

- **Error:** A través de este comando se informa a Condor del nombre del fichero, situado en la máquina local del usuario, donde debe situar la salida de los errores que produce el ejecutable del trabajo que está ejecutando en las máquinas del cluster. En el caso de `TDP-Shell` (y como también se vio para SGE) independientemente de los diferentes valores que pueda tener este comando en los ficheros de descripción de trabajos de los componentes remotos del entorno `TDP-Shell`, `tdp_console` asigna el valor *TDP_Shell.error* (y por lo tanto el del nombre del fichero de error) al *comando Error* del nuevo fichero de descripción de trabajos global.

- **Output:** Con este comando se informa a Condor del nombre del fichero, situado en la máquina local del usuario, donde debe situar la información de la salida standard que produce el ejecutable del trabajo que se ejecuta en las máquinas cluster. Como en el caso de la salida del error, el valor de este comando para el nuevo fichero de descripción de trabajos global, será siempre: *TDP_Shell_out*.

- **Log:** A través de este comando se informa a Condor del nombre del fichero, en la máquina local del usuario, donde debe situar la información relacionada con la ejecución del trabajo en las máquinas del cluster (En que maquinas se ha ejecutado el trabajo, como ha terminado o los bytes enviados y recibidos). Como en el caso de los dos comandos anteriores, independientemente de los valores que tenga en los ficheros de descripción de trabajos de los componentes remotos del entorno `TDP-Shell`, el valor de este comando (obtenido por `tdp_console`) para el nuevo fichero de descripción de trabajos global será siempre: *TDP_Shell.log*.

- **Should_transfer_files:** Este comando indica a Condor si debe o no transferir ficheros a/o desde las máquinas remotas. Si su valor es *YES*, siempre se transfieren los ficheros, si es *NO*, entonces no se transfieren (se presupone la existencia de un sistema de ficheros compartido en el cluster, como NFS o AFS). Si el valor de este comando es IF_NEEDED, entonces se indica a Condor que solo transfiera los ficheros, si no existe un un sistema de ficheros compartido en el cluster donde se va ha ejecutar el trabajo. La política que sigue `tdp_console` para asignar el valor de este comando en el nuevo fichero de descripción de trabajos global es la siguiente:

  a) Si el valor de los comandos *should_transfer_files* de los ficheros de descripción de trabajos de los componentes remotos del entorno `TDP-Shell` es el mismo (todos son YES, NO o IF_NEEDED), entonces el valor de este comando en el fichero de descripción de trabajos global será dicho valor común.

  b) Si los diferentes valores de los comandos *should_transfer_files* son diferentes (algunos son YES, otros NO o IF_NEEDED), el nuevo valor para este comando en el fichero de descripción de trabajos global será IF_NEEDED. Esto es debido a que la alternancia de valores YES o NO (o IF_NEEDED) implica que hay maquinas del cluster con un sistema de ficheros compartido (valores NO o IF_NEEDED) y máquinas que no lo poseen (valores YES). Por lo tanto, el nuevo valor de IF_NEEDED para este comando informa a Condor que no transfiera los ficheros en las maquinas con un sistema de ficheros compartido y que si que realiza la transferencia en las máquinas que no lo posean.

Como se ha explicado en el punto del comando *Executable*, a parte del ejecutable del archivo de descripción de trabajo del `tdp_agent`, también se han de transferir los ejecutables de los archivos de la aplicación de usuario y del componente remoto de la herramienta, al menos que no se indique lo contrario (comando Transfer_executable). En este caso, `tdp_console` puede encontrarse con dos casos:

a) Si el comando *should_transfer_files* ya existe en el en el nuevo fichero de descripción de trabajos global, entonces `tdp_console` deja el valor actual de dicho comando si ya informa de una transferencia (valores YES o IF_NEEDED). En caso que no la indique (valor NO), `tdp_console` cambia su valor por IF_NEEDED para indicar a Condor que se encargue, en caso

necesario (dependiendo de la existencia de un sistema de ficheros compartido), de la transferencia de los ejecutables de la aplicación de usuario y del componente remoto de la herramienta.

b) El comando *should_transfer_files* no existe en el en el nuevo fichero de descripción de trabajos, en este caso, `tdp_console` lo crea en este nuevo fichero con el valor IF_NEEDED para informar a Condor que se encargue correctamente de la gestión de la transferencia de los ejecutables de la aplicación de usuario y del componente remoto de la herramienta.

- **When_to_transfer_output:** Este comando indica a Condor cuando ha de transferir a la máquina que ha solicitado el trabajo (normalmente la máquina local del usuario), los ficheros de salida que genera dicho trabajo. Si su valor es ON_EXIT, entonces la transmisión se realiza cuando el trabajo finaliza por si mismo, si su valor es ON_EXIT_OR_EVICT, la transferencia se realiza tanto si el trabajo finaliza por si mismo como si es desalojado por Condor antes de su finalización. Para el nuevo valor de este comando en el fichero de descripción de trabajos global, `tdp_console` le asigna ON_EXIT en el caso que todos los valores de los *comandos when_to_transfer_output*, situados en los ficheros de descripción de trabajos de los componentes remotos del entorno `TDP-Shell`, tengan ese mismo valor. Si el valor de este comando en alguno de estos ficheros es ON_EXIT_OR_EVICT, entonces `tdp_console` asigna este ultimo valor al *comando when_to_transfer_output* del fichero de descripción de trabajos global. Con esta manera de proceder se consigue que los componentes remotos de `TDP-Shell` que lo necesiten, puedan transferir sus ficheros de salida en el caso que Condor los desaloje de la máquina donde se están ejecutando.

- **Transfer_input_files:** Con este comando se informa a Condor del nombre de los ficheros que se han de transmitir, antes de la ejecución del trabajo, desde la máquina local del usuario, al directorio de trabajo de Condor situado en la máquina del cluster. Estos nombres de ficheros son pasados como una lista de cadenas de caracteres (cada una es un nombre de fichero), separadas por el carácter 'coma' (,). El valor de este comando que obtiene `tdp_console` para el fichero de descripción de trabajos global, es la concatenación de las listas de los valores de los *comandos transfer_input_files* de los ficheros de descripción de trabajos de los componentes

remotos de `TDP-Shell`. Entre lista y lista concatenada, `tdp_console` sitúa una coma para indicar la separación de dos nombres de ficheros (el último de la lista anterior y el primero de la siguiente) y mantener de esta forma los requisitos del comando.
Con esta manera de proceder, `tdp_console` informa al gestor de colas Condor de todos los ficheros que necesitan estos componentes remotos de `TDP-Shell`, antes de su ejecución en la máquina del cluster.

- **Transfer_output_files:** Con este comando se informa a Condor de los ficheros que se han de copiar desde las máquinas del cluster a la máquina que ha realizado el envío del trabajo. Igual que en el caso del comando anterior, `tdp_console` obtiene el nuevo valor de este comando para el fichero de descripción de trabajos global, concatenando las listas de ficheros de los *comandos transfer_output_files* que proporcionan los ficheros de descripción de trabajos de de los componentes remotos de `TDP-Shell`.

- **Universe:** A través de este comando se especifican a Condor los diferentes entornos de ejecución para los trabajos. Estos entornos de ejecución o universos pueden ser: *vanilla, standard, scheduler, local, grid, mpi, java y vm.* Cada uno de ellos ofrece sus propias características, por ejemplo, el universo *standard* permite que en el trabajo se puedan instalar puntos de chequeo y realizar llamadas remotas al sistema (enlazando el ejecutable del trabajo con el programa Condor_compile y las librerías de Condor), el *universo vanilla* (que es el por defecto) permite ejecutar los trabajos que no requieran o necesiten utilizar el *universo standard* (por ese motivo también se les denominan *trabajos normales)* y el *universo mpi* está preparado para la ejecución de trabajos que utilicen el protocolo MPI.
Como se puede observar, cada uno de estos universos está pensado para la ejecución de trabajos cuyos ejecutables tienen unas características específicas, lo que implica que un trabajo definido para un tipo de estos universos, normalmente no se podrá ejecutar en un tipo diferente (por ejemplo un trabajo vanilla no puede ejecutarse en un universo mpi). Por este motivo, al procesar los ficheros de descripción de trabajos de de los componentes remotos de `TDP-Shell`, `tdp_console` espera que sus *comandos universe* tengan el mismo valor, o lo que es mismo, que pertenezcan al mismo universo. En caso contrario, informa de esta incompatibilidad

al usuario.

- **Requirements:** A través de este comando se informa a Condor de los requerimientos que han de cumplir las máquinas del cluster para que puedan ejecutar el trabajo. Estos requerimientos se expresan como una *expresión booleana*, la cual será evaluada a *verdadero* (la máquina cumple los requisitos) o *falso* (la máquina no los cumple). Esta *expresión booleana* esta formada por un conjunto de expresiones condicionales (A op B, donde op es $>$, $<$, $==$, $\neq$, $\geq$ y $\leq$) unidas por el operador lógico && (And). Por ejemplo, el *comando requeriments* que comunica a Condor que el trabajo necesita máquinas con más de 50 Mbytes de memoria y una arquitectura x86_64, es el siguiente: *Requirements = Memory* $\geq$ *50 && Arch == "X86_64"*.

  El componente `tdp_console` genera la nueva versión del *comando requeriments* del el fichero de descripción de trabajos global, uniendo, con el operador booleano *&&*, las expresiones booleanas de los respectivos *comandos requeriments* de los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell`. Al realizar este proceso, puede suceder que la nueva versión de este comando contenga expresiones booleanas (o requerimientos) incompatibles. Estas expresiones no son comprobadas por `tdp_console`, siendo función del usuario detectarlas de antemano (antes de utilizar el entorno `TDP-Shell`). Se ha optado por esta solución, debido a que la gran variedad de atributos que pueden contener las expresiones booleanas (Memoria, arquitectura, sistema operativo, MIPS, etc), genera un elevado número de combinaciones posibles entre ellas (validas o invalidas). Esto implica una gran cantidad de tiempo en su evaluación y un importante *componente de inteligencia* para verificar si cada combinación de requerimientos tiene o no sentido. Ejemplo de una combinación de atributos errónea es cuando un usuario ha utilizado, por un despiste, el fichero de descripción de trabajo de su aplicación a monitorizar para la *arquitectura "SUN4u"* (contiene el comando *Requirements = Arch == " SUN4u"*) y el de la herramienta de monitorización para *la arquitectura "x86_64"* (*Requirements = Arch == "x86_64"*). Con estas declaraciones, `tdp_console` genera el siguiente comando: *Requirements = Arch == "SUN4u" && Arch == " X86_64"* el cual informa a Condor que para ejecutar los componentes remotos del entorno `TDP-Shell` es necesario que las maquinas pertenezcan tanto a la arquitectura de

SUN como a la x86_64, cosa que no es posible, ya que cada máquina pertenece a una arquitectura determinada.

- **Rank:** A través de este comando se informa a Condor del rango (o conjunto) de máquinas a escoger de entre las que cumplen los requerimientos del trabajo, por lo tanto, este comando expresa preferencias. Para seleccionar este rango de máquinas, se utilizan las *expresiones flotantes*, las cuales, al ser evaluadas en cada maquina que cumple los requerimientos del trabajo, producen unos valores numéricos en punto flotante. Las maquinas que obtengan los valores del rango más grandes, serán las seleccionadas para ejecutar el trabajo en ellas. Cuando se evalúa una *expresión flotante* en una máquina concreta hay que tener en cuenta los siguientes casos especiales:

  - Si se obtiene un valor UNDEFINED o ERROR, entonces se asigna el valor de *0.0* para el rango de esa maquina concreta.
  - Si la expresión es booleana, entonces se evaluará 1.0 si es verdadera o a 0.0 si es falsa.

  Por ejemplo si un usuario desea que su trabajo se ejecute en alguna de las siguientes máquinas: *maquina1.cluster, maquina2.cluster o maquina3.cluster*, entonces definirá el siguiente *comando rank* en su fichero de descripción de trabajo: *Rank = (machine == "maquina1.cluster") || (machine == "maquina2.cluster") || (machine == "maquina1.cluster").* Estas expresiones booleanas, devuelven *verdadero* si es la máquina preguntada o falso en caso contrario, darán un valor de 1.0 cuando se evalúen en alguna de las máquinas donde se desea ejecutar el trabajo (por el operador lógico ||, *OR*) y un valor de 0.0 en el resto de máquinas.
  Para generar la nueva versión del *comando rank* del fichero de descripción de trabajos global, el componente `tdp_console` sigue un procedimiento similar al utilizado en el *comando requeriments*, el cual consiste en unir, en caso de que estén definidos, los diferentes valores de los comandos rank de los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell`. Esta unión la realiza a través del operador lógico ||, debido a que es más genérico (uno u otro) que otros operadores lógicos como el && (todos se han de cumplir). También como en el caso del *comando requeriments* (gran variedad de posibles combinaciones), `tdp_console`

no comprueba la validez del nuevo valor del *comando rank* del fichero de descripción de trabajos global, siendo función del usuario asegurarse de este hecho.
Por ejemplo, si en el fichero de descripción de trabajos de la aplicación de usuario a monitorizar se definen los siguientes requerimientos:

Requirements = Memory ≥ 128 && OpSys == "LINUX" Rank = (machine == "maquina1.cluster")|| (machine == "maquina2.cluster")

Los cuales informan a Condor que la memoria de la máquina donde se ejecute la aplicación ha de tener más de 128 Mbytes, su sistema operativo ha de ser Linux y que preferiblemente se ha de ejecutar en las máquinas *maquina1.cluster o maquina2.cluster* (las cuales tienen mas de 128 Mbytes y su sistema operativo es Linux). Además, si en el fichero de descripción de trabajos del componente remoto de la herramienta de monitorización se define el siguiente requerimiento: *Rank = (Memory ≥ 256)* El cual informa que preferentemente la máquina donde se ejecute este componente ha detener mas de 256 Mbytes. Suponiendo que en el fichero de descripción de trabajos del `tdp_agent` no se definen requerimientos especiales, los *comandos rank y requeriments* de fichero de descripción de trabajos global generado por `tdp_console`, tendrán los siguientes valores:
Requirements = Memory ≥ 128 && OpSys == "LINUX"
Rank = ((machine == "maquina1.cluster") || (machine == "maquina2.cluster")) || (Memory ≥ 256)
Si alguna de las máquinas *maquina1.cluster o maquina2.cluster* tiene más de 256 Mbytes de memoria, se podrán ejecutar correctamente los tres componentes del entorno `TDP-Shell` en alguna de ellas, debido a que se cumplirán los todos requisitos para ello, tener mas de 128 Mbytes de memoria para la aplicación de usuario (la máquina tiene mas de 256 Mbytes) y que su sistema operativo sea Linux.

4.5.2.2. Obtención de las tuplas especiales

Igual que para el gestor de colas SGE, `tdp_console` puede obtener cierta información de los archivos de descripción de trabajos de los componentes remotos del entorno `TDP-Shell` definidos para Condor. Para ello, `tdp_console` busca las líneas de estos archivos que contienen los comandos *Executable y Arguments*. Una vez encontradas,

obtiene los valores de los comandos extrayendo la cadena de caracteres situada entre los carácteres '=' y el CR del final de línea. Con estas cadenas de caracteres, `tdp_console` sigue el mismo procedimiento que utiliza para el gestor de colas SGE, las sitúa en el espacio de atributos tdp a través de las tuplas:

a) **TDP_USER_EXEC y TDP_USER_ARGS**: Para el ejecutable de la aplicación de usuario y sus posibles argumentos.

b) **TDP_TOOL_EXEC y TDP_TOOL_ARGS** :Para el ejecutable del componente remoto de la herramienta y sus posibles argumentos.

c) **TDP_SHELL_SCRIPT_AGENT**: Para el nombre del archivo `TDP-Shell` script para el componente `tdp_agent`.

Igual que en el caso de SGE, si `tdp_console` no consigue encontrar estas informaciones en los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell`, entonces sitúa **NOTHING** como valor de la tupla especial que debería contener la información no encontrada.

4.5.2.3. Ejemplo básico

En esta sección se explica como tdp_cosole obtiene el fichero de descripción de trabajos global para el gestor de colas Condor, a partir de los archivos de descripción de trabajos de los mismos componentes remotos del entorno `TDP-Shell` utilizados en el ejemplo de SGE (sección 4.5.1.3).

Archivos descripción de trabajos del entorno TDP-Shell

En los *ficheros FDT 4.5, FDT 4.6 y FDT 4.7* se muestran los tres ficheros de descripción de trabajos de los componentes remotos del entorno TDP-Shell que procesa `tdp_console`

En los ficheros de descripción de trabajos del componente remotos de la herramienta de monitorización y del `tdp_agent` se puede observar como se informa al gestor de colas Condor, que no copie los ejecutables de la maquina local del usuario a la máquina del cluster. Esto es realizado a través del comando *Transfer_executable = NO* y es debido a que estos ficheros ejecutables, por estar situados en los directorios compartidos:

FDT 4.5 Fichero de descripción de trabajos de Condor para la aplicación de usuario

```
1: Universe = vanilla
2: Executable = /home/user/bin/demo_prog
3: Output = user.out
4: Error = user.err
5: Log = user.log
6: Queue
```

FDT 4.6 Fichero de descripción de trabajos de Condor para el componente remoto de la herramienta

```
1: Universe = vanilla
2: Executable = /opt/paradyn/bin/paradynd
3: Transfer_executable = NO
4: Output = paradynd.out
5: Error = paradynd.err
6: Log = paradynd.log
7: Queue
```

/opt/paradyn/bin/ y /opt/TDP-Shell/, ya son accesibles desde las máquinas del cluster. También puede observarse, que en el archivo de descripción de trabajos del componente `tdp_agent` se informa a Condor, a través del comando *Transfer_input_files*, que copie el archivo `TDP-Shell` script de este componente, de la máquina local del usuario (la cual realiza el envío de los trabajos) a la máquina del cluster. Con esta acción, el componente `tdp_agent` puede acceder a su archivo `TDP-Shell` script por estar los dos situados en el mismo directorio de la máquina del cluster.

En el fichero FDT 4.8 se muestra el fichero de descripción de trabajos global obtenido por `tdp_console` al procesar los ficheros de descripción de trabajos FDT 4.5, FDT 4.6 y FDT 4.7. En este fichero global puede observarse como su ejecutable y los argumentos de este (comandos *Executable y Arguments*) corresponden con los del fichero de descripción de trabajos del `tdp_agent`. Esto es debido a que el gestor de colas Condor (mas concretamente, su componente remoto) es el encargado de ejecutar este componente del entorno `TDP-Shell` en la máquina del cluster. También puede observarse como se informa a Condor que no se transfiera el ejecutable de `tdp_agent` a la máquina del cluster (comando *Transfer_executable = NO*) debido a que esta situado en un directorio compartido, el */opt/TDP-Shell/*.
El archivo de descripción de trabajos de la aplicación de usuario, informa a Condor que

FDT 4.7 Fichero de descripción de trabajos de Condor para el componente tdp_agent

```
 1: Universe = vanilla
 2: Executable = /opt/tdp/tdp_agent
 3: Transfer_executable = NO
 4: Arguments = -cf:/opt/tdp/config/tdp_config.cfg -tf:tdp_agent.tdp
 5: Output = tdp_agent.out
 6: Error = tdp_agent.err
 7: Log = tdp_agent.log
 8: should_transfer_files = YES
 9: When_to_transfer_output = ON_EXIT
10: Transfer_input_files = /home/user/tdp_files/tdp_agent.tdp
11: Queue
```

FDT 4.8 Fichero de descripción de trabajos global de Condor obtenido por tdp_console

```
 1: Universe = vanilla
 2: Executable = /opt/tdp/tdp_agent
 3: Transfer_executable = NO
 4: Arguments = -cf:/opt/tdp/config/tdp_config.cfg -tf:tdp_agent.tdp
 5: Output = TDP_Shell.out
 6: Error = TDP_Shell.err
 7: Log = TDP_Shell.log
 8: should_transfer_files = YES
 9: When_to_transfer_output = ON_EXIT
10: Transfer_input_files = /home/user/bin/demo_prog,
    /home/user/tdp_files/tdp_agent.tdp
11: Queue
```

copie automáticamente su ejecutable, de la máquina local de este usuario a la máquina del cluster, más concretamente en el directorio de trabajo de Condor situado en esta. Sin embargo, al enviar tdp_conole el fichero de descripción de trabajos global a Condor, sucede que su ejecutable es el del componente `tdp_agent`, con lo cual la copia remota del ejecutable de la aplicación de usuario ya no se realiza automáticamente. Por ese motivo y para que esta copia se lleve a cabo, la dirección y el nombre de este ejecutable del usuario son situados en la en la lista del comando *Transfer_input_files* del fichero de descripción de trabajos global (archivos a transferir desde la máquina local del usuario a la del cluster).

Tuplas especiales del entorno TDP-Shell

Los valores de las tuplas especiales que obtiene `tdp_console`, al procesar los archivos de descripción de trabajos de los componentes remotos del entorno `TDP-Shell`, son los mismos valores que los mostrados en el ejemplo de SGE (punto *Tuplas especiales del entorno TDP-Shell* de la sección 4.5.1.3)

4.6. Gestión del espacio de atributos tdp

Antes de pasar a la explicación de los archivos `TDP-Shell` script (apartado 4.7), es conveniente explicar como gestiona en entorno `TDP-Shell` el espacio de atributos tdp. Como se ha explicado en el apartado 4.4, este espacio es utilizado por los componentes `tdp_console` y `tdp_agent` para intercambiar información en forma de tuplas, normalmente al ejecutar los comandos tdp de comunicación situados en los archivos `TDP-Shell` script que interpretan. En el diseño de la arquitectura base del entorno `TDP-Shell`, el espacio de atributos tdp está gestionado por un *proceso servidor único*, las funciones del cual son:

1. Aceptar o denegar las peticiones de acceso al espacio de atributos por parte de los componentes `tdp_console` y `tdp_agent`.

2. Mantener la lista de las tuplas enviadas por estos dos componentes.

3. Gestionar las peticiones, procedentes de los componentes `tdp_console` y `tdp_agent`, para las tuplas que este servidor gestiona. Estas peticiones solicitan principalmente: Añadir, obtener o eliminar tuplas.

Como puede observarse por estas funciones, la arquitectura del modelo de aplicación distribuida entre el servidor del espacio de atributos y los componentes `tdp_console` y `tdp_agent` es del tipo *cliente - servidor* [34], donde los clientes son los dos componentes del entorno `TDP-Shell` y el servidor es el proceso que gestiona el espacio de atributos tdp. En los dos siguientes puntos de este apartado se explicará como el servidor del espacio de atributos tdp implementa este modelo de aplicación distribuida y como realiza correctamente sus funciones.

4.6.1. Utilización del protocolo de comunicaciones TCP/IP

El protocolo TCP/IP [35] [36] [37] [38] gestiona las comunicaciones entre el servidor del espacio de atributos y los componentes `tdp_console` y `tdp_agent`. Se ha escogido este protocolo de comunicaciones debido a que su parte TCP garantiza:

- Mientras la conexión entre dos procesos no esté establecida correctamente, ninguno de los dos podrá transmitir datos.
- Todos los datos van a llegar a los procesos correctamente, sin errores y en el orden correcto.

La parte más conocida (por la mayoría de usuarios normales, profesionales o científicos) del protocolo TCP/IP, es el mecanismo que utiliza para que dos procesos, que se ejecutan en dos ordenadores situados en una red, puedan identificarse para poder comunicarse. Este protocolo de comunicaciones define dos términos importantes para realizar esta identificación: *la dirección IP y los puertos*. La *dirección IP* es una etiqueta numérica (de 32 bits en la versión IPv4 y de 128 en la versión IPv6) que identifica de manera lógica y única a un ordenador situado en una red. El *puerto* es un valor numérico de 16 bits que identifica a la aplicación o proceso, situado en este ordenador, que desea emitir o recibir datos. Los procesos del tipo servidor pueden utilizar diversos puertos, uno para cada servicio que ofrecen a sus posibles clientes, por ejemplo un servidor puede dar soporte a ftp a través del puerto 21 y a ssh, a través del puerto 22.
A parte de este mecanismo de identificación de procesos, para diseñar aplicaciones que trabajen con el protocolo TCP/IP, es necesario que estos procesos dispongan de un mecanismo local, o punto final de comunicaciones, a través del cual puedan gestionar y realizar su intercambio de datos por la red. Este mecanismo son los *sockets de internet* (mecanismo conocido por usuarios profesionales o científicos dedicados al desarrollo de aplicaciones distribuidas), que para el protocolo TCP su tipo es *orientado a conexión* (también existe el tipo *no orientados a conexión*, para UDP). Estos sockets, entre otras informaciones, guardan la dirección IP y el puerto de la máquina local, así como la dirección IP y el puerto de la máquina remota. Esta última información es necesaria solo si se establece una conexión remota con otro ordenador, normalmente en el caso de un servidor (este punto será explicado más adelante, en este mismo apartado).
Los sistemas operativos ofrecen a las aplicaciones una serie de primitivas que permiten

operar con los sockets de internet. Estas primitivas permiten crearlos (la llamada *socket ()*, conectarse a un determinado puerto de una máquina remota (primitiva *connect ()*) o leer o escribir a través de un socket (primitivas *send() y recv()*).
Una vez explicados como se identifican los procesos remotos y los mecanismos de conexión que utilizan, se puede pasar a explicar como se implementa la arquitectura cliente-servidor con el protocolo TCP/IP, para ello se siguen los siguientes pasos básicos:

a) El servidor se sitúa en un ordenador con una dirección IP conocida por todos los clientes y crea uno o varios sockets de internet, cada uno con un puerto también conocido. Una vez creados, el servidor declara estos sockets *de escucha (listen en ingles)* para que esperen la llegada de peticiones de los clientes (usando la primitiva *listen ()*)

b) Los clientes crean un socket y le asocian un puerto libre para realizar una petición de conexión al servidor y acceder a alguno de sus servicios (esta acciones las hacen a través de la primitiva *connect()*). El cliente identifica a este servidor a través de la dirección IP de su máquina y del puerto asociado al servicio.

c) Si el servidor acepta la petición procedente del cliente (para ello utiliza la primitiva *accept()*), entonces crea un nuevo socket, denominado *de servicio*, el cual comparte el puerto del socket que escucha peticiones. Después esta aceptación, se establece definitivamente la conexión entre el servidor y el cliente, pudiendo este ultimo, enviar peticiones al servidor y recibir las respuestas de este.

Es importante destacar que los sockets del servidor que escuchan peticiones de los clientes, comparten puertos con los que se utilizan para servir las peticiones de estos clientes. Por este motivo y como se ha comentado anteriormente, los sockets de servicio necesitan conocer la dirección IP de la máquina y el puerto del proceso cliente para poder enviarle las respuestas a sus peticiones (mientras que los de escucha no lo necesitan).
Como se ha explicado en los apartados 4.1 y 4.4, el entorno `TDP-Shell` define dos tipos de comandos tdp de comunicación para interactuar con el servidor del espacio de atributos, los síncronos (o bloqueantes) y los asíncronos (o no bloqueantes). Para gestionar las peticiones de estos dos tipos de comandos, el servidor del espacio de atributos tdp ofrece *dos servicios*, uno para las *peticiones síncronas* y otro para las *asíncronas*. Tanto `tdp_console` y `tdp_agent` pueden acceder a estos dos servicios debido a que el servidor

del espacio de atributos tdp crea, para cada uno de ellos, dos sockets de escucha que tienen asociados dos puertos que son conocidos por estos dos componentes del entorno `TDP-Shell`. Para cada petición de conexión recibida por alguno de estos dos sockets que es aceptada, el servidor crea un nuevo socket de servicio para enviar las respuestas o recibir las peticiones del componente que esta conectado a el, ya sea `tdp_console` o `tdp_agent`. Por su parte `tdp_console` y `tdp_agent`, al inicio de su ejecución, crean dos sockets con sus respectivos puertos. Un socket, denominado a partir de ahora *síncrono*, establece la conexión con el socket de servicio de las peticiones síncronas del servidor del espacio de atributos tdp y el otro socket, denominado a partir de este momento *asíncrono*, con el de las peticiones asíncronas de dicho servidor. A través de estos sockets, los comandos síncronos y asíncronos que interpretan `tdp_console` y `tdp_agent`, enviarán sus peticiones al servidor del espacio de atributos tdp.

4.6.2. Protocolo para el tratamiento de las peticiones

El diseño del servidor de atributos tdp (y en general el de la mayoría de servidores) se basa en la creación de un bucle principal que espera la llegada de las peticiones, síncronas o asíncronas, provenientes de `tdp_console` o de `tdp_agent`, para realizar las acciones necesarias que lleven a responderlas. Puede suceder que algunas de estas peticiones no puedan ser satisfechas en el momento de su llegada, debido a que soliciten el valor de alguna tupla que no esté situada en la *lista de tuplas*. Para gestionar estos casos, el servidor del espacio de atributos tdp almacena estas peticiones en la *lista de peticiones pendientes*, cuyos elementos están formados por parejas *[Atributo_pendiente, Socket_servicio_pendiente]*, donde *Atributo_pendiente* es el atributo de la tupla que esta pendiente de respuesta y *Socket_servicio_pendiente* es el socket de servicio por donde ha llegado la petición.

Para que el servidor del espacio de atributos tdp pueda identificar el tipo de petición recibida desde sus clientes y generar su correspondiente respuesta para ellos, se define un protocolo basado en el intercambio de un conjunto de mensajes propios de `TDP-Shell`. El formato de estos mensaje es *[Id_petición,Datos_mensaje]*, donde el campo *Id_petición* identifica el tipo de petición y el campo *Datos_mensaje* contiene los datos relacionados con la petición. Los diferentes mensajes que envían `tdp_console` y `tdp_agent` al servidor y las acciones que desencadenan en este último son:

- *[GET,Atributo_tupla]*: Al recibir este mensaje, el servidor busca si la tupla, cuyo atributo es *Atributo_tupla*, esta situada en la *lista de tuplas*. En caso afirmativo, extrae el valor (*Valor_tupla*) de dicha tupla y genera la respuesta a esta petición GET a través del mensaje *[ANSWER,Atributo_tupla,Valor_tupla]*. Este nuevo mensaje es enviado a través del socket de servicio por donde el servidor recibió la petición (y que está conectado con cliente que la espera). Si la tupla no existe en la *lista de tuplas*, entonces se almacena, en la *lista de peticiones pendientes*, el campo *Atributo* y el socket de servicio por donde se ha recibido el mensaje GET.

- *[PUT,Atributo_tupla,Valor_tupla]*: al recibir este mensaje, el servidor añade la tupla *(Atributo_tupla,Valor_tupla)* a la *lista de tuplas* y envía por el socket de servició donde recibió la petición, el mensaje *[ANSWER,Atributo_tupla,Resultado]*, donde *Resultado* puede ser *OK* si todo la inserción de la tupla se ha realizado con éxito o *ERROR* si no lo ha sido. A continuación, el servidor mira si en la *lista de peticiones pendientes*, existe algún elemento cuyo *Atributo_pendiente* sea igual a *Atributo_tupla*. En caso afirmativo, se envía el valor de esta tupla recientemente insertada en el espacio de atributos, a algún `tdp_console` o de `tdp_agent` que lo está esperando. Para ello, el servidor genera el mensaje *[ANSWER,Atributo_tupla,Valor_tupla]* y lo envía por el socket *Socket_servicio_pendiente* asociado al elemento de la lista *lista de peticiones pendientes* (que identifica al proceso que lo espera). Una vez realizado el envío correctamente, el elemento que identifica la petición pendiente en esta lista es eliminado.

- *[DEL,Atributo_tupla]*: Con este mensaje, el servidor elimina la tupla identificada por el atributo *Atributo_tupla* de la *lista de tuplas*. Si el valor de *Atributo_tupla* es igual a *"ALL"*, entonces elimina todas las tuplas del espacio de atributos tdp, esto es, vacía la *lista de tuplas*. El entorno `TDP-Shell` utiliza una tuplas especiales de gestión, las cuales son de uso interno y comienzan por el prefijo *"_TDP_"*. Si se utiliza la opción "ALL" estas tuplas también son eliminadas, si no se desea que esto suceda (se eliminen todas las tuplas menos las de gestión) entonces se puede utilizar el valor *"ALL_LOCAL"* en el campo *Atributo_tupla* del mensaje.

- *[TEST,Atributo_tupla]*: Al recibir este mensaje, el servidor, igual que en el mensaje GET, busca si la tupla cuyo atributo es *Atributo_tupla* esta en la *lista de tuplas*.

Si lo está, extrae su valor (*Valor_tupla*) y genera la respuesta con el mensaje *[ANSWER,Atributo_tupla,Valor_tupla]*. Si la tupla no existe en la *lista de tuplas*, entonces genera una respuesta especial *[ANSWER,Atributo_tupla,"NOTHING"]* para indicar al cliente que la tupla que solicita no esta situada en el espacio de atributos tdp. Como en el caso de otras respuestas, este mensaje también se envía a través el socket de servicio por donde se ha recibido la petición TEST.

- *[END]*: A través de este mensaje, un cliente, ya sea `tdp_console` o `tdp_agent`, informa al servidor del espacio de atributos tdp que va ha finalizar su conexión con el (tanto síncrona como asíncrona). Una vez eliminados con seguridad los recursos del sistema y los sockets de servicio síncronos y asíncronos de esta conexión, el servidor elimina de la *lista de peticiones pendientes* todos sus elementos cuyo campo *Socket_servicio_pendiente* coincida con el socket eliminado. Esto es realizado debido a que al no estar conectado el cliente, ya no es necesario que se le envíen las peticiones que pudiera tener pendientes.

- *[END_server]*: Con este mensaje, un cliente solicita la finalización de la ejecución del servidor del espacio de atributos tdp. Para evitar problemas importantes, esta acción de finalización únicamente se lleva a cavo cuando queda un solo `tdp_console` o un `tdp_agent` conectado al servidor. En caso de que haya más de uno de estos clientes conectados, este mensaje es ignorado.

En la figura 4.3 se puede observar el esquema de conexiones entre los componentes `tdp_console` y `tdp_agent` con el servidor del espacio de atributos tdp.

4.7. Archivos TDP-Shell script

En sección (4.5) se ha explicado como el componente `tdp_console` procesa los archivos de descripción de trabajos de los componentes remotos del entorno `TDP-Shell`, para obtener el fichero de descripción de trabajos global. Este fichero informa al gestor de colas, de como crear el entorno de ejecución remoto que permita ejecutar, en una máquina del cluster, el componente `tdp_agent`, el componente remoto de la herramienta (ejecutado por `tdp_agent`) y el proceso a monitorizar.

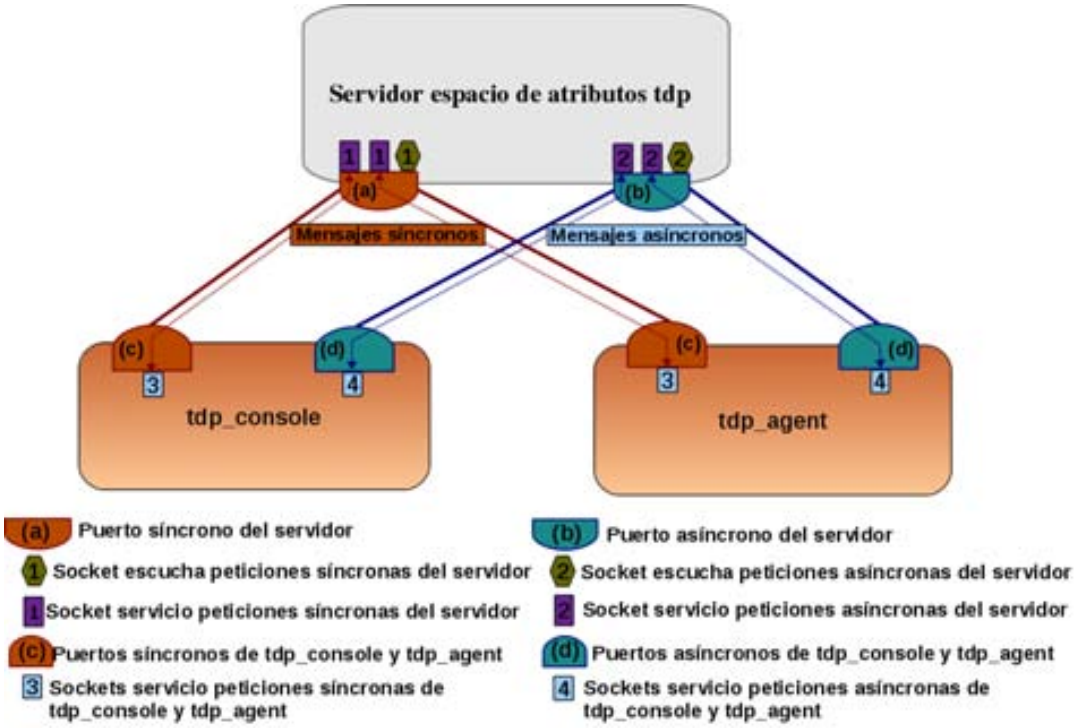


Figura 4.3: Esquema conexiones entre tdp_console y `tdp_agent` con el servidor del espacio de atributos tdp

El siguiente paso que ha de realizar en entorno `TDP-Shell` para solucionar el problema de la falta de interoperabilidad entre los gestores de colas y herramientas de monitorización, es la gestión de la ejecución sincronizada de los componentes de esta última. Para poder realizarla, `tdp_console` y `tdp_agent` han de intercambiar, a través del espacio de atributos tdp, la información necesaria que les permita ejecutar estos componentes de la herramienta en el momento adecuado. Para ello, el entorno `TDP-Shell` utiliza *los ficheros especiales TDP-Shell script*, los cuales son interpretados (no se compilan previamente y se enlazan para formar un ejecutable como en C) por sus componentes `tdp_console` y `tdp_agent`. Para facilitar el aprendizaje de la utilización de estos ficheros `TDP-Shell` por parte de los usuarios, el formato de estos ficheros se ha basado en:

- Los ficheros shell-script, por ser uno de los formatos de ficheros de comandos más utilizado por los usuarios de los sistemas basados en UNIX/Linux.
- Las instrucciones condicionales, bucles y definiciones de funciones del lenguaje de

alto nivel C (o de C++ y Java), por ser conocidas por la mayoría de programadores de aplicaciones (así como familiares a los usuarios de aplicaciones informáticas) y permitir un alto nivel de funcionalidad e iteración.

En los siguientes puntos de este apartado se mostrará el formato de los ficheros Shell-script, así como la sintaxis y significado del código tdp del que están compuestos.

4.7.1. Formato de los ficheros TDP-Shell script

El formato de los archivos `TDP-Shell` script se divide en dos secciones principales:

1. **Sección de *includes* y definición de *funciones locales*** : En esta sección se informa al componente `tdp_console` o `tdp_agent` que interpreta el archivo `TDP-Shell` script de:
   - *Los archivos TDP-Shell script que se han de incluir*: Estos archivos contienen el código tdp de un conjunto de funciones locales, que se añadirán al código del fichero `TDP-Shell` script actual y que podrán ser llamadas por este. Su funcionalidad es similar a la directiva *#include del lenguaje C*.
   - *La definición de las funciones locales*: Código tdp de las diferentes funciones locales que podrán ser llamadas desde cualquier punto del archivo `TDP-Shell` script. Esta funcionalidad es similar a la declaración y procedimientos de lenguajes de programación como C.
2. **Sección principal**: En esta sección se define el código tdp principal del archivo `TDP-Shell` script. Este código se encarga de ejecutar las acciones necesarias que controlan el intercambio de información con el espacio de atributos y la creación, en el momento adecuado, de los componentes de la herramienta y si es necesario, el proceso de la aplicación a monitorizar (esto último lo realiza el `tdp_agent`).

4.7.2. Sintaxis y funcionamiento de los diferentes componentes del Código tdp

Para realizar sus acciones y como se ha mencionado en el punto Archivos `TDP-Shell` script del apartado 4.4, el código tdp esta formado por un conjunto de *comandos tdp* e *instrucciones especiales*, los cuales se muestran en los siguientes puntos

4.7.2.1. Variables

Como en los ficheros shell-script y los lenguajes de alto nivel, los ficheros `TDP-Shell` script permiten declarar variables para almacenar datos durante cierto periodo de tiempo. Dependiendo de su ámbito de aplicación (en que parte del código son visibles o accesibles), existen dos tipos de variables :

- *Variables locales*: Su sintaxis es *$nombre_variable*. Si se declaran en la sección principal del archivo `TDP-Shell` script, su ámbito de aplicación es toda esta sección (son accesibles por todo su código tdp), pero no son accesibles desde el código tdp de las funciones locales de este archivo. Si son declaradas dentro de una función local, entonces su ámbito de aplicación se centra en el código tdp de esta función, pero no son accesibles desde el código tdp de otras funciones locales, ni desde el de la sección principal.

- *Variables globales*: Su sintaxis es: *%nombre_variable*. Su ámbito de aplicación es todo el código tdp del archivo `TDP-Shell` script. Esto es, son accesibles desde el código de las funciones locales y desde el de la sección principal.

Como en el caso de shell-scripts, las variables de `TDP-Shell` no se les define el tipo (como si se hace en C, C++ o Java), esto es no son tipadas. Por lo tanto es al asignarles un valor, mediante el operador = (varible = valor), cuando internamente (`tdp_console` o `tdp_agent`) se decide su tipo. Los tipos básicos que soporta el entorno `TDP-Shell` son: Números Enteros (positivos y negativos) y cadenas de caracteres (comprendidas entre las dobles comillas " ").

4.7.2.2. Declaración de funciones locales

Las funciones locales de un archivo `TDP-Shell` script se declaran de la siguiente manera:

tdp_fun *nombre_funcion ([$argumento]*)* **{**
código tdp de la función
[***return*** *($variable_local)]?*
}

Las funciones locales siempre se declaran con el prefijo *tdp_fun*, siendo el campo *[$argumento]**, los posibles argumentos que se le pueden pasar por valor a la función, cuya sintaxis es la misma que la de las variables locales. El comando *return ($variable_local)* (que es optativo) se utiliza para indicar que la función devuelva, al finalizar su ejecución, el valor de la variable *$variable_local*.

4.7.2.3. Declaración de instrucciones condicionales

Las instrucciones condicionales del código tdp tienen el siguiente formato:

if *(condición)* **{**
código tdp de la condición if
}
[***else*** **{**
código tdp de la condición else
}]?

Como puede observarse, la sintaxis de esta instrucción condicional esta basada en la del lenguaje C. La función de esta instrucción es ejecutar el *código tdp de la condición if* si se cumple la *condición* y en caso contrario, o bien ejecutar el *código tdp de la condición else*, si este ultimo esta declarado, o continuar la ejecución en el siguiente comando o instrucción tdp.
El campo condición tiene la siguiente sintaxis: *((A op_cond B) [op_log (A op_cond B)]+)* , donde:

- *A* puede ser una variable local o global.
- *B* puede ser una variable local, global, un número entero (positivo o negativo) o una cadena de caracteres
- *op_cond* puede ser uno de los siguientes operadores condicionales: $>$, $<$, $==$, $\neq$, $\geq$, $\leq$
- *op_log* puede ser uno de los siguientes operadores lógicos: || (O lógica) o && (Y lógica)

Debido a que el tipo de las variables se decide internamente, al comparar al contenido de una variable con otra cuyos tipos son diferentes, el resultado de esta comparación será indeterminado. Por ejemplo, si a la variable local $var_1 se le ha asignado el Entero 10 ($var_1 = 10) y a la variable $var_2 la cadena de caracteres "20" ($var_1 = "20"), el resultado de realizar la operación condicional: ($var_1 < $var_2) es indeterminado. Por este motivo, es muy importante que el programador de ficheros `TDP-Shell` script, controle las asignaciones de las variables locales o globales (para deducir su tipo) y de esta manera evitar posibles problemas en las operaciones condicionales.

4.7.2.4. Declaración de bucles

Los bucles del código tdp tienen el siguiente formato:

while *(condición)* **{**
código tdp del bucle
}

Como en el caso de los bucles condicionales, la sintaxis de los bucles también está basada en el lenguaje C. La función del bucle es repetir el *código tdp del bucle* mientras se cumpla la *condición*. Este campo *condición* tiene el mismo formato que el visto en la instrucción condicional *if*.

4.7.2.5. Declaración de expresiones matemáticas

Las expresiones matemáticas (sumar, restar, multiplicar y dividir) en los ficheros `TDP-Shell` script tienen la siguiente sintaxis:

@ Destino = (Operando op_arit Operando) [op_arit Operando]+)

Donde:

- *Destino* puede ser una variable local o global.
- *Operando* puede ser una variable local, global, un número entero (positivo o negativo).
- *op_arit* puede ser uno de los siguientes operadores aritméticos: +, -, *, /.

4.7.2.6. Comandos tdp

Estos comandos, basados en el protocolo tdp (apartado 4.1), encapsulan las funcionalidades de comunicación con el espacio de atributos, con el gestor de colas, así como la creación y gestión de procesos en la maquina local.
La mayoría de estos comandos admiten argumentos, los cuales pueden ser cadenas de caracteres, números enteros o variables locales o globales. En el caso de estas últimas, se acaba pasando su valor al argumento del comando tdp. Como en el caso de los shell-scripts y lenguajes de alto nivel, los valores que devuelven los comandos tdp pueden ser guardados en una variable local o global (en caso contrario se pierden).

Comandos tdp de comunicación con el espacio de atributos

Estos comandos realizan las funciones de situar y extraer información del espacio de atributos. Este intercambio de información se realiza a través tuplas, que como se ha explicado en el punto *Intercambio de información de la sección 4.1*, tienen el formato: *(atributo, valor)*. La mayor parte de la codificación de la funcionalidad de estos comandos esta situada en las funciones de la librería tdp, las cuales son llamadas por `tdp_console` o `tdp_agent` cuando interpretan los comandos tdp de comunicación situados en sus ficheros `TDP-Shell` script.
Dependiendo de si suspenden o no la ejecución del proceso que los llama (que será o bien `tdp_console` o `tdp_agent`) estos comandos tdp de comunicación se clasifican en:

- **Bloqueantes o síncronos:** Estos comandos, como su nombre indica, suspenden la ejecución del proceso que los llama hasta que la operación de comunicación con el espacio de atributos haya finalizado. Su sintaxis y comportamiento son:

  - **Valor_tupla = *tdp_get* Atributo_tupla** : Devuelve el valor (*Valor_tupla*) de una tupla, identificada por el atributo *Atributo_tupla*, situada en el espació de atributos. Si esta tupla no está situada en dicho espacio, el proceso que ha llamado a este comando, es suspendido hasta que dicha tupla sea situada en el. Internamente, este comando realiza los siguientes pasos:

    1. Crea el mensaje *[GET,Atributo_tupla]*
    2. Se envía este mensaje (utilizando una llamada send()) al servidor del espacio

de atributos tdp. Este envío se realiza a través del socket síncrono del componente `tdp_console` o `tdp_agent` que está interpretando este comando.

3. Suspende la ejecución del componente de `TDP-Shell` que lo ha llamado, hasta que sucedan uno de estos dos casos:
   a) Se recibe, por el mismo socket síncrono que se envío el mensaje GET, el mensaje con la respuesta del servidor del espacio de atributos tdp
   b) Ha transcurrido un intervalo de tiempo sin que esta respuesta ocurra. La ocurrencia de este último evento implica la existencia de un retardo significativo en la recepción del mensaje, pudiendo significar un posible error de comunicación con el servidor. Si esto sucede, este comando informa al usuario de este echo para que tome las medidas oportunas. Para llevar a cavo esta funcionalidad, en la implementación de UNIX/Linux del entorno `TDP-Shell`, se utiliza *la primitiva select*. Esta primitiva permite suspender la ejecución de un proceso durante un periodo de tiempo o hasta que se detecte, en ese intervalo de tiempo, un cambio de estado en algún descriptor de ficheros de dicho proceso (principalmente por haber recibido o enviado datos a través de dicho descriptor), entre ellos un socket internet (el cual seria el socket síncrono de `tdp_console` o `tdp_agent`).
4. En caso de recibir una respuesta del servidor, lee (usando una primitiva recv() sobre el socket síncrono) el mensaje asociado a dicha respuesta: *[ANSWER,Atributo_tupla,Valor_tupla]* y devuelve el campo *Valor_tupla* de este.

Como ejemplo de la división de funcionalidades entre `tdp_console`, `tdp_agent` y la librería tdp, se puede enumerar que los pasos 2,3 y 4 los realizan las funciones de la librería tdp y el paso 1 los componentes `tdp_console`, `tdp_agent` al interpretar sus comandos tdp.

- ***tdp_put*** **Atributo_tupla, Valor_tupla :** Solicita la inserción de la tupla, identificada por el atributo *Atributo_tupla*, en el espacio de atributos tdp. Mientras se espera la confirmación de la inserción de la tupla en el espacio de atributos tdp, este comando suspende la ejecución del componente `tdp_console` o `tdp_agent` que lo ha llamado. Para realizar sus acciones, este comando:

1. Crea el mensaje crea el mensaje *[PUT,Atributo_tupla,Valor_tupla]*
2. Lo envía al servidor del espacio de atributos tdp por el socket síncrono del componente `tdp_console` o `tdp_agent` que lo esta interpretando.
3. Suspende la ejecución del componentes de `TDP-Shell`, esperando a que sucedan uno de los dos siguientes sucesos:
   a) La respuesta de confirmación de inserción de la tupla procedente del servidor
   b) La ocurrencia de un error en las comunicaciones (tiempo de espera demasiado largo). Para ello sigue el mismo proceder que en el comando *tdp_get*.
4. En caso de recibir la respuesta del servidor, lee el mensaje asociado a esta respuesta *[ANSWER, Atributo_tupla, Resultado]* y comprueba que su campo *Resultado* sea OK. Si es así, entonces la inserción de la tupla ha sido correcta, si no lo es, implica un error en dicha inserción e informa al usuario de este hecho.

- **No bloqueantes o asíncronos:** Estos comandos, al contrario que los anteriores, no suspenden la ejecución del proceso que los llama mientras dura la operación de comunicación con el espacio de atributos tdp. De la finalización de esta operación de comunicación se encarga un módulo especial denominado *gestor de eventos*. La sintaxis de estos comandos tdp y el funcionamiento de este gestor de eventos se explican en los siguientes puntos:

  - ***tdp_asynget* Atributo_tupla Name_fun_asynget**: Solicita el valor de una tupla, identificada por el atributo *Atributo_tupla*, al servidor del espacio de atributos tdp. También informa del nombre de la función, *Name_fun_asynget*, que se ha de llamar, cuando llegue, al componente `tdp_console` o `tdp_agent` que lo ha interpretado, el mensaje con el valor de la tupla solicitada. Los pasos que sigue este comando para realizar estas acciones son:

    1. Crea el mensaje *[GET,Atributo_tupla]*
    2. Se envía este mensaje al servidor del espacio de atributos tdp. Este envío se realiza a través del *socket asíncrono* del componente `tdp_console` o `tdp_agent` que está interpretando este comando.

3. Almacena en la lista especial: *lista_funciones_asíncronas* que contiene las funciones asíncronas a ser llamadas por el *gestor de eventos*, el elemento formado por el par *(Atributo_tupla, Name_fun_asynget)*. Con esta lista, el *gestor de eventos* podrá averiguar que función se de llamar cuando llegue la respuesta a una petición asíncrona asociada a un determinado atributo.

- ***tdp_asynput* Atributo_tupla Valor_tupla Name_fun_asynput:** Solicita la inserción de la tupla, identificada por el atributo *Atributo_tupla*, en el espacio de atributos tdp. Igual que en el comando *tdp_asynget*, informa del nombre de la función, *Name_fun_asynput*, que se ha de llamar, cuando llegue la confirmación de la correcta inserción de la tupla. Los pasos que sigue este comando para realizar estas acciones son:

  1. Crea el mensaje *[PUT,Atributo_tupla,Valor_tupla]*
  2. Se envía este mensaje al servidor del espacio de atributos tdp. Este envío se realiza a través del *socket asíncrono* del componente `tdp_console` o `tdp_agent` que está interpretando este comando.
  3. Almacena en la *lista_funciones_asíncronas* el elemento formado por el par *(Atributo_tupla, Name_fun_asynput)*

- **Módulo gestor de eventos:** Este módulo de `tdp_console` y `tdp_agent` (codificado en la librería tdp), se encarga de la correcta finalización de las peticiones asíncronas pendientes. Para ello sigue los siguientes pasos:

  1. Periódicamente entra en ejecución y comprueba si ha llegado, por el socket asíncrono de estos componentes de `TDP-Shell`, alguna respuesta del servidor del espacio de atributos tdp. El periodo normal de entrada en ejecución de este *módulo gestor de eventos* se produce después de la ejecución de cada comando tdp que interpretan `tdp_console` o `tdp_agent`. Cuando estos componentes del entorno `TDP-Shell` están suspendidos por un comando tdp bloqueante (por ejemplo *tdp_get*), entonces este módulo entra en ejecución cada cierto intervalo de tiempo, hasta que la operación de comunicaciones síncrona finalice. Este ultimo caso se gestiona en sistemas UNIX/Linux utilizando la señal SIGALARM, la cual se genera cada cierto intervalo de tiempo y permite la ejecución de una función, asociada a esta señal, con la llamada al sistema *signal (SIGALARM, función)*. Para definir el intervalo de

tiempo de la señal SIGALRM se puede utilizar la llamada al sistema *alarm ()*, pero esta llamada tiene el problema que solo permite definir intervalos de segundos. Para definir intervalos de tiempo más pequeños, se utiliza la función *setitimer()* (esta es la que utiliza el entorno `TDP-Shell`).

2. Si ha llegado un mensaje cuyo formato es *[ANSWER,Atributo_tupla_repuesta,Valor_tupla]* (ver apartado), lo lee y extrae su campo *Atributo_tupla*.
3. Busca en la *lista_funciones_asíncronas* si existe algún elemento cuyo campo (Atributo_tupla), coincide con el del atributo del mensaje, *Atributo_tupla_repuesta*. En caso afirmativo, extrae el nombre de la función, campo *Name_fun_asyn* del elemento de la lista. Si no existe, no realiza ninguna acción e ignora el mensaje recibido, avisando al usuario de este hecho.
4. Una vez obtenido el nombre de la función, pasa a ejecutar su código. Antes de realizar esta acción, inicializa dos variables locales especiales de `TDP-Shell`: *$tdp_attribute* que contiene el atributo de la tupla contenida en el mensaje recibido (*Atributo_tupla_repuesta*) y *$tdp_value* que contiene el valor de dicha tupla (*Valor_tupla*). Estas variables son accesibles desde el código de la función como si fueran dos argumentos pasados a esta.
5. Cuando finaliza la ejecución de esta función y si este *módulo gestor de eventos* se ha llamado después de la ejecución de un comando tdp, permite que `tdp_console` o `tdp_agent` continúen interpretando el código tdp de su fichero `TDP-Shell` script (a partir del siguiente comando tdp).

En la implementación UNIX/Linux para comprobar si ha llegado alguna respuesta por el socket asíncrono, se utiliza la llamada al sistema *select ()* (igual que con los comandos síncronos) con salida inmediata de esta llamada si no se ha detectado ninguna actividad por este socket (no le ha llegado ningún mensaje).

- **De test:** Este tipo de comandos tdp comprueban la existencia de una tupla en el espacio de atributos tdp. El comando tdp de este tipo es:

  - **Valor_tupla = *tdp_test_attr* Atributo_tupla**: Consulta la existencia de una tupla, identificada por el atributo *Atributo_tupla*, en el espació de atributos. Si la respuesta es afirmativa, devuelve el valor de dicha tupla. En caso contrario

devuelve el valor *"NOTHING"*. Para realizar estas acciones, este comando tdp realiza los siguientes pasos:

1. Crea el mensaje *[TEST,Atributo_tupla]*.
2. Envía este mensaje al servidor del espacio de atributos tdp, a través del socket síncrono del componente `tdp_console` o `tdp_agent` que lo está interpretando.
3. Por este mismo socket síncrono, lee el mensaje con la respuesta del servidor, cuyo formato es *[ANSWER, Atributo_tupla, Valor_tupla]* y devuelve su campo *Valor_tupla* que contiene el posible valor de la tupla solicitada. Si esta no se encuentra en el espacio de atributos tdp, como se ha comentado, el valor de este campo del mensaje es *"NOTHING"*.

Comandos tdp de gestión de procesos locales

Los comandos tdp de gestión de procesos locales, a partir de ahora *comandos tdp de procesos*, son los encargados de crear y gestionar los procesos que solicitan `tdp_console` o `tdp_agent`, normalmente al interpretar sus respectivos archivos `TDP-Shell` script. Como se ha explicado en el punto *librería tdp* del apartado 4.4, estos comandos tdp llaman a funciones de la librería tdp, las cuales a su vez llaman a las del plug-in específico del sistema operativo de la máquina local. Estos plug-ins contienen las llamadas al sistema operativo que gestionan los procesos y son incorporados a la librería tdp dinámicamente (como si fueran una librería dinámica).
Para un mejor entendimiento del funcionamiento de este entorno basado en plug-ins que utilizan los comandos tdp de procesos, su explicación se ha dividido en los siguientes puntos:

- **Especificaciones comunes**:Para el correcto funcionamiento de la gestión de los procesos creados por los comandos tdp de procesos, el entorno `TDP-Shell` define unas especificaciones comunes a todas los comandos tdp de procesos, funciones de la librería tdp y de los plug-ins del sistema operativo. Estas especificaciones son:

  1. Un identificador único para cada proceso local creado por los comandos tdp de procesos, al que se denominara a partir de este punto: *pid_tdp* y cuyo tipo será:

Id_proceso_tdp. Este identificador es usado por los comandos tdp y funciones del entorno `TDP-Shell` para identificar al proceso que han de gestionar.

2. Cada sistema operativo define unos estados propios en los que se pueden encontrar sus procesos. Si el Entorno `TDP-Shell` escoge esta información para definir los estados de los procesos creados a través de sus comandos tdp de procesos (usando las funciones del plug-in), entonces surgirá el inconveniente de la existencia de una gran variedad de estos estados. Además también puede suceder el problema añadido, que diferentes nombres de estados representen en realidad el mismo. Para solucionar estos posibles problemas, el entorno `TDP-Shell` define un conjunto de estados propios para los procesos creados y gestionados a través de sus *comandos tdp de procesos*. Estos estados son:

   - PAUSED: El proceso está suspendido o pausado.
   - RUNNING: El proceso se está ejecutando.
   - FINISHED: El proceso ha finalizado su ejecución.
   - FAILED: El proceso a finalizado debido a un error durante su ejecución.
   - OTHER_STATE: Si está en un estado que no es ninguno de los anteriores.

- **Diseño de los plug-ins de los Sistemas Operativos locales**: Debido a que la librería tdp puede incorporar diversos plug-ins de Sistemas Operativos, es necesario que exista una interfaz en común que permita a las funciones de gestión de procesos locales de esta librería, llamar a las de cualquier plug-in para que se encarguen de llamar al las del sistema operativo que gestiona dicho plug-in. Esta interfaz está formada por un conjunto de funciones, las cuales son definidas por todos los plug-ins de Sistemas Operativos y conocidas por la librería tdp. las principales funciones de esta interfaz son:

  - Tipos usados por las funciones:
    - *Tipo_creación*: Puede ser *NORMAL* o *PAUSADO*.
    - *String:* Cadena de caracteres.
    - *Tipo_accion:* Puede ser *PAUSAR*, *CONTINUAR* y *FINALIZAR*.
    - *Retorno:* Puede ser *ERROR*, *NO* y *OK*.

- **TDP_create_process_plugin (**
  Tipo_creación:acción,
  String:Nombre_Argumentos_proceso,
  Id_proceso_tdp:&id_proc):Retorno : Realiza la o las llamadas al sistema operativo para que este cree un proceso. El argumento *acción* indica si después de su creación, el proceso continuará en ejecución (*NORMAL*), o se dejará en estado pausado *PAUSADO*. Si no se produce ningún error, esta función devuelve OK y en *id_proc*, el *pid_tdp* del proceso recién creado. Si por el contrario se produce algún error en la creación del proceso, esta función devuelve ERROR.
  En el plug-in del sistema operativo Linux, la *creación normal* usa las llamadas al sistema *fork ()* y de la familia *exec*. La primera crea un proceso hijo devolviendo su *pid* (o error) y la segunda sustituye la imagen de este proceso hijo por la del ejecutable del proceso a crear. El nombre de este ejecutable y sus posibles argumentos son pasados a través del argumento de la función *Nombre_Argumentos_proceso*.
  La *creación pausada* se realiza a través de los siguientes pasos:

  1. El proceso padre, `tdp_console` o `tdp_agent`, usa la llamada al sistema *fork ()* para crear un proceso hijo.
  2. Se informa al proceso hijo recién creado que va a ser trazado por su proceso padre, esta acción se realiza mediante la llamada al sistema *ptrace* (con el argumento *PTRACE_TRACEME*). Este trazado es interesante porque todas las llamadas de la familia *exec* que se producen en el proceso hijo, posteriores a la de *ptrace*, provocan:
     a) Que el proceso hijo emita una señal *SIGTRAP* hacia su proceso padre
     b) Que suspenda su ejecución hasta la llegada de una señal procedente del proceso padre.
  3. Por su lado, el proceso padre espera, suspendido, la llegada de la *señal SIGTRAP* desde su proceso hijo, para ello utiliza una llamada al sistema *wait* sobre el *pid* de dicho proceso hijo. Con esta captura de la señal, el proceso padre puede saber cuando se va a iniciar la ejecución del proceso hijo (por exec ejecutado por este) que se desea crear pausado.
  4. Una vez recibida la señal, el proceso padre envía la señal *SIGSTOP* al proceso

hijo, la cual le informa que se quede en estado suspendido (en realidad que siga en el estado suspendido) y deja de trazar a este proceso, ya que no necesita seguir controlándolo. Esta última acción la realiza mediante la llamada al sistema *ptrace* (con los argumentos$PTRACE_DETACH$).

Para gestionar todos los procesos creados, este plug-in para Linux utiliza una lista: *lista_procesos_TDP-Shell*, donde cada uno de sus elementos es una estructura del tipo *Estructura_proceso_TDP-Shell*. Esta estructura, identifica y mantiene la información de cada uno de estos procesos y sus campos mas importantes son:

- *Id_proceso_TDP-Shell*: El *pid_tdp* del proceso recién creado. En el caso de este plug-in para Linux coincide con el pid devuelto por este sistema operativo (mediante la llamada fork).
- *Estado_proceso_TDP-Shell*: Contiene la información del *estado tdp* del proceso.
- *Información_proceso_TDP-Shell*: Contiene información adicional sobre el proceso, la cual puede ser la devuelta por el sistema operativo.

Cuando se ha creado un proceso correctamente, la función *TDP_create_process_plug-in*, crea una nueva estructura *Estructura_proceso_TDP-Shell* y llena sus campos con la siguientes información :

- *Id_proceso_TDP-Shell*: pid_tdp del proceso
- *Estado_proceso_TDP-Shell*: Si la creación es normal, el valor de este campo es RUNNING, si es pausado, el valor es PAUSED.
- *Información_proceso_TDP-Shell*: Información devuelta por el comando *ps*

Una vez realizada esta operación, inserta este elemento en *lista_procesos_TDP-Shell*.

- **TDP_action_process_plugin (Id_proceso_tdp:id_proc, Tipo_accion:acción):Retorno**: Realiza las llamadas al sistema operativo que se encargan de cambiar el *estado tdp* del proceso, identificado por *id_proc* y solicitado a través del argumento *acción*. La relación entre estos cambios de estado tdp y los posibles valores de este último argumento son:

- *PAUSAR*: Pausa o suspende la ejecución del proceso.
- *CONTINUAR*: Reemprende la ejecución de un proceso pausado.
- *FINALIZAR*: Finaliza la ejecución de un proceso.

Si no se puede realizar el cambio de *estado tdp* del proceso, debido a que no existe la llamada al sistema operativo que lo lleve a cavo, esta función devuelve *NO*. Si se produce un error al ejecutar la llamada al sistema operativo, devuelve *ERROR* y en caso de éxito devuelve *OK*
En la implementación del plug-in del sistema operativo Linux, estas acciones se implementan a través del siguiente algoritmo:

1: Buscar un elemento: *elemento_lista_procesos_TDP-Shell* perteneciente a la *lista_procesos_TDP-Shell* cuyo campo *Id_proceso_tdp-Shell* sea igual a *id_proc* (el proceso existe)
2: **si** *elemento_lista_procesos_TDP-Shell* existe **entonces**
3: **si** el campo *Estado_proceso_TDP-Shell* del *elemento_lista_procesos_TDP-Shell* es diferente de FINISHED o FAILED (si el proceso ya ha finalizado no se hace nada) **entonces**
4: **si** el argumento *acción* es *PAUSAR* **entonces**
5: **si** el campo *Estado_proceso_TDP-Shell* del *elemento_lista_procesos_TDP-Shell* es diferente de PAUSED (si el proceso está pausado no se hace nada) **entonces**
6: Envía una señal *SIGSTOP* al proceso *id_proc* (que como se ha mencionado anteriormente, este identificador coincide con el *pid* del sistema operativo). Este envío se realiza a través de la llamada al sistema *kill ()*, la cual tiene como uno de sus argumentos el *pid* del proceso.
7: **fin si**
8: **si no**, **si** el argumento *acción* es *CONTINUAR* **entonces**
9: **si** el campo *Estado_proceso_TDP-Shell* del *elemento_lista_procesos_TDP-Shell* es igual a RUNNING (si el proceso ya está en ejecución no se hace nada) **entonces**
10: Enviar, a través de la llamada *kill*, una señal *SIGCONT* al proceso *id_proc*

```
11:          fin si
12:        si no
13:          Enviar, a través de la llamada kill, una señal SIGKILL al proceso
             id_proc.
14:        fin si
15:      fin si
16: si no
17:    Devolver ERROR
18: fin si
```

- **TDP_info_process_plugin (**
  Id_proceso_tdp:id_proc,
  Info_estado_tdp:&info_estado):Return: Devuelve la información relacionada con el estado tdp del proceso identificado por *id_proc*. *Info_estado_tdp* es una estructura de datos compuesta por dos campos, el primero con la información del *estado tdp* del proceso y el segundo con la información más completa y más relacionada con el sistema operativo. Si existe el proceso *id_proc*, esta función devuelve OK y el estado de dicho proceso en la estructura *Info_estado_tdp* pasada como argumento por referencia *info_estado*. En caso de que no se encuentre el proceso *id_proc*, esta función devuelve ERROR. En la implementación de Linux, se sigue el mismo algoritmo de búsqueda del proceso *id_proc* en la *lista_procesos_TDP-Shell* que se utiliza en la función anterior. Si se ha encontrado el elemento en esta lista, se extraen sus campos *Estado_proceso_TDP-Shell* y *Información_proceso_TDP-Shell* para ser situados en *info_estado*.

- **Cambio estado tdp de un proceso**:En la implementación de Linux, para cambiar el estado tdp de un proceso se utiliza la llamada al sistema *sigaction ()*, la cual permite que el proceso padre pueda capturar las señales que le llegan de sus procesos hijos y llamar a una función, definida en su código, cuando esto suceda. Para realizar estas acciones, la llamada *sigaction ()* permite configurar que señales procedentes de los procesos hijos quiere capturar el proceso padre y la función especial que se llamará cuando esto ocurra. En el caso de la gestión del estado tdp de los procesos, la llamada a *sigaction ()* del plug-in, permite a `tdp_console` y `tdp_agent`, capturar las señales que les envían sus procesos

cuando: Se pausan, continúan su ejecución o finalizan. Cuando estas señales llegan, una función (definida en el plug-in) es llamada para que se encargue de modificar el estado tdp del proceso situado en la *lista_procesos_TDP-Shell* (campo Estado_proceso_TDP-Shell del elemento de esta lista al que pertenece el proceso). Para conocer el proceso que ha generado la llamada, el mecanismo de la llamada *sigaction ()* permite a la función que es llamada saber el pid de dicho proceso.

- **Comandos tdp de procesos**: Los comandos tdp de procesos que define el entorno `TDP-Shell` son los siguientes:

  - **Return = tdp_create_process [Atributo_proceso]+**: Crea y pone en ejecución un proceso en la máquina local donde se ejecuta el componente `tdp_console` o el `tdp_agent` que lo ha interpretado. Sus argumentos *[Atributo_proceso]+* definen la información que se necesita para crear el proceso (suele ser: la dirección, el nombre del ejecutable y sus posibles argumentos). Este comando tdp devuelve el *pid_tdp* del proceso creado o *ERROR* en caso de que este proceso no se haya podido crear correctamente. Para la ejecución de este comando se llama a la función *TDP_create_process_plugin* del plug-in del sistema operativo utilizado, con el valor *NORMAL* en su argumento *acción*.
  - **Return = tdp_create_process_paused [Atributo_proceso]+**: Realiza la misma función y devuelve los mismos valores que el comando anterior, pero deja el proceso hijo recién creado en estado suspendido. Para la ejecución de este comando se llama a la función del plug-in *TDP_create_process_plugin* con el valor *PAUSADO* para su argumento *acción*.
  - **Return = tdp_pause_process Id_proceso_TDP**: Suspende (o pausa) la ejecución del proceso cuyo pid_tdp es *Id_proceso_TDP*. Esta comando devuelve OK si se ha ejecutado correctamente, NO si la función de pausar un proceso no esta soportada y ERROR en caso de la ocurrencia de un error durante al pausa del proceso. Para la ejecución de este comando, se llama a la función del plug-in *TDP_action_process_plugin* con el valor *PAUSAR* en su argumento *acción*.
  - **Return = tdp_continue_process Id_proceso_TDP**: Continúa la ejecución del proceso cuyo pid_tdp es *Id_proceso_TDP*. Esta comando tdp devuelve los

mismos valore que el comando tdp de pausar un proceso, pero al ejecutar la función de continuar la ejecución de un proceso. Para la ejecución de este comando, se llama a la función del plug-in *TDP_action_process_plugin* con el valor *CONTINUAR* en su argumento *acción*.

- **Return = tdp_kill_process Id_proceso_TDP**: Finaliza la ejecución del proceso cuyo pid_tdp es *Id_proceso_TDP*. Esta comando tdp devuelve los mismos valores que los dos comandos tdp anteriores, pero al realizar la acción de finalizar la ejecución de un proceso. Para la ejecución de este comando, se llama a la función del plug-in *TDP_action_process_plugin* con el valor *FINALIZAR* en su argumento *acción*.

- **Return = tdp_process_status Id_proceso_TDP**: Devuelve la cadena de carácteres que identifica el estado tdp del proceso cuyo pid_tdp es *Id_proceso_TDP*. En caso de que el proceso no exista, devuelve *ERROR*. Este comando tdp llama a la función del plug-in *TDP_info_process_plugin* para su ejecución.

- **Return = tdp_wait_process_status Id_proceso_TDP [estados_finalización]+**: Suspende el proceso padre `tdp_console` o `tdp_agent` que llama a este comando tdp, hasta que su proceso hijo (creado a través de los comandos *tdp_create_process* o, cuyo *pid_tdp* es *Id_proceso_TDP*, se encuentre en alguno de los estados tdp situados en los argumentos *[estados_finalización]+* . Este comando tdp devuelve el estado de finalización del proceso *Id_proceso_TDP* (de *[estados_finalización]+*) o ERROR si se ha producido algún error durante la espera de dicha finalización. Por ejemplo, Si se quiere esperar a que un proceso, cuyo pid_tdp está en la variable $PROC, acabe su ejecución, de forma correcta o no, el comando tdp que realizaría esta función seria: *tdp_wait_process_status $PROC "FINISHED" FAILED"*. En el plug-in de Linux, para ejecutar este comando tdp, una función de la librería tdp: *TDP_wait_status_process*, suspende la ejecución del proceso padre que la llama (usando la llamada al sistema *usleep()* admite periodos de micro segundos) durante un intervalo de tiempo, el valor estándar del cual es de 1 segundo. El proceso padre suspendido, puede salir de este estado o bien porque le ha llegado una señal de alguno de sus procesos hijos (normalmente

porque ha sufrido un cambio de estado), o porque ha pasado el intervalo de tiempo que ha de esperar (programado por usleep). Si es el primer caso, entonces *TDP_wait_status_process* comprueba si el estado del proceso, identificado por *Id_proceso_TDP*, es alguno de los situados en *[estados_finalización]+* (utilizando la función del plug-in *TDP_info_process_plugin*). En caso afirmativo, devuelve el estado tdp del proceso y en caso contrario vuelve ha esperar otro intervalo de tiempo. Como se ha comentado, el valor estándar del intervalo de suspensión del proceso padre, que es `tdp_console` o `tdp_agent`, es de 1 segundo. Este intervalo se ha escogido para asegurar que este proceso padre pueda comprobar, a través de la función de la librería tdp *TDP_wait_status_process*, si ha llegado alguna repuesta a una petición asíncrona pendiente (como máximo cada segundo se llama al *módulo gestor de eventos*). Si el proceso padre ha salido del estado de suspendido después de que hayan transcurrido n intervalos de tiempo, la función *TDP_wait_status_process* devuelve ERROR, indicando que ha pasado un considerable intervalo de tiempo y es posible que el proceso hijo encuestado sufra algún problema. Como el tiempo de ejecución de un proceso, sobretodo si se esta monitorizando, puede ser bastante importante, el número n de intervalos de tiempo de espera han de proporcionar un periodo de tiempo lo suficientemente grande (del orden de decenas de minutos), para permitir que el proceso hijo acabe su ejecución y la función *TDP_wait_status_process* no genere un falso ERROR.

Comandos tdp de comunicación con el gestor de colas

El comando tdp de comunicación con el gestor de colas se encarga de generar el fichero de descripción de trabajos global que informa al gestor de colas del entorno de ejecución global de los componentes remotos del entorno `TDP-Shell`. Como en el caso de los comandos tdp de procesos locales, la mayor parte de la funcionalidad de este comando de comunicación con el gestor de colas, recae en las funciones del plug-in específico del gestor de colas que incorpora la librería tdp. Estos plug-ins de los gestores de colas, como los de los sistemas operativos, definen una interfaz común formada por un conjunto de funciones que son llamadas desde las funciones de la librería tdp.
La función principal de estos plug-ins de los gestores de colas tiene el siguiente formato:

- TDP_submit_job_plugin (String:descripción_fdts):Retorno: Esta función procesa los tres posibles ficheros de descripción de trabajos de los componentes remotos del entorno `TDP-Shell`, pasados a través de su argumento. De este procesamiento (explicado en el apartado 4.5 de este capítulo) obtiene el fichero de descripción de trabajos global y la información necesaria para que `tdp_console` puede generar las tuplas especiales para el componente `tdp_agent`.
Si durante la ejecución de alguna de sus acciones se produce un error esta función devuelve el valor de ERROR, en caso contrario devuelve OK. La cadena de caracteres del argumento de esta función esta formada, a su vez, por un conjunto de subcadenas que definen las características de cada fichero de descripción de trabajos de los componentes remotos de `TDP-Shell`. El formato de estas subcadenas:
Identificador_trabajo:Nombre_fichero_fdt [-a:args_fichero_fdt] [-e:nombre_ejecutable]
Donde cada una de sus campos significan:

- *Identificador_trabajo*: Cadena de caracteres que identifica el fichero de descripción de trabajos a procesar e indica el comienzo de la definición de sus posibles atributos (su definición). Los valores de este campo pueden ser: *"job user:"* para el trabajo del usuario, textit"job agent:" para el del componente `tdp_agent` y *"job tool:"* para el del componente remoto de la herramienta.

- *-a:argumentos_fichero_fdt* Para gestores de colas, como SGE, basados en la utilización shell-scripts como ficheros de descripción de trabajos, esta cadena de caracteres define sus posibles argumentos. En la cadena de caracteres *argumentos_fichero_fdt* se sitúan los argumentos en el mismo orden que son pasados al fichero shell-script y separados por un (o varios) espacios en blanco. Antes de empezar a procesar el fichero shell-script (como se ha visto en el punto 4.5.1.1) esta función del plugu-in sustituye todos los argumentos (comienzan por el carácter '$') situados en la cadena de caracteres de este fichero shell-script por sus respectivos valores situados en *argumentos_fichero_fdt*.

- *-e:nombre_ejecutable* Esta cadena de caracteres también está mayoritariamente destinada a los gestores de colas basados en ficheros shell-scripts. En ella se informa del nombre *nombre_ejecutable* del ejecutable principal de estos ficheros. El motivo de dar esta información es debido a en ciertas ocasiones este ejecutable no está situado

al principio de la línea que lo define (como supone la búsqueda estándar del algoritmo visto en 4.5.1.1), obligando al usuario a suministrar esta información al entorno `TDP-Shell`. Ejemplo de esta situación es cuando el ejecutable principal del archivo shell-script de SGE utiliza otro proceso o comando que se encarga de su ejecución.

Si algún archivo de descripción de trabajos de un componente remoto de `TDP-Shell` no es necesario que sea procesado (no necesita situar información en el espacio de ejecución remoto global), entonces sencillamente no se declara en la cadena *descripción_fdts* que se pasa como argumento a la función *TDP_submit_job_plugin*. Como se ha mencionado en este capítulo (apartado 4.4), los archivos `TDP-Shell` script están diseñados para ser independientes del gestor de colas. Uno de los principales problemas para conseguir este hecho es el caso de los gestores de colas que admiten argumentos para sus ficheros de descripción de trabajos (como SGE) de los que no (como Condor). Este característica implica que la información pasada a esta función *TDP_submit_job_plugin*, a través de su argumento *descripción_fdts* (también pasada al comando tdp que la llamará a través de la librería tdp), puede depender del gestor de colas, ya que dependiendo de si sus archivos de descripción de trabajos soportan o no argumentos, existirán o no las subcadenas cuyo prefijo es "-a:" (identifica a estos posibles argumentos) en la cadena *descripción_fdts*. Para solucionar esta dependencia, si el gestor de colas no soporta argumentos para sus archivos de descripción de trabajos, las subcadenas de *descripción_fdts* que informan de estos argumentos son ignoradas. De esta forma se puede pasar la misma cadena *descripción_fdts* para la definición de los archivos de descripción de trabajos de cualquier gestor de colas, situando en ella el mismo nombre para estos ficheros de descripción de trabajos. Por ejemplo, si al fichero de descripción de trabajos de la aplicación de usuario tanto de Condor como de SGE se le denomina *user_exec.cfg* y para SGE se le pasa un argumento con el nombre de un fichero de configuración (fichero_config.txt), la subcadena de *descripción_fdts* que definiría a estos dos ficheros de descripción de trabajos seria: *"job user: user_exec.cfg -a:fichero_config.txt"* Como se ha comentado en el apartado 4.5, del procesamiento de los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell`, la función *TDP_submit_job_plugin* obtiene información útil para el componente `tdp_agent`. Para pasar esta información al comando tdp que interpreta `tdp_console` y que es el encargado de situarla en el espacio de atributos tdp, tanto la función del plug-in como este comando tdp comparten una lista común *lista_info_fdt*,

donde sitúan la información útil obtenida de los ficheros de descripción de trabajos. Los campos de cada elemento (o estructura de datos) de esta lista son:

- *id_fdt*: Identifica a que fichero de descripción de trabajos pertenece la información. Si pertenece al de la aplicación de usuario (job_user), al `tdp_agent` (job_agent) o al componente remoto de la herramienta (job_tool).
- *identificador_información*: Identifica los diferentes tipos de información, los cuales pueden ser:
  - *Ejecutable:* Es el nombre del ejecutable
  - *Argumentos:* Son los argumentos del ejecutable.
  - *Tdp-shell script:* En el caso del `tdp_agent`, el nombre de su fichero TDP-Shell script.
- *valor_información*: El valor de la información identificada por el campo anterior.

Como se ha explicado en el punto 4.5.1.1, si la función *TDP_submit_job_plugin* no encuentra la información en los ficheros de descripción de trabajos que procesa, entonces sitúa el valor de *NOTHING* en el campo *valor_información*. Por ejemplo, los campos del elemento de esta lista que contienen la información del nombre del ejecutable de la aplicación de usuario, cuyo nombre es *user_exec*, son: *id_fdt:job user, identificador_información:Ejecutable, valor_información:user_exec*. Una vez obtenido el fichero de descripción de trabajos global, la última acción que realiza la función *TDP_submit_job_plugin* es enviarlo al gestor de colas utilizando sus comandos de envío de trabajos, los cuales son *qsub* para SGE y *condor_submit* para Condor.

Después de explicar la función principal del plug-in que se encarga del procesamiento de los archivos de descripción de trabajos de los componentes remotos de `TDP-Shell`, se puede proceder a explicar el comando tdp que la llama (a través de las funciones de la librería tdp) y que permite que se defina, en un fichero `TDP-Shell` script, la información asociada a estos archivos de descripción de trabajos.

- Retorno = tdp_launch_agent [descripción_fdts]+: Este comando tdp, crea el fichero de descripción de trabajos global, lo envía al gestor de colas y sitúa las tuplas con la información para `tdp_agent` en el espacio de atributos tdp. La concatenación de

las cadenas de caracteres y de los contenido de las variables pasadas en sus argumentos *[descripción_fdts]+*, las cuales contienen las características de cada fichero de descripción de trabajo de los componentes remotos de `TDP-Shell`, forma la cadena de caracteres que se le pasa como argumento a la función *TDP_submit_job_plugin*. Se puede observar que este comando tdp exige que haya como mínimo un argumento, ya que, como se ha comentado, el fichero de descripción de trabajos del componente `tdp_agent` es obligatorio. Para obtener las tuplas especiales para el `tdp_agent`, este comando tdp lee cada elemento de la *lista_info_fdt* (situados por la función *TDP_submit_job_plugin* del plug-in) y crea la tupla asociada a el. Una vez obtenidas las tuplas, las sitúa en el espacio de atributos tdp mediante un comando *tdp_put*. Por ejemplo, para crear la tupla que informa del nombre del ejecutable del proceso (o aplicación) del usuario, primero se busca el elemento de la *lista_info_fdt* con los campos *id_fdt = job_user* e *identificador_información = Ejecutable*. A continuación se extrae el campo *valor_información* de este elemento y se crea la tupla *[TDP_USER_EXEC, valor_información]* para enviarla al espacio de atributos tdp mediante un comando *tdp_put*.

Comandos tdp útiles y de control

El entorno `TDP-Shell` ofrece unos comandos que permiten realizar ciertas operaciones útiles y de control, estos comandos son:

- **tdp_print [información_mostrar]+**: Este comando muestra por la salida estándar (normalmente de una consola) la concatenación de las cadenas de caracteres o el contenido de las variables pasadas a través de sus argumentos *[información_mostrar]+*. Por ejemplo, si la variable *$HOLA contiene "Bienvenido"*, la interpretación del comando tdp: *tdp_print $HOLA " a TDP-Shell"* muestra "Bienvenido a TDP-Shell".

- **tdp_exit**: Con este comando tdp se fuerza el final de la interpretación de comandos tdp del fichero `TDP-Shell` script y se devuelve el control al componente `tdp_console` o `tdp_agent`. En el caso de este ultimo componente, normalmente también finaliza su ejecución por haber acabado la interpretación de su fichero `TDP-Shell` script.

- **tdp_end**: Este comando finaliza la ejecución de `tdp_console` (o de `tdp_agent`,

aunque no es habitual utilizarlo en este componente) y lo desconecta (tanto para los sockets síncronos como asíncronos) del servidor del espacio de atributos tdp.

- **tdp_end_server**: Este comando realiza las mismas funciones que *tdp_end* pero también comunica al servidor del espacio de atributos tdp que finalice su ejecución. Como se ha comentado en la explicación de este servidor, su finalización solo es realizada si existe un solo componente `tdp_console` o `tdp_agent` en ejecución (es el último).

4.7.3. Ejemplo ilustrativo de archivos TDP-Shell script

En la ficheros 4.9 y 4.10 se muestran los dos `TDP-Shell` scripts que interpretan `tdp_console` y `tdp_agent` para poder ejecutar sincronizadamente, el componente local y remoto de la herramienta Paradyn.

TDP-Shell script 4.9 Fichero TDP-Shell script para tdp_console

```
1: tdp_fun error_paradynd () {
2:   tdp_print "ERROR en la ejecución remota de paradynd"
3:   tdp_exit
4: }
5: tdp_asynget ERROR_PARADYND error_paradynd
6: $JOB_USER="job user:" "/home/user/job_user.cfg"
7: $JOB_AGENT="job agent:" "/TDP-Shell/config/job_agent.cfg"
8: $JOB_TOOL= "job tool:" "/home/user/job_gdbserver.cfg"
9: tdp_launch_agent $JOB_USER $JOB_AGENT $JOB_TOOL
10: $FILE_INFO="paradyn_ports.txt"
11: $TOOL_EXEC="/paradyn/x86_64-unknown-linux2.4/bin/paradyn"
12: $PARADYN=tdp_create_process $TOOL_EXEC "-x" $FILE_INFO
13: if ($PARADYN==ERROR){
14:   tdp_print "ERROR al ejecutar paradyn"
15:   tdp_put ERROR_PARADYN="ERROR"
16:   tdp_exit
17: }
18: $INFO=TDPRead_contents_from_file ($FILE_INFO,"1"," ","3","6")
19: tdp_put PARADYN_INFO=$INFO
20: tdp_create_process "/bin/sh" "-c" "rm" + $FILE_INFO
21: tdp_wait_process $PARADYN "FINISHED"
```

En ellos se pueden observar los siguientes puntos:

TDP-Shell script 4.10 Fichero TDP-Shell script para tdp_agent

```
1: tdp_fun error_paradyn () {
2:   tdp_print "ERROR en la ejecución de paradyn"
3:   tdp_exit
4: }
5: tdp_asynget ERROR_PARADYN error_paradyn
6: $PARADYN_INFO = tdp_get PARADYN_INFO
7: $USER_EXEC = tdp_get TDP_USER_EXEC
8: $TOOL_D_EXEC = tdp_get TDP_TOOL_EXEC
9: $USER_ARGS = tdp_get TDP_USER_ARGS
10: if ($USER_ARGS == "NOTHING") {
11:   $USER_ARGS
12: }
13: PARADYND = tdp_create_process $TOOL_D_EXEC "-z unix"
    $PARADYN_INFO "-runme" $USER_EXEC $USER_ARGS
14: if ($PARADYND==ERROR){
15:   tdp_print "ERROR al ejecutar paradynd"
16:   tdp_put ERROR_PARADYND="ERROR"
17:   tdp_exit
18: }
19: tdp_wait_process $PARADYND "FINISHED"
```

1. *Gestión de eventos*: En este ejemplo se utiliza el comando tdp de comunicación asíncrona *tdp_asynget* (línea 5 de cada fichero `TDP-Shell` script) para informar a `tdp_console` y a `tdp_agent` de la ocurrencia de un error durante la ejecución de sus respectivos componentes de la herramienta (caso típico de utilización de eventos). Como puede observarse en el fichero `TDP-Shell` script que interpreta el componente `tdp_agent`, si se produce un error en la ejecución del componente remoto de la herramienta (línea 13), este componente de `TDP-Shell` inserta en el espacio de atributos tdp (utilizando el comando *tdp_put* de la línea 16), la tupla identificada por *ERROR_PARADYND*. Esta inserción provoca que el servidor del espacio de atributos tdp, envié una respuesta al componente `tdp_console`, el cual le había informado previamente que esperaba a través del comando tdp de comunicación asíncrona *tdp_asynget* de la línea 5 de su fichero `TDP-Shell` script. Cuando `tdp_console` recibe la respuesta, su modulo gestor de eventos sabe que debe ejecutar la función *error_paradynd* (definida en la línea 1 de su

archivo `TDP-Shell` script), porque el comando *tdp_asyncget* línea 5 del archivo `TDP-Shell` script de `tdp_console`) le ha informado de este echo a través de sus dos argumentos, el primero le informa de la tupla por la cual ha de esperar, en este caso *ERROR_PARADYND* y el segundo de la función que ha de llamar si se produce este evento, la cual es *error_paradynd*.
El mismo procedimiento sigue el componente `tdp_agent` si sucede un error cuando `tdp_console` crea el componente local de Paradyn, pero en este caso la tupla que provoca la ejecución de la función *error_paradyn* (definida en la línea 1 del fichero `TDP-Shell` script de `tdp_agent`) es *ERROR_PARADYN*.

2. *Envío petición al gestor de colas*: El componente `tdp_console`, a través del comando *tdp_launch_agent* (línea 9 de su fichero `TDP-Shell` script) procesa los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell` para crear el fichero de descripción de trabajos global y situar las tuplas para el componente `tdp_agent` en el espacio de atributos tdp. Como puede observarse, este comando *tdp_launch_agent* recibe como argumentos las características de los ficheros de descripción de trabajo de la aplicación de usuario, identificado por "job user:", del componente remoto de la herramienta Paradyn, identificado "job tool:" y del componente `tdp_agent`, identificado por "job agent:". Estos ficheros de descripción de trabajos son los explicados en los puntos 4.5.1.3 para SGE y 4.5.2.3 para Condor.

3. *Sincronización e intercambio de información de los componentes de la herramienta*: Como se ha explicado en el punto 2.4.2, la herramienta Paradyn primero ejecuta su componente local, el cual crea los puertos de conexión y después ejecuta su componente remoto para que pueda conectarse a este componente local. `tdp_console` y `tdp_agent` consiguen realizar esta sincronización de los componentes de la herramienta Paradyn de la siguiente manera:

   a) El componente `tdp_agent` solicita, al espacio de atributos tdp, la información que necesita para ejecutar el componente remoto de la herramienta Paradyn (paradynd). Para ello utiliza el comando tdp *tdp_get* sobre las siguientes tuplas:

      - PARADYN_INFO (línea 6 fichero `TDP-Shell` script de `tdp_agent`): Para

obtener la Información de conexión del componente local de Paradyn (paradyn)

- TDP_USER_EXEC (línea 7 fichero `TDP-Shell` script de `tdp_agent`): Para obtener la Información del ejecutable del proceso del usuario a monitorizar.
- TDP_TOOL_EXEC (línea 8 fichero `TDP-Shell` script de `tdp_agent`): Para obtener la Información del ejecutable del componente remoto de la herramienta.
- TDP_USER_ARGS (línea 9 fichero `TDP-Shell` script de `tdp_agent`): Para obtener la Información de los posibles argumentos del proceso del usuario a monitorizar.

Si algunas de estas tuplas no esta situada en el espacio de atributos tdp, el comando *tdp_get* que la ha solicitado, suspende la ejecución del componente `tdp_agent` hasta que esta tupla este situada en dicho espacio de atributos. Cuando ha obtenido todas las tuplas (todos los *tdp_gets* han finalizado), el componente `tdp_agent` ejecuta el componente remoto de la herramienta Paradyn con el comando *tdp_create_process* de la línea 13 de su fichero `TDP-Shell` script. Si la variable $USER_ARGS (con los posibles argumentos del proceso de usuario) no tiene ningún valor (se ha declarado sin asignación, línea 10 `TDP-Shell` script de `tdp_agent`), entonces esta variable es ignorada si se pasa como argumento de un comando tdp (y en general, todas las variables que cumplen esta condición).
Con este proceder, `tdp_agent` se asegura de que puede ejecutar correctamente este componente remoto de Paradyn, porque tiene la información de todos los ejecutables y está seguro de que el componente local de esta herramienta ya esta en ejecución, debido a que `tdp_console` ha situado la tupla PARADYN_INFO (con la información de conexión) en el espacio de atributos tdp.

b) El componente `tdp_console`, después de solicitar al gestor de colas la ejecución de `tdp_agent`, interpreta el comando *tdp_create_process* de la línea 12 de su fichero `TDP-Shell`, para ejecutar el componente local de Paradyn. Con el argumento *-x* se solicita a este componente de la herramienta que sitúe su

información de conexión en el fichero *paradyn_ports.txt*. Esta información tiene el siguiente formato: *paradynd -zflavor -l1 -m<nombre_host> -p<puerto_1> -P<puerto_2>* (ver punto 2.4.2 para más información). A continuación `tdp_console` procesa este fichero con la función especial del entorno `TDP-Shell` *TDPRead_contents_from_file* (línea 18 del fichero `TDP-Shell` script de `tdp_console`) para concatenar las tres últimas cadenas de caracteres de su 1ª (y única) línea. De esta concatenación se obtiene la cadena de caracteres *"-m<nombre_host> -p<puerto_1> -P<puerto_2>"* con la información de conexión que necesita el `tdp_agent` para poder ejecutar el componente remoto de Paradyn. Una vez obtenida dicha información, `tdp_console` la sitúa en el espacio de atributos tdp utilizando el comando *tdp_put*(línea 19 de su archivo `TDP-Shell` script) sobre la tupla *PARADYN_INFO* (e indicando también a `tdp_agent` que el componente local de Paradyn ya está en ejecución).

c) Una vez en ejecución los componentes de la herramienta, tanto `tdp_console` como `tdp_agent` esperan la finalización de dichos componentes con el comando tdp *tdp_wait_process* (línea 21 fichero `TDP-Shell` script de `tdp_console` y 19 del fichero `TDP-Shell` script de `tdp_agent`)

4.8. Conclusiones

Para solucionar el problema de la falta de interoperabilidad entre los gestores de colas y herramientas de monitorización cuando se monitorizan aplicaciones serie (formadas por un sola proceso) la arquitectura base del entorno `TDP-Shell` utiliza los siguientes tres componentes principales:

1) **Componente tdp_console**: Normalmente se ejecuta en la máquina local del usuario y se encarga de crear el componente local de de la herramienta y de solicitar al gestor de colas que ejecute el componente remoto del entorno `TDP-Shell` (`tdp_agent`).

2) **Componente tdp_agent**: Se ejecuta en la máquina remota (creado por el gestor de colas), encargándose de la creación del componente remoto de la herramienta de monitorización y en caso necesario, del proceso del usuario a monitorizar.

3) **Espacio de atributos tdp**: Almacena la información, en formato de tuplas, que

necesitan intercambiar `tdp_console` y `tdp_agent`, para que puedan realizar sus funciones. Ofrece unas primitivas síncronas y asíncronas (get y put) para situar y extraer las tuplas ubicada en el.

`tdp_console` procesa la información contenida en los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell` (aplicación de usuario, componente remoto de la herramienta y `tdp_agent`) para obtener un nuevo *fichero de descripción de trabajos global* el cual describe el *entorno de ejecución global* para estos componentes remotos. Este entorno de ejecución permite, al gestor de colas, realizar las acciones necesarias que le permiten ejecutar al componente `tdp_agent` y ayudar a la correcta ejecución del resto de componentes remotos del entorno `TDP-Shell`. Los archivos especiales `TDP-Shell` script, uno interpretado por el `tdp_console` y el otro por el `tdp_agent`, definen el orden temporal de envío y recepción de la información que necesitan ambos componentes del entrono `TDP-Shell` para ejecutar, de una forma sincronizada, el componente local y remoto de la herramienta de monitorización. Estos ficheros tienen un formato parecido a los ficheros shell-script y están formado por un conjunto de:

- *Comandos tdp de comunicación con el espacio de atributos tdp*: Encargados de realizar el intercambio de información (situar y extraer) con el espacio de atributos tdp. Existen dos tipos de estos comandos, los bloqueantes que suspenden la ejecución del proceso hasta la finalización de la operación de comunicación y los no bloqueantes, que no la suspenden y una función especial es la encargada de finalizar la comunicación.

- *Comandos tdp creación y control de procesos locales*: Son los encargados de operar (crear, pausar, continuar la ejecución y obtener la información del estado) con los procesos que se van a ejecutar en una maquina determinada (local del usuario o remota del cluster). Para realizar estas funciones utilizan los servicios del sistema operativo local de dicha máquina.

- *Comandos tdp de comunicación con el gestor de colas*: Propios del entorno de trabajo `TDP-Shell`, son utilizados por `tdp_console` para procesar los ficheros de descripción de trabajos de los componentes remotos del entorno `TDP-Shell` y obtener el fichero de descripción de trabajos global.

- *Instrucciones especiales* : Se encargan de añadir funcionalidad a los ficheros `TDP-Shell` script (basados en el lenguaje C). Pueden ser *bucles (while)*, *instrucciones condicionales (if, then, else)*, *variables*, *asignaciones y declaraciones de funciones locales.*

La separación del entorno de ejecución de los componentes remotos del entorno `TDP-Shell` del proceso de sincronización de los componentes de la herramienta de monitorización, permite simplificar las posibles soluciones que ofrece el entorno `TDP-Shell` al problema de la interoperabilidad entre los gestores de colas y herramientas de monitorización. Si se cambia de gestor de colas, solo es necesario declarar los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell`, manteniendo los ficheros `TDP-Shell` script de las herramientas de monitorización.

5

Arquitectura de TDP-Shell para entornos distribuidos MPI

En el capítulo anterior se ha explicado el diseño de la arquitectura base del entorno `TDP-Shell`, centrado en solucionar el problema de falta de interoperabilidad entre gestores de colas y herramientas de monitorización para aplicaciones serie, las cuales solo ponen en ejecución un proceso. Pero la mayoría de aplicaciones que están pensadas para ejecutarse en un cluster de ordenadores no cumplen esta condición, son aplicaciones distribuidas que ponen en ejecución un conjunto de procesos en las máquinas de dicho cluster. Dentro de la programación distribuida, una de las técnicas más utilizadas es el *paso de mensajes*. Esta técnica se basa en que los procesos, para realizar sus tareas, se intercambien datos a través de mensajes (de ahí su nombre). El paradigma basado en la técnica del paso de mensajes que se ha transformado prácticamente en un estándar es MPI. Este paradigma propone una interfaz que define, a través de un conjunto de rutinas agrupadas en una librería, la sintaxis y la semántica de los sistemas de paso de mensajes. Entre las características y objetivos que ofrece MPI se pueden destacar:

- Portabilidad de su código para que sus rutinas puedan ser eficientemente implementadas sobre un gran número de plataformas y que sean fácilmente utilizables por los usuarios.

- Ofrecer mecanismos de comunicación entre procesos, tanto los que involucran a dos procesos (punto a punto) como a un grupo de procesos (colectivos).

- Permitir agrupar a los procesos en grupos, así como definir las comunicaciones entre los procesos del mismo grupo (intra-comunicaciones) y entre procesos de diferentes grupos (inter-comunicaciones).
- Permitir definir diferentes topologías para los procesos de un grupo (anillo, malla, redes n-dimensionales, etc), con la finalidad que se adapten correctamente a sus patrones de comunicaciones.
- El standard MPI (versiones 1.X) no define como crear y gestionar procesos dinámicamente (en tiempo de ejecución, como lo hace PVM [39] [40]), pero la versión 2 de MPI, que incluye todas las funciones de la versión 1, si que lo permite.

En los últimos 15 años han aparecido bastantes implementaciones de la librería MPI (a partir de ahora librerías MPI). De estas implementaciones destacan *MPICH [41] , LAM [42] y OpenMPI[43]* por su gran aceptación entre los usuarios, profesionales e investigadores. Los motivos mas importantes de esta aceptación son los siguientes:

- Son *open source*, estas librerías se pueden instalar libremente en las máquinas donde se quieran utilizar.
- Han sido diseñadas para que puedan ser utilizadas en una gran variedad de plataformas (tanto a nivel de sistema operativo como de hardware) y de lenguajes de alto nivel (existen versiones de las funciones de la librería MPI para Fortran, C o C++).

Mientras que la interfaz que ofrecen estas librerías MPI para los diferentes lenguajes de alto nivel, debe ser la misma para las diferentes plataformas, su diseño e implementación interna normalmente es diferente. Por ejemplo MPICH utiliza, por defecto el protocolo *ssh* para la creación remota de los procesos, mientras que LAM y OpemMPI utilizan para este mismo propósito, unos componentes o demonios remotos denominados *lamd*, para LAM y *orted*, para OpenMPI.

Debido a la gran aceptación de las implementaciones de MPI, tanto los gestores de colas Condor o SGE, como las herramientas de monitorización estudiadas en este trabajo (Gdb, Paradyn o Totalview) se han adaptado para dar soporte a aplicaciones distribuidas basadas en estas implementaciones MPI, principalmente MPICH, LAM u

OpenMPI.
Todas estas implementaciones de MPI soportan las dos versiones de MPI, pero debido a que la gestión dinámica de procesos que implementa su versión 2, puede provocar problemas con la gestión de los trabajos que realizan los gestores de colas (igual que sucede con las herramientas de monitorización), estos normalmente soportan completamente la versión 1 de MPI (o esta parte en las implementaciones de MPI 2). Por ejemplo, Condor ofrece la posibilidad de lanzar trabajos basados en la librería MPICH1 (implementación de MPI 1), sin embargo no garantiza que funcionen correctamente los basados en MPICH2 (implementación de MPI 2). Por este motivo, para este trabajo de tesis se ha escogido trabajar con la implementación de la versión 1 de MPI que ofrecen MPICH, LAM u OpenMPI, ya que es soportada por la mayoría de gestores de colas y herramientas de monitorización.

Como en el caso de las aplicaciones serie, en la monitorización de aplicaciones distribuidas basadas en MPI, tampoco existe la interoperabilidad entre los gestores de colas y las herramientas de monitorización. Como solucionar este problema será el tema de estudio de los siguientes apartados de este capitulo. Primero se explicará como trabajan los gestores de colas y las herramientas de monitorización con aplicaciones distribuidas MPI, para explicar, a continuación, como partiendo de la estructura de la arquitectura base del entorno `TDP-Shell` (implementada para el caso serie), se obtiene la nueva versión de este entorno, que consigue la interacción entre los gestores de colas y las herramientas de monitorización para permitir que estas ultimas puedan monitorizar aplicaciones distribuidas MPI [44] [45].

5.1. aplicaciones MPI sobre gestores de colas y herramientas de monitorización

Antes de explicar las diferentes formas que ofrecen los gestores de colas Condor y SGE así como las herramientas de monitorización Gdb, Paradyn y Totalview para ejecutar y monitorizar (en caso de estas últimas) aplicaciones MPI, es interesante destacar que para hacerlo aprovechan los entornos de ejecución de las librerías MPI. Las características mas destacadas de estos entornos de ejecución son:

1. Ofrecen un programa, normalmente llamado *mpirun (o mpiexec)*, el cual es el encargado de crear el entorno de ejecución que permita la correcta puesta en marcha de los diferentes procesos de una aplicación MPI. Para que pueda crear este entorno, a este programa mpirun se le han de suministrar ciertas informaciones, las mas destacadas son:

   - Las máquinas donde se han de ejecutar los diferentes procesos de la aplicación MPI. Normalmente, esta información se pasa a través de un fichero, donde cada una de sus líneas define a una máquina. Su formato básico es:

     <nombre máquina> <número de CPU's/cores o procesos por máquina>

   - El número de procesos que va a generar la aplicación MPI, pasado a través del argumento de mpirun "-np" y que se distribuirán por las diferentes máquinas declaradas en el fichero del punto anterior. La distribución no tiene que ser necesariamente 1 proceso por máquina o CPU, puede haber varios procesos por máquina (sobretodo con las CPU's multicore).

2. La metodología que ofrecen MPICH1, LAM u OpenMPI para ejecutar los procesos en las máquinas remotas es diferente. MPICH1 pone en ejecución, normalmente en la máquina local donde se ejecuta el programa mpirun, un proceso de la aplicación MPI denominado padre (con identificador 0). Este proceso se encarga de poner en ejecución al resto de los procesos de la aplicación MPI, denominados procesos hijos (con identificadores >0). Esta creación se realiza dentro de la función de la librería *MPI_Init* situada en el código ejecutable del proceso padre. Para ello, la versión standard de MPICH1 (ch-p4) utiliza la herramientas de conexión remota ssh (también permite rsh, pero por motivos de seguridad no se recomienda su uso). Para que esta herramienta de conexión remota pueda ejecutar los procesos en las máquinas remotas, es necesario que estén en ejecución sus demonios sshd en ellas. Por lo tanto, en aplicaciones distribuidas basadas en la librería MPICH, es imprescindible que estos demonios sshd estén activos en cada máquina antes de proceder a la ejecución de los procesos MPI hijos. Si no están activos, el programa mpirun no se encarga de ponerlos en ejecución, suponiendo que alguna aplicación externa o persona los pondrá en ejecución.
   Como se ha comentado en la introducción de este capítulo, las implementaciones

de la librería MPI de LAM y OpenMPI no utilizan ssh para ejecutar los procesos de sus aplicaciones, utilizan unos demonios propios denominados lamd para LAM y orted para OpenMPI. En el caso de LAM, antes de que su programa mpirun ejecute cualquier proceso de su aplicación MPI, deben estar activos los servidores lamd en todas las máquinas donde se van a ejecutar dichos procesos. Como en el caso de MPICH, el programa mpirun de LAM tampoco se encarga de realizar esta función, para hacerlo se dispone de un programa especial, denominado *lamboot*, el cual utiliza el fichero de máquinas para conocer las máquinas donde se han de ejecutar los demonios lamd. En el caso de OpenMPI las direcciones donde están situados los ejecutables del demonio orted han de ser conocidas en todas las máquinas (normalmente son las del directorio donde se instala la distribución de OpenMPI por defecto).

3. Tanto MPICH, como LAM y OpenMPI suponen que en las máquinas donde se ejecutan los procesos MPI, los códigos ejecutables de estos están situados en directorios conocidos (puede ser a través de un sistema de ficheros distribuido) y con sus correspondientes permisos de ejecución correctos. Al programa mpirun se le pasa el directorio (si es necesario) y el nombre del ejecutable de la aplicación MPI, este directorio y nombre es utilizado por ssh (MPICH), por lamd (LAM) o orted (OpenMPI) para conocer la situación de los ejecutables de los procesos de esta aplicación MPI y poderlos ejecutar en cada máquina.

Una vez explicados los entornos de ejecución de las diferentes implementaciones de MPI, en los siguientes puntos de este apartado se explicarán como los aprovechan los gestores de colas y las herramientas de monitorización

5.1.1. Ejecución de trabajos MPI sobre Condor y SGE

Condor y SGE ofrecen unos entornos especiales para trabajos distribuidos. Por su parte, Condor ofrece el *universo paralelo* [46] y SGE el *entorno paralelo o pe* [47]. Dentro de estos entornos distribuidos se pueden configurar de diversos tipos, entre ellos el mpi (MPICH, LAM para Condor y SGE así como OpenMPI para SGE). Para que los entornos de trabajos paralelos de Condor y SGE puedan ejecutar correctamente los trabajos MPI, es necesario que primero puedan poner en marcha el entorno de ejecución que utilizan

las diferentes implementaciones de la librera MPI que soportan. Por ejemplo, antes de ejecutar la aplicación MPI, LAM necesita que estén en ejecución sus demonios lamd y en el caso de MPICH1, los demonios sshd que necesita la herramienta de conexión remota ssh. Por lo tanto, para la correcta puesta en marcha de estos entorno de ejecución de MPI, los dos gestores de colas necesitan obtener cierta información de ellos (variables de entorno, fichero de máquinas, situación de los servidores sshd, lamd u orted, etc). Para ello tanto Condor como SGE ofrecen unos scripts especiales configurables, para que sus administradores (o usuarios) puedan introducir esta información. Condor ofrece un script para MPICH1 (*mp1script*) y otro para LAM *lamscript*. Por su parte, SGE ofrece, para cada tipo de su entorno paralelo, dos clases de scripts especiales, unos para su inicialización (acciones a realizar antes de la ejecución del script del trabajo distribuido), los cuales suelen denominarse con el prefijo *start* y otros para su finalización (acciones a realizar después de la ejecución del script del trabajo distribuido), los cuales suelen denominarse con el prefijo *stop*. Dentro del entorno distribuido MPI de SGE se ofrecen los scripts de inicio y finalización para MPICH, LAM y OpenMPI.

Estos scripts especiales que utilizan Condor y SGE, afectan a la manera en que se declaran sus ficheros de descripción de trabajos para aplicaciones distribuidas MPI. Debido a este motivo, es necesario el estudio de las modificaciones que sufren estos ficheros para que el componente `tdp_console` los pueda procesar correctamente y obtener el fichero de descripción de trabajos global. En el caso de Condor, los ficheros de descripción de trabajos para aplicaciones distribuidas MPI se declaran a través de los siguientes comandos:

- **universe = parallel** : Se indica a Condor que el trabajos es del tipo distribuido.
- **executable = [mp1script | lamscript]** : Como ejecutable se declara el script especifico de la implementación de la librería MPI : mp1script para MPICH y lamscript para LAM. Este script se encarga del lanzamiento de la aplicación MPI definida en el trabajo (utiliza el comando mprun).
- **arguments = <Ejecutable de la aplicación MPI> [args]***: Como argumentos al script especifico se le pasan el ejecutable de los procesos de aplicación MPI y sus posibles argumentos.
- **machine_count = n** : Este comando indica el número de procesos (n) que

generará la aplicación MPI (valor del argumento -np de mpirun).

Para el caso del gestor de colas SGE, la utilización de shell-scrips como ficheros de descripción de trabajos MPI, permite la existencia de una gran variedad de posibilidades a la hora de declarar estos trabajos. Pero existe una forma básica (punto de partida de muchos scripts) de realizar esta declaración, la cual se muestra en los siguientes puntos:

1. Declaración de las directiva especial de SGE para entornos distribuidos MPI, la cual es:

   - **-pe mpi n1-n2 :** Directiva de SGE a través del cual se le informa de que el trabajo utiliza el entorno paralelo (pe) y que es del tipo MPI, además también indica que estará formado por entre n1 y n2 procesos(n1 y n1 pertenecen a los números Naturales). Al interpretar esta directiva, SGE ejecuta sus scripts especiales del entorno paralelo MPI. El de inicialización antes de ejecutar el cuerpo principal de shell-script de descripción del trabajo MPI y el de finalización al terminar la ejecución de este shell-script.

2. Declaraciones e información útil situada en el cuerpo principal del script:

   - **Fichero $TMPDIR/machines:** Fichero, devuelto por el script especial de inicialización del para el entorno MPI de SGE, el cual informa de las máquinas escogidas para la ejecución de los procesos de la aplicación MPI definida en el trabajo. Este número de máquinas dependerá de los números n1 y n2 pasados en el comando de SGE *-pe*. Tanto el nombre del directorio temporal *$TMPDIR*, como en nombre del fichero, denominado por defecto *machines*, son configurables a través del el script especial de inicialización del para entornos MPI de SGE.
   - **El Comando mpirun (o mpiexec)** es utilizado para poner en marcha la ejecución de la aplicación MPI (como si se tratará de un cluster no controlado por un gestor de colas). En caso necesario, se puede utilizar el fichero *$TMPDIR/machines* para informar de las máquinas donde se va a ejecutar los procesos de la aplicación MPI. Por ejemplo, para la versión MPICH1 se utiliza el argumento especial de mpirun *-machinefile $TMPDIR/machines*, para informarle de la dirección y nombre del fichero que contiene la descripción de estas de máquinas.

Una vez estudiado como definir aplicaciones distribuidas MPI a través de los archivos de descripción de trabajos de Condor y SGE, el ultimo punto de estudio es como estos gestores de colas, ponen en ejecución los procesos de las aplicaciones MPI definidas en sus respectivos trabajos. Este estudio es importante porque definirá la manera en que Condor y SGE van a ejecutar los componentes `tdp_agent` cuando gestionan este tipo de aplicaciones distribuidas bajo el entorno `TDP-Shell`. Como se verá a continuación, el procedimiento que siguen ambos gestores de colas son muy similar y se ha resumido en los siguientes pasos:

1. Localizar las máquinas que tengan los suficientes recursos para ejecutar los procesos de la aplicación MPI definida en el trabajo del usuario (figura 5.1, punto segundo). Una vez localizadas, los scripts especiales del entorno distribuidos MPI de Condor y SGE informan a los entornos de ejecución de la distribución de MPI que utilizan (MPICH, LAM u OpenMPI) de estas máquinas seleccionadas, normalmente a través de un fichero, denominado *de máquinas*, con el formato que se ha explicado al principio de este apartado. En Condor el encargado de localizar las máquinas con suficientes recursos es un planificador especial ,denominado dedicado [48], instalado en una máquina del conjunto que controla dicho gestor. En SGE, el proceso de selección de las máquinas se realiza dentro del script de inicialización del entorno distribuido mpi, el cual procesa el fichero especial de SGE, denominado *sge_hostfile*, que contiene la lista de todas las máquinas de su cluster que pueden ejecutar procesos de las aplicaciones MPI. Este fichero, creado normalmente por el administrador del sistema, se le puede pasar como argumento a los scripts de inicialización o estar definido en una variable de entorno especial de SGE (la forma de hacerlo la definirá el script de inicialización).

2. Ejecutar en las máquinas seleccionadas en el punto anterior, los demonios (o componentes remotos) de las herramientas de conexión remota (sshd para el caso de ssh y MPICH1) o los demonios (lamd para LAM y orted para OpenMPI) propios de la implementación de MPI utilizada (figura 5.2, punto 1). En el caso de LAM, para ejecutar sus demonios remotos, los scripts especiales del entorno distribuido MPI de los gestores de colas utilizan el programa lamboot, pasándole como argumento el fichero de máquinas generado en el paso 1. Hay que destacar que Condor, después de la selección de las máquinas del paso 1, ejecuta en una máquina el proceso cuyo

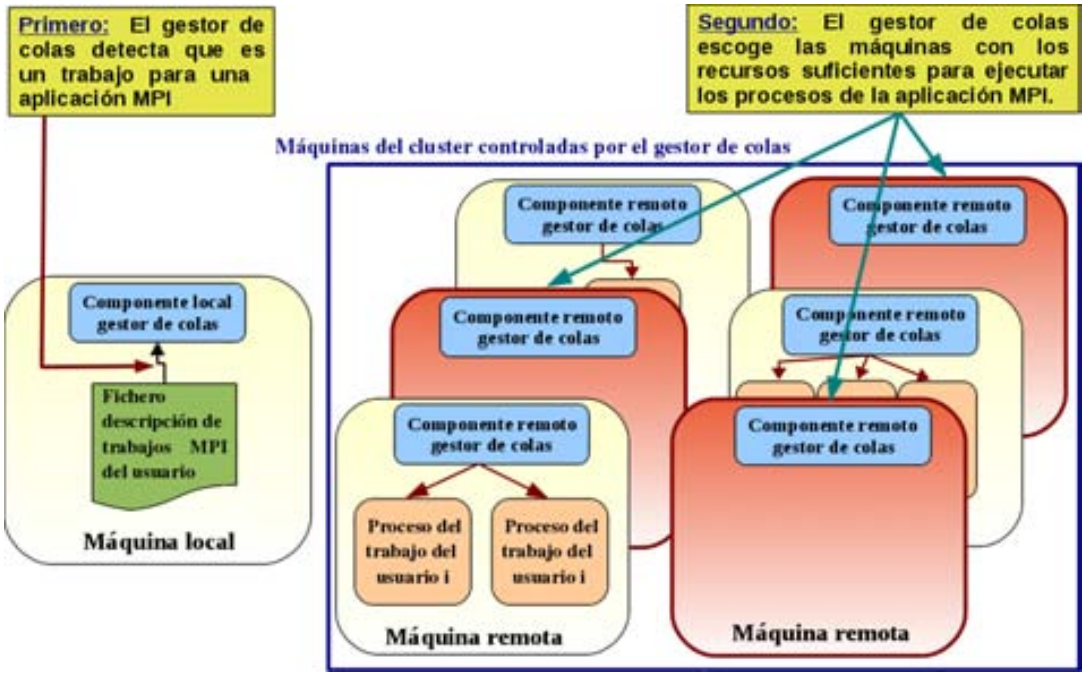


Figura 5.1: Selección de las máquinas con los suficientes recursos

código ejecutable se ha declarado en el comando execute del fichero de descripción de trabajos enviado por el usuario. En este caso, el código de este proceso será el script especial del entorno paralelo (mp1script o lamscript), consiguiendo de esta manera que su ejecución realice las acciones explicadas en este paso para el entorno de ejecución de la librería MPI escogida.

3. Una vez están activos estos demonios ya se puede ejecutar la aplicación MPI del trabajo definido por el usuario. La manera de proceder respecto a esta ejecución es diferente en cada gestor de colas. Condor utiliza su script especial del el entorno MPI para ejecutarla. Este script llama al programa mpirun, cuyo argumento del código ejecutable de los procesos de la aplicación MPI es obtenido gracias a que también se le pasa como argumento de este script especial de Condor (como se ha explicado en la declaración de los archivos de descripción de trabajos MPI de este punto 5.1.1). SGE por su lado, utiliza su metodología de ejecución standard, ejecutando en una máquina de las seleccionadas, el script que define el trabajo paralelo MPI que usuario le ha enviado y el cual contiene, entre otros comandos, el programa mpirun y el ejecutable de dicha aplicación MPI (figura 5.2, punto 2).

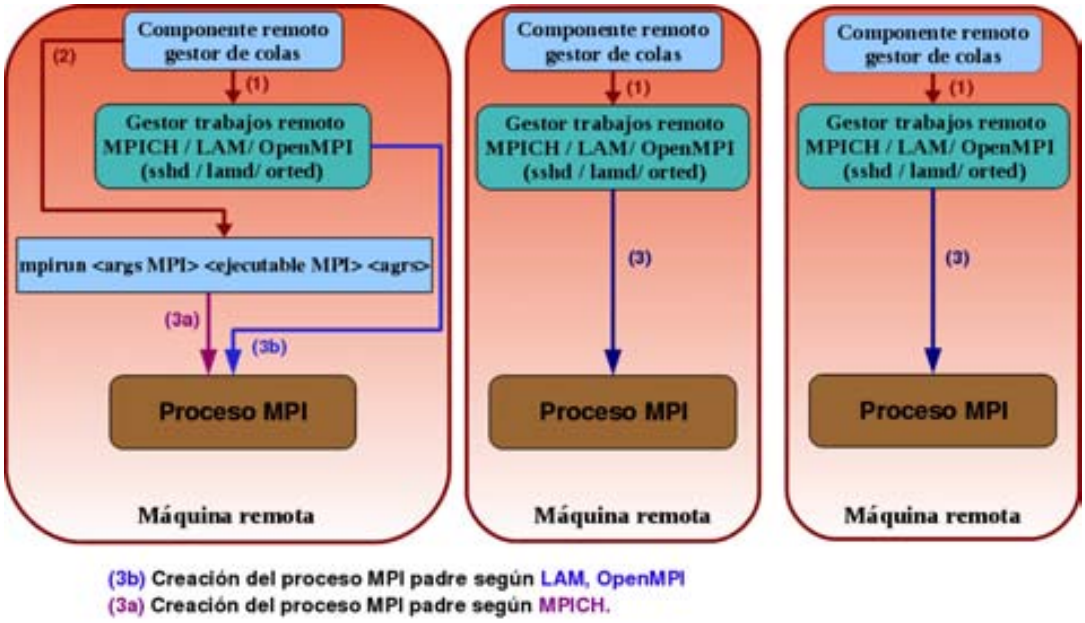


Figura 5.2: Ejecución de los procesos MPI bajo el control del gestor de colas

Una vez entre en ejecución el programa mpirun, este se encargará de la correcta puesta en marcha de los procesos de la aplicación MPI, como si hubiera sido ejecutado directamente por el usuario y no a través del gestor de colas . En la figura 5.2, se pueden observar las diferencias que existen entre LAM, OpenMPI con MPICH respecto a la metodología seguida en ejecución los diferentes procesos MPI, sobre todo con el proceso padre. En la implementación de MPICH este proceso es ejecutado por mpirun para que posteriormente se encargue de la ejecución de los procesos hijos (figura 5.2, punto 3a y 3), por el contrario en el caso de LAM y OpemMPI son sus respectivos demonios lamd y orted, los encargados de la ejecución de todos los procesos MPI (figura 5.2, puntos 3b y 3).

5.1.2. Monitorización de aplicaciones MPI sobre Paradyn, Gdb y Totalview

Las herramientas de monitorización objeto de estudio en este trabajo de tesis, Paradyn, Totalview y Gdb, se han adaptado para soportar el tipo de aplicaciones distribuidas basadas en las diferentes implementaciones de MPI, entre ellas MPICH, LAM y OpenMPI. Esta adaptación se ha hecho principalmente de dos formas:

a) **Integrada dentro del entrono de ejecución de la librería de MPI**
La herramienta de monitorización adapta (y normalmente modifica) el entorno de ejecución de la librería MPI para que sus componentes puedan realizar correctamente su función de monitorización con los procesos de la aplicación MPI. Esta funcionalidad integrada requiere, tanto la modificación del código de los componentes de la herramienta como de los de la librería MPI (por ejemplo el código del programa mpirun). La herramienta de monitorización Totalview utiliza esta funcionalidad integrada para monitorizar aplicaciones basadas en la librería MPICH1 (igualmente con LAM u OpenMPI). Esta herramienta, a través un programa especial denominado *mpirun_dbg.totalview*, el cual es llamado desde el programa mpirun (con el argumento -tv), crea el entorno de ejecución adecuado para realizar las operaciones de monitorización sobre aplicaciones MPI que ella necesita. Este programa especial crea el componente local de la herramienta y a continuación, este componente local, ejecuta el proceso MPI padre (como hace el programa mpirun en la versión standard de MPICH1), pasándole el parámetro especial *-mpichtv*, el cual le informa que prepare a los procesos MPI hijos que el genere, para que puedan ser monitorizados por esta herramienta.

b) **Aprovechando el entorno de ejecución de la librería MPI**
En este caso no se modifica el código de la librería MPI, sino que es el componente local de la herramienta de monitorización o el usuario (si la herramienta permite la ejecución manual de sus componentes remotos), el encargado de ejecutar el programa mpirun, cambiando el nombre del ejecutable de la aplicación MPI, que se le pasa como argumento, por el del componente remoto de la herramienta o el de un script especial que ejecute estos componentes remotos. De esta forma se consigue que se pongan en ejecución, en las diferentes máquinas del cluster, los componentes remotos de la herramienta para que posteriormente creen los procesos MPI, los monitoricen e intercambien la información con su componente local.
Este sistema es utilizado por la herramienta Paradyn, la cual permite monitorizar tanto aplicaciones basadas en la librería MPICH1 como LAM. Igual que se explicó para el caso de monitorización de aplicaciones serie, esta herramienta ofrece dos métodos para la puesta en ejecución de sus componentes remotos que monitorizarán los procesos de la aplicación MPI. El primero es el automático desde su

componente local (el recomendado) y el segundo es el manual, realizado por un usuario o una aplicación externa. De estos dos métodos, el automático tiene el mismo problema que cuando se monitorizan aplicaciones serie, normalmente utiliza la herramienta de conexión remota ssh para poner en ejecución sus componentes remotos, entrando en conflicto con la gestión que realiza el gestor de colas. Por este motivo, el método a utilizar para el diseño del entorno `TDP-Shell` para la herramienta Paradyn cuando monitorizan aplicaciones MPI, es el manual. Este método implica seguir los siguientes pasos importantes:

a) Obtener la información (argumentos) para los componentes remotos de la herramienta: Para ello realizamos el mismo proceder que se vio en el apartado 2.4.2 para monitorizar aplicaciones serie, ejecutamos el componente local de Paradyn con el argumento -x y el nombre del fichero donde debe situar la información de conexión.

b) De este fichero con la información de conexión, se obtienen la mayoría de argumentos para el componente remoto de Paradyn, *paradynd*, pero como se van a monitorizar aplicaciones MPI, algunos de estos argumentos difieren de los que se le pasan a este componente en el caso de aplicaciones serie, los cuales son:

- *-zmpi:* El tipo de aplicación que se va a monitorizar es MPI
- *[-MMPICH | -MLAM]:* Informa de la implementación de MPI que utiliza la aplicación distribuida, *-MMPICH* si es MPICH o *-MLAM* si es LAM.

Por lo tanto la línea del ejecutable del componente remoto de Paradyn para aplicaciones MPI será:

*> paradynd -zmpi -l<n> -m<nombre máquina> -p0 -P<puerto comunicaciones> [-MMPICH | -MLAM] -runme <ejecutable mpi> [<argumentos ejecutable mpi>]**

Donde los campos *<ejecutable mpi> y <argumentos ejecutable mpi>* son el nombre del ejecutable y los posibles argumentos de los procesos de la aplicación MPI.

c) Una vez obtenida la información para poder ejecutar el componente remoto

de paradyn, ya se puede utilizar el programa mpirun para que se encargue de ejecutar estos componentes remotos y que estos, a su vez, se encarguen de ejecutar los procesos de la aplicación MPI a monitorizar. Para realizar este procedimiento se puede sustituir el nombre del ejecutable de los procesos de la aplicación MPI, pasado como argumento de este programa mpirun, por el del ejecutable del componente remoto de Paradyn.
Dependiendo de la implementación de MPI que se utilice, este proceder puede no funcionar directamente. En el caso de MPICH1, existe el problema que no se permiten pasar argumentos de usuario a los procesos hijos de la aplicación MPI (solo admiten argumentos de comunicación de la propia librería). Esto significa que usando el entrono de ejecución de esta implementación de MPI, los componentes remotos de la herramienta Paradyn hijos (creados en lugar de los procesos hijos de la aplicación MPI) no pueden recibir los argumentos de comunicación con su componente local. Para solucionar este problema, se puede crear un fichero shell-script que se encargará de ejecutar paradynd con estos argumentos de conexión y que será ejecutado por mpirun. Por lo tanto, con este shell-scrip especial, la ejecución de los componentes remotos de Paradyn sobre MPICH1 se realizará de la siguiente manera:

1) mpirun ejecuta, en la máquina local del usuario, el shell-script especial para los componentes paradynd padre.
2) Este shell-script especial padre, ejecuta el componente paradynd padre y este a su vez el proceso de la aplicación MPI padre, pasándole como uno de sus argumentos, el shell-script especial como el nombre del ejecutable de los procesos hijos que ha de crear este proceso MPI padre.
3) En cada máquina seleccionada para la ejecución de un proceso de la aplicación MPI, se ejecuta el shell-script especial hijo, puesto en ejecución por el proceso MPI padre usando la herramienta de conexión remota ssh.
4) Cada uno de estos shell-scripts hijos debe capturar los argumentos de comunicación especiales de la librería MPICH1 (a partir de ahora argumentos de MPICH) que les ha pasado el proceso padre MPI y que necesitan los procesos hijos de dicha aplicación. Una vez tiene estos argumentos de MPICH, cada shell-script especial hijo ejecuta el componente

paradynd hijo con sus correspondientes argumentos de conexión con el componente local de la herramienta y como su argumento del ejecutable, el proceso de la aplicación MPI con sus argumentos MPICH.

5) Cada componente paradynd pone en ejecución el proceso MPI hijo y se conecta con su componente local de Paradyn para realizar el proceso de monitorización.

En el caso de la librería LAM, no existe esta restricción respecto a pasar argumentos de usuario a sus procesos hijos, por lo tanto, pasando a mpirun el componente paradynd con sus argumentos (entre ellos el del ejecutable de los procesos de la aplicación MPI), se consigue que, a través del demonio lamd, este componente remoto de Paradyn se ejecute en las máquinas seleccionadas y que pueda ejecutar el proceso MPI a monitorizar.

Con la herramienta Gdb también se puede utilizar un método parecido al se utiliza con Paradynd, este método consiste en que mpirun ponga en ejecución, en las máquinas del cluster seleccionadas, los componentes gdbserver (directamente o a través de un shell-script, si se utiliza la librería MPICH). Posteriormente, para cada uno de estos componentes remotos de la herramienta Gdb, se ejecuta, en la máquina local del usuario, un componente local de esta herramienta que se conecta al gdbserver recién creado. Por lo tanto, al final se tienen tantos componentes locales de Gdb como gdbservers se han creado y cada uno de ellos monitoriza un proceso de la aplicación MPI .
Para poder realizar este método, un usuario o una aplicación externa, han de solucionar los siguientes problemas:

a) Como se vio en 2.4.1 a los componentes remotos de Gdb se les ha de pasar, como argumento, un puerto de comunicaciones libre, que tiene que conocer el componente local de esta herramienta para poder conectarse con el. Para obtener este puerto libre y ejecutar correctamente el componente gdbserver se pueden seguir dos métodos:

1) Usar un shell-script o aplicación especial (puede ser el mismo que se utiliza para MPICH) que obtenga un puerto libre de la máquina donde

se ha de ejecutar el gdbserver, para pasárselo como argumento y de esta manera, poderlo ejecutar correctamente. Posteriormente, este shell-scrip o aplicación especial, ha de publicar el puerto de comunicaciones escogido para el componente gdbserver, para que sea conocido por el componente local de Gdb que se le va a conectar. Esta publicación se puede realizar a través de un fichero especial, lo cual obliga al usuario o a la aplicación externa, que utilizan a mpirun para ejecutar los componentes gdbserver, a acceder a las máquinas donde se ejecutan estos componentes (normalmente vía terminal remoto) y esperar a que se creen estos ficheros especiales. Una vez creados, pueden leer su información y obtener el puerto de conexión. Para realizar esta publicación de los puertos de conexión, se pueden usar otros métodos más elaborados (dependerá de los conocimientos técnicos del usuario), como la utilización de conexiones TDP/IP entre la aplicación externa y los shell-scrip o aplicaciones especiales que obtienen los puertos de conexión y ejecutan los gdbservers.

2) Escoger el mismo puerto de conexión libre (si es posible) en las máquinas donde se han de ejecutar los gdbservers. De esta forma esta información de conexión es directamente accesible por los dos componentes de la herramienta Gdb.

b) El usuario o la aplicación externa deben esperar a que estén en ejecución, en las máquinas seleccionadas, los componentes remotos de Gdb para poder ejecutar los componentes locales de esta herramienta y que estos se puedan conectar a estos componentes remotos. Este proceder obliga al acceso a las máquinas remotos donde se ejecutan los gdbservers y utilizar llamadas al sistema operativo, para comprobar cuando estos componentes están en ejecución.

En la figura 5.3 pueden observarse todos los puntos explicados anteriormente sobre la utilización del entorno de ejecución propio de la librería MPI, para ejecutar los componentes remotos de las herramientas de monitorización Gdb y Paradyn. En los puntos 1m al 6m de esta figura, pueden observarse los pasos seguidos por la librería MPICH1 para ejecutar los componentes remotos de Gdb o Paradyn y que estos ejecuten los procesos a monitorizar.
En los puntos 1 l/o al 3 l/o de la figura 5.3 pueden observarse la misma funcionalidad

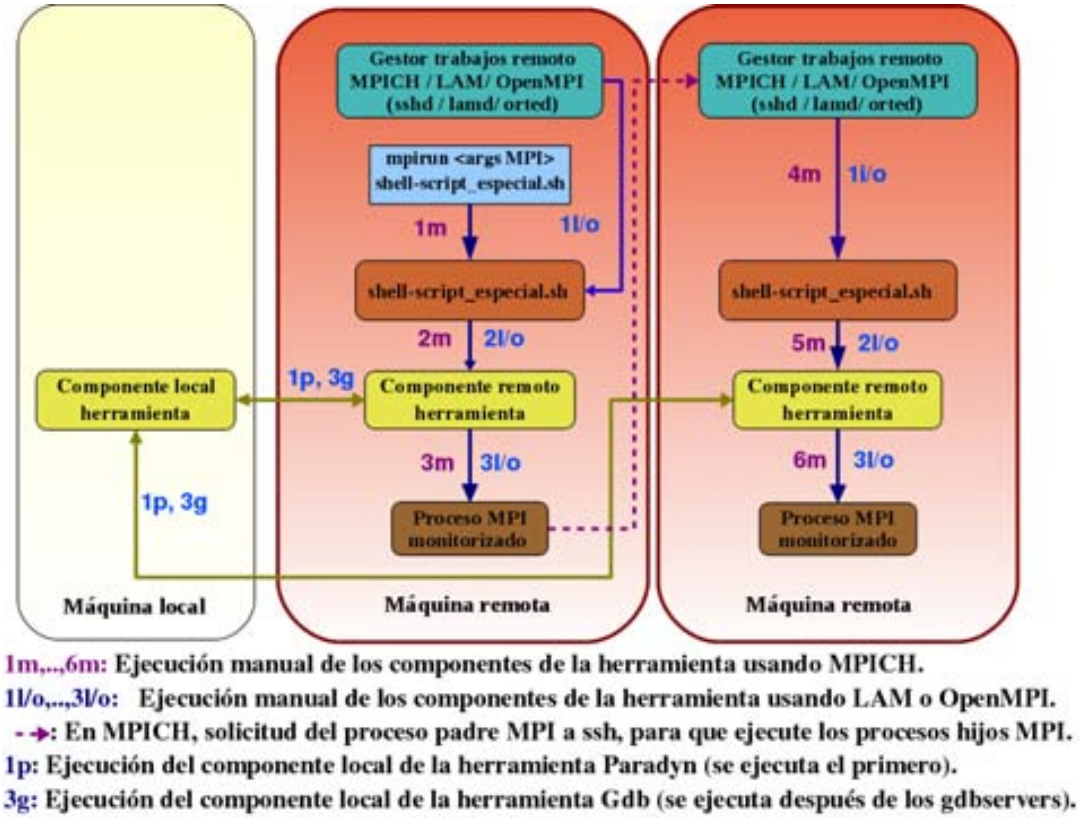


Figura 5.3: Ejecución de los componentes remotos de la herramienta usando la librería MPI

pero para las librerías LAM u OpenMPI. Para finalizar en los puntos 1p y 3g de esta misma figura pueden observarse los dos momentos temporales en los que se han de ejecutar cada componente local de cada herramienta. En el caso de Paradyn (punto 1p) su componente local se ha de ejecutar antes que sus componentes remotos (ya que les ha de proporcionar la información de conexión), sin embargo, en el caso de Gdb (punto 3g) es al contrario, se han de ejecutar primero sus componentes remotos y después su componente local (ya que estos componentes remotos suministran la información de conexión al componente local).

Como puede observarse, la mayoría de estos métodos manuales de ejecución de los componentes remotos de las herramientas, implican un esfuerzo de diseño de componentes intermedios (ya sean shell-scripts o aplicaciones especiales) por parte de los usuarios que los alejan de su objetivo de monitorizar sus aplicaciones en un cluster controlado por un gestor de colas.

5.2. TDP-Shell para herramientas que aprovechan el entorno de ejecución de MPI

Debido a la posibilidad de ejecución manual de sus componentes remotos, las herramientas de monitorización que aprovechan el entorno de ejecución de la librería MPI (vistas en apartado anterior), han sido el centro del estudio para solucionar el problema de la falta de interoperabilidad entre ellas y los gestores de colas en el caso de monitorización de aplicaciones MPI. Igual que para el caso de monitorización de aplicaciones serie, la explicación de como el entorno `TDP-Shell` soluciona el problema de esta falta de interoperabilidad se ha dividido en dos apartados, el primero con el procesamiento de los ficheros de descripción de trabajos y el segundo con la explicación de los pasos que realiza el entorno `TDP-Shell` (utilizando los ficheros `TDP-Shell` script), para sincronizar la ejecución los componentes locales y remotos de la herramienta de monitorización, en el caso de aplicaciones MPI

5.2.1. Procesamiento de los ficheros de descripción de trabajos

Los ficheros de descripción de trabajos de Condor y SGE tienen el mismo formato que el utilizado para las aplicaciones serie. Esto quiere decir que para el caso del gestor de colas SGE también se utiliza la división de estos ficheros en la sección pre-ejecutable, donde se sitúan las acciones a realizar antes de la ejecución del programa principal, la sección ejecutable, donde se declara el ejecutable del programa principal y la sección post-ejecutable, donde se sitúan las acciones que se realizan después de la ejecución del programa principal.

El punto diferencial respecto a los ficheros de descripción de trabajos para aplicaciones serie, es que el archivo de descripción del trabajo de la aplicación de usuario, define como ejecutar la aplicación MPI utilizando para ello las declaraciones especificas para cada gestor de colas. También hay que destacar que los archivos de descripción de trabajos del componente remoto de la herramienta y del componente `tdp_agent`, no son del tipo distribuido MPI (no se utiliza un programa tipo mpirun para ejecutarlos). Estos ficheros informan del entorno de ejecución (principalmente del ejecutable y sus argumentos) que necesitan estos componentes para poder realizar correctamente la monitorización de los

procesos de una aplicación MPI.

5.2.1.1. Condor

En el caso de Condor se utiliza la misma metodología que para el caso de aplicaciones serie, se procesa cada comando de los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell`, obteniendo su nueva versión que será situada en el fichero de descripción de trabajos global. La diferencia principal respecto al caso de aplicaciones serie, es la forma de procesar los comandos que definen el comportamiento de la aplicación distribuida MPI. Como se ha explicado en el punto 5.1.1 de este capitulo, el fichero de descripción de trabajos MPI declara, como ejecutable, un script especial, denominado mp1script para MPICH y lamscript para LAM, al cual se le pasa el nombre del ejecutable de la aplicación MPI como uno de sus argumentos (a través del comando *Arguments*). Por lo tanto, como Condor ha de ejecutar los componentes `tdp_agent`, o el shell-script que se encargue de hacerlo correctamente (al que denominaremos a partir de ahora *tdp_shell_MPI.sh*), el fichero de descripción de trabajos global (obtenido por `tdp_console`) ha de tener como ejecutable, a uno de los scripts especiales mp1script o lamscript y la misma lista de argumentos que la del fichero de descripción de trabajos MPI del usuario, sustituyendo el nombre del ejecutable de los procesos MPI por el ejecutable del `tdp_agent` o del *tdp_shell_MPI.sh*. A continuación se muestra un ejemplo de como quedarían los comandos *Execute* y *Arguments* en el archivo de descripción de trabajos global.

- Archivo de descripción de trabajos MPI del usuario:
  universe = parallel
  executable = [mp1script | lamscript]
  arguments = <ejecutable aplicación MPI usuario> [argumentos]*
  machine_count = n

- Archivo de descripción de trabajos global obtenido por `tdp_console` (suponiendo que el ejecutable del archivo de descripción de trabajos de `tdp_agent` es tdp_shell_MPICH.sh) :
  universe = parallel
  executable = [mp1script | lamscript]

arguments = tdp_shell_MPICH.sh [argumentos]*
machine_count = n

Normalmente, el nombre del ejecutable de la aplicación MPI, es la primera cadena de caracteres (separada entre espacios en blanco) del valor del comando Arguments. Si esto no fuera así, para que `tdp_console`, a través del comando tdp *tdp_launch_agent* (explicado en el punto 4.7.2.6), pueda encontrar esta cadena de caracteres dentro del comando Arguments, se aprovecha el argumento *-e* que se puede pasar a este comando tdp, el cual indica precisamente el nombre del ejecutable del trabajo. Con esta modificación la declaración del fichero de descripción de trabajos de una aplicación MPI para el comando tdp *tdp_launch_agent* en Condor quedaría: *"job user:" -e <ejecutable aplicación MPI usuario>*

5.2.1.2. SGE

Para el caso de SGE, con las secciones pre-ejecutables y post-ejecutables también se sigue la misma metodología que para caso serie. Se sitúan la concatenación de las cadenas de estas secciones de cada fichero de descripción de trabajos, en la sección pre-ejecutable y post-ejecutable del fichero de descripción de trabajos global. Como en el caso de Condor, la diferencia con el caso serie, viene dada por el código ejecutable que `tdp_console` sitúa en la sección ejecutable del archivo de descripción de trabajos global. El procedimiento seguido por `tdp_console` para informar, a SGE, que ejecute el componente `tdp_agent` o el *tdp_shell_MPICH.sh* en las máquinas seleccionada, consiste en:

- Obtener la cadena de caracteres del ejecutable del archivo (o shell-script) de descripción de trabajo MPI del usuario (situada a continuación de la directiva `TDP-Shell` #tdp_exec), cuyo formato estándar es:

  *mpirun -np <número de procesos MPI> <atributos para mpirun> <ejecutable aplicación MPI usuario> [argumentos]**

- Obtener la nueva cadena del ejecutable del archivo de descripción de trabajos global, sustituyendo el nombre del ejecutable de los procesos MPI <ejecutable aplicación MPI usuario> por el *tdp_shell_MPICH.sh* (suponiendo que es el ejecutable del archivo de descripción de trabajos de `tdp_agent`). El formato de

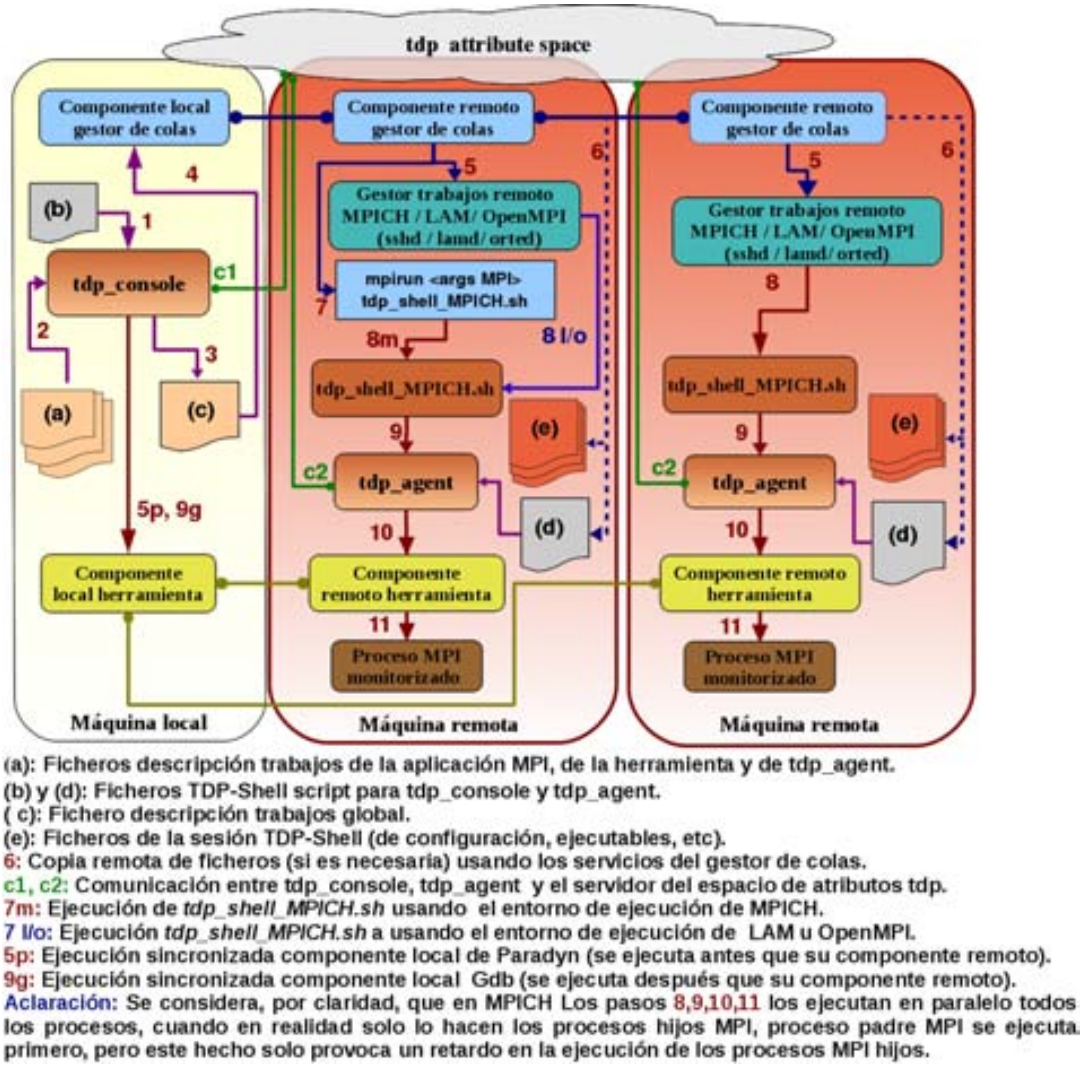
esta nueva cadena de caracteres es:
*mpirun -np <número de procesos MPI> <atributos para mpirun> tdp_shell_MPICH.sh [argumentos]**

5.2.2. Proceso de Sincronización de la ejecución de los componentes de la herramienta

Los pasos que siguen `tdp_console` y `tdp_agent`, indicados en sus respectivos archivos `TDP-Shell` script, para ejecutar sincronizadamente los componentes de la herramienta de monitorización son:

1. El componente `tdp_console` interpreta los comandos e instrucciones tdp su archivo `TDP-Shell` script (figura 5.4, punto 1). Uno de estos comandos es *tdp_launch_agent*, el cual es el encargado de procesar los archivos de descripción de trabajos de los componentes remotos de `TDP-Shell` y obtener el nuevo fichero de descripción de trabajos global. Este fichero informa al gestor de colas como ejecutar el componente `tdp_agent` en las máquinas del cluster, en este caso en un entorno de monitorización de aplicaciones MPI. En la versión actual de `TDP-Shell` para estos entornos, se utiliza, por defecto, el shell-script *tdp_shell_MPICH.sh*, el cual se encarga de la ejecución de los componentes `tdp_agent`, solucionando los posibles problemas de la implementación de la librería MPI utilizada (por ejemplo, el hecho que MPICH no permite pasar argumentos de usuario a sus procesos hijos). Esta característica puede ser modificada por los usuarios del entrono `TDP-Shell`.

2. El gestor de colas procesa el fichero de descripción de trabajos global y detecta que es del tipo MPI (figura 5.4, punto 4). Entonces elije las máquinas que disponen de los suficientes recursos para ejecutar la aplicación MPI y después, a través de sus componentes remotos (figura 5.4, punto 5), pone en ejecución en cada máquina seleccionada, los demonios de la herramienta de conexión remota o los demonios propios la implementación de MPI (ssh, si se trata de MPICH, lamd si es LAM u orted si es OpenMPI). Después de estas acciones y si se ha solicitado, realiza la copia remota de archivos (en SGE es una de las acciones situadas en las secciones pre-ejecutable), como el fichero de configuración tdp, el `TDP-Shell` script de los `tdp_agent` o el código ejecutable de algún componente (figura 5.4, punto 6). Si no



(a): Ficheros descripción trabajos de la aplicación MPI, de la herramienta y de tdp_agent.
(b) y (d): Ficheros TDP-Shell script para tdp_console y tdp_agent.
(c): Fichero descripción trabajos global.
(e): Ficheros de la sesión TDP-Shell (de configuración, ejecutables, etc).
6: Copia remota de ficheros (si es necesaria) usando los servicios del gestor de colas.
c1, c2: Comunicación entre tdp_console, tdp_agent y el servidor del espacio de atributos tdp.
7m: Ejecución de *tdp_shell_MPICH.sh* usando el entorno de ejecución de MPICH.
7 l/o: Ejecución *tdp_shell_MPICH.sh* a usando el entorno de ejecución de LAM u OpenMPI.
5p: Ejecución sincronizada componente local de Paradyn (se ejecuta antes que su componente remoto).
9g: Ejecución sincronizada componente local Gdb (se ejecuta después que su componente remoto).
Aclaración: Se considera, por claridad, que en MPICH Los pasos 8,9,10,11 los ejecutan en paralelo todos los procesos, cuando en realidad solo lo hacen los procesos hijos MPI, proceso padre MPI se ejecuta primero, pero este hecho solo provoca un retardo en la ejecución de los procesos MPI hijos.

Figura 5.4: Esquema entorno TDP-Shell para la monitorización de aplicaciones MPI

se han definido más acciones, ejecuta proceso principal del archivo de descripción de trabajos global, que en este caso y por tratarse de una aplicación MPI es el comando mpirun con el script especial *tdp_shell_MPI.sh* como su argumento que define el ejecutable de la aplicación MPI.

3. El comando mpirun utilizando ssh (el proceso padre si es MPICH1) o los demonios lamd u orted, pone en ejecución, en las máquinas escogidas, los scripts especiales *tdp_shell_MPI.sh* (figura 5.4, puntos 8, 8m y 8l/o) para que estos scripts ejecuten los procesos `tdp_agent` (figura 5.4, punto 9).

4. Una vez están en ejecución los componentes `tdp_agent`, estos interpretan los códigos tdp situados en sus archivos `TDP-Shell` scripts, los cuales les informan como sincronizar la ejecución de los componentes remotos de la herramienta de monitorización con el componente local de la misma, el cual es ejecutado por el componente `tdp_console` (al interpretar el código tdp de su archivo `TDP-Shell` script). Esta proceso de sincronización depende de la herramienta de monitorización que se esté utilizando.

   Si es Gdb, el fichero `TDP-Shell` script de cada `tdp_agent` le informa que primero obtenga un puerto libre en la máquina donde se ejecuta (usando la función especial de `TDP-Shell`: TDPGet_port()), que ejecute el componente gdbserver (con el comando *tdp_create_process*) y que situé en el espacio de atributos tdp (con el comando *tdp_put*) las tuplas con la información de la máquina y el puerto donde debe conectarse el componente local de la herramienta Gdb.

   Por su parte, el fichero `TDP-Shell` script que interpreta `tdp_console` le informa que espere bloqueado (usando un comando *tdp_get*) a que algún `tdp_agent` sitúe las tuplas con la información de conexión de algún gdbserver en el espacio de atributos. Una vez sucede este hecho, el componente `tdp_console` ejecuta el componente local de Gdb (con el comando `tdp_console`) con la información de conexión del gdbserver recién creado. Este proceso de creación de parejas componente local, remoto de Gdb se repite para cada proceso MPI a monitorizar. (figura 5.4, puntos 9g y 10).

   En el caso de la herramienta Paradyn, el proceso de sincronización es en sentido contrario al de Gdb. El fichero `TDP-Shell` script del componente `tdp_console`, le informa que ejecute el componente local de esta herramienta (usando el comando tdp_create_process) con el argumento *-x fichero_conexión_paradyn*, donde se sitúa la información de conexión para sus componentes remotos *paradynd*. Una vez creado este *fichero_conexión_paradyn*, `tdp_console` obtiene de el la información de conexión para los componentes paradynd (utilizando la función de `TDP-Shell` TDPRead_contents_from_file) y la sitúa (con una tupla) en el espacio de atributos tdp (con el comando *TDP_put*).

   Por su parte, el fichero `TDP-Shell` script que interpreta cada `tdp_agent` que se ejecuta en las máquinas cluster, le informa que espere suspendido (con un comando *tdp_get*) a que se sitúe, en el espacio de atributos tdp, la tupla con la información de conexión con el componente local de Paradyn. Una vez sucede este hecho, cada

`tdp_agent` ejecuta (con el comando *tdp_create_process*) los componentes remotos de Paradyn para que se conecten con su componente local y monitoricen la aplicación MPI.

Como puede apreciarse, el diseño del entorno `TDP-Shell` para solucionar el problema de interoperabilidad entre gestores de colas y herramientas de monitorización que aprovechan el entorno de ejecución de la distribución de MPI, sigue las mismas pautas que para las aplicaciones serie: Procesamiento de los ficheros de descripción de trabajos y utilización de ficheros `TDP-Shell` script, en este caso adaptados a las diferentes implementaciones de la librería MPI (tanto si utilizan herramientas de conexión remotas, tipo ssh, como demonios propios, lamd u orted). Este planteamiento permite que aprender el funcionamiento de esta nueva versión del entorno `TDP-Shell` sea fácil y rápido para los usuarios de este entorno.

5.3. TDP-Shell para herramientas integradas el entorno de ejecución de MPI

Las herramientas integradas el entorno de ejecución de MPI, como Totalview, no facilitan la ejecución manual de sus componentes remotos. Es el programa mpirun quien crea el entorno de ejecución para que ejecute los componentes de la herramienta y los procesos a monitorizar (modificando el código de ambos). Este hecho dificulta (o incluso impide) que se pueda utilizar el gestor de colas (sin modificar el código de los componentes de Totalview), para que ejecute los procesos `tdp_agent` y estos, los componentes remotos de la herramienta (como ocurre en el caso de Paradyn) Que no se pueda adaptar esta solución, no implica que no exista otra solución para conseguir que el Entrono `TDP-Shell` se pueda aplicar a las herramientas integradas en el entorno de ejecución de MPI. Para la herramienta Totalview, representante de este tipo de herramientas, se puede implementar la siguiente solución:

a) Ejecutar todo el entorno de monitorización de la herramienta Totalview en las máquinas remotas controladas por el gestor de colas.

b) Mostrar la/s ventana/s del componente local de la herramienta, que se ejecuta en una de las máquinas del cluster, en la máquina local del usuario. Esto en entornos de

ventanas X11 (Unix/Linux) se realiza exportando la salida de las aplicaciones gráficas ejecutadas sobre un servidor de ventanas, en este caso en la máquina del cluster, a otro servidor situado en una máquina diferente, en este caso el de la máquina local del usuario.

5.4. Nuevos comandos tdp para entornos distribuidos MPI

La ejecución de diversos componentes `tdp_agent` que conlleva la monitorización de aplicaciones MPI, implica nuevas funcionalidades relacionadas con las operaciones que se realizan con el espacio de atributos tdp, las cuales son:

1. *Control de acceso a tuplas compartidas por varios* ***`tdp_agent`****:* Durante la creación de los componentes de la herramienta, puede ser necesario que diversos `tdp_agent` o el componente `tdp_console`, accedan a una tupa compartida situada en el espacio de atributos tdp, interpreten ciertos comandos tdp que modifiquen su valor y que finalmente vuelvan a situarla en el espacio de atributos tdp (por ejemplo incrementen el valor de una tupla compartida). Esto puede provocar que sea necesario un acceso secuencial a esta tupla compartida, esto es, mientras un componente `tdp_agent` o `tdp_console` la está modificando ningún otro componente del entorno `TDP-Shell` puede acceder a ella, ya que podían obtener datos inconsistentes. En el entorno de sistemas operativos, este tema es conocido como el acceso a zonas críticas o de exclusión mutua (donde se sitúan los accesos a recursos compartidos) y normalmente dicho acceso está controlado por semáforos. Estos semáforos garantizan (dependiendo de su estado) que solo un o un número de procesos accederá a la zona crítica que comparten.
   Aprovechando el comportamiento de los semáforos que ofrecen los sistemas operativos, el entorno `TDP-Shell` ha desarrollado dos comandos tdp especiales que lo implementan, los cuales son:

   - **tdp_lock identificador_semaforo**: Este comando tdp se sitúa al comienzo de una zona critica (código tdp que modificará una o varias tuplas compartidas), garantizando el acceso secuencial a dicha zona critica. Este tipo de acceso

implica que mientras un componente `tdp_console` o `tdp_agent` interpreta el código tdp de su zona critica (esta dentro de esta), el resto de componentes se espera (suspendido) a que este componente finalice la ejecución de este código tdp (salga de la zona critica).
Para implementar esta funcionalidad, el comando *tdp_lock* envía una petición (por el socket síncrono del componente `tdp_console` o `tdp_agent` que lo interpreta) al servidor de atributos tdp. Para ello utiliza el mensaje especial *[TDP_LOCK, identificador_semaforo]*, donde *identificador_semaforo* es la cadena de carácteres que identifica al semáforo. Una vez realizado este envío, se queda suspendido esperando su repuesta.
Cuando el servidor del espacio de atributos recibe este mensaje TDP_LOCK, comprueba si en la lista de semáforos *lista_semaforos_activos*, existe un elemento (identificador_semaforo, socket_sincrono). En caso negativo, esto es, ningún componente del entorno `TDP-Shell` esta en la zona crítica controlada por el semáforo *identificador_semaforo*, envía la respuesta *[ANSWER, OK]* por el socket, *socked_sincrono_recibido*, donde ha recibido el mensaje TDP_LOCK. De esta forma el componente `TDP-Shell` que ha enviado este mensaje puede entrar en su zona critica. Una vez realizadas todas estas acciones, sitúa el elemento (*identificador_semaforo, socked_sincrono_recibido)* en la textitlista_semaforos_activos. En el caso de que exista el elemento en la *lista_semaforos_activos*, esto significa que algún componente de `TDP-Shell` está en la zona crítica controlada por el semáforo *identificador_semaforo*, no envía ninguna respuesta al `tdp_console` o `tdp_agent` que le ha enviado el mensaje TDP_LOCK por el socket, *socked_sincrono_recibido* y sitúa el elemento *(identificador_semaforo, socked_sincrono_recibido)* en la lista lista_semaforos_pendientes, la cual contiene las peticiones de acceso a las zonas criticas pendientes. Con este proceder se impide que el componente `tdp_console` o `tdp_agent` que ha enviado la petición, entre en la zona critica.

- **tdp_unlock identificador_semaforo**: Este comando tdp se sitúa a la salida de la zona critica e informa de este hecho al resto de componentes del entorno `TDP-Shell` que puedan estar esperando entrar a esta zona critica. Para implementar esta funcionalidad, el componente `tdp_console` o `tdp_agent` que interpreta este comando, envía el mensaje especial *[TDP_UNLOCK, identifi-*

cador_semaforo] (por su socket síncrono) al servidor del espacio de atributos tdp.

Cuando el servidor del espacio de atributos tdp recibe este mensaje TDP_UNLOCK, comprueba si en la lista *lista_semaforos_pendientes* existe el elemento *(identificador_semaforo, socket_sincrono_pendiente)*. En caso afirmativo, esto es, algún componente del entorno `TDP-Shell` esta esperando entrar a la zona crítica controlada por el semáforo *identificador_semaforo*, envía el mensaje *[ANSWER, OK]* por el socket, *socket_sincrono_pendiente*, para que el `tdp_console` o el `tdp_agent` que realizó esta petición pueda entrar en la zona critica. Posteriormente borra este elemento de la *lista_semaforos_pendientes* y lo añade a la *lista_semaforos_activos* para indicar que algún componente de `TDP-Shell` esta en la zona critica controlada por el semáforo *identificador_semaforo*. Si el elemento no esta en *lista_semaforos_pendientes*, el gestor del espacio de atributos no realiza ninguna acción porque no hay ningún componente de `TDP-Shell` esperando entrar a la zona critica controlada por el semáforo *identificador_semaforo*.

2. **Almacenamiento de diversos valores en una sola tupla**: Igual que sucede con el control de acceso a zonas criticas, cuando se utiliza el entorno `TDP-Shell` en la monitorización de aplicaciones MPI, diversos `tdp_agents` o el `tdp_console` pueden necesitar intercambiarse información situada en una salo tupla, implicando que esta pueda tener diversos valores. Por ejemplo en el caso de la herramienta Gdb, diversos `tdp_agent` pueden situar, en el espacio de atributos tdp, la información de los puertos de conexión de sus gdbservers a través de una tupla. Posteriormente el proceso `tdp_console` puede acceder a esta tupla situada en el espacio de atributos tdp para adquirir la información de conexión. Para poder realizar esta situación de diversos datos en una tupla, se ha desarrollado un nuevo tipo de estas denominado *tupla con atributo array*, cuyo formato es:
(identificador_tupla[índice_tupla],valor_indice).
Donde el valor,*valor_indice* de la tupla identificada por *identificador_tupla* esta situado en la posición identificada por *índice_tupla* de esta tupla. Este ultimo campo, también se conoce como el índice y en el entorno `TDP-Shell` es una cadena de caracteres. Como puede observarse, este formato es parecido a la de-

claración del tipo *array* que realizan los lenguajes de alto nivel como C o C++ (exceptuando que su campo índice es un numero entero) y su funcionamiento es parecido al que realizan estos lenguajes. Por ejemplo, los diferentes ficheros `TDP-Shell` script que interpretan los componentes `tdp_agent`, podrían utilizar la tupla *PORTS_GBBERVER[$ID_GDBSERVER]=<valor puerto comunicaciones>* para situar el valor del puerto de comunicaciones de los gdbservers en el espacio de atributos. Es importante destacar que la variable $ID_GDBSERVER tiene un valor diferente en cada fichero `TDP-Shell` script que interpretan los diferentes `tdp_agents`. Si archivo `TDP-Shell` script que interpreta el componente de `TDP-Shell`, ha de obtener los valores de una *tupla con atributo array* del espacio de atributos tdp, puede encontrarse con el problema que desconozca orden en que estos valores se situaran en la tupla (ya que normalmente, se desconoce el orden de ejecución de los diferentes `tdp_agents` que sitúan valores en la tupla). Para solucionar este problema, se puede utilizar la cadena de carácteres especial "ANY" en el índice (*índice_tupla*) del atributo de la *tupla con atributo array*. Con este índice especial, se asegura que para cada petición a este tipo de tuplas, realizada por un `tdp_console` o `tdp_agent` al espacio de atributos tdp, el servidor que lo controla devuelva un nuevo valor situado en esta tupla, diferente a los que ya se habían enviado anteriormente al `tdp_console` o `tdp_agent` que realiza la petición. Siguiendo el ejemplo de la herramienta Gdb, si en es espacio de atributos tdp, los `tdp_agents` han situado las siguientes tres tupas:
PORTS_GBBERVER["GDBSERVER_1"]="2000".
PORTS_GBBERVER["GDBSERVER_2"]="2020".
PORTS_GBBERVER["GDBSERVER_3"]="3000".
Para leer estos 3 valores, en el `TDP-Shell` script que interpreta `tdp_console` se podría utilizar, en un bucle while, el siguiente comando tdp: *tdp_get PORTS_GBBERVER["ANY"]*. Con este proceder, `tdp_console` conseguiría que a cada llamada sucesiva del comando *tdp_get*, se le devolviera primero "2000", después "2020" y por último "3000". Puede observarse que a la segunda llamada al comando *tdp_get PORTS_GBBERVER["ANY"]* no se devuelve "2000" porque el servidor del espacio de atributos tdp ya lo ha enviado como repuesta a la petición del comando *tdp_get* anterior.

Como puede observarse, estas nuevas funcionalidades de semáforos y de estructura de tuplas con atributo array, se han basado en las que ofrecen tanto los sistemas operativos como los lenguajes de alto nivel, para que resulten familiares a los usuarios del entrono `TDP-Shell` permitiendo su rápido aprendizaje.

5.5. Conclusiones

Los gestores de colas y las herramientas de monitorización estudiadas en este trabajo de tesis se han adaptado para dar soporte a las aplicaciones distribuidas basadas en la librería MPI. Para realizar este soporte, estos gestores de colas y herramientas de monitorización aprovechan el entorno de ejecución propio de cada implementación de la librería MPI, de las cuales las más utilizadas son *MPICH* , *LAM* y *OpenMPI*. Este entorno de ejecución se basa en la utilización de un programa, normalmente denominado mpirun (o mpiexec), que se encarga de lanzar la ejecución los diferentes procesos de la aplicación MPI en un conjunto de máquinas seleccionadas. Dependiendo de la implementación de MPI, para realizar este lanzamiento, este programa se apoya en las herramientas de conexión remota como ssh (caso de MPICH) o en los demonios propios como lamd (en el caso de LAM) u orted (caso de OpenMPI).
Hay que destacar que ciertas herramientas de monitorización, como Paradyn y Gdb, permiten que los usuarios puedan ejecutar sus componentes remotos manualmente, permitiendo aprovechar el entorno de ejecución de la implementación de MPI para que el programa mpirun ponga en ejecución estos componentes remotos de la herramienta y estos a su vez ,los procesos de la aplicación a monitorizar. Sin embargo, otras herramientas no permiten (o facilitan) esta ejecución manual de sus componentes remotos, debido a que se integran dentro del entorno de ejecución de la implementación de MPI (modificando su código), como es el caso de Totalview. La solución del entorno `TDP-Shell` para el problema de interoperabilidad entre gestores de colas y herramientas de monitorización para aplicaciones MPI, se ha centrado en el primer tipo de herramientas de monitorización (las que permiten la ejecución manual de sus componentes remotos). Para ello, se ha aprovechado que los gestores de colas utilizan (y gestionan) el entorno de ejecución de la implementación de MPI, para que este entorno ponga en ejecución (usando mpirun), en lugar de los componentes remotos de la herramienta, los componentes `tdp_agent` para que junto con el componente `tdp_console`,

ejecuten sincronizadamente (utilizando sus respectivos ficheros `TDP-Shell` script) los componentes de la herramienta de monitorización. Este nuevo entorno de monitorización para aplicaciones MPI ha implicado añadir nuevas funcionalidades y estructuras de datos al entorno `TDP-Shell` base, estas son:

- Control de acceso a las zonas criticas o de exclusión mutua, cuando varios componentes `tdp_agent` o `tdp_console` intentan modificar una variable compartida. Para ello se han implementado dos comandos tdp nuevos que actúan como semáforos, controlando la entrada a la zona critica, comando *tdp_lock* y su salida, comando *tdp_unlock*.

- Almacenamiento de diversos valores en una sola tupla utilizando un nuevo tipo de tuplas denominadas *tupla con atributo array*, cuyo formato: *(identificador_tupla[índice_tupla],valor_indice)* permite que cada *[índice_tupla]* de la tupla *identificador_tupla* contenga un valor diferente

Todas estas modificaciones realizadas en el entorno `TDP-Shell` base se han realizando pensando que sean fácilmente asumibles por los usuarios y no deban dedicar una considerable cantidad de tiempo en su aprendizaje.

6

TDP-Shell y el caso especial de monitorización retardada

Hasta este capitulo se han explicado los diseños del entorno TDP-Shell para solucionar el problema de interoperabilidad entre los gestores de colas y las herramientas de monitorización tanto para aplicaciones serie (formadas por un solo proceso) como aplicaciones distribuidas basadas en la librería MPI. El funcionamiento de estos diseños se basa en que sus dos componentes principales, `tdp_console` y `tdp_agent`, ejecuten de una forma predeterminada (descrita por los archivos `TDP-Shell` script de ambos componentes), los componentes de la herramienta de monitorización y los procesos de la aplicación de usuario. Esto significa que una vez comienza una sesión de monitorización de `TDP-Shell`, el usuario simplemente espera a que se le confirme cuando puede empezar a monitorizar su aplicación, evento que sucede cuando todos los componentes de la herramienta de monitorización han sido creados por `tdp_console` y `tdp_agent`. La situación de monitorización descrita en el párrafo anterior no tiene porque ser única, pueden existir otras formas de monitorización, donde su funcionamiento no sea tan predefinido y el usuario tenga un papel más activo en ellos. Uno de estos entornos, denominado *ejecución retardada de la herramienta de monitorización o monitorización retardada*, consta de los siguientes pasos importantes:

1. Se ejecutan los procesos de la aplicación a monitorizar (puede que en estado pausado) en las máquinas del cluster.

2. Los componentes remotos de la herramienta no se ejecutan inmediatamente después

de estos procesos para que se les adjunten. Se espera a que lo indique el usuario.

3. Cuando el usuario lo considera oportuno, informa a la herramienta de monitorización que cree sus componentes remotos (o lo hace el propio usuario), para que se adjunten a los procesos de la aplicación que han sido puestos en ejecución con anterioridad.

En este capítulo se explicarán las modificaciones realizadas en el diseño del entorno `TDP-Shell`, para ofrecer una solución adecuada para este nuevo tipo de monitorización.

6.1. Diseño del Entorno TDP-Shell para la monitorización retardada

Para poder realizar los pasos que requiere la monitorización retardada, explicados en la introducción de este capítulo, el diseño del entorno `TDP-Shell` visto hasta ahora debe incorporar principalmente dos nuevas funciones:

1. Permitir que el usuario pueda informar, desde el proceso `tdp_console`, al proceso `tdp_agent` cuando quiere que se ejecute el componente remoto de la herramienta y en consecuencia empezar el proceso de monitorización. Por su parte, el proceso `tdp_agent` tiene que estar pendiente de la llegada de esta petición.

2. Una vez el usuario informe que quiere empezar a monitorizar los procesos de su aplicación, cada componente `tdp_agent` ha de ejecutar el componente remoto de la herramienta, para que se adjunte correctamente al proceso de aplicación. Además y como se ha explicado en los capítulos anteriores tanto `tdp_console` como `tdp_agent` han de sincronizar la ejecución de los componentes de la herramienta.

Como se puede observar en estas dos nuevas funciones hay un importante componente asíncrono : Informar del instante, a priori desconocido, en que el usuario quiere empezar la monitorización de los procesos de su aplicación. Para implementar este componente asíncrono se pueden aprovechar los comandos tdp asíncronos y el sistema de eventos que ofrece el entorno `TDP-Shell`. Para encapsular y simplificar la utilización de estos comandos asíncronos, se han definido los siguientes nuevos comandos tdp:

- Para el archivo `TDP-Shell` script que interpreta el componente `tdp_console`:

  - **tdp_remote_create_process [id_del_proceso_remoto]?:** Informa a los componentes `tdp_agent` que están esperando una monitorización retardada de algún proceso, que el usuario quiere comenzar dicha monitorización. El argumento de este comando tdp, que es optativo, informa del identificador del proceso situado en la maquina remota, *id_del_proceso_remoto*, que se quiere monitorizar (a través del `tdp_agent` situado en dicha máquina). Si solo hay un proceso remoto a monitorizar (es así en la mayoría de los casos), entonces no es necesario identificarlo y este argumento no es necesario declararlo. Para realizar este aviso, comando *tdp_remote_create_process*, sitúa en el espacio de atributos (a través de un comando *tdp_put*) la tupla especial del entorno `TDP-Shell`:
    TDP_START_REMOTE_PROCESS= [id_del_proceso_remoto | "NO"]
    La cadena de caracteres "NO" se utiliza en el caso de que no se pase ningún identificador de proceso remoto.

  - **tdp_wait_local_invocation función_componente_local:** Informa que se llame a la función, *función_componente_local*, al recibirse la respuesta desde el servidor del espacio de atributos debido a la inserción de la tupla especial *TDP_START_REMOTE_PROCESS* (Lo cual indicará que el usuario quiere comenzar el proceso de monitorización retardada).
    Para realizar esta acción utiliza el comando tdp asíncrono:
    tdp_asyncget TDP_START_REMOTE_PROCESS función_componente_local
    El código tdp de la función, *función_componente_local*, es el encargado de sincronizar la ejecución del componente local de la herramienta con los componentes remotos de la misma (estos últimos ejecutado por los `tdp_agents`)

- Para el archivo `TDP-Shell` script que interpreta el componente `tdp_agent`:

  - **tdp_wait_remote_invocation función_componente_remoto:** Informa que se llame a la función, *función_componente_remoto*, al recibirse la respuesta desde el servidor del espacio de atributos debido a la inserción de la tupla especial cuyo atributo es textitTDP_START_REMOTE_PROCESS (indicando

que el usuario quiere comenzar el proceso de monitorización retardada).
Igual que para el caso del comando *tdp_wait_local_invocation*, para realizar esta acción utiliza el comando tdp asíncrono:
tdp_asyncget TDP_START_REMOTE_PROCESS función_componente_remoto
El código tdp de la función *función_componente_remoto* es el encargado de realizar dos acciones importantes:

- Sincronizar la ejecución del componente remoto de la herramienta con el local de la misma (este ultimo ejecutado por `tdp_console`).
- Informar al componente remoto de la herramienta que se adjunte al proceso que ha creado `tdp_agent` y que se desea monitorizar a partir de aquel instante. Normalmente, para que un componente remoto de una herramienta se adjunte a un proceso en ejecución, hay que suministrarle (habitualmente como argumento) el pid de dicho proceso. Por lo tanto cuando esta función, *función_componente_remoto*, ejecute el componente remoto de la herramienta (con un comando *tdp_create_process*), necesitará conocer el pid del proceso que se le va ha adjuntar. Para ello puede utilizar la función especial del entorno `TDP-Shell`: *TDPProcess_local_SO_id ($varible_pid_tdp_proceso)*, la cual devuelve el pid del proceso identificado por la variable *$varible_pid_tdp_proceso*, que en este caso es la del proceso al que se quiere adjuntar el componente remoto de la herramienta para monitorizarlo (y que contiene su identificador tdp de proceso).
  Si se ha pasado el identificador de proceso *id_del_proceso_remoto* como valor de la tupla TDP_START_REMOTE_PROCESS, esta función puede acceder a el mediante la variable especial *$tdp_value.*

Por lo tanto, el código tdp del archivo `TDP-Shell` script que interpreta el componente `tdp_console` para la monitorización retardada tiene el siguiente formato básico:

tdp_fun *función_componente_local ()* **{**
/ código creación componente local herramienta */*
}

tdp_launch_agent <*descripción trabajos componentes remotos* `TDP-Shell`>
tdp_wait_remote_invocation *función_componente_local*

tdp_interactive
tdp_remote_create_process [id_del_proceso_remoto]?

El comando *tdp_interactive* permite al usuario que mientras se esta interpretando un archivo `TDP-Shell`, el pueda introducir (interactivamente) nuevos comandos tdp para que sean interpretados. Para salir de este modo interactivo se utiliza el comando tdp *tdp_exit_interactive*, el cual devuelve el control de interpretación de comandos tdp al fichero `TDP-Shell` script. Estos comandos tdp son interesantes porque permiten al usuario introducir comandos tdp mientras se decide a empezar la monitorización retardada.

Una vez visto el código tdp del componente `tdp_console`, se puede mostrar el formato del código tdp del archivo `TDP-Shell` script que interpreta el componente `tdp_agent` para la monitorización retardada:

tdp_fun *función_componente_remoto ()* **{**
/ Código creación sincronizada del componente remoto herramienta*
más adjuntarse al proceso identificado por $id_proceso/*
}

$id_proceso=[**tdp_create_process** | **tdp_create_paused_process**] *descripción_proceso*
tdp_wait_remote_invocation *función_componente_remoto*

Como puede observarse, el código tdp de los archivos `TDP-Shell` que interpretan `tdp_conosole` y `tdp_agent` para la monitorización retardada, es sencillo (pocos comandos tdp nuevos), aprovechando las funciones de gestión de eventos del entorno `TDP-Shell`, para encapsular el comportamiento asíncrono propio de esta monitorización retardada.

6.2. Conclusiones

El entorno `TDP-Shell`, a parte de ayudar a realizar la monitorización estándar de aplicaciones serie y distribuidas en clusters controlados por un gestor de colas,

también da soporte a un tipo nuevo de motorización, denominado *ejecución retardada de la herramienta de monitorización o monitorización retardada*. Este nuevo tipo de monitorización se basa en crear los procesos en las máquinas del cluster y cuando el usuario la decida, comenzar la monitorización de estos.
Para soportar este nuevo tipo de monitorización, con un fuerte componente asíncrono (depende de la decisión temporal del usuario), el diseño del entorno `TDP-Shell` debe incorporar dos nuevas funciones:

1. Permitir que el usuario pueda informar al componente `tdp_agent` cuando quiere empezar el proceso de monitorización y en consecuencia que se ejecute el componente remoto de la herramienta.

2. Una vez el usuario informe de este hecho, cada componente `tdp_agent` ha de ejecutar el componente remoto de la herramienta, para que se adjunte al proceso a monitorizar. Además tanto `tdp_console` como `tdp_agent` han de sincronizar la ejecución de los componentes de la herramienta.

Tres nuevos comandos tdp del entorno `TDP-Shell` encapsulan estas funcionalidades, aprovechando el sistema de gestión de eventos asíncronos que ofrece el propio entorno. Estos comandos son:

- Para el archivo `TDP-Shell` script que interpreta el componente `tdp_console`:

  - **tdp_remote_create_process:** Informa a los componentes `tdp_agent` que están esperando una monitorización retardada , que el usuario quiere comenzar dicha monitorización.

  - **tdp_wait_local_invocation** *función_componente_local*: Informa que se llame a la función, *función_componente_local*, cuando el usuario decida comenzar la monitorización retardada. El código tdp de la función, *función_componente_local*, es el encargado de sincronizar la ejecución del componente local de la herramienta con los componentes remotos de la misma.

- Para el archivo `TDP-Shell` script que interpreta el componente `tdp_agent`:

  - **tdp_wait_remote_invocation** *función_componente_remoto*: Informa que se llame a la función, *función_componente_remoto*, cuando el usuario decida

comenzar la monitorización retardada. El código tdp de la función *función_componente_remoto* es el encargado de sincronizar la ejecución del componente remoto de la herramienta con el local de la misma e informar, al componente remoto de la herramienta, que se adjunte al proceso que ha creado `tdp_agent` y que se desea monitorizar a partir de aquel instante.

Este código tdp de los archivos `TDP-Shell` que interpretan `tdp_conosole` y `tdp_agent` para la monitorización retardada, se ha desarrollado para que sea sencillo (pocos comandos tdp nuevos) y que aproveche las funciones de gestión de eventos del entorno `TDP-Shell`, para encapsular el comportamiento asíncrono propio de esta monitorización retardada.

7

Casos prácticos del entorno de trabajo TDP-Shell

En este capítulo se mostraran unos casos prácticos de utilización del entorno `TDP-Shell`, que completarán el ejemplo mostrado en el capitulo 4. Para ello se tratará el caso de monitorización de una aplicación MPI y de una monitorización retardada.

7.1. Monitorización de aplicaciones MPI

En este apartado se mostrará un caso práctico de la utilización del entorno `TDP-Shell` para poder monitorizar aplicaciones MPI en un cluster controlado por un gestor de colas. Para este caso práctico se ha escogido el gestor de colas Condor con la herramienta de monitorización Gdb. En los siguientes puntos se mostraran los archivos de descripción de trabajos de los componentes remotos de `TDP-Shell`, el archivo de descripción de trabajo global obtenido por `tdp_console` al procesarlos y finalmente los archivos `TDP-Shell` script que controlan la ejecución sincronizada de los componentes de las herramienta Gdb.

7.1.1. Archivos de descripción de trabajos de Condor

En el fichero de descripción de trabajos de la aplicación MPI del usuario, FDT 7.1, se puede observar la utilización del script especial de Condor *mp1script* (líneas 2) que se encarga de la gestión del entorno de ejecución de la librería MPICH1 (para LAM seria

lamscript). Unos de los puntos destacables del archivo de descripción de trabajos del `tdp_agent`, FDT 7.1, es que informa al gestor de colas Condor que copie (líneas 9) , en la máquinas del cluster donde se ejecutarán los `tdp_agent`, todos los archivos que estos necesitan (de configuración, el `TDP-Shell` script, la librería tdp y el plug-in de Condor). Esto es hecho de esta forma porque el entorno `TDP-Shell` no está instalado en el cluster que controla Condor.

El archivo de descripción de trabajos del componente remoto de la herramienta Gdb, FDT 7.3, informa al gestor de colas Condor del nombre del ejecutable de este componente (líneas 2).

FDT 7.1 fichero de descripción de trabajos de Condor para la aplicación de usuario

```
1: Universe = parallel
2: Executable = /opt/condor/bin/mp1script
3: Arguments = user_exec_MPI
4: Output = user_MPI.out
5: Error = user_MPI.error
6: Log = user_MPI.log
7: machine_count = 4
8: should_transfer_files = YES
9: when_to_transfer_output = ON_EXIT
10: transfer_input_files = /home/user/user_exec_MPI
11: Requirements = Memory ≥ 64 && OpSys == "LINUX"&&Arch =="x86_64"
12: Queue
```

FDT 7.2 fichero de descripción de trabajos de Condor para `tdp_agent`

```
1: Universe = vanilla
2: Executable = /TDP_Shell/bin/tdp_shell_MPI.sh
3: Arguments = -tf:tdp_agent_MPI_gdb.tdp
4: Output = tdp_agent_MPI.out
5: Error = tdp_agent_MPI.error
6: Log = tdp_agent_MPI.log
7: should_transfer_files = YES
8: when_to_transfer_output = ON_EXIT
9: transfer_input_files = /TDP_Shell/bin/tdp_agent, /TDP_Shell/cfg/tdp_shell_config.tcf,
   /TDP_Shell/tdp/tdp_agent_MPI.tdp, /TDP_Shell/lib/lib/libTDP.so,
   /TDP_Shell/plugins/tdp_lib_job_condor_plugin.so
10: Queue
```

FDT 7.3 fichero de descripción de trabajos de Condor para gdbserver

```
1: Universe = vanilla
2: Executable = gdbserver
3: transfer_executable = False
4: Output = gdbserver.out
5: Error = gdbserver.error
6: Log = gdbserver.log
7: Queue
```

FDT 7.4 fichero de descripción de trabajos global obtenido por `tdp_console` para Condor

```
1: Universe = parallel
2: Executable = /opt/condor/bin/mp1script
3: Arguments = /TDP_Shell/bin/tdp_shell_MPI.sh
4: Log = TDP_Shell_error
5: Output = TDP_Shell_out
6: Error = TDP_Shell_error
7: Should_transfer_files = YES
8: When_to_transfer_output = ON_EXIT
9: Transfer_input_files = /home/user/user_exec_MPI, /TDP_Shell/bin/tdp_agent,
   /TDP_Shell/cfg/tdp_shell_config.tcf, /TDP_Shell/tdp/tdp_agent_MPI.tdp,
   /TDP_Shell/lib/lib/libTDP.so, /TDP_Shell/plugins/tdp_lib_job_condor_plugin.so
10: Machine_count = 4
11: Requirements = Memory ≥ 64 && OpSys == "LINUX"&&Arch =="x86_64"
12: Queue
```

El fichero de descripción de trabajos global, FDT 7.4, es obtenido por `tdp_console` después de procesar los ficheros FDT 7.1 , FDT 7.2 y FDT 7.3. En este fichero se puede observar como se aprovecha el script especial de Condor para MPICH1 (comando Execute, líneas 7.2) para que Condor ejecute, en las máquinas remotas de su cluster, el script especial de `TDP-Shell` `tdp_shell_MPI.sh` (pasado como argumento del ejecutable, comando Arguments, líneas 3), el cual se encargará a su vez, de ejecutar el componente `tdp_agent`. En el archivo de descripción de trabajos de `tdp_agent`, al no utilizar el comando *transfer_executable = False*, se informa a Condor que realice la copia del ejecutable de este componente de `TDP-Shell`, de la máquina local del usuario a las máquinas del cluster. Por lo tanto, en el archivo de descripción de trabajos global, obtenido por `tdp_console`, se tiene que informar de que se realice esta copia remota.

Para ello se sitúa la dirección y el nombre, en la máquina local del usuario, del ejecutable del componente `tdp_agent` , en la lista contenida en el comando de Condor *Transfer_input_files* (líneas 9) con la lista de ficheros a transmitir desde esta máquina a las del cluster (que ya contiene los que necesita el componente `tdp_agent`).
En el archivo de descripción de trabajos global se pueden observar los nombres estándar que da el entorno `TDP-Shell` a los archivos con la salida estándar (líneas 5), del error (líneas 6) y de log (líneas 4). Cambien se observa como, en este archivo global, se conservan los requerimientos (comando Requeriments, líneas 11) del archivo de descripción de trabajos de la aplicación MPI de usuario. Este hecho implica, que el usuario ya tiene en cuenta que tanto el componente remoto de la herramienta como el `tdp_agent` cumplirán con estos requerimientos. Del procesamiento de los ficheros de descripción de trabajos de los componentes remotos de `TDP-Shell`, `tdp_console` obtiene las tuplas especiales con la información para los `tdp_agents`. Para este caso los valores de estas tuplas son:

- **TDP_USER_EXEC**: *user_exec_MPI*. El nombre del ejecutable de la aplicación MPI de usuario. En el caso de trabajos MPI, este ejecutable es la primera cadena de caracteres del comando Arguments del archivo de descripción de trabajos de la aplicación del usuario.

- **TDP_USER_ARGS**: *NOTHING*. Posibles argumentos del ejecutable de la aplicación de usuario

- **TDP_TOOL_EXEC**: *gdbserver*. Nombre del ejecutable del componente remoto de la herramienta.

- **TDP_TOOL_ARGS**: *NOTHING*. Posibles argumentos del ejecutable del componente remoto de la herramienta.

- **TDP_SHELL_SCRIPT_AGENT**: *-tf:tdp_agent_MPI_gdb.tdp*. Archivo `TDP-Shell` script para el componente `tdp_agent`.

- **TDP_NUM_PROGS**:*4*. Número de procesos que generará la aplicación MPI del usuario.

7.1.2. Archivos TDP-Shell script para tdp_agent y tdp_console

En los dos archivos `TDP-Shell` scrip para `tdp_console`,`TDP-Shell` script 7.5 y `tdp_agent`, `TDP-Shell` script 7.6 se pueden observar: El envío de la petición al gestor de colas para que cree el entorno de ejecución de los componentes remotos de `TDP-Shell`, la sincronización de la ejecución de los componentes de la herramienta Gdb y la utilización de eventos asíncronos para informar de posibles errores durante la ejecución de dichos componentes.

En la líneas 9 del archivo `TDP-Shell` script para `tdp_console`, se utiliza el comando *tdp_launch_agent* para que procese los archivos de descripción los componentes remotos de `TDP-Shell`, pasados como argumentos de este comando tdp y genere el archivo de descripción de trabajos global. Este ultimo fichero informa al gestor de colas del entorno de ejecución de los componentes remotos de `TDP-Shell` y como ejecutar los componentes `tdp_agents`(aprovechando el entorno de ejecución de la librería MPI). Puede observarse como se informa del nombre del ejecutable de la aplicación de usuario (argumentos "-e:" "user_exec_MPI"), para que tanto Condor como SGE, sepan como encontrarlo en sus respectivos ficheros de descripción de trabajos (independencia de los archivos `TDP-Shell` script del gestor de colas). Para el caso de las llamadas asíncronas, estos dos scripts siguen el mismo proceder que en el ejemplo explicado en el apartado 4.7. El archivo `TDP-Shell` script que interpreta `tdp_console` le informa, a través del comando *tdp_asynget* (líneas 5, `TDP-Shell` script 7.5), que se llame a la función *error_gdbserver* (líneas 1, `TDP-Shell` script 7.5) si se produce la inserción de la tupla ERROR_GDBSERVER en el espacio de atributos tdp. Este inserción de la tupla sucede si algún componente `tdp_agent` (comando `tdp_put`, líneas 20, `TDP-Shell` script 7.6), no puede ejecutar correctamente su componente remoto de Gdb. Esta última acción la realiza `tdp_agent` al interpretar el comando `tdp_create_process` (líneas 18) de su fichero `TDP-Shell` script. El componente `tdp_agent` también informa, al resto de `tdp_agents`, del error producido al ejecutar su componente remoto de Gdb, para ello utiliza la tupla *END_GDBSERVERS* (21, `TDP-Shell` script 7.6) y la función asíncrona asociada a esta tupla *end_gdbservers* (líneas 5, `TDP-Shell` script 7.6). Por su parte, el componente `tdp_console` informa de un error al ejecutar su componente local de Gdb (comando tdp_create_process, líneas 17,`TDP-Shell` script 7.5), a través de la tupla *ERROR_GDB* (comando tdp_put, líneas 20, `TDP-Shell`

TDP-Shell script 7.5 Fichero TDP-Shell script para tdp_console

```
1: tdp_fun error_gdbserver () {
2:   tdp_print "ERROR en gdbserver: ", $tdp_value
3:   tdp_exit
4: }
5: tdp_asyngetERROR_GDBSERVER error_gdbserver
6: $JOB_USER="job user:" "/home/user/mpi/job_user.cfg"
7: $JOB_AGENT="job agent:" "/TDP-Shell/config/job_agent.cfg"
8: $JOB_TOOL= "job tool:" "/home/user/mpi/job_gdbserver.cfg"
9: tdp_launch_agent "MPI" $JOB_USER "-e:" "user_exec_MPI" $JOB_AGENT
   $JOB_TOOL
10: $USER_EXEC= "/home/user/user_exec_MPI"
11: $CONT=0
12: $NUM_PROCS=tdp_get TDP_NUM_PROGS
13: while ($CONT < $NUM_PROCS){
14:   $REMOTE_PORT=tdp_get REMOTE_MACHINE_PORT[ANY]
15:   $SH_PROC=tdp_create_process "sh" "-c" "echo target remote "
      + $REMOTE_PORT + " > gdb_remote_" + $REMOTE_PORT + ".cfg"
16:   tdp_wait_process_status $SH_PROC "FINISHED"
17:   $GDB=tdp_create_process "/usr/bin/xterm" "-e" "gdb" "-x" "gdb_remote_" +
      $REMOTE_PORT + ".cfg" $USER_EXEC
18:   if ($GDB == ERROR){
19:     tdp_print "Error creating Gdb"
20:     tdp_put ERROR_GDB = "ERROR"
21:     tdp_exit
22:   }
23:   $CONT= @ $CONT + 1
24: }
25: $GDB_SERVERS_FINISH = 0
26: tdp_print "Waiting gdbservers finish....."
27: while ($GDB_SERVERS_FINISH < $NUM_PROCS){
28:   tdp_get GDBSERVER_END[ANY]
29:   $GDB_SERVERS_FINISH= @ $GDB_SERVERS_FINISH + 1
30: }
```

TDP-Shell script 7.6 Fichero TDP-Shell script para tdp_agent

```
1: tdp_fun error_gdb () {
2:   tdp_print "ERROR from Gdb"
3:   tdp_exit
4: }
5: tdp_fun end_gdbservers () {
6:   tdp_print "ERROR from another Gdbserver"
7:   tdp_exit
8: }
9: tdp_asynget ERROR_GDB error_gdb
10: tdp_asynget END_GDBSERSERS end_gdbservers
11: $USER_EXEC_ARGS = TDPGet_Arguments ("1", "ALL")
12: $USER_EXECUTABLE = tdp_get TDP_USER_EXEC
13: $TOOL_EXECUTABLE = tdp_get TDP_TOOL_EXEC
14: $LOCAL_MACHINE=TDPGet_LocalHost()
15: $PORT=Get_Port()
16: $AGENT_ID=Get_IdConsole()
17: $LOCAL_MACHINE_PORT=$LOCAL_MACHINE + ":" + $PORT
18: $GDBSERVER=tdp_create_process $TOOL_EXECUTABLE
    $LOCAL_MACHINE_PORT $USER_EXECUTABLE $USER_EXEC_ARGS
19: if ($GDBSERVER == ERROR){
20:   tdp_put ERROR_GDBSERVER = "ERROR"
21:   tdp_put END_GDBSERVERS = $AGENT_ID
22:   tdp_exit
23: }
24: tdp_put REMOTE_MACHINE_PORT[$AGENT_ID]=$LOCAL_MACHINE_PORT

25: $GDBSERVER_STATUS = tdp_wait_process_status $GDBSERVER
    "FINISHED" "FAILED"
26: tdp_put GDBSERVER_END[$AGENT_ID] = $GDBSERVER_STATUS
```

script 7.5). Los archivos `TDP-Shell` que interpretan los componentes `tdp_agent` informan que, si se inserta esta tupla *ERROR_GDB* en el espacio de atributos tdp, se llame la función *error_gdb* 10, la cual a sido asociada a esta tupla, utilizando del comando `tdp_asynget` situado en la líneas 5 del fichero `TDP-Shell` script 7.6.

Para la sincronización de los componentes de la herramienta Gdb, el `TDP-Shell` script que ejecuta el componente `tdp_console` le informa, a través del comando `tdp_get` (líneas 14), que espere bloqueado a la inserción de algún valor nuevo, que no se

haya adquirido ya, en la tupla con atributo array *REMOTE_MACHINE_PORT[ANY]* (el índice especial ANY informa de esta funcionalidad). La inserción de este nuevo valor en la tupla REMOTE_MACHINE_PORT, implica que algún `tdp_agent` ha creado su componente remoto de Gdb y que su puerto de comunicaciones está listo para recibir peticiones. Esto implica que `tdp_console` ya puede crear el componente local de Gdb para que se conecte a este componente gdbserver recién creado. Por su parte, el archivo `TDP-Shell` script del componente `tdp_agent`, le informa que después de ejecutar el gdbserver, sitúe la información de su puerto de comunicaciones en el espacio de atributos tdp. Para ello utiliza el comando *tdp_put* de líneas 24 sobre la tupla con atributo array, *REMOTE_MACHINE_PORT*. El valor del índice de esta tupla está contenido en la variable *$AGENT_ID*, la cual contiene el identificador único asociado al componente `tdp_agent`. Este identificador es obtenido gracias a la función especial de `TDP-Shell` *TDPGet_LocalHost()* (líneas 14, `TDP-Shell` script 7.6) y esta formado por la concatenación de las cadenas de caracteres del nombre de la máquina donde se ejecuta el `tdp_agent` más el pid (dado por el sistema operativo) de este componente. El último bucle del archivo `TDP-Shell` script que interpreta `tdp_console`, le informa que espere suspendido, a través del comando *tdp_get* sobre la tupla con atributo array *GDBSERVER_END[ANY]* (líneas 28), a que los componentes `tdp_agent` le confirmen que se ha acabado la ejecución de sus componentes remotos de Gdb. Esto último lo realizan utilizando el comando *tdp_wait_process_status* (líneas 25, `TDP-Shell` script 7.6) para esperar que sus respectivos gdbserver finalicen su ejecución (estén en el estado tdp "FINISHED" o "FAILED") y posteriormente, situando el estado de esta finalización en el espacio de atributos tdp a través de la tupla GDBSERVER_END[$AGENT_ID] (comando *tdp_put*, líneas 26, `TDP-Shell` script 7.6). Es interesante observar el uso de la función especial de `TDP-Shell` *TDPGet_Arguments*, situada en la líneas 11 del archivo `TDP-Shell` script que interpretan los `tdp_agents`. Esta función especial devuelve los argumentos pasados a `tdp_agent`, comprendidos entre los dos argumentos de esta función, que en este caso practico que se está analizando son todos, del primero (primer argumento de la función, "1") a el ultimo ("ALL" significa el último). Con este proceder, se consigue que en el caso de utilizar la librería MPICH1, el componente `tdp_agent` pueda obtener los argumentos conexión propios de esta librería (pasados por el proceso MPI padre a los scripts `tdp_shell_MPI.sh` hijos y de estos a los `tdp_agents`) para pasarlos al proceso de la aplicación MPI que se va a monitorizar.

7.2. Monitorización retardada

En este caso practico se mostrará la utilización de la monitorización retardada utilizando la herramienta de monitorización Totalview sobre el gestor de colas SGE. La monitorización remota consiste en que el componente `tdp_agent` ponga en ejecución el proceso a monitorizar y el usuario decida cuando inicializar su monitorización. Cuando este sucede, el componente `tdp_agent` crea el componente remoto de la herramienta para que este se adjunto al proceso creado anteriormente. Para que el componente local de la herramienta de Totalview pueda adjuntarse a un proceso que esta en ejecución en una máquina remota, necesita conocer la información de conexión de su componente remoto (maquina:puerto) y el pid del proceso a monitorizar

7.2.1. Archivos de descripción de trabajos de SGE

En el archivo de descripción de trabajos (o shell-script) de la aplicación de usuario (FDT 7.7) informa a SGE, en su sección pre-ejecutable (entre las líneass 5 y 8), que copie, desde la máquina local del usuario, el ejecutable de su aplicación a su directorio de trabajo, situado en las máquinas del cluster y que posteriormente se den permisos de ejecución a dicho ejecutable. En la sección post-ejecutable de este shell-script de la aplicación de usuario (entre las líneass 11 y 13) se informa a SGE que elimine el fichero del ejecutable de la aplicación del usuario. El archivo de descripción de trabajos del componente `tdp_agent` (FDT 7.8), igual que el de la aplicación de usuario, también informa a SGE, en su sección pre-ejecutable (entre las líneass 5 y 7), que copie el archivo `TDP-Shell` script para el `tdp_agent`, de la máquina local del usuario a la del cluster. En al sección post-ejecutable (entre las líneass 10 y 12) de este fichero de descripción de trabajos, se informa a SGE que cuando finalice la ejecución del `tdp_agent`, elimine su fichero `TDP-Shell` script de la máquina del cluster. El archivo de descripción de trabajos del componente remoto de la herramienta Totalview, simplemente informa del nombre del ejecutable de este componente. Es interesante observar que estos ficheros de descripción de trabajos suponen que los ejecutables y los entornos de ejecución para `tdp_agent` (exceptuando su fichero `TDP-Shell` script) y el componente remoto de la herramienta, están instalados y disponibles en las máquinas del cluster.

En el fichero de descripción de trabajos global (FDT 7.10), obtenido por `tdp_console`,

FDT 7.7 fichero de descripción de trabajos de SGE para la aplicación de usuario

```
1: #!/bin/sh
2: #$ -N user_exec
3: #$ -o user_exec_out
4: #$ -e user_exec_error
5: #TDP_begin_pre_block
6: scp user@user_localhost:/home/user/user_exec user_exec
7: chmod u+x user_exec
8: #TDP_end_block
9: #TDP_exec
10: ./user_exec
11: #TDP_begin_post_block
12: rm user_exec
13: #TDP_end_block
```

FDT 7.8 fichero de descripción de trabajos de SGE para el componente tdp_agent

```
1: #!/bin/sh
2: #$ -N tdp_agent
3: #$ -o tdp_agent_out
4: #$ -e tdp_agent_error
5: #TDP_begin_pre_block
6: scp user@user_localhost:/home/user/TDP-Shell/tdp/tdp_agent_ret.tdp
   tdp_agent_ret.tdp
7: #TDP_end_block
8: #TDP_exec
9: /TDP_Shell/bin/tdp_agent -cf:/TDP_Shell/cfg/tdp_shell_config.tcf
   -tf:tdp_agent_ret.tdp
10: #TDP_begin_post_block
11: rm tdp_agent_ret.tdp
12: #TDP_end_block
```

FDT 7.9 fichero de descripción de trabajos de SGE para el componente remoto de Totalview

```
1: #!/bin/sh
2: #$ -N tvdsvr
3: #$ -o tvdsvr_out
4: #$ -e tvdsvr_error
5: #TDP_exec
6: tvdsvr
```

se puede observar que en su sección pre-ejecutable (de la líneas 5 a la 7), se han situado los comandos de las secciones pre-ejecutables de los shell-scripts de la aplicación del usuario y de `tdp_agent`. Con esta unión, se informa a SGE del entorno de ejecución previo a la ejecución de estos dos componentes remotos de `TDP-Shell`, las acciones del cual son: Copiar todos los archivos que necesitan la aplicación de usuario y el `tdp_agent`, así como dar permisos de ejecución al fichero ejecutable de esta aplicación de usuario. La parte post-ejecutable de este ficheros de descripción de trabajos global (de la líneas 9 a la 10), realiza las mismas acciones que la sección pre-ejecutable de este archivo, pero con los comandos que se han de ejecutar después de la ejecución del componente `tdp_agent` y de la aplicación de usuario.
Por último, es interesante destacar que la líneas del ejecutable del fichero de descripción de trabajos global (la líneas 8), es la misma que la del fichero del `tdp_agent`. Esto es debido a que el gestor de Colas SGE es el encargado de ejecutar, en una máquina de su cluster, el componente `tdp_agent`.

FDT 7.10 fichero de descripción de trabajos global obtenido por `tdp_console` para SGE

```
1: #!/bin/sh
2: #$ -N TDP_Shell
3: #$ -o TDP_Shell_out
4: #$ -e TDP_Shell_error
5: scp user@user_localhost:/home/user/user_exec user_exec
6: chmod u+x user_exec
7: scp user@user_localhost:/home/user/TDP-Shell/tdp/tdp_agent_ret.tdp
   tdp_agent_ret.tdp
8: /TDP_Shell/bin/tdp_agent -cf:/TDP_Shell/cfg/tdp_shell_config.tcf
   -tf:tdp_agent_ret.tdp
9: rm user_exec
10: rm tdp_agent_ret.tdp
```

Para este caso práctico, las tuplas especiales con la información para los `tdp_agents`, obtenidas `tdp_console`, son:

- **TDP_USER_EXEC**: *user_exec* (Nombre del ejecutable de la aplicación de usuario).
- **TDP_USER_ARGS**: *NOTHING* (Posibles argumentos de la aplicación de usuario).

- **TDP_TOOL_EXEC**: *tvdsrv*. (Nombre del ejecutable del componente remoto de la herramienta).
- **TDP_TOOL_ARGS**: *"-server -set_pw 0:0"* (Posibles argumentos del componente remoto de la herramienta).
- **TDP_SHELL_SCRIPT_AGENT**: *-tf:tdp_agent_ret.tdp* (`TDP-Shell` script para el `tdp_agent`).

7.2.2. Archivos TDP-Shell script para tdp_agent y tdp_console

El los archivos de descripción de trabajos que interpretan `tdp_console` (`TDP-Shell` script 7.11) y `tdp_agent` (`TDP-Shell` script 7.11) para la monitorización retardada, se pueden observar dos partes en común (o muy parecidas) con este tipo de archivos para el caso de aplicaciones MPI (punto 7.1.2). El primero de estos puntos, es el mecanismo para informar de los posibles errores en la creación de los componentes de la herramienta (y de la aplicación de usuario), el cual consiste en que la inserción de una tupla especial, que identifica al error, produzca la llamada a una función que trata este error. El segundo punto en común, en este caso para los archivos `TDP-Shell` script de `tdp_console`, es la utilización del comando *tdp_launch_agent* (líneas 25) para informar al gestor de colas (en este caso, SGE) que ejecute el `tdp_agent` y cree el entorno de ejecución de los componentes remotos de `TDP-Shell`, en una máquina de su cluster.

La parte diferenciadora de estos archivos `TDP-Shell`, respecto a otros ficheros de este tipo, es el procedimiento de monitorización remota. Una vez en ejecución el componente `tdp_agent` en la maquina remota, se puede observar que el archivo `TDP-Shell` script que interpreta, le informa que ejecute el proceso de la aplicación de usuario a monitorizar(comando *tdp_create_process*, líneas 20) y que obtenga el pid (identificador del sistema operativo) de este proceso, ya que esta información la necesita el componente local de la herramienta. Para obtener este pid utiliza la función especial de `TDP-Shell` *TDPGet_Process_local_OS_id()* (cuyo argumento es el *pid_tdp* del proceso a monitorizar) situada en la líneas 26 del fichero `TDP-Shell` que interpreta `tdp_agent`. Una vez realizadas estas acciones, el `TDP-Shell` script que interpreta `tdp_agent`, le informa que sitúe la información de este pid en el espacio de atributos tdp, utilizando el comando *tdp_put* situado su líneas 27 y que se quede esperando (suspendido) a que el

TDP-Shell script 7.11 Fichero TDP-Shell script para tdp_console

```
1: tdp_fun error_tvdsvr () {
2:   tdp_print “ERROR in tvdsvr : ”
3:   tdp_exit
4: }
5: tdp_fun error_user_exec () {
6:   tdp_print “ERROR in user process : ”
7:   tdp_exit
8: }
9: tdp_fun exec_totalview () {
10:   $MACHINE_PORT=tdp_get REMOTE_MACHINE_PORT
11:   $PID=tdp_get REMOTE_PID
12:   $TOTAL=tdp_create_process “totalview” %USER_EXEC $MACHINE_PORT
      “-pid” $PID
13:   if ($TOTAL == ERROR){
14:     tdp_print “Error creating Totalview”
15:     tdp_put ERROR_TOTALVIEW= “ERROR”
16:     tdp_exit
17:   }
18: }
19: tdp_asyngetERROR_TVDSVR error_tvdsvr
20: tdp_asyngetERROR_USER_EXEC error_user_exec
21: %USER_EXEC= “/home/user/user_exec”
22: $JOB_USER=“job user:” “/home/user/job_user.cfg”
23: $JOB_AGENT=“job agent:” “/TDP-Shell/config/job_agent.cfg”
24: $JOB_TOOL= “job tool:” “/home/user/job_tvdsrv.cfg”
25: tdp_launch_agent $JOB_USER $JOB_AGENT $JOB_TOOL
26: tdp_wait_local_invocation exec_totalview
27: tdp_interactive
28: tdp_remote_create_process
29: tdp_wait_process_status $TOTAL “FINISHED”
```

usuario decida comenzar la monitorización. Esta última acción la realiza al interpretar el comando *tdp_wait_remote_invocation*, situado en la líneas 28 del archivo `TDP-Shell` script para `tdp_agent`. Este comando tdp también informa de la función que hay que llamar, exec_tvdsrv (líneas 5), cuando el usuario decida comenzar la monitorización de su aplicación.

TDP-Shell script 7.12 Fichero TDP-Shell script para tdp_agent

```
1: tdp_fun error_totalview () {
2:   tdp_print "ERROR in Totalview"
3:   tdp_exit
4: }
5: tdp_fun exec_tvdsrv () {
6:   $TOOL_ARGS=tdp_get TDP_TOOL_ARGS
7:   $LOCAL_MACHINE=TDPGet_LocalHost()
8:   $PORT=Get_Port()
9:   $LOCAL_MACHINE_PORT=$LOCAL_MACHINE + ":" + $PORT
10:  $TVDSRV=tdp_create_process "tvdsrv" $TOOL_ARGS
11:  if ($TVDSRV == ERROR){
12:    tdp_print "Error creating Tvdsrv"
13:    tdp_put ERROR_TVDSVR = "ERROR"
14:    tdp_exit
15:  }
16:  tdp_put REMOTE_MACHINE_PORT=$LOCAL_MACHINE_PORT
17: }
18: tdp_asyngetERROR_TOTALVIEW error_totalview
19: $USER_EXECUTABLE = tdp_get USER_EXECUTABLE
20: $EXEC=tdp_create_process $USER_EXECUTABLE
21: if ($EXEC == ERROR){
22:   tdp_print "Error creating user exec"
23:   tdp_put ERROR_USER_EXEC = "ERROR"
24:   tdp_exit
25: }
26:  %PROC_PID=TDPGet_Process_local_OS_id($EXEC)
27: tdp_put REMOTE_PID= %PROC_PID
28: tdp_wait_remote_invocation exec_tvdsrv
29: tdp_wait_process_status $TVDSRV "FINISHED"
```

Por su parte y después de la ejecución del comando *tdp_launch_agent*,el archivo `TDP-Shell` script que interpreta el componente `tdp_console` le informa, usando el

comando *tdp_wait_local_invocation* (líneas 26), del nombre de la función, el cual es exec_totalview (líneas 9), que ha de ejecutar cuando el usuario quiera comenzar la monitorización. Después de estas acciones, `tdp_console` entra en modo interactivo con el comando *tdp_interactive* (líneas 27 de su archivo `TDP-Shell` script). Este comando permite, a `tdp_console`, interpretar comandos tdp del usuario mientras se esta interpretando un archivo `TDP-Shell` script.
Cuando el usuario desea comenzar la monitorización de su aplicación, sale del modo interactivo (introduciendo el comando *tdp_exit_interactive*) y a continuación, `tdp_console` interpreta el comando *tdp_remote_create_process* (líneas 28) de su archivo `TDP-Shell` script, el cual desencadena el procedimiento de monitorización remota.
Esta petición del usuario provoca que el componente `tdp_agent` ejecute la función *exec_tvdsrv* de su fichero `TDP-Shell` script, la cual obtiene el nombre de la máquina (función especial *TDPGet_LocalHost()*, líneas 7), escoge un puerto de comunicaciones libre (función especial TDPGet_LocalHost(), líneas 8) y ejecuta el componente de la herramienta Totalview (comando *tdp_create_process*, líneas 10). Una vez realizadas estas acciones, esta función *exec_tvdsrv* sitúa, en el espacio de atributos tdp, la información de conexión del componente remoto de Totalview, utilizando para ello un comando *tdp_put* (líneas 16) sobre la tupla REMOTE_MACHINE_PORT.
Mientras el componente `tdp_agent` realiza estas acciones, `tdp_console`, interpretando su fichero `TDP-Shell`, llama a la función exec_totalview, la cual obtiene, del espacio de atributos tdp, los dos datos que necesita para que la herramienta Totalview pueda monitorizar remotamente un proceso que ya está en ejecución. El primero de estos datos es la información de conexión del componente remoto de Totalview. Esta acción la realiza a través del comando *tdp_get* (líneas 10) sobre la tupla REMOTE_MACHINE_PORT. El segundo dato es el pid del proceso que se desea monitorizar y que ha sido creado por el `tdp_agent`. Igual que con la información de conexión, esta acción la realiza con el comando *tdp_get* (sobre la tupla REMOTE_PID) de la líneas 11) del archivo `TDP-Shell` script de `tdp_console` . Una vez tiene esta información, el componente `tdp_console` ejecuta el componente local de la herramienta Totalview, interpretando el comando *tdp_create_process* situado en la líneas 12 de este fichero `TDP-Shell` script. A este componente local de Totalview, `tdp_console` le pasa como argumentos, la información de conexión con su componente remoto y el pid del proceso al que se tiene que adjuntar este componente remoto. Cuando el componente remoto de Totalview recibe esta información

del pid del proceso, se le adjunta y empieza la monitorización.

7.3. Conclusiones

En este capítulo se han mostrado dos casos prácticos, donde se han podido constatar los dos puntos mas importantes de la arquitectura `TDP-Shell`:

- El procesamiento que realiza `tdp_console` de los archivos de descripción de trabajos de los componentes remotos de `TDP-Shell`, para obtener el fichero de descripción de trabajos global. Este procesamiento, como se ha podido observar, es diferente para los archivos de descripción de trabajos de Condor de los de SGE, debido a su formato diferente (SGE utiliza shell-scripts y Condor un formato propio basado en comandos).

- Como definir la sincronización de los componentes de la herramienta y la información que necesitan intercambiarse. Para ello, `tdp_console` y `tdp_agent` interpretan sus respectivos ficheros `TDP-Shell` script. El de `tdp_console` define la parte del componente local de la herramienta y el del `tdp_agent`, la parte del componente remoto de la misma. También se puede observar que, para facilitar su aprendizaje, el formato de estos archivos `TDP-Shell` script está basado en los shell-scripts con instrucciones y definición de funciones locales basados en los lenguajes de alto nivel como C.

Por lo tanto, estos casos prácticos ayudan a mostrar una de las características más importantes del entorno `TDP-Shell`, que sus usuarios no deban dedicar mucho tiempo al aprendizaje del funcionamiento de este entorno.

8

Conclusiones y líneas futuras

En este capitulo se explicarán las conclusiones extraídas de este trabajo de tesis, así como las líneas abiertas que este deja.

8.1. Conclusiones

En este trabajo de tesis se ha propuesto y desarrollado una solución al problema de la falta de interoperabilidad entre herramientas de monitorización y gestores de colas. Este problema se produce debido a las grandes dificultades que tienen los usuarios al intentan utilizar, en los clusters controlados por estos gestores de colas, las herramientas de monitorización en su forma estándar. Su causa principal es debida a que las herramientas de monitorización y los gestores de colas, no están diseñados para compartir la información y los recursos necesarios que les permitan realizar, conjuntamente, el proceso de monitorización.
Para solucionar este problema de falta de interoperabilidad se ha desarrollado en entorno de trabajo TDP-Shell, cuyas características más destacables son la facilidad de uso (el usuario no debe dedicar mucho tiempo a su aprendizaje), su flexibilidad para adaptarse a diferentes gestores de colas y herramientas de monitorización, así como no tener que (o evitar al máximo) modificar sus códigos ejecutables (para evitar el tiempo que esto representa, además de que algunos no son accesibles por ser productos comerciales).
El diseño de la arquitectura base de `TDP-Shell`, diseñada para solucionar el problema de la falta de interoperabilidad entre gestores de colas y herramientas de monitorización

para aplicaciones serie (un solo proceso), se ha basado en el hecho de que el diseño de sus arquitecturas utiliza dos componentes principales : El *local*, que normalmente se ejecuta en la máquina local del usuario y se encarga de interactuar con este usuario (recoger sus peticiones y mostrar los resultados de estas) y el componente *remoto*, que se ejecuta en las máquinas del cluster y se encarga de realizar las acciones que resuelven las peticiones del usuario.
Aprovechando esta característica, el diseño base de TDP-Shell define los dos componentes principales: El local o `tdp_console`, que se ejecuta en la máquina local del usuario y se encarga de sincronizar la ejecución del componente local de de la herramienta, con el componente remoto de esta. El segundo componente de TDP-Shell es el remoto o `tdp_agent`, el cual se ejecuta en las máquinas remotas y se encarga de sincronizar la ejecución del componente remoto de la herramienta de monitorización, con el componente local de esta. También se puede encargar, en caso necesario, de crear el proceso del usuario a monitorizar. El componente `tdp_console` también realiza la petición, al gestor de colas, para que ejecute el componente `tdp_agent`. Para ello, procesa los ficheros de descripción de trabajos de los componentes remotos de TDP-Shell, los cuales informan al gestor de colas, del entorno de ejecución la aplicación de usuario, el componente `tdp_agent` y el componente remoto de la herramienta. De este procesamiento obtiene el *fichero de descripción de trabajos global*, el cual describe el *entorno de ejecución global* que permite, al gestor de colas, realizar las acciones necesarias para que se puedan ejecutar, directamente por el o a través de otro componente, estos componentes remotos del `entorno TDP-Shell` (entre ellos el tdp_agent).

Los archivos especiales `TDP-Shell script`, uno interpretado por el `tdp_console` y el otro por `tdp_agent`, definen el orden temporal de envió y recepción de la información que ambos componentes necesitan para poder ejecutar, sincronizadamente, el componente local (`tdp_console`) y el remoto (`tdp_agent`) de la herramienta de monitorización. El formato de estos ficheros es parecido al de los ficheros shell-script (para su fácil aprendizaje). Para definir sus acciones, estos ficheros `TDP-Shell script` utilizan, para el intercambio de información, los comandos tdp (específicos del entrono TDP-Shell) de comunicación síncrona (o bloqueante) y asíncrona (no bloqueante), de gestión de procesos locales, para la creación de los componentes de la herramienta y de comunicación con el gestor de colas, para solicitar a dicho gestor, la ejecución de `tdp_agent` en su cluster.

Para el intercambio de información, los comandos tdp de comunicaciones utilizan tuplas (cuyo formato es atributo, valor)) que sitúan y extraen del espacio de atributos tdp que se utiliza para almacenarlas.
La publicación asociada al tema del diseño del entorno `TDP-Shell` base es:

- V. Ivars, A. Cortes, and M.A. Senar. *TDP SHELL: An Interoperability Framework for Resource Management Systems and Run-time Monitoring Tools*, LNCS 4128, pp. 15 - 24, 2006, Springer-Verlag.

Como en el caso de las aplicaciones serie, en la monitorización de aplicaciones distribuidas basadas en MPI, tampoco existe la interoperabilidad entre los gestores de colas y las herramientas de monitorización. Para solucionar este problema, el entorno TDP-Shell aprovecha la característica de que, para operar con aplicaciones MPI, la mayoría de los gestores de colas y herramientas de monitorización, utilizan el entorno de ejecución de la de la librería MPI que estas utilizan (el cual se encarga de ejecutar los procesos MPI en las maquinas seleccionadas).
Por lo tanto, el nuevo diseño del entorno `TDP-Shell` (adaptado del diseño base) que soluciona este problema de falta de interoperabilidad para aplicaciones MPI, se basa en solicitar al gestor de colas (a través del fichero de descripción de trabajo global obtenido por `tdp_console`) que utilice el entorno de ejecución propio de la librería MPI, para ejecutar en las máquinas seleccionadas, los componentes `tdp_agent` en lugar de los procesos MPI de la aplicación de usuario. Una vez que estos componentes `tdp_agent` están en ejecución ya pueden interpretar sus archivos `TDP-Shell` script, para que les informen de como sincronizar la ejecución de los componentes remotos de la herramienta, con el componente local de la misma (el cual es ejecutado por el `tdp_console`). Una vez están en ejecución los componentes de la herramienta, ya se pueden iniciar las actividades de monitorización de la aplicación MPI.
La ejecución de diversos componentes `tdp_agent` que conlleva la monitorización de aplicaciones MPI, implica añadir nuevas funcionalidades al entorno `TDP-Shell`, entre ellas el control de acceso a tuplas compartidas (semáforos) o el almacenamiento de diversos valores en una sola tupla (nuevo tipo de tuplas denominado *tupla con atributo array*).
Las publicaciones asociadas al tema del diseño del entorno `TDP-Shell` para la monitorización de aplicaciones MPI sobre gestores de colas, son las siguientes :

- Vicente-José Ivars, Miquel Angel Senar, Elisa Heymann. *TDP-Shell: Achieving interoperability between Resource Management Systems and Monitoring Tools for MPI Environments.* CEDI 2010: Actas de las XXI Jornadas de Paralelismo, pp 651-658. Editorial IBERGARCETA PUBLICACIONES, S.L.

- Vicente-José Ivars, Miquel Angel Senar, Elisa Heymann. *TDP-Shell: A Generic Framework to Improve Interoperability between Batch Queue Systems and Monitoring Tools*. 2011 IEEE International Conference on Cluster Computing, pp 522 - 526. E-ISBN : 978-0-7695-4516-5, Print ISBN: 978-1-4577-1355-2.

El entorno TDP-Shell se ha adaptado a un nuevo tipo de monitorización denominado *ejecución retardada de la herramienta de monitorización o monitorización retardada*, el cual consiste en que se cree el proceso de la aplicación en la máquina remota y que el usuario decida cuando comenzar su monitorización. Este hecho implica, que el componente remoto de la herramienta se adjunte a este proceso creado anteriormente. Para dar soporte a este nuevo tipo de monitorización, el entorno TDP-Shell ha desarrollado dos comandos para informar, a `tdp_console` y `tdp_agent`, de las funciones que han de ejecutar cuando el usuario decida comenzar la monitorización retardada. Los códigos tdp de estas funciones, una ejecutada por `tdp_conole` y la otra por `tdp_agent`, son los encargados de sincronizar la ejecución del componente local de la herramienta con los componentes remotos de la misma. También se ha desarrollado otro comando tdp que permite, al usuario, avisar a `tdp_console` y a los `tdp_agent`, cuando quiere comenzar el procedimiento de monitorización retardada.

8.2. líneas abiertas

Debido a que el tema tratado en este trabajo de tesis es la primera versión del entorno de trabajo `TDP-Shell`, deja varias líneas de investigación abiertas, las más importantes son:

- Implementar el procesamiento de la mayoría de los comandos y primitivas de los gestores de colas, contenidos en los archivos de descripción de trabajos de los componentes remotos de `TDP-Shell`. De esta forma, `tdp_console` puede generar un archivo de descripción de trabajos más completo, que describa el entorno de

ejecución global (para los componentes remotos de `TDP-Shell`) con un mayor número de acciones.

- Ampliar el diseño del entorno `TDP-Shell` para que incluya la monitorización de otros tipos de aplicaciones distribuidas, como por ejemplo basadas en OpenMPI. Esto hecho seguramente implicará añadir nuevos comandos tdp e instrucciones a los ficheros `TDP-Shell` script.

- Investigar el diseño de nuevos servidores del espacio de atributos tdp que se basen en una arquitectura distribuida (diversos servidores interconectados entre si). Con este diseño, se puede conseguir que los servidores adapten mejor a las diferentes topologías de los clusters y mejorar de esta manera, la escalabilidad de este servidor del espacio de atributos. Este tema es importante para el caso que se desea utilizar el entorno `TDP-Shell` en la monitorización de aplicaciones que generan un conjunto elevado de procesos y por tanto, de componentes `tdp_agent` (como puede suceder en entornos Grid).

- Implementar un acceso multiusuario al espacio de atributos tdp. Este tipo de acceso permite que un servidor del espacio de atributos tdp (ya sea único o distribuido) gestione diversas sesiones `TDP-Shell` de diferentes usuarios, permitiendo una gestión mas precisa de los recursos del sistema (comparada con la un servidor, un solo usuario). Otra característica es que permite compartir cierta información común (o global) a todas las sesiones `TDP-Shell` de los diferentes usuarios, como por ejemplo los archivos de configuración de las herramientas o de los componentes de TDP-Shell. Por último, este acceso multiusuario también permite que la información local de una sesión de un usuario, no sea accesibles desde las sesiones `TDP-Shell` de otros usuarios.

- Ofrecer el entorno de trabajo `TDP-Shell` como un estándar que puedan adoptar las herramientas de monitorización (desde su diseño inicial), para indicar a los gestores de colas como lanzar sus componentes remotos. También seria interesante ampliar la utilización de `TDP-Shell`, a otras aplicaciones con las mismas necesidades que las herramientas de monitorización y a otros gestores de recursos, con un comportamiento parecido a los gestores de colas.

Apéndices

Apéndice A

Nomenclaturas usadas en este documento

La nomenclatura usada en la representación sintáctica de las diferentes estructuras de este documento es la siguiente:

- *[elemento]* : Diferenciar el *elemento* del resto de *elementos*.
- *[elemento]?* : Puede haber 0 o 1 *elementos*.
- *[elemento]+* : Puede haber 1, 2 o n repeticiones de *elemento* (n pertenece a los números Naturales).
- *[elemento]** : Puede haber 0, 1, 2 o n repeticiones de *elemento*.
- *[elemento 1 | elemento 2]* : Opcionalidad, un *elemento* o el otro, pero nunca a la vez.

La nomenclatura usada en la sintaxis de declaración de funciones es la siguiente:

nombre_función ([tipo:[nombre_argumento | &nombre_argumento]]*):[retorno]?
Donde:

- *nombre_función:* Es el nombre que identifica la función .
- *tipo:* Define el tipo del argumento.

- *nombre_argumento:* Nombre del argumento pasado como valor, su contenido no se modifica dentro de la función.
- *&nombre_argumento:* Nombre del argumento pasado como referencia, su contenido se modifica dentro de la función.
- *retorno:* Define el tipo del dato que devuelve la función.

Bibliografía

[1] M.J.Mutka, M.Livny, and M.W. Litzkow. Condor, a hunter of idle workstations. *8th Int'l Conf. on Distributed Systems, San Francisco*, Junio 1988. Página web del proyecto Condor: http://research.cs.wisc.edu/condor/.

[2] Oracle grid engine. url = http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html.

[3] Open grid scheduler. url= http://sourceforge.net/projects/gridscheduler/.

[4] Pbs works. url = http://www.pbsworks.com/default.aspx.

[5] Adaptative computing. url = http://www.adaptivecomputing.com/products/ torque.php.

[6] Gdb: The gnu project debugger. url=http://www.gnu.org/software/gdb/.

[7] Rogue wawe software, totalview. url = http://www.roguewave.com/products/ totalview-family/totalview.aspx.

[8] B. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K.Kunchithapadam, and T.Newhall. The paradyn parallel performance measurement tools. *IEEE Computer 28*, Noviembre 1995.

[9] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Stack trace analysis for large scale applications. *International Parallel and Distributed Processing Symposium, Long Beach, California*, Marzo 2007. Página web de la herramienta Stat: http://www.paradyn.org/STAT/STAT.html.

[10] Message passing interface forum. url = http://www.mpi-forum.org/.

[11] Message Passing Interface Forum. University of Tennessee, Knoxville, Tennessee. *MPI: A Message-Passing Interface Standard, Version 2.2*, septiembre 2009. Página web: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[12] Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Gregory Lee, Barton P. Miller, and Martin Schulz. Overcoming scalablility challenges for tool daemon launching. *37th International Conference on Parallel Processing (ICPP-08), Portland, Oregon*, Septiembre 2008.

[13] Jr. John DelSignore. Technical article: Mpi debugging standards. what took so long? *TotalView Technologies*, Febrero 2009.

[14] R. Wismuller, J. Trinitis, and T. Ludwig. Ocm - a monitoring system for interiperable tools. *in Proc. 2nd SIGMETRICS Symposium on Parallel and Distrubuted Tools, Welches, USA*, Agosto 1998.

[15] R. Prodan and J. M. Kewley. A framework for an interoperable tool environment. *Proc. of EuroPar 2000, Lecture Notes in Computer Science*, Volumen 1900:65–69, 2000.

[16] T.Fahringer, A. Jugravu, S. Pllana, R. Prodan, Jr. C. Seragiotto, and Hong-Linh Truong. Askalon: A tool set for cluster and grid computing. *Concurrency and Computation: Practice and Experience*, Volumen 17 , edición 2-4 Grid Performance:142–169, 2005.

[17] B. Miller, A. Cortes, M. A. Senar, and M. Livny. The tool daemon protocol (tdp). *Proceedings of SuperComputing*, Noviembre 2003.

[18] Submitting a job. url = http://research.cs.wisc.edu/condor/manual/vx.y/2_5submitting_job.html, donde x.y es la versión de condor.

[19] Command reference manual (man pages), url = http://research.cs.wisc.edu/condor/manual/vx.y/9_command_reference.html, donde x.y es la versión de condor.

[20] Grid engine man pages, url = http://gridscheduler.sourceforge.net/htmlman/htmlman1/qsub.html.

[21] Grid engine man pages, url = http://gridscheduler.sourceforge.net/htmlman/manuals.html.

[22] Condor administrators'manual, introduction url = http://research.cs.wisc.edu/condor/manual/vx.y/3_1introduction.html, donde x.y es la versión de condor.

[23] Oracle Corporation, World Headquarters, 500 Oracle Parkway, Redwood Shores, CA 94065, U.S.A. *An Oracle White Paper, Beginner's Guide to Oracle Grid Engine 6.2*, Agosto 2010. Página web : http://www.oracle.com/technetwork/oem/host-server-mgmt/twp-gridengine-beginner-167116.pdf.

[24] Sun Microsystems, Inc. 4150 Network Circle Santa Clara, CA 95054 U.S.A. *N1 Grid Engine 6 User's Guide*, Mayo 2005. Página web :http://docs.oracle.com/cd/E19080-01/n1.grid.eng6/817-6117/817-6117.pdf.

[25] Richard Stallman, Roland Pesch, Stan Shebs, and et al. *Debugging with gdb. Tenth Edition, for gdb version 7.4.50.20120313.* Free Software Foundation51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA ISBN 978-0-9831592-3-0, 2011. Versión on-line del documento: http://sourceware.org/gdb/current/onlinedocs/gdb/index.html.

[26] Computer Sciences Department, University of Wisconsin Madison, WI 53706-1685. *Paradyn Parallel Performance Tools, User's Guide, Release 5.1*, 2007.

[27] Computer Sciences Department, University of Wisconsin Madison, WI 53706-1685. *Paradyn Parallel Performance Tools, Developer's Guide, Release 5.1*, 2007.

[28] Computer Science Department University of Wisconsin-Madison Madison, WI 53711. Computer Science Department University of Maryland College Park, MD 20742. *Paradyn Parallel Performance Tools, Dyninst Programmer's Guide, Release 7.0*, 2011. Página web : http://www.dyninst.org/sites/default/files/manuals/dyninst/dyninstProgGuide.pdf.

[29] Rogue Wave Software, Inc. *TotalView Installation Guide,*, 2011. Página web : http://www.roguewave.com/documents.aspx?entryid=1142&command =core_download.

[30] Rogue Wave Software, Inc. *TotalView User Guide*, 2011. Página web : http://www.roguewave.com/documents.aspx?entryid=1138&command =core_download.

[31] Rogue Wave Software, Inc. *TotalView Reference Guide*, 2011. Página web : http://www.roguewave.com/documents.aspx?entryid=1139&command =core_download.

[32] Vicente Ivars, Ana Cortes, , and Miquel A. Senar. Tdp_shell: An interoperability framework for resource management systems and run-time monitoring tools. *Europar 2006, LNCS 4128*, pages 15 – 24, 2006.

[33] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM, 32, 4,*, pages 444–458, 1989.

[34] George Reese. *Database Programming with JDBC and Java, capítulos 7 y 8.* O'Reilly and Associates, 2000. Página web capitulo 7: http://java.sun.com/developer/Books/jdbc/ch07.pdf, página web capitulo 8: http://java.sun.com/developer/Books/jdbc/ch08.pdf.

[35] W. Richard Stevens. *Unix Network Programming, volúmenes 1-2.* Prentice Hall, 1998.

[36] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP, volúmenes I-III.* Prentice Hall.

[37] Brian "Beej Jorgensen"Hall. *Beej's Guide to Network Programming, Using Internet Sockets.* Jorgensen Publishing, 2011.

[38] Beej's guide to network programming, using internet sockets. url = http://beej.us/guide/bgnet/.

[39] Pvm (parallel virtual machine). url = http://www.csm.ornl.gov/pvm/.

[40] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Book Order Department, 55 Hayward Street, Cambridge, MA 02142, 1994.

[41] Mpich2. url = http://www.mcs.anl.gov/research/projects/mpich2/index.php.

[42] Lam/mpi parallel computing. url = http://www.lam-mpi.org/.

[43] Open mpi: Open source high performance computing. url = http://www.open-mpi.org/.

[44] Vicente-José Ivars, Miquel Angel Senar, and Elisa Heymann. Tdp-shell: Achieving interoperability between resource management systems and monitoring tools for mpi environments. *CEDI 2010: Actas de las XXI Jornadas de Paralelismo*, pages 651–658, 2010.

[45] Vicente-José Ivars, Miquel Angel Senar, and Elisa Heymann. Tdp-shell: A generic framework to improve interoperability between batch queue systems and monitoring tools. *2011 IEEE International Conference on Cluster Computing*, pages 522–526, 2011.

[46] Parallel applications (including mpi applications). url = research.cs.wisc.edu/condor/manual/vx.y/2_9parallel_applications.html, donde x.y es la versión de condor.

[47] Sun Microsystems, Inc. 4150 Network Circle Santa Clara, CA 95054 U.S.A. 650-960-1300. *Sun ONE Grid, Enterprise Edition Reference Manual*, Octubre 2002.

[48] 3.13.10 condor's dedicated scheduling. url = http://research.cs.wisc.edu/condor/manual/vx.y/3_13setting_up.html#section0041310000000000000000, donde x.y es la versión de condor.