Un Framework Basado en Componentes para Aplicaciones Master/Worker en CORBA–$\mathcal{LC}^\star$

José Antonio Pérez Sánchez[1], Diego Sevilla Ruiz[2], José M.García[2] y Antonio F. Gómez Skarmeta[1]

Resumen— **En este artículo se presenta un *framework* tolerante a fallos basado en componentes para aplicaciones *Master/Worker*. Mostramos cómo la aplicación de componentes puede mejorar el diseño, reusabilidad, escalabilidad y robustez de este tipo de aplicaciones. Este *framework* se ha construido utilizando el modelo de componentes CORBA *Lightweight Components*, CORBA–$\mathcal{LC}$. Se mostrará cómo este modelo se adapta perfectamente a las necesidades de este tipo de aplicaciones y nos permitirá diseñar distintas estrategias para desarrollarlas. A su vez, al ser CORBA–$\mathcal{LC}$ un modelo todavía en desarrollo, el *framework* sirve como un banco de pruebas para mostrar su validez.**

Palabras clave— **CORBA, Componentes Software, Master/Worker**

I. Introducción

EL paradigma de programación *Master/Worker* ha sido ampliamente utilizado en el ámbito de las aplicaciones de cálculo científico y de computación de altas prestaciones y de *Grid* [1], [2], [3]. Proyectos como distributed.net [4] o SETI@home [5] también utilizan variaciones de esta arquitectura para conseguir aglutinar el poder de computación de cientos e incluso miles de ordenadores.

Aunque el paradigma de aplicaciones *Master/Worker* es ampliamente conocido y utilizado, resultando muy adecuado para un rango de aplicaciones particulares, la mayoría de las implementaciones de este paradigma se realizan *ad hoc* para cada aplicación particular y para una arquitectura, máquina paralela o *cluster* en particular, en muchas ocasiones utilizando software de paso de mensajes como PVM o MPI [6].

Además, el paradigma *Master/Worker* permite en teoría una perfecta separación entre la parte común a todas las aplicaciones (manejo de recursos) y la parte específica de cada aplicación (el cálculo o proceso específico), sin embargo, es frecuente que las aplicaciones *Master/Worker* contienen ambas partes mezcladas sin posibilidad de reutilización para posteriores aplicaciones.

En el mejor de los casos, el usuario/programador todavía tiene que encargarse de instalar el software adecuado en cada uno de los distintos nodos que incurren en la computación y enfrentarse a problemas de versiones y dependencias.

Gracias al diseño basado en componentes, podemos afrontar el diseño de un *framework* que permita reutilizar las partes comunes del paradigma *Master/Worker* y a la vez permita integrar nuevas aplicaciones centrándose sólo en el proceso particular que éstas tienen que realizar. El soporte de programación basada en componentes lo proporcionará el modelo de componentes CORBA–$\mathcal{LC}$ (*CORBA Lightweight Components*) [7], [8].

En la sección II se introduce el *framework* para aplicaciones *Master/Worker*, exponiendo, por un lado, las características de lo que denominamos un «*Componente*» y cómo la filosofía de diseño basado en componentes se aplica a este tipo de aplicaciones. En la sección III se introduce el modelo de componentes CORBA–$\mathcal{LC}$, y cómo sus características ayudan a implementar el *framework* descrito sobre una red de ordenadores distribuida y heterogénea. La sección IV muestra una evaluación del modelo desde varios puntos de vista, desde la adecuación desde el punto de vista del diseño del software hasta las consideraciones de red, escalabilidad, tolerancia a fallos, etc. Finalmente, la sección V muestra las conclusiones extraídas de la implementación actual del *framework*, cómo mejorarlo y qué características de CORBA–$\mathcal{LC}$ son deseables a raíz de este estudio.

II. Un framework basado en componentes para aplicaciones Master/Worker

Las aplicaciones *Master/Worker* siguen un patrón regular y particular en las que un programa principal o «*Master*» utiliza a un conjunto de programas trabajadores o «*Worker*» para distribuir una tarea grande en otras más pequeñas. Los programas trabajadores, a su vez, se distribuyen entre las distintas máquinas de una red, que, al funcionar en paralelo, reducen el tiempo de resolución de la tarea en total, ya que cada uno de los trabajadores realiza una parte de la misma.

El conjunto de aplicaciones que se pueden beneficiar de esta organización es muy amplio, sobre todo en el ámbito del problemas con cálculo intensivo. El problema debe permitir una división en un número finito de «trabajos» de cuya solución se pueda construir la solución final al problema, además de tener un alto *ratio* cómputo/comunicación, es decir, que dividir el trabajo y enviarlo a los distintos trabajadores para su cálculo sea menos costoso en tiempo que si el cómputo se realiza localmente.

Teniendo en cuenta esta organización, construimos

⋆Este trabajo se recoge bajo el proyecto PB/13/FS/99 de la Fundación Séneca.

1. Departamento de Ingeniería de la Información y las Comunicaciones, Facultad de Informática, Universidad de Murcia, 30171 Murcia (España).

2. Departamento de Ingeniería y Tecnología de Computadores, Facultad de Informática, Universidad de Murcia, 30171 Murcia (España).

una primera versión del *framework* con la que podremos evaluar (1) cuan buena es la tecnología de desarrollo basado en componentes para conseguir una máxima reutilización, una división clara de responsabilidades y en concreto para aplicarla al problema *Master/Worker*, y (2) cómo el modelo de componentes utilizado (CORBA–$\mathcal{LC}$, §III) ayuda a la programación basada en componentes en general y a las aplicaciones *Master/Worker* en particular.

Este primer diseño se esquematiza en la figura 6 de la página 6. Hemos utilizado diagramas UML en el que se representan a la vez los componentes y las instancias de los mismos. En ella se muestran los principales componentes del *framework.* Los componentes ofrecen («*provide*») y utilizan («*use*») un conjunto de «puertos». Estos puertos pueden ser, bien *interfaces* para comunicación síncrona, bien *eventos* para comunicación asíncrona.

En la figura se pueden identificar los siguientes componentes:

- `GenericMaster`. Este componente implementa un *Master genérico.* Así, encapsula la funcionalidad que es común a todas las aplicaciones *Master/Worker.* El usuario utiliza el interface `JobAcceptor` implementado por este componente para pedir la realización de un trabajo en particular. Para cada aplicación, es el encargado de controlar el *Master* y el conjunto de *Workers* específicos para la misma, utilizándolos a través de sus interfaces `Master` y `Worker` respectivamente (Figura 7, pág. 6). También realiza la distribución de los trabajos requeridos por el *Master* para la aplicación particular entre el conjunto de *Workers* disponible.
- `MatrixVector::MVMaster`. Cada componente *Master* debe implementar el interface `Master`. En particular, este componente implementa la funcionalidad específica de un *Master* en una multiplicación matriz-vector. Así, sabe exactamente cómo dividir el trabajo de una multiplicación matriz-vector entre distintos trabajos independientes (dividiendo la matriz en subconjuntos de filas junto con el vector), y cómo agrupar las soluciones parciales en una solución completa final. El interface `Master` es utilizado por el `GenericMaster` para pedirle al componente el conjunto de trabajos a asignar a los *Workers* y para obtener la solución final.
- `MatrixVector::MVWorker`. De la misma forma, este componente sabe cómo realizar el trabajo específico del problema, la multiplicación de una submatriz por el vector. Todo componente *Worker* debe implementar el interface `Worker`, que también será utilizado por el componente `GenericMaster` para pedirle la realización de un trabajo.

En la figura también aparece un «paquete» UML llamado `Fractal`. Este paquete implementa el cálculo de un fractal Mandelbrot haciendo uso del «framework». Estos componentes también implementan los interfaces necesarios, con lo que se pueden integrar sin ningún problema, abriendo la posibilidad de implementar nuevas aplicaciones *Master/Worker* simplemente añadiendo los componentes necesarios. Estos nuevos componentes sólo se tienen que ocupar de la funcionalidad específica del problema que resuelven.

El diseño es adecuado no sólo porque permite la reutilización de la funcionalidad básica del *framework*, sino porque además, al mantener concentrada la funcionalidad común en el componente `GenericMaster`, se pueden implementar distintas estrategias de asignación de trabajos a *Workers*, distintas estrategias de tolerancia a fallos (en el caso de que algún trabajador falle, etc.).

Aunque adecuado, este diseño de componentes no sería factible sin una infraestructura de soporte en tiempo de ejecución. En este artículo utilizamos CORBA–$\mathcal{LC}$.

III. CORBA Lightweight Components, CORBA–$\mathcal{LC}$

CORBA Lightweight Components es un modelo de componentes distribuido basado en CORBA. De ésta hereda su soporte para múltiples lenguajes de programación, y para ejecutarse en y comunicarse con distintas arquitecturas y sistemas operativos.

En CORBA–$\mathcal{LC}$, los «componentes» son unidades binarias intercambiables que especifican sus requisitos y ofrecimientos en forma de puertos. Así, los componentes ofrecen a la vez una visión de *tipo* (definiendo los interfaces y operaciones), así como de *implementación* lista para ser usada en cualquier aplicación. Ambas visiones del componente están descritas utilizando XML, lo que permite que los componentes sean completamente autodescriptivos, y por ello manejados automáticamente por servidores y repositorios de componentes. La Figura 1 muestra la definición de características externas del componente, mientras que la Figura 2 muestra la definición de características del paquete binario, como son la librería dinámica que se cargará al activar el componente, los ficheros con definiciones IDL de CORBA, el ORB soportado, etc.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<component>
   <repoid>IDL:CORBALC/MasterWorker/GenericMaster:1.0<
/repoid>

   <factory>
      <option>keyless< /option>
   < /factory>

   <ports>
      <provides>
         IDL:CORBALC/MasterWorker/JobAcceptor:1.0
      < /provides>

      <uses>IDL:CORBALC/MasterWorker/Master:1.0< /uses>
      <uses>IDL:CORBALC/MasterWorker/Worker:1.0< /uses>
   < /ports>
< /component>
```

Fig. 1. Definición XML del componente `GenericMaster`.

Además, CORBA–$\mathcal{LC}$ ofrece un modelo de componentes «*distribuido*»: también tiene en consideración qué componentes, qué instancias de componentes residen en cada «*host*» que forma parte del sistema, y cómo estas instancias están conectadas entre ellas

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE package SYSTEM "package.dtd">

<package>
  <authors>
    <author>
      <name>Diego Sevilla< /name>
      <email>dsevilla@ditec.um.es< /email>
    < /author>
    <author>
      <name>José Antonio Pérez Sánchez< /name>
      <email>japs2@alu.um.es< /email>
    < /author>
  < /authors>
  <props>
    <idl>GenericMaster.idl< /idl>
  < /props>
  <files>
    <file>
      <filename>GenericMaster.so< /filename>
      <system>
        <arch>x86< /arch>
        <os>
          <name>Linux< /name>
          <version>2.4< /version>
        < /os>
      < /system>
      <orb>
        <name>ORBACUS_ORB< /name>
        <version>4.1.0< /version>
      < /orb>
    < /file>
  < /files>
< /package>
```

Fig. 2. Definición XML del paquete binario del componente `GenericMaster`.

para formar aplicaciones. En concreto, el modelo de red que ofrece CORBA–$\mathcal{LC}$:

- Mantiene constancia del conjunto de componentes instalados en cada *host* («nodo») del sistema, con lo que se libera al usuario de tener que manejar el conjunto de componentes y versiones necesarias en cada *host*.
- Permite que los componentes, al iniciar su ejecución, pidan a la red nuevos componentes que necesitan para completar su ejecución (esto es, permite que la conexión entre componentes se haga de forma dinámica y en tiempo de ejecución). Esto, al contrario de otros modelos como el modelo de componentes de CORBA (CCM) [9], [10] o *Enterprise JavaBeans* (EJB) [11] que obligan a que las conexiones entre componentes estén establecidas de antemano en documentos fijos, permite que las aplicaciones se adapten a los recursos disponibles en cada momento. Así, por ejemplo, CORBA–$\mathcal{LC}$ permite que se seleccionen en un momento dado los componentes residentes en aquellas máquinas con menos carga, o de un determinado tipo de hardware, etc.
- Además, CORBA–$\mathcal{LC}$ ofrece transparencia local/remota y posibilidad de instalar componentes bajo demanda. Así, cuando un componente necesita a otros para su ejecución, los pide a la infraestructura de CORBA–$\mathcal{LC}$, que puede decidir entre ejecutar los componentes requeridos de forma remota en otros *hosts* o traer el componente desde otro *host* remoto, instalarlo localmente y ponerlo a ejecutar. Los componentes también pueden exponer requisitos más específicos, como que necesitan una instancia de un componente ejecutándose en cada *host* posible del sistema, etc.
- Finalmente, CORBA–$\mathcal{LC}$ mantiene continuamente un conjunto de nodos conectados, manteniendo una coherencia «débil» de red [12]. Esto permite una tolerancia a fallos en los distintos nodos que componen el sistema, permitiendo conexiones y desconexiones en caliente.

IV. Evaluación de la arquitectura

Cada vez que un cliente quiere ejecutar una aplicación *Master/Worker* particular para resolver un problema, contacta con la infraestructura de CORBA–$\mathcal{LC}$ para pedirle la instanciación de un componente `GenericMaster`, indicándole además el identificador de repositorio CORBA (*Repository Id*)[1] del problema específico a resolver. El cliente también le envía los datos a resolver codificados como el tipo genérico `any` de CORBA.

Una vez que el `GenericMaster` conoce qué componentes utilizará para el cálculo, puede pasar a pedir CORBA–$\mathcal{LC}$ las instancias necesarias: el `Master` y tantas instancias de componentes `Worker` como sea posible en nodos diferentes. Para ello, la infraestructura puede determinar si es más productivo utilizar los *Workers* que sabe instalados en distintos nodos o bien iniciar un proceso de «difusión» de los distintos componentes a los nodos. Como CORBA–$\mathcal{LC}$ permite empaquetar en el mismo binario librerías dinámicas para distintas arquitecturas o distintos ambientes de programación como Java, o lenguajes de *script*, las posibilidades de utilizar más nodos aumentan. Además, la comunicación se realiza a través de CORBA, por lo que se pueden integrar sin problemas todas estos ambientes de programación.

El proceso *Master/Worker* propiamente dicho lo realiza el `GenericMaster` pidiéndole al `Master` particular las tareas a realizar y enviándoselas a los `Worker` para su realización, siguiendo los interfaces de la Figura 7 (p. 6).

A. Evaluación de Prestaciones

Para evaluar la eficiencia del «*framework*», implementamos dos problemas *Master/Worker* como se muestra en la Figura 6, una multiplicación matriz-vector y un cálculo distribuido de un fractal. Estas dos aplicaciones son típicamente utilizadas bajo el paradigma *Master/Worker*.

Ambos problemas se comportan diferente con respecto al *speedup*, como era de esperar, ya que el problema de la multiplicación tiene un bajo *ratio* computación/comunicación y no escala bien debido a las exigencias de comunicación y el poco cálculo realizado en cada elemento (una simple multiplicación). Sin embargo, en el cálculo del fractal, cada *Worker* debe realizar más trabajo, sobre todo si se quiere calcular el fractal con bastante precisión. Aún así, el *ratio* computación/comunicación sigue siendo bajo, ya que cada trabajador debe retornar un conjunto de valores para cada punto.

En las pruebas utilizamos un *cluster* heterogéneo que incluía ordenadores en la Universidad de Murcia: dos PIII a 1.0GHz conectados por una red Ethernet

[1] Un «*Repository Id*» representa en CORBA la definición de un interface. En CORBA–$\mathcal{LC}$ también puede representar la definición de un componente.

de 100Mbps (números 1 y 2 en la gráfica), un Athlon XP a 1.8GHz (3), un PIII a 550MHz (4), un PIV a 1.5GHz (5) (estos tres últimos conectados por una red de 10Mbps); y tres ordenadores conectados por Internet (Universidad de Indiana, EEUU): un PIII a 450MHz (6), y dos PIV a 1.5GHz (7 y 8). Estas pruebas se realizaron durante el uso normal de estos ordenadores, y a través del tráfico normal de Internet.

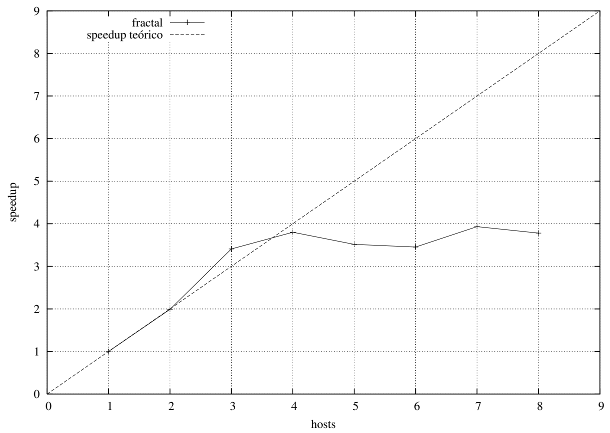

Fig. 3. Speedup para el cálculo del fractal.

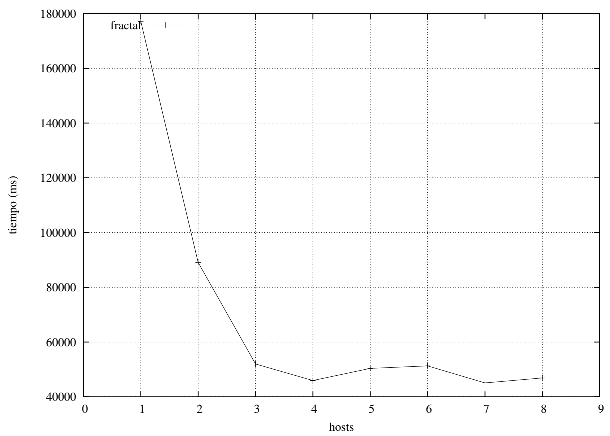

Fig. 4. Tiempos de ejecución para el cálculo del fractal.

Las gráficas de las figuras 3 y 4 muestran los resultados de *speedup* y tiempos de ejecución respectivamente para el cálculo del fractal. El primer punto a resaltar de las gráficas es el *speedup super-lineal* que se consigue con 3 *Workers*. Esto es debido a que el tiempo secuencial se midió con un *host* de 1.0GHz y el *host* 3 es de 1.8GHz. Aunque se podría argüir en contra de esta decisión, la hemos tomado debido a que, en nuestra opinión, seleccionar el nodo más rápido como caso base para calcular la ganancia de *speedup* es una solución «igual de aceptable» que seleccionar el nodo más lento, ya que todo depende de qué se quiera medir. En nuestro caso, estábamos interesados en medir hasta qué punto el *framework* era lo suficientemente flexible como para admitir y utilizar eficientemente tanto nodos con alta velocidad de proceso como nodos lentos. La Figura 3 muestra que al añadir un recurso de mayor poder de computación, el *framework* es capaz de «absorberlo» para redundar en un mejor tiempo de proceso. Es por esto que tomamos el tiempo secuencial de un nodo «medio». Además, al añadir el nodo 4, el *speedup* vuelve a estar muy cerca del lineal (aunque por debajo) aún integrando un recurso de menor capacidad de proceso (550MHz).

Los parámetros elegidos para el problema hacen que no escale bien a partir de cuatro *hosts*. Sin embargo, la Figura 4 indica que a pesar de que seguir añadiendo más *hosts*, incluso a través de Internet, algunos con menor capacidad de proceso (*hosts* 6, 7 y 8), el tiempo total de cálculo de la aplicación no aumenta, sino que se mantiene relativamente constante. Esto da una perfecta idea de la escalabilidad y adaptabilidad del *framework*, que es capaz no sólo e integrar *hosts* con diferentes velocidades a través de una red local, sino además de integrar recursos (*hosts*) a través de Internet.

Por último, medimos la sobrecarga que introduce la distribución automática de componentes sobre la red. En promedio, para el componente `FractalWorker` (194KBytes), la instalación se realiza en 130ms a través de la red local, lo cual es un tiempo muy razonable.

B. Tolerancia a Fallos

Además de las características de tolerancia a fallos de red que ofrece CORBA–$\mathcal{LC}$, expuestas en la sección III, el *framework* también ofrece un nivel de tolerancia a fallos ante posibles *hosts* que fallen o se desconecten en el momento en el que estaban haciendo alguna computación.

El objetivo de la tolerancia a fallos en un sistema distribuido es permitir una «degradación sostenible» a medida que van fallando recursos del sistema. Sin embargo, este soporte para tolerancia a fallos, por lo general, afecta al rendimiento general de la aplicación, ya que el *framework* debe llevar cuenta de qué *hosts* han fallado, se han desconectado, o incluso, tardan mucho en responder.

La tolerancia a fallos en el *framework* la implementa el componente `GenericMaster`. Para cada *Worker* utilizado, se crea un hilo de ejecución o «*thread*» encargado del diálogo con ese *Worker*. Por otro lado, cada trabajo puede estar uno de los tres estados siguientes: **Pendiente** (el trabajo no ha sido asignado), **Asignado** (el trabajo ha sido asignado a un *Worker* que todavía no ha devuelto el resultado) y **Terminado** (el trabajo se ha terminado).

El algoritmo que sigue cada *thread* es el siguiente: Busca el siguiente trabajo a enviar a su *Worker*. De entre todos los trabajos que están en estado *pendiente*, selecciona uno, le cambia el estado a *pendiente* y lo envía al *Worker* para su resolución. Cuando recibe el resultado, le asigna un estado de *terminado*. Si no hay ningún trabajo en *pendiente*, pero hay trabajos en *asignado*, el *thread* selecciona uno de esos trabajos y lo envía al *Worker*, actualizando a *terminado* cuando recibe la respuesta. Las respuestas duplicadas del resultado de un trabajo se ignoran.

Este sencillo algoritmo permite, por un lado, que los trabajos emprendidos por *Workers* que se hayan desconectado o hayan caído se reasignen a otros *Workers*, y por otro, que los *hosts* con mayor poder de computación sean capaces de adquirir más trabajos que los *hosts* más lentos, dando una mejor utilización de los recursos de la red.

Para determinar la eficiencia de este soporte de tolerancia a fallos, realizamos pruebas con varios *hosts* haciendo que uno de ellos (el nodo 1, de 1.0GHz) nunca respondiera. Los resultados de *speedup* para 4 *hosts* se muestran en la Figura 5.

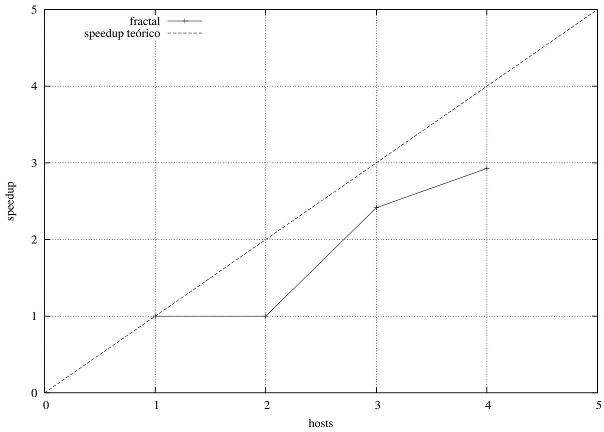

Fig. 5. Speedup para el cálculo del fractal (activada la tolerancia a fallos, 1 *host* siempre falla).

La Figura muestra la validez de la aproximación, ya que los valores de *speedup* son muy similares a los de la gráfica sin fallo de ningún componente (similares al *speedup* de $n-1$ nodos).

V. Conclusión y Trabajos Futuros

Aunque esta primera versión del *framework Master/Worker* ha demostrado ser adecuada para implementar eficientemente este tipo de aplicaciones, también nos ha servido para identificar una serie de vías futuras y puntos a mejorar y a investigar. Entre las mejoras posibles se encuentran:

- Actualmente no se soporta una comunicación directa entre los *Workers*. Sería interesante permitir una comunicación y sincronización entre los distintos trabajadores. Aunque esta comunicación se puede hacer conectando los distintos componentes, sería interesante proporcionarla desde el *framework*.
- El hecho de que el `GenericMaster` controle el conjunto de *Workers* hace que la comunicación entre éstos y el `Master` se realice de forma indirecta a través del `GenericMaster`. Esto aumenta el número de copias de los datos. Sería interesante modificar el *framework* para hacerlo más eficiente en cuanto a copias, permitiendo así tamaños del problema mayores, y una mayor escalabilidad.
- Actualmente, se utiliza un *thread* para contactar con cada *Worker*. Para ejemplos con cientos de *hosts*, esto no es un problema. Sin embargo, cuando se utilizan miles de ellos, la escalabilidad con el número de *threads* puede ser importante. Sería interesante desarrollar mecanismos de *thread pool*, etc.
- Sería interesante también que los *Workers* utilizasen *threads*, para así poder aumentar el solapamiento comunicación/computación.
- Aunque el *framework* ofrece tolerancia a fallos, una posible mejora sería añadirle soporte de «*checkpointing*», no excesivamente complicado. Dada la naturaleza de los componentes del *framework*, bastaría con que éstos implementaran un hipotético interface `Checkpoint`.
- CORBA–$\mathcal{LC}$ ofrece un modelo de componentes con mejoras sobre el modelo de componentes de CORBA (CCM) [10], y permite probar nuevas tecnologías como los mensajes asíncronos (muy importantes para este tipo de aplicaciones de red), el servicio de tolerancia a fallos y el servicio de grupos. Estos nuevos servicios no están soportados por todos los ORBs, por lo que desarrollamos un entorno para permitir el uso de diferentes ORBs llamado *ORBframe* [13].
- Igualmente, CORBA–$\mathcal{LC}$ soporta eventos, aunque actualmente no están implementados en el prototipo. Los eventos permiten conectar en caliente nuevos componentes, de particular importancia para aplicaciones gráficas de *front-end* y para monitorización de aplicaciones en ejecución. Un punto de investigación muy interesante es añadir soporte de eventos.

Referencias

[1] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.

[2] G. Shao, F. Berman, and R. Wolski, "Master/Slave Computing on the Grid," in *Proceedings of the 2000 Heterogeneous Computing Workshop*, 2000, http://gcl.ucsd.edu/amwat/.

[3] J. Linderoth, S. Kulkarni, J-P. Goux, and M. Yoder, "An Enabling Framework for Master-Worker Applications on the Computational Grid," in *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, Pittsburgh, PN, August 2000, pp. 43–50, http://www.cs.wisc.edu/condor/mw/.

[4] *distributed.net*, 2002, http://www.distributed.net.

[5] *SETI@home*, 2002, http://setiathome.ssl.berkeley.edu/index.html.

[6] *MPI Software Technology, INC*, 2001, http://www.mpi-softtech.com.

[7] D. Sevilla, J. M. García, and A. Gómez, "CORBA Lightweight Components: A Model for Distributed Component-Based Heterogeneous Computation," in *EUROPAR'2001*, Manchester, UK, August 2001.

[8] D. Sevilla, J. M. García, and A. Gómez, "Design and Implementation Requirements for CORBA Lightweight Components," in *Metacomputing Systems and Applications Workshop (MSA'01)*, Valencia, Spain, September 2001.

[9] D. Sevilla, *The CORBA & CORBA Component Model (CCM) Page*, 2002, http://www.ditec.um.es/~dsevilla/ccm/, 2002.

[10] Object Management Group, *CORBA Component Model*, 1999, OMG Document ptc/99-10-04.

[11] SUN Microsystems, *Enterprise Java Beans specification*, 1.1 edition, December 1999, http://java.sun.com/products/ejb/index.html.

[12] K. Birman and T. Joseph, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 47–76, 1987.

[13] D. Sevilla and J. A. Pérez, *ORBframe*, http://sf.net/projects/orbframe.

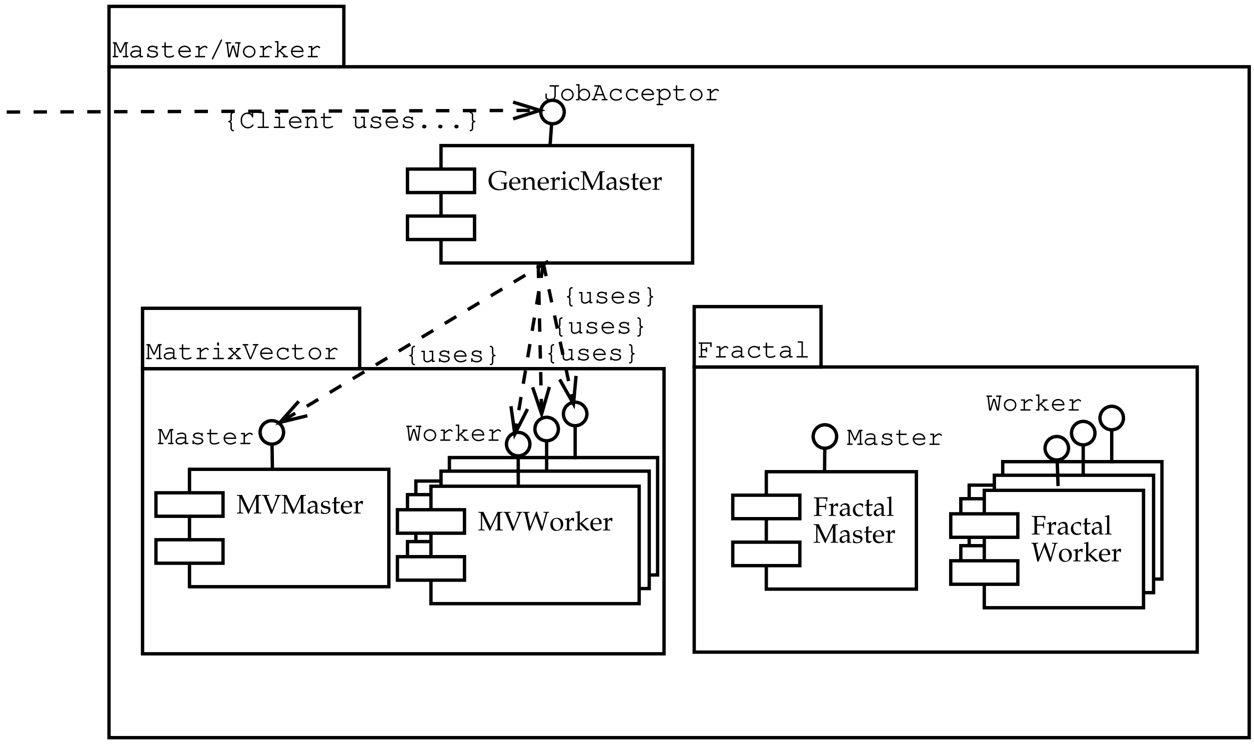

Fig. 6. Diagrama UML de componentes del *framework Master/Worker*.

interface Master : CORBALC::lcIface
{
    //*Retorna el repoId del componente Worker asociado*
    string getWorkerId ();

    //*Indica al Master cuántos Workers hay disponibles*
    //*@param howMany número de Workers disponibles*
    //*@param jobs número de trabajos a realizar*
    void availableWorkers (**in unsigned long** howMany, **out unsigned long** jobs)
        raises (AlreadyConfigured);

    //*Pedir un trabajo*
    //*@param jobId identificador del trabajo (en 0..jobs-1)*
    JobUnit getJob (**in unsigned long** jobId) **raises** (JobIdOutOfRange);

    //*Establece la solución de un trabajo y devuelve true cuando se han finalizado todos los trabajos*
    boolean finalizeJob (**in** JobUnit job) **raises** (AlreadyDoneJob, JobIdOutOfRange);

    //*Retorna el resultado final*
    any getResult () **raises** (NotCompletedYet);
};

interface Worker : CORBALC::lcIface
{
    JobUnit doWork (**in** JobUnit jobData);
};

Fig. 7. Interfaces `Master` y `Worker`.