POLÍTICAS DINÁMICAS PROACTIVAS DE BALANCEO DE CARGA EN J2EE

*Jorge Humberto Arias Bedoya**

Resumen: las implantaciones de alta disponibilidad provistas hoy en día por los fabricantes de contenedores J2EE (IBM, Bea, Sun, Oracle, JBoss) se estructuran alrededor de dos conceptos básicos: la replicación de servicios en un *cluster* y el balanceo de carga implantado sobre políticas estáticas de distribución. Las políticas estáticas no tienen en cuenta el estado interno de ejecución del *cluster* para llevar a cabo sus decisiones de balanceo y distribución. Por tanto, no evitan la presencia de *hotspots* (servidores sobrecargados). En este artículo se presenta una solución a este problema basada en un modelo de implantación de políticas de balanceo de carga dinámicas proactivas. Estas políticas tienen en cuenta el estado interno de cada miembro del *cluster*, como por ejemplo, el número de *threads*, %CPU disponible, el número de transacciones en ejecución o la disponibilidad de memoria. Conocer estas variables permite realizar un balanceo y distribución de carga más equilibrado y eficiente que garantice el uso óptimo de los recursos del *cluster* y ofrezca los niveles de disponibilidad y escalabilidad requeridos por una solución empresarial compuesta de servicios del negocio replicados. Aun cuando la implantación se realizó y probó sobre el servidor de aplicaciones JBoss, el modelo es lo suficientemente general como para que pueda ser usado en otras implantaciones J2EE.

Palabras clave: balanceo de carga, *clustering, proxies* inteligentes, servidores de aplicaciones, J2EE, políticas proactivas de balanceo de carga, JBoss.

Abstract: The high availability implementations provided by J2EE container manufacturers are based on two basic concepts: replication of services over a cluster and load balancing using static policies. Static policies do not use the internal execution state of each machine in

* *Ingeniero de sistemas y magíster en ingeniería de sistemas y computación, Universidad de los Andes. Profesor asistente, Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana. Correo electrónico: jo-arias@javeriana.edu.co.*

the cluster to decide which computer will be employed by the load balancing system. This feature allows the presence of hotspots - overloaded server. In this paper we present a solution to this problem based on a model of dynamic proactive policies. A dynamic policy uses the cluster's internal execution state to make the decisions about load balancing. The execution state considered by the proactive policy consists of variables related to the hardware platform, operating system and J2EE platform where the business application is deployed. Some of the variables considered are: threads loaded, %CPU available, business transaction per second, used memory among others. The load balancing system can take advantage of knowing these variables to make more equilibrated and efficient decisions. The solution is general and could be implemented on any certified J2EE implementation. However the model was implemented on JBoss Application Server 3.0.0.

Key words: Load balacing, clustering, high availability, smart stub, proxies, application server, J2EE, java, proactive policies, JBoss.

1. Introducción

El reciente crecimiento de sistemas transaccionales migrados de un esquema *host* o cliente/servidor a un esquema multi-capas basado en *middleware* y con soporte *web* ha despertado un fuerte interés por el diseño e implantación de plataformas de componentes distribuidos [Allamaraju et al., 2001][Roman et al., 2002]. Estas plataformas ofrecen soporte al manejo de los aspectos no funcionales de la aplicación, como por ejemplo: manejo transaccional, manejo multi-hilos, escalabilidad, alta disponibilidad, ejecución distribuida e integración entre otros importantes servicios [Roman et al., 2002]. Dichas plataformas proveen una infraestructura robusta para implantar aplicaciones empresariales críticas y vitales para el funcionamiento de cualquier negocio, caracterizadas por altas cargas de trabajo, acceso simultáneo y rápidos tiempos de respuesta. En esta situación es importante considerar el diseño de aplicaciones escalables y de alta disponibilidad que puedan soportar cargas aceleradas de trabajo los 365 días del año. Escalabilidad y alta disponibilidad sólo se logran con una adecuada e inteligente implantación de *clustering* y balanceo de carga [Bea, Scalability, 2001] [Labourey y Burke, Clustering, 2002] [Labourey y Burke, Jboss 3.0, 2002].

J2EE (Java 2 Enterprise Edition) es la especificación propuesta por Sun Microsystems para desarrollo de aplicaciones multi-nivel [Sun, 2002] y existen en el mercado varias implantaciones de la especificación: IBM, Bea, Sun, Oracle, JBoss. Las soluciones de alta disponibilidad provistas por estos fabricantes de contenedores J2EE se estructuran alrededor de una plataforma de *clustering* que permite la replicación de servicios del negocio encapsulados como componentes Enterprise Java Beans (EJB) y de un sistema de balanceo de carga basado en políticas estáticas.

Las políticas de balanceo de carga ofrecidas por las plataformas J2EE existentes no tienen en cuenta el estado interno de ejecución de cada

miembro del *cluster* para llevar a cabo de una manera óptima y eficiente la distribución y el balanceo de carga sobre los servidores que contienen los servicios del negocio replicados. Es decir, se basan en políticas estáticas de distribución de carga que emplean heurísticas o métodos aleatorios y cíclicos como Round Robin, First Available y Random [Bea, Cluster, 2001][Jewell, 2002]]][Labourey y Burke, Clustering, 2002] [Labourey y Burke, Jboss 3.0, 2002]. Es así como este tipo de políticas no garantiza el empleo óptimo de los recursos replicados causando *hotspots*. Un *hotspot* es una situación en la que un servidor particular del *cluster* puede saturarse de requerimientos y presentar los más altos niveles de carga de trabajo mientras los demás servidores permanecen con índices bajos de carga.

En este artículo se presenta un modelo de implantación de políticas de balanceo de carga dinámicas proactivas en J2EE. Las políticas dinámicas tienen en cuenta el estado interno de ejecución de cada miembro del *cluster* desde varias dimensiones o contextos de ejecución como: sistema operativo (número de *threads*, páginas de memoria, tamaño de *buffers)*, recursos de *hardware* (%CPU disponible, número de procesadores, capacidad de memoria principal y secundaría) y plataforma operacional J2EE (número de transacciones en ejecución, disponibilidad de memoria en *heap* y *stack*, requerimientos en espera y ejecución). Este estado interno de ejecución se tiene en cuenta en la política para efectuar una distribución y balanceo de carga más equilibrado y eficiente, logrando de esta manera una utilización óptima de los recursos del *cluster*, ya que se puede llegar a un estado de equilibrio perfecto en el cual cada miembro del *cluster* mantiene un mismo nivel de carga de trabajo. Adicionalmente, el hecho de ser proactivas garantiza que el tiempo que toma llevar a cabo un proceso de distribución y balanceo de carga es mínimo, ya que con antelación a un requerimiento o solicitud de uno de los servicios ofrecidos por el *cluster*, el sistema sabe cuál es el miembro del *cluster* más apto para dar respuesta a dicha solicitud.

La solución propuesta se compone de dos partes. Primero, del lado del cliente se utilizan *proxies* inteligentes, configurables en el momento de hacer el *deployment* del componente y de un sistema de escuchas de eventos de estado publicados sobre un canal IP Multicasting[1] empleando un *framework* de comunicación en grupo llamado Java Groups [Ban, 2001]. Esta infraestructura es totalmente transparente para el desarrollador de una aplicación cliente que hace uso de servicios remotos del negocio funcionales y no funcionales replicados. Segundo, del lado servidor se utiliza un modelo de replicación y publicación de eventos a un canal IP Multicasting también empleando JavaGroups; estos eventos encapsulan el estado de ejecución de cada uno de los miembros del *cluster*. Adicionalmente, la solución del lado servidor emplea un proceso de introspección sobre el contenedor J2EE

[1] *Direccionamiento IP posible en un rango de direcciones que inicia en 224.0.0.0 y termina en 239.255.255.255.*

donde se ejecuta la aplicación. Este proceso permite obtener en tiempo de ejecución información relacionada con variables del sistema operativo, plataforma *hardware* y sistema operacional J2EE. Estas variables se obtienen dinámicamente del entorno cada cierto intervalo de tiempo. El proceso de introspección se llevó a cabo empleando los servicios de conectividad a la *virtual machine* ofrecidos por la infraestructura de depuración de la edición estándar de Java 2 (J2SE), llamada Java Protocol Debugger Architecture (JPDA)[Java Guide, 2002]. El modelo de políticas dinámicas proactivas descrito se puso en funcionamiento sobre JBoss 3.x [Fleury y Stark, 2002] [Stark, 2002], para lo cual fue necesario extender el *framework* de alta disponibilidad (Jboss High Availabity Framework [Labourey y Burke, Clustering, 2002]) ofrecido por esta implantación J2EE.

Este artículo se encuentra organizado de la siguiente forma: en la primera parte se enuncian las diversas técnicas de balanceo de carga existentes y la implantación J2EE de JBoss 3.0.x y su *framework* de alta disponibilidad. En la segunda parte se explica el diseño y arquitectura de la solución construida para el soporte de políticas dinámicas proactivas de balanceo de carga en J2EE y la extensión realizada al *framework* de alta disponibilidad propuesto en el servidor de aplicaciones JBoss 3.0.x. Por último, se presentan las conclusiones del trabajo y se esboza un conjunto de ideas relacionadas que podrían ser llevadas a cabo alrededor de este proyecto.

2. TÉCNICAS DE BALANCEO DE CARGA

2.1 DEFINICIÓN

Una política de balanceo de carga es la implantación de un algoritmo que determina el miembro del *cluster* más apto para recibir y procesar un requerimiento realizado por una aplicación cliente. La decisión sobre cuál miembro del *cluster* seleccionar se hace utilizando métodos heurísticos o información de estado del entorno de ejecución dentro del cual residen y se ejecutan los servicios del negocio replicados. Dependiendo de cómo se tome la decisión las políticas de balanceo de carga se clasifican en dos tipos: políticas o técnicas estáticas o dinámicas de balanceo de carga.

2.2. TÉCNICAS ESTÁTICAS

Las técnicas estáticas son aquellas que emplean métodos heurísticos para determinar el miembro o servidor del *cluster* sobre el cual dirigir una solicitud o procesamiento de una tarea específica. Estos métodos no tienen en cuenta el estado interno de ejecución de cada uno de los miembros del *cluster* para calcular y determinar el servidor más apto para llevar a cabo la ejecución de un requerimiento o solicitud. Los métodos heurísticos más comunes para el diseño e implanta-

ción de técnicas estáticas son: *Round Robin* (RR) y *Random*. En el método *Round Robin*, a partir de una lista de miembros del *cluster* se selecciona de manera cíclica uno de ellos. Ante la llegada de una primera solicitud selecciona el primer miembro de la lista; la segunda solicitud selecciona el segundo miembro y así sucesivamente hasta el final de la lista, momento en el que volverá a asignar el primer servidor de la lista. En el método *Random* se emplea una lista de los miembros del *cluster*, se calcula de manera aleatoria un valor entero entre uno y el tamaño del *cluster*; el valor calculado indica la posición del servidor en la lista de miembros del *cluster* responsable de recibir y procesar la solicitud o tarea.

Las técnicas estáticas no garantizan un óptimo aprovechamiento de los recursos del *cluster* ya que dan lugar a situaciones que afectan negativamente el desempeño de un sistema replicado. Por ejemplo, al emplear métodos *Random* es posible que todas las solicitudes entrantes sean redireccionadas al mismo servidor del *cluster*, presentándose una situación de desbalanceo y saturación conocida como *hotspot* [Harchol-Balter, 2002][Labourey y Burke, Clustering, 2002]. Cuando se emplean técnicas tipo *Round Robin* se presentan situaciones en las cuales ante la presencia de diferentes tipos de solicitudes (en algunas situaciones estas solicitudes se clasifican por su complejidad en cuanto a duración en tiempo, recursos de *software* y de *hardware* necesarios para ser procesadas), las de menor complejidad siempre sean direccionadas al mismo servidor o conjunto de servidores del *cluster* y las de mayor complejidad al mismo servidor o conjunto de servidores, presentándose así una situación de balanceo en número de tareas pero desbalanceo en el uso de recursos lógicos y físicos.

2.3 Técnicas Dinámicas

A partir de estos inconvenientes surge la necesidad de tener técnicas de balanceo de carga que garanticen el empleo óptimo de los recursos físicos y lógicos de *cluster* y mantengan este último en un estado de verdadero equilibrio. Es así como nacen las técnicas dinámicas de distribución y balanceo de carga que tienen en cuenta el estado interno de ejecución de cada uno de los miembros del *cluster* para calcular y determinar de una manera más inteligente el miembro del *cluster* más apto para llevar a cabo la ejecución de una tarea específica.

El estado interno de ejecución tenido en cuenta por la política dinámica se compone de tres tipos de variables: plataforma *hardware*, sistema operativo y plataforma J2EE. Las variables de la plataforma *hardware* están asociadas a las características *hardware* de cada uno de los miembros del *cluster*, por ejemplo: número de procesadores, capacidad en memoria principal, porcentaje de CPU en uso y disponible, entre otros indicadores de ejecución propios del sistema. Las variables del sistema operativo están relacionadas con la ejecución del sistema operacional en cada servidor miembro del *cluster*, por ejemplo,

número de procesos en ejecución y en espera programados por el sistema operativo, número de *threads*, memoria principal usada y libre, número de páginas de memoria asignadas y no asignadas, entre otras. Por último, las variables relacionadas con la plataforma J2EE donde se ejecuta la aplicación replicada, por ejemplo, número de transacciones o solicitudes en estado de espera, ejecución y terminadas, transacciones y solicitudes procesadas en una unidad de tiempo, tamaño de los *pools* de *threads* y objetos dentro del contenedor.

La información provista por las variables descritas anteriormente se tiene en cuenta por la política de balanceo de carga para determinar cuál es la máquina más adecuada para llevar a cabo la ejecución de una determinada tarea. Conocer el valor de estas variables permite garantizar el óptimo empleo de los recursos del *cluster*, eliminando la presencia de *hotspots* y manteniendo cada uno de los miembros del *cluster* con la misma carga real de trabajo. Un aspecto importante que debe ser considerado en cualquier solución de balanceo de carga es que en un entorno de ejecución del mundo real existen diferentes tipos de solicitudes o requerimientos, situación en la que el sistema de balanceo debe tener la capacidad para determinar cuál es la máquina que puede responder más efectivamente según el tipo de solicitud o requerimiento efectuado [Bea, Scalability, 2001][Harchol-Balter, 2002]. No es lo mismo que una solicitud compleja sea dirigida a un miembro del *cluster* con poco poder de procesamiento y capacidad de memoria principal, a que ésta sea dirigida a un servidor con una considerable capacidad de procesamiento. Indudablemente el desempeño percibido por una aplicación cliente de los servicios replicados es diferente en ambos casos, siendo mejor en el último caso.

Ahora, la pregunta que surge es: ¿cómo el sistema de balanceo de carga que emplea técnicas o políticas dinámicas obtiene el estado de ejecución de cada uno de los miembros del *cluster*? Existen dos modelos o maneras en que un sistema de balanceo puede llegar a acceder a esta información.

2.3.1 Técnicas dinámicas reactivas

El primero de ellos es un modelo reactivo en el cual, ante la llegada de una solicitud o requerimiento de balanceo de carga el sistema enviará mensajes de *broadcast* a cada miembro del *cluster* solicitándole su estado interno. Una vez éstos respondan la política dinámica de balanceo de carga se procederá a calcular cuál servidor del *cluster* es el más apto para recibir y procesar la solicitud. Este modelo tiene la desventaja de ser una solución lenta y poco eficiente ya que el proceso de recolección de la información y su posterior procesamiento puede tardar bastante tiempo. Incluso puede presentarse el caso en que el tiempo de ejecución de una tarea sea bastante inferior al tiempo requerido para establecer el servidor del *cluster* más apto para ejecutarla.

2.3.2 Técnicas dinámicas proactivas

El segundo es el modelo proactivo en el que, ante la llegada de una solicitud de balanceo de carga, la política dinámica sabe con anterioridad cuál es el servidor del *cluster* más apto y con la menor carga de trabajo sobre el cual dirigir la solicitud o tarea. De esta manera, se eliminan los tiempos asociados a la recolección de la información de estado y al cálculo de la decisión, permitiendo tener una solución de balanceo más rápida, inteligente y eficiente. Este grado de proactividad se logra solicitando cada cierto intervalo de tiempo el estado de ejecución de cada miembro del *cluster* y, con base en éste, calcular y determinar el servidor más apto y con la menor carga real de trabajo.

3. Servidor de aplicaciones JBoss

El modelo de políticas dinámicas proactivas descrito en este artículo se implementó y probó sobre el servidor de aplicaciones JBoss 3.0.0. A continuación se expone la arquitectura conceptual de este servidor de aplicaciones y el *framework* de alta disponibilidad ofrecido por esta implantación J2EE.

3.1 Generalidades de JBoss

El servidor de aplicaciones JBoss es una implantación gratuita, *open source*, de la especificación J2EE, que se distribuye bajo la licencia de código abierto *Lesser General Public License* (LGPL) [Fleury y Stark, 2002] y es liderado directamente por una organización conocida como JBoss Group LLC [Stark y JBoss Group, 2002] e indirectamente por más de mil desarrolladores alrededor del mundo. La arquitectura interna de JBoss es altamente modular y se fundamenta en un diseño de *plug-ins* [Liu, 2002] [Stark y JBoss Group, 2002], los cuales son manejados e integrados por medio del estándar de la industria Java Management Extensions (JMX) [Java Extensions, 2002][Sullins y Whipple, 2002].

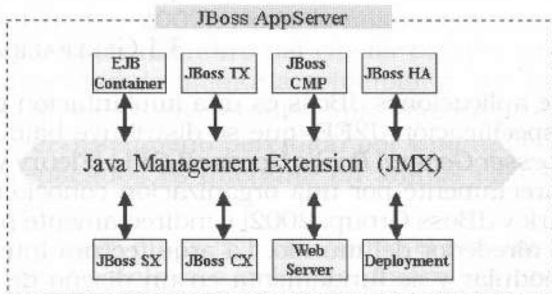
Como se enunció en el párrafo anterior, JBoss se estructura arquitectónicamente alrededor de varios módulos [Liu, 2002], los cuales se ejecutan de manera independiente y se integran por medio de JMX. Los principales módulos que componen el servidor de aplicaciones son:

- EJB *Container*: implantación de la especificación EJB 1.1 y 2.0. Sobre ésta se ejecutan los componentes del negocio EJB provistos por el proveedor de la aplicación J2EE.
- JbossTX: módulo que implementa la especificación JTA/JTS, ofreciendo los servicios necesarios para soportar transacciones locales y distribuidas sobre JBoss.
- *Deployment*: módulo responsable de llevar a cabo el *deployment* de un componente EJB, *web* o servicio sobre el servidor de aplicaciones.
- *JBoss Naming Service:* implantación *Java Naming Directory Interface* (JNDI) que forma el servicio de directorio y nombres de JBoss.

- *WebServer*: contenedor *web* que da soporte a los componentes *web* especificados por los APIs Java Servlets y Java Server Pages (JSP).
- JBoss CX: arquitectura que da soporte a la especificación JCA (Java Connector Architecture) empleada para la integración de componentes del negocio de una aplicación J2EE con una infraestructura de sistemas de información empresarial (Legacy Systems, CRM, ERP).
- JBoss Sx: *framework* de seguridad que soporta diversas implantaciones de seguridad, como por ejemplo modelo declarativo y programático propuesto por la especificación J2EE, especificación Java Authorization and Authetication Service (JAAS), entre otros.
- JBoss Message Queue (MQ): implantación del servicio de mensajería asincrónico ofrecido en JBoss.
- JBossCMP: implantación del servicio de persistencia manejada por el contenedor, ofrecida por JBoss
- JBoss High Availability (HA): *framework* de alta disponibilidad, diseñado alrededor de replicación de servicios y balanceo de carga.

La Figura 1 muestra la arquitectura conceptual descrita en el párrafo anterior.

Figura 1. Arquitectura conceptual de JBoss



A diferencia de la mayoría de implantaciones J2EE existentes, JBoss no emplea *stubs* o *proxies client-side* compilados a mano por el proveedor de la aplicación J2EE. Por el contrario, emplea *proxies* dinámicos, los cuales son creados dinámicamente en el momento que se hace *deployment* de la aplicación sobre el servidor de aplicaciones. Estos proxies dinámicos arquitectónicamente están diseñados sobre el patrón de diseño Proxy que se basa en la clase *java.lang.reflect Proxy* presente desde la versión JDK 1.3 [Blosser, 2000] y emplean introspección (*java reflection*) sobre las interfases remotas *Home* y *Remote* provista por el proveedor de la aplicación para generarse automáticamente.

3.2 *Framework* de alta disponibilidad, disponible en JBoss

La implantación *open source* de JBoss, desde su versión 3.0.0, incluye un *framework* de alta disponibilidad que se compone de una solución de *clustering* o replicación de servicios como componentes del negocio EJB, servicios JNDI, entre otros y una solución de balanceo de

carga que se estructura alrededor de políticas estáticas, empleando métodos heurísticos como *RoundRobin*, *Random* y *First Available* [Labourey y Burke, 2002].

El modelo de *clustering* y replicación implementado en el *framework* de alta disponibilidad de JBoss se apoya en una plataforma *open source* para el soporte de IP Multicasting confiable en Java, llamada JavaGroups [Ban, 2001]. Esta plataforma ofrece varios servicios para implementar soluciones de replicación de una manera simple y confiable. Uno de estos servicios es el *Channel* [Ban, 2001], el cual permite administrar la comunicación entre un conjunto de servidores, componentes o servicios agrupados bajo una dirección IP Multicasting. Entre las múltiples funcionalidades ofrecidas por el *Channel*, se encuentran:

- Almacenamiento de mensajes en una cola hasta que los receptores de éstos decidan recuperar o consumir dichos mensajes.
- Permitir la suscripción de procesos *listener* de mensajes, que pueden ser sincrónicos *Push* o asicrónicos *Pull*.
- Notificar a los miembros del grupo *IP Multicasting* el ingreso de un nuevo miembro o la salida de uno de ellos.
- Medio de comunicación sobre el que fluyen objetos del mismo *framework* conocidos como *Building Blocks* [Ban, 2001], los cuales permiten encapsular y manejar de una manera transparente objetos distribuidos replicados, como por ejemplo *Distributed Hashtable, RPCHandler y PullPushAdapter* [Ban, 2001].

Este *framework* emplea un concepto llamado *Partition* [Labourey y Burke, Clustering, 2002], que a bajo nivel es un objeto *JavaGroups Channel*. Las particiones agrupan servicios o componentes del negocio replicados a lo largo del *cluster*. Cada vez que uno de los componentes EJB o servicios replicados cambia su estado la partición es responsable de replicar el nuevo estado a los demás miembros o servicios de la partición.

La solución de balanceo de carga propuesta por este *framework* se estructura alrededor de *proxies* inteligentes de objetos remotos y servicios replicados. Estos *proxies* se fabrican de manera dinámica en JBoss en el momento de inicialización del servidor de aplicaciones o en el momento en que se hace *deployment* del componente EJB. Durante el proceso de fabricación de este tipo de *proxies* el contenedor suministra por medio de uno de los métodos constructores del *proxy* una lista de los miembros del *cluster* que alojan una réplica del componente EJB para quien se está fabricando el *proxy* y una instancia de la política estática de balanceo de carga. El nombre de la clase a partir de la cual se crea la instancia de la política se especifica en uno de los descriptores del componente EJB. Para el caso de JBoss se emplea el *jboss.xml* como descriptor para establecer configuraciones propias y avanzadas del servidor de aplicaciones.

Cada vez que se hace una invocación remota al componente EJB, la respuesta es empleada por el *framework* de alta disponibilidad para encapsular y enviar el nuevo estado de la partición a la instancia del

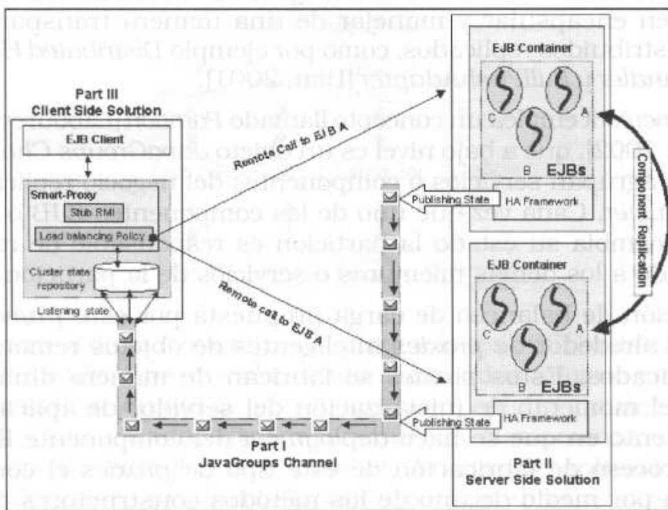
smart proxy residente en la aplicación cliente en caso de que la partición o la configuración del *cluster* haya cambiado, por ejemplo, debido al ingreso de nuevos miembros y al retiro de miembros existentes.

Como se mencionó, las políticas de balanceo de carga empleadas por JBoss son estáticas. Cuando una aplicación cliente hace una invocación remota, el *proxy* inteligente intercepta este llamado e inmediatamente solicita a la instancia de la política de balanceo de carga que contiene que seleccione un servidor sobre el cual despachar el llamado remoto. El balanceo de carga se lleva a cabo en la aplicación cliente dentro del *proxy* del objeto remoto que obtuvo vía JNDI.

4. Políticas dinámicas proactivas de balanceo de carga en J2EE

La solución de políticas dinámicas proactivas diseñada e implementada se estructura alrededor de una arquitectura compuesta por tres partes, tal como se muestra en la Figura 2.

Figura 2. Arquitectura del modelo de políticas dinámicas proactivas de balanceo de carga en J2EE



4.1 Canal *JavaGroups*

La primera parte de la arquitectura es un canal *JavaGroups*, en el que se encuentran registrados cada uno de los miembros del *cluster* y cada uno de los *proxies* inteligentes residentes en la aplicación-cliente. Este canal tiene como función principal servir de medio de replicación por el cual fluye información asociada al estado de ejecución interno de cada miembro del *cluster* y solicitudes de estado requeridas por el *proxy* inteligente. Este estado es obtenido de manera asincrónica y sincrónica por el *proxy* inteligente residente en la máquina cliente,

el cual emplea un componente preconstruido *(building block)* ofrecido por *JavaGroups*, llamado *PullPushAdapter* para observar asincrónicamente el canal y notificar de manera sincrónica el nuevo estado al *proxy*. Internamente, un canal administra un grupo de servicios o miembros agrupados bajo una dirección *IP Multicasting*. En caso de tener configuraciones en las cuales los *smart-stub* y los miembros del *cluster* se comuniquen por medio de una red TCP/IP que no soporte *IP Multicasting*, *JavaGroups* provee la funcionalidad de tener canales sobre protocolos orientados a la conexión sobre IP como TCP [Ban, 2001]. Adicionalmente, ofrece la funcionalidad de implementar canales que atraviesen *firewalls*, para lo cual se apoya en el diseño de túneles empleando TCPGOSSIP [Ban, 2001].

4.2 Solución *Server Side*

La segunda parte de la arquitectura se refiere a la solución *server side*. Ésta se encuentra diseñada alrededor de un proceso activo de notificación de estado y de un proceso de introspección al entorno de ejecución. El proceso activo es un *java thread* que se crea durante la fase de inicialización del servidor de aplicaciones y tiene asignadas varias responsabilidades: i) suscribirse al canal de replicación explicado anteriormente; ii) obtener vía JPDA cada cierto intervalo de tiempo variables de ejecución asociadas al *hardware*, al sistema operativo y a la plataforma J2EE donde se ejecuta dicho objeto activo, iii) comparar el último estado de ejecución publicado sobre el canal con el estado que acaba de obtener y procesar, si ambos son relativamente diferentes de acuerdo con un umbral de referencia, el nuevo estado es publicado sobre el canal; en caso contrario no se lleva a cabo ningún proceso de notificación; iv) por último, este *thread* debe actuar como un proceso servidor de consultas de estado, ya que en cualquier momento un *proxy* inteligente puede requerir el último estado de ejecución calculado por el servidor, situación en la cual el *thread* principal crea un *thread* hijo que se encarga de atender y resolver dicha solicitud de estado.

La solución *server side* posee una consola JMX[JX] que permite cambiar en cualquier instante vía HTML, *Single Network Management Protocol* (SNMP) o *Remote Method Protocol* (RMI) el intervalo de tiempo empleado por el *thread* principal para obtener un nuevo estado de ejecución del servidor, además de cambiar el umbral de referencia empleado para llevar a cabo el proceso de comparación a partir del cual se determina si el estado de ejecución recientemente obtenido y calculado ha cambiado lo suficiente respecto al anterior para que amerite ser replicado sobre el canal.

4.3 Solución *Client Side*

La última parte es la solución *client side*. Ésta se compone de los siguientes elementos: *proxy* inteligente *(smart-stub)*, instancia de la

política de balanceo de carga, repositorio compartido de estado de ejecución del *cluster* y proceso activo *listener* de eventos de estado.

El primero de los elementos enunciados se refiere a la instancia del *proxy* del componente EJB. Este *proxy* se fabrica de manera dinámica en el servidor de aplicaciones durante el *deployment* del componente o durante la fase de inicialización del contenedor. En el momento en que el contenedor J2EE fabrica el *proxy* cliente envía a la clase sobre la que actuará los siguientes parámetros:

- Instancia de una *collection* que contiene la lista de las referencias remotas o *proxies* de las réplicas del componente EJB para el cual se está fabricando el *proxy*. Esta lista de referencias es mantenida y administrada por el *framework* de alta disponibilidad de la plataforma J2EE.
- Instancia de la política dinámica proactiva. El nombre de la clase a partir de la cual se crea esta instancia se define en el descriptor en formato *eXtensible Markup Language (XML)* del servidor de aplicaciones durante el *assembly* del componente. Esta instancia es creada por el módulo y proceso de *deployment* de componentes de la plataforma J2EE.

Una vez el proceso de fabricación del *proxy* termina, éste es registrado en el directorio de servicios vía JNDI de donde se puede obtener por una aplicación cliente que necesite interactuar con los servicios ofrecidos por el componente EJB.

El tipo de *proxy* descrito recibe el nombre de *proxy* inteligente *(smart stub)*, ya que estos objetos, aparte de servir de delegadores o fachada a servicios remotos expuestos por un componente EJB, encapsulan objetos serializables especializados, los cuales en un momento dado pueden llegar a ejecutar cierta funcionalidad o lógica, permitiendo de esta manera tener objetos *proxies* con cierto grado de inteligencia.

La instancia de la política dinámica proactiva de balanceo de carga es creada por el contenedor J2EE, pero se ejecuta bajo el contexto de la máquina virtual de la aplicación cliente que obtuvo vía JNDI el *proxy* inteligente. Cada invocación remota hacia un método del negocio del componente EJB es delegada en primera instancia al *proxy* inteligente. Este último solicita a la instancia de la política que contiene que le asigne el servidor del *cluster* más apto sobre el cual direccionar la invocación remota de acuerdo con los parámetros establecidos por la política dinámica o tipo de solicitud. La instancia de la política solicitará el último estado de ejecución del *cluster* notificado al *proxy* inteligente, el cual reside en un repositorio compartido de estados; en caso de que el repositorio no tenga asociado ningún estado del *cluster* se genera un mensaje de solicitud de estado a cada uno de los miembros del *cluster*. Esta solicitud es publicada sobre el canal *JavaGroups* que agrupa tanto los *proxies* inteligentes como los servidores que contienen réplicas del componente. Este estado del *cluster* es empleado por el algoritmo encapsulado en la instancia de la política dinámica proactiva para determinar y calcular el servidor del *cluster* más apto para recibir y procesar la invocación remota.

El repositorio compartido de estado de ejecución del *cluster* tiene como función principal mantener el último estado o vista de ejecución de cada una de las máquinas que componen el *cluster*. Este estado es publicado sobre el canal cada cierto intervalo de tiempo por cada uno de los miembros del *cluster*. El estado almacenado en el repositorio no es persistente, reside en la memoria principal como un objeto estático y a él puede tener acceso cualquiera de los *proxies* que se ejecuten bajo el contexto de la máquina virtual de la aplicación-cliente. Es por esto que recibe el nombre de repositorio compartido.

Por último, la solución *client side* se compone de un proceso activo –un *thread* hijo del *thread* principal– bajo el cual se ejecuta la aplicación-cliente. Este *thread* es inicializado en el momento en que se realiza la primera invocación remota en la aplicación cliente, es decir, cuando se hace la búsqueda del objeto remoto en el directorio de servicios vía JNDI. La función principal de este *thread* es la de escuchar de manera asincrónica *(pull)* los eventos de notificación de estado publicados por los miembros del *cluster* sobre el canal *JavaGroups* y almacenarlos de manera sincrónica *(push)* en el repositorio compartido de estados de ejecución. Esta funcionalidad de sincronía y asincronía se llevó a cabo empleando el *PullPushAdapter*, uno de los *building blocks* ofrecidos en el *framework* de *JavaGroups*. Este *thread* es destruido en el momento en que la ejecución de la aplicación-cliente termina.

5. Conclusiones y futuros desarrollos

El papel de las políticas dinámicas proactivas en el diseño e implantación de *frameworks* de alta disponibilidad en J2EE será cada vez más relevante e importante. La necesidad de optimizar el uso de cada uno de los recursos del *cluster*, de mantenerlo en un verdadero estado de equilibrio de la carga de trabajo y no del número de tareas y la capacidad de hacer balanceo de carga selectivo, entre otras importantes características, convergerá en una solución que tenga presente en sus decisiones de distribución y balanceo de carga el estado interno de ejecución de cada uno de los miembros de un *cluster* desde diferentes facetas de ejecución (sistema operativo, *hardware* y contenedor o plataforma J2EE). Otra característica fundamental es que toda la arquitectura y el modelo son totalmente transparentes tanto para quien escribe la aplicación del negocio en J2EE, como para quien escribe la aplicación cliente que emplea los servicios del negocio que se encuentran replicados.

Como posibles trabajos a desarrollar a partir de este aporte se pueden mencionar los siguientes:

- Validar el modelo de políticas dinámicas proactivas propuesto, implementando un sistema transaccional multi-capas desarrollado bajo la especificación J2EE, el cual se deberá estructurar alrededor de servicios del negocio replicados sobre un *cluster* y de una solución de balanceo de carga que emplee técnicas de balanceo estáticas, dinámicas reactivas y dinámicas proactivas. El proceso de validación se llevará a cabo por medio de la aplicación de prue-

bas de desempeño tipo *benchmark* como por ejemplo TCP-W [TPC Benchmark, 2001], ECPerform [ECPerf Benchmark, 2001], RUBiS [Rice University, 2002] y la posterior comparación y análisis de cada uno de los resultados arrojados por este tipo de pruebas.

- Promover un *Java Communities Proccess* (JCP)[Ban, 2001] que permita incluir dentro de la especificación J2EE el soporte al servicio de lógica no funcional de alta disponibilidad (HA) y dentro de éste, una solución de balanceo de carga que se fundamente en técnicas dinámicas proactivas.

Referencias

ALLAMARAJU, S., K. AVEDAL, R. BROWETT, J. Diamond (2001), Professional Java Server Programming: J2EE edition, Birmingham, Wrox Press.

BAN, B. (2001), JavaGroups User's Guide. Fujitsu Network Communications, en: http://www.javagroups.org

BEA System (2001), Achieving Scalability and high Availability for e-Business. Clustering en Bea WebLogic Server. Bea White Papers. San Jose, California, en: http://www.bea.com/whitepapers/

BEA System (2001), Planning WebLogic Server Clusters. San José, California, en: http://edocs.bea.com/wls/docs61/cluster/planning.html

BEA System (2001), Cluster Features and Infrastructure. San José, California, en: http://edocs.bea.com/wls/docs61/cluster/features.html

BLOSSER, J. (2001), Explore the Dynamic Proxy API Use dynamic proxies to bring strong typing to abstract data types" Javaworld, en: http://www.javaworld.com/javaworld/jw-11-2000/jw-1110-proxy_p.html

CECCHET, E., J. MARGUERITE, W. ZWAENEPOEL (2002), Performance and Scalability of EJB Applications, Rice University, OOPSLA '02. ACM 1-58113-417.

ECPerf Benchmark Specification (2001), en: http://java.sun.com/j2ee/ecperf/

FLEURY, M., N. Stark (2002), JBoss Administration and Development *"Jboss 2.4.x"*, Sams.

GAMMA, E., R. Helm, R. Johnson (1999), Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

HANIK, F. (2002), Clustering Technologies in Memory Session Replication in Tomcat 4.0. The ServerSide J2EE Community, en: http://www2.theserverside.com/resources/article.jsp?l=Tomcat

HARCHOL-BALTER, M. (2002), Task assignment with unknown Duration. Carnegie Mellon University. ACM 0004 -5411/02/0300, Pittsburgh.

Java Community Process (s.f.), Community Development of Java Technologies Specifications, en: http://www.jcp.org

Java Management Extensions Specification (2002), en: http://jcp.org/aboutJava/communityprocess/final/jsr003/index2.html.

Java Platform Debugger Architecture Guide (2002), en: http://java.sun.com/j2se/1.4.1/docs/guide/jpda/.

JEWELL, T. (2000), EJB 2 Clustering with Application Servers. The O'really Network, en: http://www.onjava.com/lpt/a//onjava/2000/12/15/ejb_clustering.html.

LABOUREY, S., B. Burke (2002), JBoss Clustering. Atlanta: JBoss Group, LCC.

LABOUREY, S., B. Burke (2002), Clustering with JBoss 3.0. The O'Reilly Network, en: http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html.

LIU, J. (2002), Research Project: An Analysis of JBoss Architecture. School of Information Technologies. University of Sydney, en: http://www.cs.usyd.edu.au/~jennyliu/jboss.html

Rice University Department of Computer Science, INRIA (2002), Rice University Bidding System Benchmark (RUBIS), en: http://www.cs.rice.edu/CS/Systems/DynaServer/RUBiS/

ROMAN, E., S. Ambler, T. Jewell (2002), Mastering Enterprise Java Beans, second edition, New York, Willey Computer Publishing.

STARK, S., The JBoss Group (2002), JBoss Administration and Development *"Jboss 3.0.x"*, second edition, Atlanta, JBoss Group.

SULLINS, B., M. Whipple (2002), JMX in Action, Manning Publications.

Sun Microsystems (2002), J2EE Platform Specification, en: http://java.sun.com/j2ee/

TPC Benchmark W. (Web Commerce) Specification (2001), Transaction Processing Performance Council, San José, California, en: http://www.tpc.org/tpcw/default.asp.