DEFINICIÓN DE UN API DE PROGRAMACIÓN PARALELA Y CONCURRENTE DE ALTO NIVEL SOBRE CLUSTERS DE ALTO DESEMPEÑO

OLGA ISABEL VILLAMIZAR SANCLEMENTE

UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN
BOGOTÁ, D.C.
2005

DEFINICIÓN DE UN API DE PROGRAMACIÓN PARALELA Y CONCURRENTE DE ALTO NIVEL SOBRE CLUSTERS DE ALTO DESEMPEÑO

OLGA ISABEL VILLAMIZAR SANCLEMENTE

Trabajo de Grado presentado como requisito para optar al título de Magíster en Ingeniería de Sistemas y Computación

Director: Rafael Gómez
Profesor Asociado

UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN
BOGOTÁ, D.C.
2005

CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

1. INTRODUCCIÓN

El cómputo en red es una de las alternativas que pueden ser tomadas para solucionar los problemas que se presentan al momento de querer realizar grandes cálculos y correr aplicaciones que requieran capacidades de cómputo bastante altas. Se unen varios computadores en red y se aprovecha la capacidad de cómputo de cada uno de ellos, formando un *cluster*. En este caso se habla de computación paralela, que puede entenderse como el cómputo en varias unidades de procesamiento para aprovechar los recursos y reducir el tiempo invertido en un proceso.

Si se desea correr aplicaciones sobre el *cluster*, es necesario realizar una programación paralela para poder migrar los procesos a todos los nodos que conforman el cluster y balancear la carga de procesos presentes. Existe en este momento la estandarización de prácticas comunes de programación paralela mediante *MPI* y *PVM*, las cuales proporcionan herramientas necesarias para la construcción de programas paralelos. Sin embargo, estas librerías son de muy bajo nivel y los desarrolladores de aplicaciones tendrían que conocer mucho acerca de migración de procesos y manejo de paso de mensajes entre otras cosas.

Hoy en día existen *API* y *Middlewares* que ayudan a ver el *cluster* como una sola máquina, difundir la información en el *cluster*, migrar los procesos para realizar balanceo de carga, tener un control centralizado y descentralizado del cluster, pero aun así, para los programadores sigue siendo difícil hacer desarrollos sobre *cluster*, debido a que tienen que conocer muchas librerías y comandos de muy bajo nivel que les permitan interactuar con el *cluster*.

Existe un *Middleware* llamado *OpenMosix* el cual pretende conseguir una imagen única del sistema *cluster*, mostrándolo al usuario como un todo. Cada nodo presente en el *cluster* es capaz de auto – gobernarse, esto quiere decir que no existe un nodo maestro que se encarga de manejar los procesos de los demás nodos. Este *middleware* se instala en el sistema operativo Linux y lo convierte en un sistema de ejecución paralela.

Por otra parte, existe un *API* desarrollado en *C++,* que es una implementación de un modelo de concurrencia llamado *CSP* que maneja la comunicación de procesos a través de canales. Este *API* llamado *C++CSP* fue tomado como base para el desarrollo de la presente tesis, tomando las principales funciones de procesos y canales para la concurrencia local.

Con este proyecto, se definió un modelo de programación paralela que permite desarrollar programación de alto nivel sobre *cluster* de alto desempeño, para así poder realizar

aplicaciones paralelas de una forma más fácil y amigable, sin que el programador deba conocer la configuración hardware del *cluster*. Se quiso facilitar la programación sobre el cluster, brindando un modelo claro de programación basado en el modelo existente *CSP* y utilizando herramientas que ayudaron a la definición del *API*, ya que se tomaron las ventajas del *OpenMosix* para permitir un balanceo de carga y distribución de procesos, el *API* desarrollado *C++CSP* para la concurrencia local y las principales ventajas de las primitivas de envío y recepción de mensajes de *PVM* para la comunicación entre los procesos que se encuentran distribuidos en nodos diferentes en el *cluster*.

En los siguientes capítulos, se explica un poco de la teoría de *cluster*, las principales características de *OpenMosix*, las primitivas de envío y recepción de mensajes de *PVM*, el modelo de *CSP* y las principales características del *API C++CSP*. Al igual que la extensión del modelo *CSP* para permitir concurrencia en diferentes nodos del *cluster*, el diseño e implementación del *API* definido.

2. OBJETIVOS

OBJETIVO GENERAL

El objetivo general de este trabajo de investigación es definir y validar un modelo de programación paralela que permita realizar desarrollos de aplicaciones sobre cluster utilizando librerias y primitivas de alto nivel, permitiendo así que la programación sobre clusters de computadores sea más amigable y sencilla.

OBJETIVOS ESPECIFICOS

- Instalación y configuración de un cluster de alto rendimiento.
- Definición del modelo de programación.
- Definir las diferentes alternativas de implementación del modelo.

3. CLUSTERS

La complejidad de ciertos problemas en algunas ramas de la ciencia requiere capacidades de cálculo mayores que las proporcionadas por los computadores personales. Existen máquinas mucho más poderosas que han permitido realizar estos cálculos, sin embargo, estas máquinas tienen el inconveniente de ser muy costosas y de rápida obsolescencia. Por lo tanto, ha sido necesario buscar nuevas alternativas que soporten estos problemas.

Una de las alternativas que se ha explorado, consiste en utilizar equipos de cómputo conectados en red que se puedan presentar al usuario como un solo sistema, teniendo una capacidad de cómputo notoria con costos razonables.
A este conjunto de equipos se les ha dado el nombre de cluster.

A continuación, se dará una breve introducción a los clusters, su definición, usos, clasificación, componentes entre otros.

3.1 DEFINICION DE CLUSTER

El cluster se define como un grupo de componentes distribuidos que colaboran en conjunto para presentarse al usuario como un solo sistema [1].

A cada uno de los componentes distribuidos se les da el nombre de “Nodo” y cada nodo tiene su propio disco duro y memoria, por lo tanto se puede entender el cluster como un sistema de memoria distribuida.

Entre algunas de las principales características de los clusters se tienen:

- Permiten escalabilidad, disponibilidad, rendimiento y computo en paralelo.
- Estructura relativamente liviana en costos y recursos.
- Seguridad sólo requerida si está expuesto al “exterior”.
- Comunicación a través de paso de mensajes.

Uno de los primeros clusters fue construido por la Nasa en 1994 con el propósito de resolver problemas computacionales que aparecen en las ciencias de la Tierra y el Espacio. Se le dio el nombre de *Beowulf* y fue construido con 16 computadoras personales con procesadores Intel 486 a 200 MHz, dando un rendimiento teórico de 3.2 Gflop/s.

Una de las características principales de este cluster es haber sido construido corriendo sistema operativo Linux, lo cual, debido a que es de fuente abierta, ha permitido que se le añadan nuevas características al sistema para poder responder a las necesidades de los usuarios en los clusters.

Después de la construcción del *Beowulf,* se han construido muchos otros clusters que han dado gran desempeño y han solucionado muchos problemas de astrofísica, dinámica molecular, dinámica no-lineal y ecuaciones diferenciales, fases de transición en el universo cercano, movimientos de la tierra, entre otros.

3.2 USOS

Según Thomas Sterling en su artículo "Una Introducción a los Clusters de PC para Cómputo de Alto Rendimiento", los clusters permiten aprovechar los recursos; las tecnologías de hardware y software que fueron desarrollados para aplicaciones amplias y mercados comerciales pueden también servir en el área del cómputo de alto rendimiento [2].

Hoy en día los clusters están siendo utilizados para correr diversas aplicaciones y para diferentes usos, entre ellos:

- Aplicaciones Científicas y de ingeniería.
  - Simulaciones.
  - Aplicaciones de Negocio.
  - Aplicaciones de Misión Crítica.
  - Control de reactores nucleares, sistemas de control.
- Aplicaciones de Internet.

Los clusters han evolucionado para apoyar actividades en aplicaciones que van desde la supercomputación hasta el software adaptado a misiones críticas, pasando por los servidores Web, el comercio electrónico y las bases de datos de alto rendimiento.

3.3 CLASIFICACION

Los clusters se pueden clasificar por diferentes criterios: según su uso, según la configuración de los nodos y según la arquitectura de los nodos, entre otras posibilidades.

Según la arquitectura de los nodos, es decir, la configuración hardware, los clusters pueden ser *Homogéneos* (todos los nodos pertenecientes al cluster tienen la misma arquitectura) o *Heterogéneos* (no todos los nodos pertenecientes al cluster tienen la misma arquitectura). De igual forma se clasifican según la configuración software (según el sistema operativo que se encuentre instalado en los nodos).

Por otra parte, según su uso, los clusters se pueden clasificar de la siguiente forma:

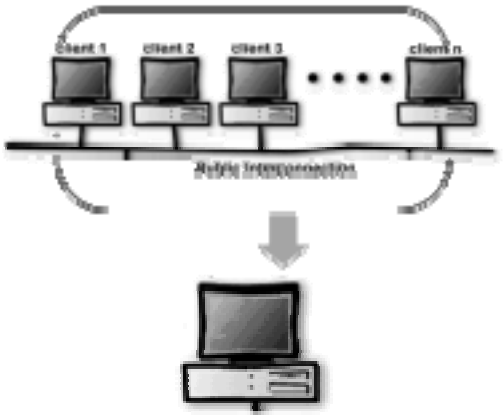
- Alto desempeño.
- Servidores Virtuales y balanceo de carga.
- Alta disponibilidad.
- Cómputo en Malla.

A continuación se dará una breve descripción de cada uno de ellos:

3.3.1 Clusters de Alto desempeño.

Algunas veces se deben correr aplicaciones que necesitan mucho procesamiento y es necesario compartir recursos, memoria, disco y red de máquinas individuales para obtener la capacidad de cómputo que se requiere, es por eso que se tiene un conjunto de máquinas individuales que se comportan como una sola máquina muy potente y se le da el nombre de cluster de alto desempeño.

Figura 1. Cluster de Alto Desempeño



3.3.2 Clusters de Servidores Virtuales.

En empresas que ofrecen servicios Web, generalmente tienen el problema del aumento de tráfico en determinadas horas del día, los clusters de servidores virtuales, ayudan a compartir la carga de trabajo y tráfico. Su propósito consiste en mejorar el tiempo de acceso y la confiabilidad, distribuyendo las tareas entre los nodos que hagan parte del

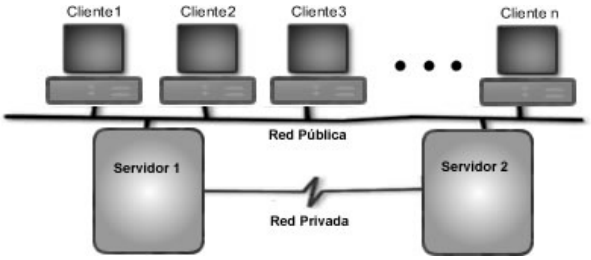
cluster, es así que si un nodo se encuentra con mucha carga, otro nodo está en la capacidad de atender la solicitud.

3.3.3 Clusters de Alta Disponibilidad.

En algunas empresas, es indispensable que los servicios que ofrecen a sus clientes, se encuentren disponibles siempre, por lo tanto este tipo de clusters ayudan a mantener en funcionamiento los servidores y tener los servicios disponibles.

Este tipo de clusters son llamados también como Clusters de redundancia. Su función es la de estar listos para entrar en funcionamiento inmediatamente se presente un fallo en algún otro servidor.
Los clusters de alta disponibilidad permiten un fácil mantenimiento de servidores, debido a que existe redundancia entre ellos (por lo menos un servidor igual al otro) y se puede sacar de funcionamiento un servidor sin necesidad de que deje de funcionar el sistema ya que se tendrá el respaldo del otro servidor.

Figura 2. Cluster de Alta Disponibilidad



En la anterior figura se muestra que existen 2 servidores que en un tiempo periódico se están monitoreando mutuamente; en el momento en que uno falle, entra el otro en funcionamiento.

Este tipo de clusters es utilizado generalmente para correr aplicaciones que necesiten un intercambio intensivo de información y en aplicaciones que necesiten estar disponibles todo el tiempo.

3.3.4 Cómputo en Malla.

Como vimos anteriormente, un cluster puede ofrecer diferentes servicios como lo son disponibilidad, rendimiento, balanceo de carga entre otros, sin embargo estos servicios los ofrece dentro de una misma organización, si se desean compartir recursos entre diferentes dominios corporativos, es necesario utilizar una malla (grid).

Una Malla es un tipo de sistema paralelo y distribuido que permite compartir, seleccionar y añadir recursos que se encuentran distribuidos a lo largo de dominios administrativos múltiples.

Una Malla es un tipo de sistema paralelo y distribuido que permite compartir, seleccionar y añadir recursos, se basa en la disponibilidad capacidad, rendimiento, costo de los recursos distribuidos así como en la calidad de servicio que requiere el usuario.

Ahora bien, si los recursos distribuidos se encuentran bajo la administración de un sistema central, estamos hablando de cluster.

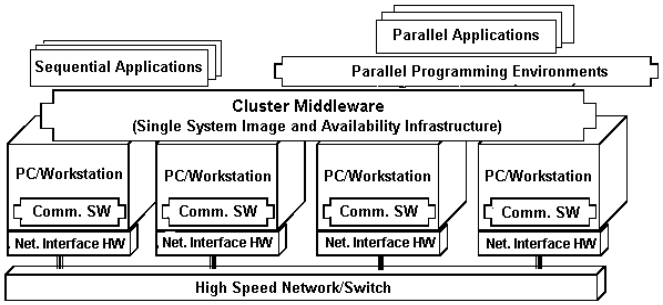
En una malla, cada nodo tiene su propio administrador de recursos y política de asignación.

Este tipo de cluster, más conocido como Grid, está siendo muy utilizado para compartir recursos entre universidades, trabajos de investigación entre naciones, por ejemplo se está haciendo un desarrollo de un Grid Europeo.

3.4 COMPONENTES

En la figura que se presenta a continuación, se encuentra la arquitectura de un cluster; ahí se pueden observar los diferentes componentes que conforman el cluster.

Figura 3. Arquitectura del Cluster



En la figura anterior tomada de una presentación de Rajkumar Buyya del 27th simposio anual de arquitectura de computadores en el año 2000 titulada: *High Performance Cluster Computing (Architecture, Systems, and Applications),* se puede observar que un cluster se compone de: nodos (*PC/Workstation*), la red, el middleware, entornos y herramientas de programación paralela, entre otros componentes.

3.4.1 Nodos.

Se llaman nodos a cada uno de los computadores que hacen parte del cluster. Un nodo puede ser un sistema con uno o más procesadores (PC, estaciones de trabajo o SMP), con memoria, recursos de entrada / salida y sistema operativo.

Los nodos se pueden basar en diferentes arquitecturas y correr diferentes sistemas operativos, además se encuentran conectados unos a otros a través de cables de red.

3.4.2 Red.

La red es un componente muy importante en los clusters, ya que transfiere los paquetes de datos, permitiendo así la comunicación entre nodos.

Algunos problemas requieren gran ancho de banda y baja latencia para ser más eficientes. Hoy en día existen diversos tipos de red que pueden ser utilizados para tratar estos problemas, entre ellos: Ethernet, Fast Ethernet, Gigabit Ethernet, Myrinet e Infiniband.

Las principales características técnicas que se deben tener en cuenta para elegir el tipo de red que se va a utilizar en un cluster son las siguientes:

- Alto ancho de banda.
- Baja latencia.
- Disponibilidad / Fiabilidad.
- Escalabilidad

3.4.3 Middleware.

Es una interfaz entre las aplicaciones, el hardware del cluster y el sistema operativo. Reside entre el sistema operativo y la aplicación.

Según Sacha Krakowiak en una presentación realizada en el 2002 titulada: Patterns and Frameworks for Middleware construction, un middleware tiene cuatro principales funciones [3]:

- Proveer una interfaz de programación para las aplicaciones (API).
- Ocultar la heterogeneidad del hardware y sistema operativo de los nodos que conforman el cluster.
- Hacer la distribución de procesos transparente al usuario.
- Simplificar el manejo y administración del cluster.

Entre las principales características del Middleware están que permite proveer una sola imagen del sistema al usuario (Single System Image – SSI), difundir la información en el cluster, migrar los procesos para realizar balanceo de carga, tener un control centralizado y descentralizado del cluster entre otras.

3.4.4 Herramientas de programación paralela.

Los más altos grados de rendimiento se han alcanzado a través del cómputo en paralelo. Asegurar el paralelismo en los programas de aplicación para alcanzar ganancias en el rendimiento ha sido un reto que el cómputo ha atacado desde los años 70.

El cómputo paralelo implica tomar una tarea y dividirla en subtareas para luego trabajar sobre cada una de ellas simultáneamente. En algunos casos es necesaria la comunicación entre subtareas para poder finalizar la operación; en el cluster, esta comunicación se realiza a través del paso de mensajes.

El paso de mensajes es un modelo de interacción entre procesadores dentro de un sistema paralelo. En general, un mensaje es construido por software en un procesador y se envía a través de la red a otro procesador, el cual debe aceptar el mensaje y actuar sobre su contenido.

Existen algunas librerías que soportan el paso de mensajes entre los equipos de una red. Dos de las más conocidas son MPI (Message Passing Interface – Interfaz de Paso de Mensajes) y PVM (Parallel Virtual Machine – Máquina Virtual Paralela).

3.5 FUNCIONAMIENTO

Como se ha podido observar, desde un punto de vista general, un cluster consta de dos partes: la primera es el software y la segunda es la interconexión hardware entre las máquinas (nodos) que pertenecen al cluster.

"El software está compuesto por el sistema operativo, compiladores y aplicaciones especiales que permiten que el sistema explote las ventajas del cluster" [4].

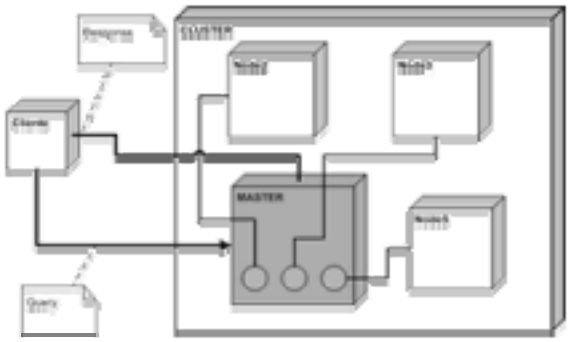
Cuando se trata de resolver un problema en paralelo, el software debe ser capaz de dividir el problema en tareas más pequeñas, repartirlas entre los nodos y elaborar los resultados. Puesto que las subtareas se van a ejecutar en paralelo, se consigue un aumento de velocidad, aunque hay que tener en cuenta el retardo que genera la división, reparto y transmisión de mensajes (resultado, coherencia, estados).

La distribución de tareas se puede hacer mediante los siguientes esquemas:

3.5.1 Distribución Maestro /Esclavo.

La distribución de tareas mediante este esquema se hace de la siguiente forma:

Figura 4. Distribución Maestro/Esclavo



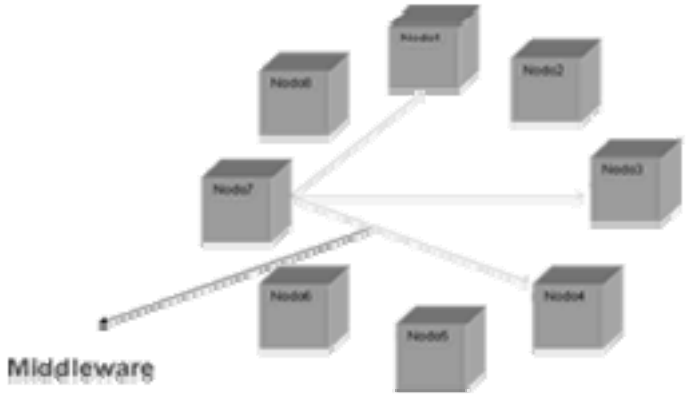
El cliente hace una solicitud o lanza un proceso. Existe un nodo Master(maestro), el cual hará la distribución de las tareas a los nodos Slaves (esclavos), los cuales se encargarán de ejecutar las tareas y pasar la respuesta al nodo maestro; este último le entregará la solución al cliente.

El nodo maestro hace la migración de los procesos dependiendo de la carga que tengan los demás nodos pertenecientes al cluster; es decir, si un nodo se encuentra ejecutando una tarea y existe otro nodo que en ese determinado momento se encuentra en reposo, entonces el nodo maestro enviará el proceso al nodo que se encuentra en reposo.

3.5.2 Distribución de tareas, procesamiento simétrico.

La distribución de tareas mediante este esquema se hace de la siguiente forma:

Figura 5. Distribución Simétrica



En este tipo de esquema, todos los nodos cumplen la misma función dentro del cluster.

Se llama nodo Raíz o Home el nodo donde se lanza una tarea o proceso. Los procesos migran a un nodo remoto y usan sus recursos locales, pero se encuentran en continua comunicación con el nodo raíz.

Después que un proceso migra, muchas de las llamadas al sistema son interceptadas por un link layer en el nodo remoto; si la llamada al sistema es independiente del sitio, se realiza en el nodo remoto, de otra forma, la llamada al sistema se redirecciona al nodo raíz para ser ejecutada.

En este tipo de esquema, existe un middleware que se encarga de distribuir los procesos o tareas entre los nodos que forman el cluster. Además, se encarga de: entregar los resultados, hacer balanceo de carga, hacer transparente al usuario la forma de migración de los procesos y hacer ver el sistema como uno sólo.

Para el usuario que ejecuta la tarea o el proceso, es transparente el lugar donde se está ejecutando el mismo.

3.6 PROBLEMÁTICA DE PROGRAMACIÓN SOBRE CLUSTERS

En las secciones anteriores se pudo observar que los clusters ofrecen algunas ventajas como lo son el poder compartir recursos, la buena relación costo/rendimiento, la tolerancia a fallos y el manejo de paralelismo entre otras. Sin embargo, también existen desventajas como: necesidad de utilizar software más complejo para ver el sistema como uno solo, problemas con la red (latencia y velocidad), dificultad de configuración, programación de aplicaciones a bajo nivel dado que el programador debe conocer información acerca de la configuración del cluster para poder utilizar las primitivas de programación, tal como se mostró anteriormente en la descripción de las librerías de paso de mensajes existentes, entre otras.

Cuando se quiere utilizar un cluster para aprovechar el desempeño del mismo, es necesario que el programador tome en cuenta el paralalelismo y la concurrencia al momento de desarrollar las aplicaciones. Dado esto, el programador debe familiarizarse con librerías de paso de mensajes de bajo nivel y se hace necesario la creación de API o middlewares que le permitan que se interese únicamente por el desarrollo de la aplicación y no por cómo interactuar con el cluster.

El API que permita al programador realizar aplicaciones paralelas sobre cluster debería ocultarle la configuración del cluster, es decir, para el programador debería ser transparente el número de procesadores presentes en el cluster, la configuración de cada uno de ellos entre otras características. Además sólo debería ser importante para él definir o identificar la granuralidad que se va a manejar, ya sea por proceso, por hilo, por nodo.

En los siguientes capítulos, se describen posibles soluciones encontradas para las desventajas antes mencionadas que ayudan a lograr que la programación sobre el cluster sea un poco más sencilla.

4. OPENMOSIX

OpenMosix (*Multicomputer Operating System for Unix*) es una extensión del *Kernel* de Linux. Hace parte de la implementación de un proyecto académico desarrollado en el centro de Computación Distribuida de la Universidad Hebrea.

Se encuentra clasificado como un *cluster* SSI (*Single System Image*) y hace parte de uno de los tipos de funcionamiento de *cluster* explicados anteriormente, *Distribución de tareas, procesamiento simétrico*, es utilizado para distribución de carga y permite distribuir los procesos que hacen parte de una aplicación entre varios nodos del *cluster*, migrando procesos entre los nodos y teniendo en cuenta la carga de los nodos para distribuir los procesos.

Generalmente es utilizado para correr sistemas que necesiten de mucha potencia de computo repartida en muchos procesos que puedan ser migrados de unas máquinas a otras usando un algoritmo de balanceo de carga automático y haciendo transparente para el usuario la migración de los procesos.

En su funcionamiento, cuando un usuario inicia un proceso lo lanza en determinado nodo del cluster y *OpenMosix* dependiendo de la carga de ese nodo puede migrarlo a otro nodo que se encuentre con menos carga, así se puede ejecutar más rápidamente y el sistema puede asignar y reasignar los nodos en los cuales se ejecuta.

Cada proceso tiene un único nodo *home* (donde inicia y es creado). Cuando un proceso migra a un nodo remoto, siempre se mantiene en contacto con el nodo *home* ya que éste es el que realiza los llamados al sistema.

A continuación se explicarán características internas de *OpenMosix*, configuración, ventajas y desventajas al momento de usarlo, entre otros aspectos importantes.

4.1 ASPECTOS INTERNOS DE OPENMOSIX

Existen diversos aspectos a ser tenidos en cuenta en *OpenMosix*, la descripción de estos aspectos ha sido tomada de un estudio realizado por Moshe Bar [5].

4.1.1 Distribución de tareas, procesamiento simétrico.

La granularidad de la distribución de trabajo en *OpenMosix* es el proceso, por lo tanto, no puede ejecutar multiples *threads* en nodos separados y no puede ejecutar un solo proceso en nodos separados.

Muchas aplicaciones son diseñadas para crear procesos hijos (*fork()*), *OpenMosix* puede migrar cada uno de estos procesos creados.

4.1.2 Comunicación

La comunicación a través de los nodos del *cluster* trabaja por encima de TCP/UDP, usando un protocolo TCP/IP optimizado para la migración y la comunicación entre procesos.

La arquitectura de la comunicación de *OpenMosix* tiene 2 partes principales que son:

- Mecanismo de migración preferente de procesos.
- Algoritmos adaptativos para compartir de recursos.

Ambas partes son implementadas como módulos recargables a nivel del *kernel* y a nivel de aplicación son completamente transparentes.

El mecanismo de migración de procesos se encarga de evaluar cada uno de los nodos del *cluster* y realizar un balanceo de carga uniforme entre los nodos, usualmente, las migraciones son basadas en la información dada por los algoritmos para compartir recursos

Los algoritmos adaptativos para compartir recursos son dos:

- *Balanceo de Carga*: Reduce la diferencia de carga entre los nodos; esto lo hace migrando procesos. Revisa la carga de procesos en cpu y el porcentaje de memoria libre.

- *Gestión de memoria*: Intenta mantener el máximo número de procesos en memoria del *cluster*. Se invoca cuando un nodo empieza paginación excesiva dado que no hay suficiente memoria libre.

Estos algoritmos están diseñados para responder en línea a las variaciones en el uso de los recursos a través de los nodos.

4.1.3 Procesos

Un proceso es la parte mínima en *OpenMosix* y puede ser dividido en 2 partes:

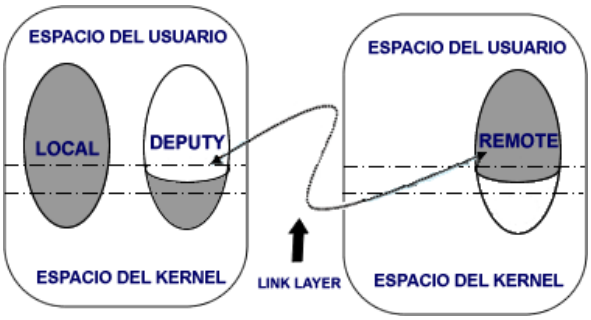
- El contexto del sistema de un proceso el cual es llamado *deputy* y no es migrado del nodo *home*; encapsula el proceso cuando está ejecutándose en el *Kernel*.

- El contexto del usuario el cual es llamado *remote* y es la parte del proceso que realiza el trabajo. Contiene el código del programa, los datos, los mapas de memoria. El *remote* es la parte del proceso que puede ser migrada a otro nodo en el *cluster*.

Después de la migración, ambas partes del proceso se comunican entre sí, utilizando un API entre el contexto del sistema y el contexto del usuario. Otra forma de interactuar entre los contextos es a través de señales y eventos.

Las llamadas entre los procesos pueden ser interceptadas a través del *"link layer"*, que mantiene la comunicación entre las 2 partes del proceso. A continuación en la Figura 6 se muestran las partes de un proceso en OpenMosix.

Figura 6. Partes de un proceso en OpenMosix



Las llamadas al sistema, deben ser ejecutadas en el nodo *home*, por lo tanto el contexto del usuario debe contactar al contexto del sistema, enviarle una petición con todos los parámetros y datos que el nodo *home* necesitará procesar. El nodo *home*, procesará la llamada y enviará de vuelta al nodo *remote* el valor de éxito o fracaso de la llamada y los resultados que necesite el *remote.* Las llamadas al sistema son una forma de sincronización entre los dos contextos.

En su funcionamiento, si un proceso se encuentra ejecutándose en un nodo A y *OpenMosix* lo migra a otro nodo B, entonces en el nodo A quedará el *deputy* (contexto del sistema) y en el nodo B quedará el *remote* (contexto del usuario). Si el proceso que se encuentra corriendo en B debe hacer una llamada al sistema, entonces contacta al contexto del sistema del proceso en el nodo A y le envía la petición para que sea ejecutada en ese nodo y luego recibe la respuesta de A.

El contexto del usuario (*remote*) no puede ser accedido por otros procesos que se encuentran ejecutándose en el mismo nodo y viceversa, esto se hace para poder mantener la consistencia de los datos.

4.2 CONFIGURACION

Para poder utilizar las ventajas que ofrece el *OpenMosix*, es necesario instalarlo en cada uno de los nodos pertenecientes al *cluster* y ejecutar el demonio que lo habilita.

Existen 3 tipos de configuraciones:

- *Single-pool:* Todos los equipos (incluyendo servidores y workstations) se usan como una sola máquina: Cada máquina hace parte del *cluster* y puede migrar procesos.

- *Server-pool:* Los servidores son parte del *cluster* mientras los workstations no.

- *Adaptive-pool*: Los servidores hacen parte del *cluster* mientras los workstations pueden entrar y salir del *cluster* dependiendo del la hora, por ejemplo.

Para que los nodos se reconozcan entre sí, es necesario configurar un archivo llamado *openmosix.map*, en el cual se colocarán las direcciones IP de los nodos que pertenecen al *cluster*. Sin embargo también se puede realizar autodescubrimiento utilizando mensajes multicast.

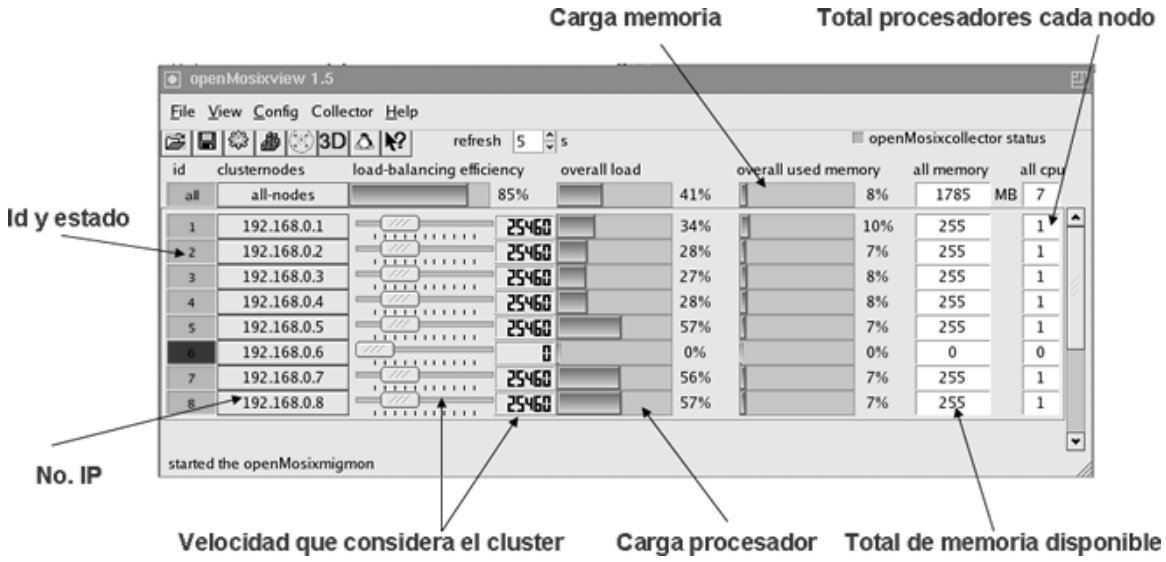
4.3 HERRAMIENTAS DE USUARIO

Una de las principales herramientas que ofrece el *OpenMosix* al usuario es *OpenMosixView*, que permite administrar el *cluster* de forma gráfica. Esta herramienta contiene 5 aplicaciones muy útiles para monitorear y administrar el *cluster*.

- *OpenMosixview:* Es la aplicación principal para monitorear y administrar el *cluster*.
- *OpenMosixproc:* Es una aplicación gráfica para el manejo de procesos.
- *OpenMosixcollector*: Es un demonio que guarda los *logs* e información de los nodos pertenecientes al *cluster*.
- *OpenMosixanalyzer*: Es una aplicación que sirve para analizar los datos recolectados por el *OpenMosixcollector*.
- *OpenMosixhistory*: Es una aplicación que guarda un historial de los procesos del *cluster*.

A continuación en la Figura 7 se muestra la ventana principal del *OpenMosixview* en la que se detallan cada uno de sus componentes principales.

Figura 7. Ventana principal del OpenMosixview



4.4 VENTAJAS Y DESVENTAJAS

El *OpenMosix* ofrece muchas ventajas para permitir que varios computadores conectados en red funcionen como un *cluster*, sin embargo por ser un proyecto relativamente nuevo, también tiene algunas desventajas que en futuras versiones podrían ser resueltas.

4.4.1 Ventajas.

- No se requieren paquetes extras y no se necesitan cambios en el código de las aplicaciones ya existentes.
- Los nodos pertenecientes al *cluster* pueden entrar y salir del *cluster* sin interrumpir el funcionamiento del *cluster*
- Los usuarios pueden correr aplicaciones paralelas inicializando múltiples procesos en un nodo y *OpenMosix* se encargará de distribuir esos procesos siendo transparente para el usuario el sitio donde se está corriendo el proceso.
- Permite un máximo aprovechamiento de los recursos existentes.
- Permite flexibilidad y escalabilidad debido a que puede ser configurado en tiempo de ejecución y se le pueden agregar más nodos.
- Migración de procesos automática y transparente para el usuario.

- Si en determinado momento, un nuevo nodo ingresa al *cluster*, *OpenMosix* tiene la capacidad de hacer un balanceo de carga de los procesos.

- Ofrece una alternativa para la programación paralela.

4.4.2 Desventajas.

- Dependiente del *Kernel*.

- No todos los procesos pueden ser migrados, por ejemplo los procesos que comparten memoria y contienen threads. Tiene limitaciones de funcionamiento.

- Un mismo proceso no puede ser ejecutado en múltiples nodos.

- Los *Green Threads* de java permiten migración ya que cada java *thread* es un proceso separado. Sin embargo estos *threads* ya no vienen con las versiones actuales de java.[6]

- No permite migrar *sockets*. Aunque en futuras versiones se está estudiando la posibilidad e incluir *sockets* migrables.

OpenMosix tiene definido un API para la interacción con el cluster, este API permite saber a través de comandos, que procesos están corriendo en que nodo del cluster, permite bloquear la ejecución de procesos en determinado nodo o nodos del cluster, además permite sacar estadísticas de ejecución de procesos, cantidad de CPU utilizada en los nodos, cantidad de memoria, permite migrado manual de procesos entre otras[7].

5. PVM – PARALLEL VIRTUAL MACHINE

PVM es un paquete de programas que permite usar computadores unidos por red (heterogéneos) como un solo computador para así explotar las ventajas de la programación paralela, concurrente y distribuida.

PVM es una de las varias soluciones presentadas para la programación sobre clusters de computadores, permite la comunicación entre los diferentes nodos del cluster siempre y cuando estos estén conectados a la máquina virtual paralela.

El modelo computacional de *PVM* se basa en que una aplicación consiste de varias tareas; cada tarea es responsable de una parte del programa. Las tareas se pueden ejecutar en paralelo, pueden necesitar sincronizarse o intercambiar datos y, en cualquier momento de la ejecución, una tarea puede comenzar o parar otras tareas, además de agregar o borrar computadores de la máquina virtual. La forma de intercambiar datos y sincronizarse es por paso de mensajes.

PVM está compuesto por dos componentes principales: el primero es un demonio que controla el funcionamiento de los procesos (tareas) y coordina las comunicaciones; el segundo componente consta de unas librerías de rutina de interfaz las cuales tienen primitivas para: control de procesos, información del estado de los procesos, configuración dinámica de los nodos, envío y recepción, control de *buffers* de mensajes, empaquetamiento de datos, entre otras.

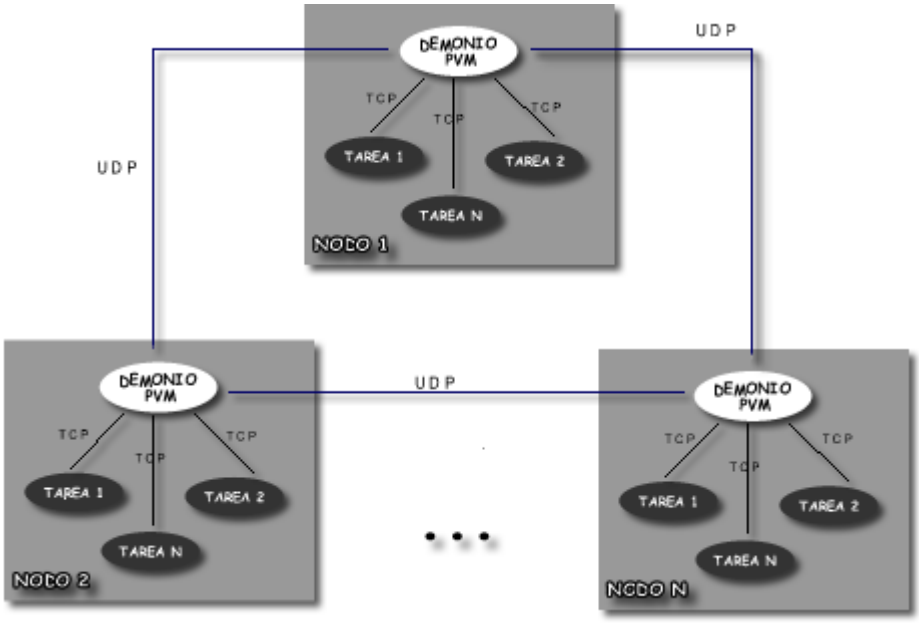
En las siguientes secciones, se explicará un poco acerca de la arquitectura de *PVM*, el modelo de comunicaciones que utiliza y las funciones de *PVM*, analizando más a fondo las primitivas de envío y recepción de mensajes que hacen parte importante del desarrollo del presente trabajo de investigación.

5.1 ARQUITECTURA DE PVM

En la arquitectura de *PVM* los principales componentes son los demonios que corren en cada una de las máquinas del cluster, las tareas que están ejecutando cada uno de estos demonios y los protocolos de comunicación utilizados para interactuar entre cada componente.

En la Figura 8 se muestra cómo es la interacción entre cada uno de los componentes y los protocolos de comunicación utilizados entre demonios y entre demonios y procesos (tareas); más adelante se explicará cada uno de ellos.

Figura 8. Arquitectura de PVM



5.1.1 Demonio PVM (*pvmd*).

Es un proceso Unix que debe correr en cada uno de los nodos que hacen parte del cluster.

Es el encargado de asignar un identificador a cada una de las tareas que están corriendo en su nodo. El identificador es llamado *TID* (*task integer ID)* y es único, se asigna al momento de inicializar la tarea; su función es la de establecer las comunicaciones entre las tareas sirviendo como enrutador y controlador de mensajes, además provee un punto de contacto, autenticación, detección de fallos y control de procesos.

Un demonio puede ser configurado como *master* o *slave,* dependiendo del momento de inicialización; es decir, la primera máquina que inicialice su demonio *PVM* va a ser el *master*, los demás demonios que entren a ser parte de la máquina virtual serán *slaves.*

5.1.2 Tareas PVM.

Una tarea es la unidad más pequeña en *PVM.* Hace parte de una aplicación que usa las funciones de la librería *PVM* para interactuar con otras tareas, la forma de interacción es: enviando y recibiendo mensajes, inicializando subtareas, detectando errores entre otras.

Cada tarea tiene un identificador único que es proporcionado por el demonio local y es utilizado para interactuar con otras tareas ya que cuando se desea enviar y recibir mensajes, se debe hacer utilizando los *TIDS*.

Las tareas pueden ser asignadas para ejecutar en diferentes nodos del cluster; esto se hace utilizando la primitiva *pvm_spawn*, la cual lanza una tarea en la ubicación que se especifique.

5.1.3 Protocolos de Comunicación.

La comunicación en *PVM* está basada en *TCP*, *UDP* y *sockets* de Unix.

Existen 3 tipos de conexiones en *PVM*: entre demonios, entre demonio y tareas y entre tareas. La primera utiliza protocolo *UDP* para la comunicación, la segunda utiliza *TCP* y la tercera usa los *sockets* de Unix, aunque generalmente el flujo de comunicación va de la tarea al demonio local, luego del demonio local al demonio remoto y de éste a la tarea remota.

Tomando como ejemplo la Figura 8, si en determinado momento la tarea 1, que se encuentra en el nodo 2, desea interactuar con la tarea 2, que se encuentra en el nodo 1, debe realizar los siguientes pasos:

- Se comunica con su demonio local, el cual se encarga de establecer comunicación con los demás demonios existentes en el cluster para hallar la localización de la tarea 2, esto lo hace a través de *UDP*.
- Luego cuando ya se tiene la localización de la tarea, entonces el demonio local se comunica con el demonio del nodo 1 el cual le envía el mensaje a través de *TCP* a la tarea 1.

5.2 MODELO DE COMUNICACION DE PVM

Según lo explica el documento *The PVM Concurrent Computing System Evolution* de V_ S_ Sunderam del departamento de Matemáticas y Ciencias de la Computación de la Universidad de Atlanta – USA, G_A Geist del laboratorio Oak Ridge y J Dongarra R Manchek del departamento de Ciencias de la Computación de la Universidad de Tennessee, el modelo de comunicación de *PVM* supone que una tarea puede enviar un mensaje a cualquier otra tarea sin límite de tamaño, además se pueden enviar un número ilimitado de mensajes. Sin embargo, cada nodo tiene una cantidad limitada de memoria física para el *buffer*, pero *PVM* supone que existe suficiente memoria disponible.

También, el modelo provee funciones de envío asíncrono de mensajes, recepción asíncrona de mensajes y recepción de mensajes particionados. Además garantiza el orden de los mensajes: si la tarea 1 envía un mensaje A a la tarea 2, después la tarea 1 envía el mensaje B a la tarea 2, el mensaje A llegará a la tarea 2 antes que el mensaje B.

Los *buffers* de mensajes son asignados dinámicamente, aunque, como se dijo anteriormente, el tamaño depende de la cantidad de memoria disponible en el nodo.

Para poder establecer la comunicación entre nodos pertenecientes a la máquina virtual, es necesario utilizar *rsh* para la autenticación entre nodos; de no tener instalado y corriendo el demonio de *rsh,* no será posible acceder desde un nodo a otro y utilizar sus recursos.

5.3 FUNCIONES DE PVM

Como se mencionó anteriormente, existen funciones de: control de procesos, información, configuración dinámica, empaquetamiento de datos y control de *buffers*, envío de mensajes y recepción de mensajes. A continuación se define cada una de ellas.

5.3.1 Funciones de Control de Procesos.

Son funciones que le permiten al programador tener un control de los procesos y tareas que se encuentran ejecutándose en la máquina virtual.

- *pvm_mytid*: Regresa el *TID* de este proceso; puede ser llamada varias veces. Inicializa el proceso en la máquina virtual si es la primer llamada de *PVM*. Esto quiere decir que asocia el proceso con el demonio local.

- *pvm_exit*: Le dice al demonio (*pvmd*) local que este proceso se retira de *PVM*. Esta rutina no mata el proceso; el proceso puede volver a asociarse a la máquina virtual haciendo un llamado como *pvm_mytid.*

- *pvm_spawn*: Inicializa un nuevo proceso *PVM*. El proceso que se inicializa puede hacerlo en el nodo local o en un nodo diferente. Además puede inicializarse más de un solo proceso.

- *pvm_kill*: Mata la tarea *PVM* identificada por el *TID* pasado como parámetro a la función.

5.3.2 Funciones de Información.

Las funciones de información, como su nombre lo indica, le sirven al programador para obtener información que le pueda servir para continuar ejecutando la aplicación. La información obtenida es acerca de los procesos que se están ejecutando en la máquina virtual.

- *pvm_parent* : Regresa el *TID* del proceso que lanzó la tarea actual.

- *pvm_tidtohost*: Esta función es útil para determinar en qué nodo está ejecutandose determinada tarea.

- *pvm_config*: Regresa información acerca de la máquina virtual incluyendo el número de nodos.

- *pvm_tasks*: Regresa información acerca de las tareas *PVM* corriendo en la máquina virtual.

5.3.3 Funciones de Información.

Las funciones de configuración dinámica son funciones que permiten cambiar la configuración de la máquina virtual sin tener que detener su funcionamiento.

Estas rutinas son usadas algunas veces para configurar la máquina virtual, pero es más común que sean usadas para incrementar la flexibilidad y la tolerancia a fallas. Estas rutinas permiten a la aplicación incrementar la potencia de cómputo (agregando nodos) si determina que la aplicación necesita mayor capacidad de cómputo para resolverse.

- *pvm_addhosts*: Agrega un nodo a la máquina virtual.

- *pvm_delhosts*: Elimina un nodo de la máquina virtual.

5.3.4 Funciones de Empaquetamiento y Control de Buffers.

Las funciones de empaquetamiento y control de los *buffers* de mensajes son funciones muy utilizadas para el envío y recepción de mensajes, ya que primero se debe crear el *buffer* y luego empaquetar los datos, al momento de enviar, y, al momento de recibir, desempaquetar dichos datos.

En *PVM* existe sólo un *buffer* de envío activo y un *buffer* de recepción activo por proceso en determinado momento. El programador puede crear varios *buffers* en determinado momento, pero únicamente trabaja con uno para el empaquetamiento y envío de datos, ya que las funciones de empaquetamiento, envío, recepción y desempaquetamiento de datos afectan únicamente los *buffers* activos.

- *pvm_pkdatatype*: Empaqueta un tipo de dato dentro de un *buffer* (send) activo.

- *pvm_upkdatatype*: Desempaqueta un tipo de dato dentro de un *buffer* (receive) activo.

- *pvm-mkbuf*: Crea un nuevo *buffer* de envío vacío, especificando el método de codificación usado para los mensajes.

- *pvm-initsend*: Limpia el *buffer* de envío y crea uno nuevo, cuyo identificador devuelve, para empaquetar un nuevo mensaje.

5.3.5 Funciones de Envío de Mensajes.

En *PVM*, el envío de un mensaje comprende 3 pasos: primero el *buffer* de envío debe ser inicializado con la función *pvm-initsend* o *pvm-mkbuf*; segundo, el mensaje debe ser empaquetado dentro del *buffer* inicializado con la función *pvm_pkdatatype* y, tercero, se debe enviar el mensaje.

Para enviar un mensaje PVM, una tarea debe especificar el identificador de la tarea a la cual desea enviarle el mensaje (receptor) y el tag del mensaje. El tag es usado para

etiquetar el contenido del mensaje. Mediante la etiqueta del mensaje se identifica el mensaje para que el receptor identifique qué tipo de mensaje se está recibiendo.

Las funciones para el envío de mensajes son las siguientes:

- *pvm_send(int tid, int msgtag)* : Etiqueta el mensaje con un identificador entero *msgtag* y lo envía al proceso identificado por *TID*. Esta rutina no bloquea al proceso que la llama. La comunicación es asíncrona.

- *pvm_mcast(int *tids, int ntask, int msgtag)*: Etiqueta el mensaje con el identificador msgtag y lo difunde a todas las tareas especificadas en el vector tids de números enteros, cuya longitud viene dada por ntask.

La signatura de *pvm_send* es la siguiente:

int info = pvm_send(int tid, int msgtag)

Si se hizo el envío exitosamente, la variable *info* tomará el valor de 0. Si ocurre cualquier error, el valor de *info* será < 0.

El modelo de PVM garantiza lo siguiente acerca del orden de los mensajes: Si la tarea 1 envía un mensaje A a la tarea 2 y luego la tarea 1 envía un mensaje B a la tarea 2, el mensaje A llegará a la tarea 2 antes que el mensaje B.

5.3.6 Funciones de Recepción de Mensajes.

En *PVM,* un mensaje puede ser recibido por una función bloqueante o no y luego se debe desempaquetar los datos que vienen en el mensaje usando funciones de desempaquetamiento.

Las funciones de recepción de mensajes pueden aceptar mensajes enviados desde cualquier tarea, mensajes enviados desde una tarea específica, mensajes con un *tag* específico o mensajes con cualquier *tag.*

Las funciones de recepción de mensajes son las siguientes:

- *pvm_recv(int tid, int msgtag)* : Esta función de recepción de mensajes va a esperar hasta que un mensaje con la etiqueta *msgtag* llegue desde *TID*. Un valor de -1 en *msgtag* o en *TID* quiere decir que viene de cualquier tarea y con un *tag* que no interesa. Entonces pone el mensaje en un nuevo *buffer* de recibo que es creado. Esta función es bloqueante, es decir, la tarea detiene su ejecución, hasta que llegue el mensaje.

- *pvm_nrecv(int tid, int msgtag)* : Si el mensaje no ha llegado, entonces *bufid = 0* (parámetro que devuelve la función). Si un mensaje con la etiqueta *msgtag* ha llegado de *TID*, coloca el mensaje en un nuevo *buffer* de recibo activo y regresa el

identificador del *buffer*. Esta función es no bloqueante y puede ser llamada múltiples veces hasta que el mensaje haya llegado.

5.4 VENTAJAS Y DESVENTAJAS DE PVM

Una de las principales ventajas que ofrece *PVM* es poder configurar un gran número de computadores heterogéneos y hacer que puedan comportarse como uno solo, permitiendo así a los programadores desarrollar aplicaciones paralelas y aprovechar la capacidad de cómputo ofrecida por estos computadores. Además es un software de libre distribución que puede ser usado por todas las personas que deseen aprovechar sus características.

PVM ofrece una gran variedad de funciones que le ayudan al programador a conocer la arquitectura del cluster en el cual se encuentra trabajando, funciones para poder distribuir las tareas que desea sean ejecutadas en su aplicación. Sin embargo, esto es un poco de bajo nivel, ya que el programador no tiene por qué familiarizarse con la arquitectura del cluster; él debería únicamente estar preocupado por el desarrollo de su aplicación sin tener en cuenta características de hardware. Es decir, en PVM no se tiene una programación y distribución de tareas transparente, ya que se le debe especificar en las funciones el lugar en donde se desea sean ejecutadas las tareas. Dado esto, el programador debe conocer con anterioridad la forma como están configurados los nodos pertenecientes al cluster y el cluster en general.

Así que estudiando las principales características de PVM, nos damos cuenta que es una solución de bajo nivel para la programación sobre clusters que no permite que el programador solo se concentre en lo que tiene que hacer la aplicación a correr sobre el cluster, sino también conocer la arquitectura y número de nodos pertenecientes a éste.

6. CSP – COMMUNICATING SEQUENTIAL PROCESSES

En muchas ocasiones para realizar la programación sobre clusters, es necesario seguir un modelo de programación que ayude al programador a organizar mejor los componentes de la aplicación y la forma de interacción entre estos componentes, además de tomar en cuenta los problemas que se puedan presentar de comunicación y sincronización de las tareas o procesos que estén siendo ejecutados, por lo tanto en este capítulo se explica un modelo de programación que ha servido para definir aplicaciones distribuidas y concurrentes.

CSP es un modelo de programación que describe la computación concurrente y distribuida. Un programa desarrollado utilizando este modelo comprende un grupo de procesos que se comunican unos con otros a través de mensajes, por ejemplo, utilizando primitivas de *input* y *output* a través de canales de comunicación. La comunicación se ejecuta de forma asíncrona y el flujo de información, en un canal, es en una sola dirección. Un proceso que ejecuta una función de comunicación (*input u ouput)* se bloquea hasta que el proceso con el cual se está intentando comunicar ejecuta la correspondiente primitiva.

Entre las principales características que tiene CSP están:

- CSP encapsula los principios fundamentales de comunicación (se debe tener un emisor, un receptor y un mensaje a enviar).

- Se encuentra definido semánticamente en términos de un modelo matemático estructurado.

6.1 PRINCIPALES CARACTERISTICAS DE LOS PROCESOS, ELEMENTOS DEL MODELO Y LA COMUNICACION

El corazón del modelo CSP son los procesos y los canales, siendo los procesos piezas de código que pueden ser ejecutadas en secuencia o en paralelo y los canales el medio por el cual los procesos se comunican. Sin embargo, existen más elementos que ayudan en la declaración de variables, ejecución paralela, secuencial y condicionales, entre otros.

A continuación se explicará la forma como CSP hace el manejo de los procesos, las comunicaciones entre procesos y la sincronización, dado que son los puntos más importantes que CSP trata.

6.1.1 Manejo de los Procesos y las Comunicaciones.

Los procesos CSP son objetos o entidades que existen independientemente pero se pueden comunicar. Durante su tiempo de vida, un proceso puede ejecutar varias acciones. Entre las acciones que pueden ejecutar está la comunicación con otros procesos.

Los procesos en CSP se comunican nombrándose el uno al otro en sentencias *input* u *output.* En un *output*, se especifica el valor que va a ser tomado del canal; en un *input,* el valor a ser enviado al canal. La comunicación fluye de forma unidireccional: desde el proceso que efectúa el *output* hacia el proceso que efectúa el *input.*

En CSP la forma de representar un *Input* es ***?*** y la forma de representar un *Output* es **!.**

Por ejemplo, se supone que se tienen 2 procesos A y B. A desea recibir un valor desde B y colocarlo en la variable x. Entonces A ejecuta un *input* utilizando el comando ***B ? x***. B desea hacer el respectivo *output* del valor que puede ser enviado y ejecuta el siguiente comando ***A ! exp*** (donde *exp* puede ser una expresión que arroje un valor). La comunicación ocurre cuando los dos procesos están listos para ejecutar sus comandos. La ejecución de ***B ? x*** o ***A ! exp*** bloquea la ejecución del proceso hasta que el otro proceso ejecuta el comando correspondiente.

Adicionalmente a las primitivas de comunicación presentadas anteriormente, **?** (*input*) y **!** (*output*), CSP tiene declaración de variables, asignación de sentencias, secuencias, ejecución concurrente, repeticiones y condicionales. Por ejemplo, CSP delimita sentencias secuenciales con puntos y coma (**;**) y delimitas bloques de sentencias utilizando paréntesis (**[]**). Además para ejecuciones de procesos concurrentes y paralelas utiliza la el operador //. Cabe resaltar que los procesos que realizan las ejecuciones concurrentes no pueden compartir variables: CSP está orientado a comunicación por paso de mensajes, y no por variables compartidas.

A continuación se muestra un ejemplo de la forma como se comunican los procesos, se tiene el siguiente código CSP en el que se está declarando un proceso *C* que tiene un entero *n* al que le asigna el valor de 9 y luego le envía al proceso *D* el valor de 2 veces *n*. Si el proceso *D* ejecuta el *input* correspondiente (*C* ? *x* , siendo *x* la variable en la cual va a guardar el valor), el *output* termina y luego se le asigna a *n* el valor de 3.

```
C::     n :     integer;
        n :=    9;
        D !     2*n;
        n :=    3
```

Canales

Un canal es el principal mecanismo de comunicación entre procesos. El canal es el encargado de unir los procesos y sirve de intermediario entre ellos, además únicamente puede estar asociado a 2 procesos. Generalmente los canales son *zero-buffered* y punto a punto, sin embargo pueden definirse canales *uno a muchos*, *muchos a uno* y *muchos a muchos.*

Por otra parte, los canales pueden ser pasados de proceso en proceso, es decir, si un proceso está utilizando un canal y este proceso crea un subproceso, entonces el canal que tenía asociado puede ser pasado al subproceso para ser utilizado.

La forma como se expresa la asignación de canales a procesos en CSP es la siguiente:

P(c)
Q(c)

Los procesos P y Q comparten el canal C entre ellos. Los canales son usados dentro de los procesos como cualquier otro parámetro, por lo tanto pueden ser pasados a procesos creados a partir de otros procesos.

6.1.2 Manejo de la Sincronización.

Los procesos en determinado momento deben sincronizarse para efectuar tareas concurrentes o simplemente para finalizar, así que CSP define algunos elementos para permitir la sincronización de los procesos. A continuación se explicará la definición y uso de guardas, alternativas y barreras.

Guardas y Alternativas

Las sentencias condicionales son conocidas como guardas; se representan por →. La condición de la cláusula es una serie de expresiones booleanas que van antes de →. Si todas las expresiones son verdaderas, entonces el proceso ejecuta la acción de la guarda (la serie de expresiones que van después de →).

Por ejemplo, ***x ≥ y → max:= x.*** Si el valor de *x* es mayor o igual que el valor de *y*, entonces se le asigna a la variable *max* el valor de *x*.

Las alternativas son construidas uniendo sentencias de guardas encerradas en paréntesis (**[]**) y separadas por **||**. Una alternativa especifica la ejecución de uno de sus componente; si todos fallan, el comando falla, de otra forma, si un componente cumple la condición, se ejecuta.

Por ejemplo **[Q ? msg → <*Espera mensaje desde Q*> || R ? msg → <*Espera mensaje desde R*> || S ? msg -> <*Espera mensaje desde S*>].** El proceso puede esperar mensaje desde cualquiera de los 3 procesos, Q, R o S. Los comandos repetitivos son las mismas sentencias de alternativas precedidas por un ***.**

6.2 IMPLEMENTACIONES DE CSP

Existen diversas implementaciones del modelo antes explicado, a continuación explicaremos dos de ellas:

6.2.1 JAVA - JCSP.

Es una instancia de un tipo simple de dato (enteros, cadenas de caracteres, reales, fecha, hora, JCSP es una librería de Java que es una implementación del modelo de CSP. La versión actual, soporta concurrencia en una sola máquina virtual (puede ser un multiprocesador). Sin embargo se está trabajando en la creación de JCSP *Networking* que facilita la construcción de redes CSP a través de ambientes distribuidos como Internet. Detalles como direcciones IP, conexiones de *sockets*, entre otros, son ocultos para el programador JCSP, quien únicamente se preocupa por las primitivas CSP al momento de crear la aplicación. JCSP aprovecha los *java-threads* para la construcción de aplicaciones utilizando el modelo de CSP.

Los procesos en JCSP interactúan únicamente a través de primitivas de eventos como lo son los canales, las barreras, *timers* entre otros; no invocan métodos de otros procesos. Pueden combinarse para correr en secuencia o en paralelo. Además pueden ser combinados para ejecutar una serie de condiciones y eventos alternativos. Es decir, JCSP implementa todos los elementos del modelo CSP mencionados anteriormente.

6.2.2 C++ - C++CSP.

Según Neil Brown y Peter Welch en su artículo "An Introduction to the Kent C++CSP Library", C++CSP es una librería para C++ que combina un API orientado a objetos derivado de JCSP con algunas modificaciones para tomar ventaja de las capacidades de C++ (como los *templates*) y un soporte para el *kernel* [9].

La librería fue desarrollada como un proyecto en la Universidad de Kent y está escrita toda en C++.

Los procesos en C++CSP, al igual que en JCSP, interactúan entre ellos a través de primitivas de eventos y no invocan métodos de otros procesos. Pueden ejecutarse en paralelo o secuencialmente.
Pueden utilizar primitivas de sincronización como guardas, alternativas y barreras. Las barreras son eventos que se utiliza para sincronizar procesos. Los procesos que pertenecen a una barrera son bloqueados hasta que todos los procesos que pertenecen a la barrera terminan su ejecución.

Entre las desventajas que presenta el API de C++CSP se encuentra que únicamente soporta concurrencia en una sola máquina, sin embargo, se está terminando de desarrollar el API de C++CSP *Networked*.

Características principales del API de C++CSP

C++CSP adopta un acercamiento al diseño orientado a objetos para implementar CSP. Un proceso es una subclase de *CSProcess* y los canales son clases como *One2OneChannel* que es uno a uno *unbuffered*. Los canales son clases templates que pueden ser usadas para comunicar cualquier tipo de dato. En un escenario típico, un proceso padre crea un canal y pasa un *puerto de entrada* (también una clase *Chanin*) a un proceso y un *puerto de salida* (también una clase *Chanout*) a otro proceso. De esta forma, ningún proceso hijo puede acceder al canal directamente.

La librería C++CSP tiene: procesos, canales, alternativas, ejecuciones paralelas, barreras, objetos móviles entre otros elementos de CSP.

***Manejo de Procesos**:*
Un proceso *CSP* es un componente que encapsula estructuras de datos y algoritmos para la manipulación de datos. Los procesos interactúan entre sí a través de canales, y pueden ser ejecutados en forma secuencial o paralela.

Para crear un Proceso en el API de C++CSP se debe crear una clase que contenga los métodos que se deseen sean ejecutados por el proceso y algo muy importante, es que esta clase debe heredar de *CSProcess* y debe implementar el método *run*, el cual tiene el código de implementación de lo que se quiere hacer con el proceso.
La definición de un proceso se realiza de la siguiente forma:

class processA : public CSProcess

Como se dijo anteriormente, los procesos pueden ser ejecutados de forma secuencial o paralela, esto se hace utilizando unas clases especiales creadas en el API de C++CSP, estas clases son Sequential y Parallel.

Para definir una ejecución secuencial, es necesario listar, en el constructor S*equential*, los procesos que se desean que corran secuencialmente, un proceso no comienza a ejecutarse hasta que otro no haya finalizado.

Para una ejecución paralela, al igual que en Sequential, es necesario listar en el constructor de *Parallel* los procesos que se desean que corran en paralelo, la ejecución del *parallel* finaliza cuando todos los procesos que se ejecutaron en paralelo han finalizado y la aplicación sigue corriendo de forma normal.

A continuación se explica la clase principal para el manejo de procesos en el presente API.

CSProcess [10]:
Es la clase que define los procesos y una de las más importantes de la librería.

Un proceso tiene estructuras de datos y algoritmos que manejan esos datos. Estos son privados y no pueden ser accedidos por otros procesos.

- Uso:

Para usar esta clase, es necesario extenderla e implementar el método *run()*, el cual provee la funcionalidad del proceso. Esta implementación debe ser hecha por el programador.

Los canales que se van a utilizar deben ser pasados por el constructor, al igual que los parámetros que se van a utilizar.

- Componentes de la clase:

Tabla 1. Componentes de la clase CSProcess

MÉTODO	DESCRIPCIÓN
CSProcess (const stackCommit, const stackReserve=0)	El constructor recibe como parámetro el tamaño de la pila del proceso. No existe un número fijo para este tamaño, si se da un tamaño muy pequeño, el programa puede fallar y dejar de ejecutar. Si es muy alto, no sucede esto, pero puede requerir más memoria.
virtual void run()	Este método contiene la funcionalidad del proceso. Cuando se extiende de la clase, se tiene que implementar este método.
void yield ()	Este método puede ser llamado desde el método *run*() para tener un control del proceso.
void sleepTo (const Time &sleepUntil)	Este método permite al proceso dormir hasta determinado tiempo. El parámetro indica hasta cuando.
void sleepFor (const Time &sleepFor)	Este método permite al proceso dormir por una cantidad de tiempo especificado. El parámetro indica el tiempo que se desea que el proceso duerma.
bool CSP_API Start_CSP(const csp::Time &t, Logger::LoggerImpl *impl)	Este método, inicia el sistema *CSP*, debe ser llamado antes de usar el sistema. Si el sistema fue iniciado devuelve un valor de true. Si se va a ejecutar un *Parallel o Sequential*, no es necesario inicializar.
void CSP_API End_CSP ( )	Este método finaliza el sistema; debe ser llamado al finalizar la programación.

***Manejo de Comunicaciones**:*
El manejo de las comunicaciones en C++CSP se hace a través de canales, un canal es el principal mecanismo de comunicación entre procesos. Una comunicación normal incluye un proceso escribiendo en el canal y otro leyendo del canal, también se puede tomar como una forma distribuida de asignación, un valor es computado en un proceso y es asignado a una variable en otro proceso.

El API de C++CSP define diferentes tipos de canales: Canal uno a uno, uno a muchos, muchos a uno y muchos a muchos.

Cada uno de estos canales tienen puertos de entrada y puertos de salida que son utilizados por los procesos para comunicarse, estos puertos son llamados *Channels ends o puertos.*

- *One2OneChannel* [11]*:*
  Es un canal de comunicación uno a uno.

  Los canales son usualmente creados y luego se pasan su puerto de lectura o puerto de escritura por el constructor del proceso. El canal en sí mismo no se puede usar, se usan sus puertos.

  - Componentes de la clase:

Tabla 2. Componentes de la clase One2OneChannel

MÉTODO	DESCRIPCIÓN
Chanin <DATA_TYPE> reader()	Obtiene un puerto de lectura para el canal.
Chanout <DATA_TYPE> writer()	Obtiene un puerto de escritura para el canal.

Existen varias clases similares a esta, con funcionalidad parecida, estas clases heredan las características de la clase descrita anteriormente.

Figura 9. Diagrama de clases de los canales C++CSP

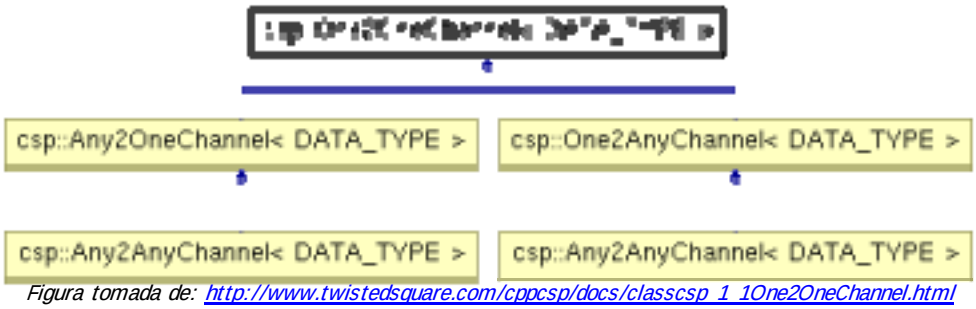


Figura tomada de: http://www.twistedsquare.com/cppcsp/docs/classcsp_1_1One2OneChannel.html

- *One2AnyChannel:*
  Con esta clase se pueden distribuir puntos de lectura del canal a múltiples procesos.

- *Any2OneChannel*:
  Con esta clase se pueden distribuir puntos de escritura del canal a múltiples procesos.

- *Any2AnyChannel*:
  Con esta clase se pueden distribuir los puntos de lectura y escritura del canal a múltiples procesos.

Los puertos de entrada y salida de los canales son clases llamadas *Chanin* y *Chanout*.

- *Chanin* [12]**:**
  Es el punto de lectura de un canal.

  Puede ser usado para *inputs* normales. Se usa de la siguiente manera.

```
One2OneChannel<int> chan;
Chanin<int> in = chan.reader();
int a,b;
in.input(&a);
in.input(&b);
```

  Primero se debe definir el tipo de canal a ser utilizado, en este caso es un canal uno a uno, al tener el canal definido, se obtiene el reader (puerto de lectura del canal) y luego se realiza el input de la variable que se desee.

  - Componentes de la clase:

Tabla 3. Componentes de la clase Chanin

MÉTODO	DESCRIPCIÓN
void input (DATA_TYPE *const dest)	Ejecuta un input al canal. El parámetro *dest*, indica dónde se guarda el input. No debe ser nulo.

- *Chanout* [13]*:*
  Es el punto de escritura de un canal.

Puede ser usado para *outputs* normales. Se usa de la siguiente manera.

```
One2OneChannel<int> chan;
Chanout<int> out = chan.writer();
int a,b;
out.output(&a);
out.output(&b);
```

Primero se debe definir el tipo de canal a ser utilizado, en este caso es un canal uno a uno, al tener el canal definido, se obtiene el writer (puerto de escritura del canal) y luego se realiza el output y se guarda en una variable.

- Componentes de la clase:

Tabla 4. Componentes de la clase Chanout

MÉTODO	DESCRIPCIÓN
void output (DATA_TYPE *const dest)	Ejecuta un output del canal. El parámetro *dest* indica dónde va a ser guardado el valor.

Manejo de la Sincronización:
Los procesos en determinado momento necesitan sincronizarse para ejecutar alguna tarea al mismo tiempo, C++CSP tiene algunos componentes que ayudan al programador a realizar la sincronización. Entre estos componentes se tiene:

- *Barreras* [14]*:*
  Una barrera es un evento que se utiliza para sincronizar procesos. Los procesos que pertenecen a una barrera son bloqueados hasta que todos los procesos que pertenecen a la barrera terminan su ejecución.

  - Componentes de la clase:

Tabla 5. Componentes de la clase Barrier

MÉTODO	DESCRIPCIÓN
virtual void reset (const int setEnrolled)	Resetea la barrera. Elimina de la barrera todos los procesos que se hayan ligado a ella.
virtual void enroll (const int toEnroll=1)	Liga más procesos a la barrera. El valor de *toEnroll* es el número de procesos a ligar a la barrera.
virtual void sync ()	Sincroniza los procesos que se encuentran ligados en la barrera.

- *Alternativas* [15]*:*
  Permite a un proceso esperar a que se cumpla un determinado evento (*guard).* Un ALT es una primitiva que toma una lista de condiciones o eventos (guardas) y retorna cuando una de esas condiciones está lista. Típicamente, se puede tomar como guardas *inputs* a canales o *timeouts*.

  - Componentes de la clase:

Tabla 6. Componentes de la clase Alternativa

MÉTODO	DESCRIPCIÓN
unsigned int virtual void priSelect ()	Retorna un PRI ALT, un PRI ALT retorna cuando una de las guardas está activa, si hay más de una activa, entonces selecciona la que primero se encuentre en el arreglo. Retorna el valor del index de la guarda seleccionada
unsigned int sameSelect ()	Retorna la guarda que fue activada de último.
Guard * replaceGuard (unsigned int index, Guard *guard)	Reemplaza una guarda en el arreglo. Este método es útil cuando se quiere reemplazar una guarda de timeout sin necesidad de crear un nuevo alt. *Index:* El índice de la guarda a reemplazar. *Guard:* La nueva guarda.

Ejemplo de Aplicación del API de C++CSP

A continuación se presenta un ejemplo tomado del API de C++CSP donde se hace una definición de 3 procesos, la definición de sus canales y respectivos *channels ende (puertos de lectura y escritura)*, además de la definición de ejecución paralela y secuencial de dichos procesos.

Se definen 3 clases, una por cada uno de los procesos y la clase proceso que los une realiza la ejecución.

Tabla 7. Ejemplo de C++CSP

```
class OneIOneO : public CSProcess
{
       chanin<int>  in;
       chanout<int>  out;
       int id;
```

```
protected:
	void run()
	{
		int n;
		in >> n;
		out << id;
	}

public OneIOneO(const chanin<int>& i, const chanout& o, int n)
		:	CSProcess(65536*16), in(i), out(o), id(n)
		{
		};
}



class OneOOneI : public CSProcess
{
	chanin<int>  in;
	chanout<int>  out;
	int id;
protected:
	void run()
	{
		int n;
		out << id;
		in >> n;

	}

public OneOOneI(const chanin<int>& i, const chanout& o, int n)
		:	CSProcess(65536*16), in(i), out(o), id(n)
		{
		};
}


class CompositeIO : public CSProcess
{
	int id;
protected:
	void run()
	{
		One2OneChannel<int> c0, c1;

		Parallel ( CSP_NEW OneIOneO(c0.reader(), c1.writer(), id),
			   CSP_NEW OneOOneI(c1.reader(), c0.writer(), id),
			   NULL);

	}

public CompositeIO (int i) :	CSProcess(65536*16), id(n)
		{
		};
};


void SimpleCommsTest()
{
	Start_CSP();
```

```
        Sequential (
                CSP_NEW CompositeIO(0),
                CSP_NEW CompositeIO(3),
                CSP_NEW CompositeIO(6),
                NULL);

        Parallel (
                CSP_NEW CompositeIO(10),
                CSP_NEW CompositeIO(20),
                CSP_NEW CompositeIO(30),
                NULL);

        End_CSP();
}
```

En la clase *OneIOneO*, se define un puerto de entrada *in* y un puerto de salida *out*, en el método run se usa el puerto de entrada para colocar en el canal el valor de *n (in >> n)* y se saca del canal con el puerto de salida el valor de *id* y se coloca en la variable *id (out << id)*.

En la clase *OneOOneI*, se define un puerto de salida para tomar el valor que hay en el canal y colocarlo en la variable *id* y se usa el puerto de entrada para colocar en el canal el valor de *n*.

En la clase *CompositeIO* se instancia el tipo de canal, *One2OneChannel* y se definen 2 canales de este tipo *(c0 y c1*), además se define una ejecución paralela de los procesos *OneIOneO* y *OneOOneI*. Como podemos ver, en el constructor de *parallel*, se pasan los procesos y en el constructor de cada uno de los procesos se pasan los puertos de entrada y salida del canal.

En el método *SimpleCommsTest*(), se inicializa la ejecución de los procesos y se ejecuta *CompositeIO* de manera secuencial y paralela.

El ejemplo presentado anteriormente, es un ejemplo sencillo que ilustra la definición de los principales componentes del API de C++CSP.
A medida que avanzamos en los capítulos, hemos podido observar que existen diversas herramientas creadas para aprovechar las ventajas que ofrece un cluster, sin embargo siempre hemos encontrado algunas desventajas de estas herramientas; en los siguientes capítulos veremos que haciendo una extensión del modelo de CSP y tomando las ventajas de las herramientas vistas podemos generar una solución interesante para el problema de la programación sobre clusters.

7. EXTENSION DEL MODELO DE CSP

En las secciones anteriores se pudo observar que los *clusters* ofrecen algunas ventajas como son: el poder compartir recursos, la buena relación costo/rendimiento, la tolerancia a fallos y el manejo de paralelismo, entre otras.

Cuando se quiere utilizar un *cluster,* para aprovechar el desempeño del mismo, es necesario que el programador tome en cuenta el paralelismo y la concurrencia en el momento de desarrollar las aplicaciones; debe conocer la arquitectura del *cluster*, la configuración, el número de nodos y procesadores que lo conforman. Algunas veces es necesario realizar la programación basado en esto y dejar inmerso en el código muchos de estos datos; además es necesario que familiarizarse con librerías de paso de mensajes de bajo nivel como *PVM*.

La idea principal de la extensión del modelo *CSP,* y la definición de un *API* para la programación sobre *cluster,* es brindarle al programador una forma transparente de realizar su programación; sin tener que preocuparse por muchos detalles que a la hora de programar no son realmente importantes.

Se sugiere que el programador defina los componentes de su programa y la forma de interacción entre ellos, dejando a un lado si la aplicación se va a correr en uno, dos o más nodos del *cluster*. Por aparte, el programador puede definir qué procesos desea que corran en el nodo local y cuáles desea que se ejecuten en otro nodo del *cluster*, sin tener que especificar cuál, dejando así que el balanceo de carga sea realizado por el *cluster* (*Openmosix*).

Para la creación del *API* de programación, se tomó como base el modelo de programación concurrente *CSP* desarrollado por *C.A.R. Hoare*, ya que este ofrece la facilidad de definición de procesos e interacción entre ellos. Además se utilizó la implantación en *C++* del modelo *C++CSP*, para tomar algunos componentes básicos como lo son los procesos, canales, barreras entre otros; sin embargo, dado que el *API* original de *C++CSP* únicamente soporta concurrencia dentro de un solo nodo, fue necesario modificar algunas clases y crear otras para permitir un ambiente distribuido utilizando el paso de mensajes de PVM.

La unión de todos estos componentes dio como resultado la definición de un *API* de programación que ayuda a desarrollar aplicaciones paralelas de un modo más sencillo y transparente y lograr que dichas aplicaciones corran en un ambiente paralelo y distribuido. Además se utilizaron las ventajas de *Openmosix* para permitir realizar balanceo de carga sobre el cluster

Esta definición permite al programador crear procesos locales o procesos distribuidos, dependiendo de dónde quiera que se ejecuten. Además esta localización de procesos se puede redefinir sin necesidad de que afecte el diseño de la aplicación.

7.1 CARACTERISTICAS DEL MODELO

El modelo propuesto para extender el modelo original de CSP se centra en el manejo de los procesos, las comunicaciones y la sincronización entre procesos, debido a que hay que tener en cuenta que los procesos pueden estar corriendo en nodos diferentes del *cluster* y, por lo tanto, se debe tomar en cuenta la localización para poder comunicar los procesos.

A continuación, se analizarán más a fondo cómo se realiza el manejo de los procesos, las comunicaciones y la sincronización en la extensión del modelo de CSP.

7.1.1 Manejo de Procesos.

Como vimos en los capítulos 5 y 6, un proceso *CSP* es un componente que encapsula estructuras de datos y algoritmos para la manipulación de datos. Los procesos interactúan entre sí a través de canales, y pueden ser ejecutados en forma secuencial o paralela.

La extensión del modelo permite tener 2 tipos de procesos: *procesos locales y procesos distribuidos;* debido a que, como se comentó anteriormente, se le permite al programador especificar los procesos que desea que se ejecuten de manera distribuida, es decir, en un nodo diferente al que se está corriendo la aplicación. Aunque no necesariamente un proceso distribuido se ejecutará en un nodo diferente al que se está corriendo la aplicación.

Fue necesario diferenciar entre los dos tipos de procesos, ya que la forma de manejar cada uno de ellos es diferente; los procesos *locales* utilizan el *API* de *C++CSP* para comunicarse con otros procesos locales y los procesos *distribuidos* deben ejecutar algoritmos de localización para poder establecer comunicación entre procesos *local – distribuido y distribuido – distribuido*.

Para manejar los procesos *distribuidos*, fue necesario utilizar como base algunas clases del *API* de *C++CSP* y crear nuevas clases que permitieran la comunicación entre los diferentes tipos de procesos, dependiendo de dónde se encontrarán ejecutándose.

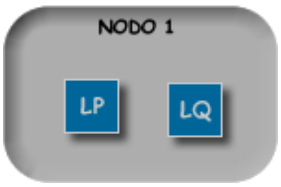
A continuación se definirán cada uno de los tipos de procesos y la forma de representación en el modelo.

Procesos Locales

Se define como proceso *local,* un proceso que se ejecuta en el mismo nodo en el cual fue lanzado. En este caso, estos procesos son ejecutados como threads.

Para facilitar la identificación de cada uno de los procesos en las siguientes gráficas, un proceso *local* se identifica con la notación *Lproceso*. Por ejemplo, si se tiene un proceso *P* que se quiere que sea local, entonces la forma de identificarlo es *LP*.

Figura 10. Proceso Local



En la Figura 10 presentada, se tienen 2 procesos *locales* en el Nodo1.

Para crear un proceso *Local,* el programador debe seguir los siguientes pasos:

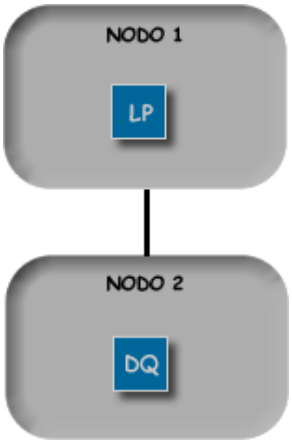
- Crear una clase que herede de la clase CSProcess del API de C++CSP.
- Crear en esta clase todos los atributos y métodos e implementar el método *run*().

Procesos Distribuidos

Se define como proceso *distribuido,* un proceso que se lanza en el nodo local y se ejecuta en el mismo nodo o en un nodo diferente pero con un identificador de proceso diferente.

Un proceso distribuido se identifica con la notación *Dproceso*. Por ejemplo, si se tiene un proceso *Q* que se quiere que sea distribuido, entonces la forma de identificarlo es *DQ*.

Figura 11. Proceso Distribuido



En la Figura 11 se tienen 2 procesos: un proceso *local* P en el nodo 1 y un proceso *distribuido* Q que se está ejecutando en el nodo 2.
Sin embargo, el proceso *Q (DQ)* puede también estarse ejecutando en el nodo 1 y ser un proceso distribuido. En este caso, contrario a un proceso local, el proceso CSP es un proceso nuevo de Unix y OpenMosix puede migrar o no dicho proceso a otro nodo del cluster.

Para crear un proceso *distribuido*, el programador debe seguir estos pasos:

- Se debe crear un proceso local.
- Esta clase (proceso) se debe pasar por el constructor a una nueva clase llamada *CSProccesDistributed*; esta clase se encargará de duplicar el proceso, creando un nuevo proceso Unix que puede ser o no migrado a otro nodo del cluster; sin embargo por ser un proceso Unix diferente al inicial, se considera como un proceso distribuido.

El nuevo proceso continuará su ejecución y el padre del proceso esperará hasta que su hijo finalice para finalizar la ejecución.

Parallel

Como se mencionó en el capítulo 6, existen varias formas de ejecutar procesos: una de ellas es ejecutarlos en paralelo. Para definir una ejecución paralela, es necesario listar en el constructor de *parallel* los procesos que se desean que corran en paralelo. En la extensión del modelo, un *parallel* puede tener procesos *locales y distribuidos* en la misma ejecución.

La ejecución del *parallel* finaliza cuando todos los procesos que se ejecutaron en paralelo han finalizado y la aplicación sigue corriendo de forma normal.

Por ejemplo, si se desea que 2 procesos A y B se ejecuten en forma paralela, entonces se deben seguir los siguientes pasos:

- Crear las clases A y B que hereden de CSProcess y tengan sus atributos y métodos necesarios para la ejecución.
- En el constructor *Parallel* pasar las 2 clases A y B así:

Parallel (new A, new B)

- Parallel solo finaliza su ejecución cuando tanto el proceso A como el proceso B hayan finalizado la ejecución.

En el caso que el programador desee que el proceso B se ejecute paralelo a A y que sea un proceso distribuido, se deben seguir los siguientes pasos:

- Crear las clases A y B que hereden de CSProcess y tengan sus atributos y métodos necesarios para la ejecución.
- En el constructor de *Parallel* se pasan los procesos A y B asi:

Parallel (new A, CSProcessDistributed(new B))

- El proceso A se ejecuta en el nodo local y *CSProcessDistributed* hace un *fork()*[1], por lo tanto el proceso B puede ser migrado para ser ejecutado en otro nodo del cluster. Sin embargo el proceso padre del proceso B migrado queda a la espera de que el hijo termine su ejecución para poder finalizar: el padre sólo termina cuando el hijo ha finalizado.

- Parallel solo finaliza su ejecución cuando tanto el proceso A como el proceso B han finalizado la ejecución.

Sequential

Otra forma de ejecutar procesos es de forma secuencial; un proceso no comienza a ejecutarse hasta que otro no haya finalizado. Para definir una ejecución secuencial, es necesario listar, en el constructor, los procesos que se desean que corran secuencialmente. Un *sequential* puede tener procesos *locales y distribuidos* en la misma ejecución.

En este caso la forma de ejecutarse es la siguiente:

- Crear las clases A y B que hereden de CSProcess y tengan sus atributos y métodos necesarios para la ejecución.
- En el constructor *Sequential* pasar las 2 clases A y B así:

Sequential (new A, new B)

- La clase A comienza su ejecución cuando finalice, la clase B inicia su ejecución.
- Sequencial solo termina su ejecución cuando el último proceso que se colocó en el constructor finaliza su ejecución.

Al igual que en el Parallel, si el programador desea que el proceso B se ejecute secuencial pero que sea un proceso distribuido, se deben seguir los siguientes pasos:

- Crear las clases A y B que hereden de CSProcess y tengan sus atributos y métodos necesarios para la ejecución.
- En el constructor de *Sequential* se pasan los procesos A y B asi:

Sequential (new A, CSProcessDistributed(new B))

- El proceso A se ejecuta en el nodo local y cuando ha finalizado su ejecución, CSProcessDistributed hace un *fork()* del proceso B y este puede ser migrado para ser ejecutado en otro nodo del cluster. Sin embargo el proceso padre del proceso B

[1] La forma de ejecutarse del CSProccesDistributed es descrita más adelante en el capítulo de diseño.

migrado queda a la espera de que el hijo termine su ejecución para poder finalizar: el padre sólo termina cuando el hijo ha finalizado.

- *Sequential* solo finaliza su ejecución cuando el proceso B ha finalizado la ejecución.

7.1.2 Manejo de las Comunicaciones.

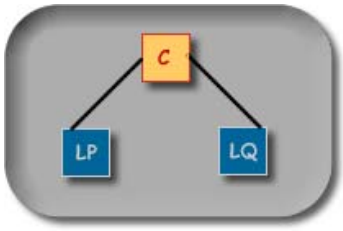
Comunicaciones

En el modelo extendido la comunicación entre los procesos es transparente para el programador: para él, comunicar dos procesos locales es igual que comunicar un proceso local y uno distribuido o dos procesos distribuidos. Sin embargo, al interior del modelo existen 3 formas de comunicación:

- *Local - Local:*
  Esta forma de comunicación se da entre procesos que se encuentran ejecutándose en un mismo nodo, y son dos threads de un mismo proceso Unix; entre dos procesos *locale.*

  En la **¡Error! No se encuentra el origen de la referencia.** se muestra la interacción de estos procesos.

  Figura 12. Procesos Local - Local

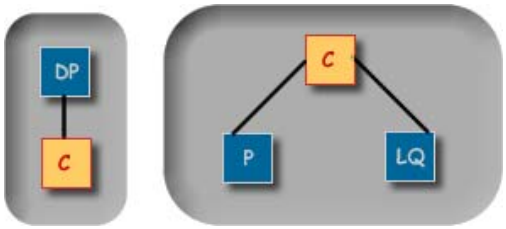
  

  Ambos procesos son locales y utilizan el mismo canal. Este tipo de comunicación se maneja de igual forma como lo hace el *API* de *C++CSP*: a través de *inputs* y *outputs* a un canal.

- *Local – Distribuida*:
  Esta forma de comunicación se da entre un proceso *local* y un proceso *distribuido.* Para realizar el manejo de esta comunicación, es necesario extender la funcionalidad del *API* de *C++CSP.*

  En la Figura 13 se muestra la interacción de estos procesos. Se tienen 2 procesos P y Q. P es un proceso distribuido y Q es un proceso Local.

Figura 13. Procesos Local - Distribuido



El proceso P se pasa por el constructor de CSProcessDistributed, éste hace un fork() y se crea la copia del proceso como otro proceso del sistema. El proceso hijo de P —la copia— continúa la ejecución y el padre —P— queda en espera hasta que el proceso hijo finalice; el proceso hijo puede ser ejecutado en el mismo nodo o en otro nodo del cluster (depende de Openmosix). En la figura 4 se puede ver que P sería el proceso padre y DP el proceso hijo.

Debido a que el proceso *DP* puede ser migrado y además es un proceso nuevo del sistema, es necesario establecer su ubicación para poder inicializar la comunicación con el proceso Q.

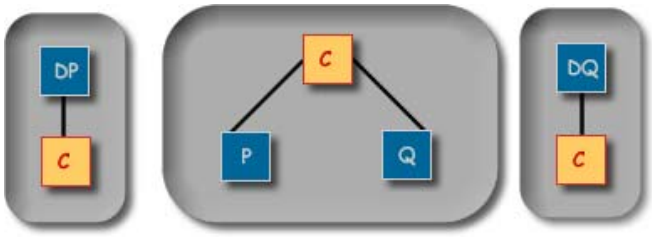
Como se puede observar en la figura, cuando se crea copia del proceso, también se crea copia del canal y esta también puede ser migrada junto con el proceso; por lo tanto, cuando el proceso Q quiere comunicarse con el proceso P (en este caso DP), normalmente se comunicaría con el canal C, pero, en este caso, DP no recibiría el mensaje; por esto es necesario crear componentes que permitan crear la conexión entre estos procesos.

- *Distribuida – Distribuida*:
  Esta forma de comunicación se da entre dos procesos *distribuidos.* Igual que el tipo de comunicación anterior, es necesario extender la funcionalidad del *API* de *C++CSP.*

  Para estos dos tipos de comunicaciones, se maneja la misma filosofía: los procesos son nuevos procesos del sistema y es necesario establecer la ubicación de cada uno de ellos para poder iniciar la comunicación.

  En la Figura 14 se muestra la interacción de estos procesos.

Figura 14. Procesos Distribuido - Distribuido



Los procesos *P* y *Q* se duplican para que puedan ser *distribuidos*, y se crean *DP* y *DQ*, los cuales son nuevos procesos del sistema y pueden ser migrados a otros nodos del *cluster* (depende de Openmosix). Estos procesos migran con una copia de su canal.
Es necesario establecer la ubicación de cada uno de ellos para poder inicializar la comunicación.

Canales

Un canal es el principal mecanismo de comunicación entre procesos. Una comunicación normal incluye un proceso escribiendo algo en el canal y otro leyendo del canal.

Dado que el *API* de *C++CSP* soporta concurrencia únicamente en un solo nodo, y que el comportamiento de las comunicaciones entre 2 procesos *locales* no es el mismo que el comportamiento entre un proceso *local* y otro *distribuido* o entre 2 *distribuidos*, fue necesario crear clases similares a las clases *Chanin y Chanout* de C++CSP para poder establecer las comunicaciones remotas que se dan entre procesos creados por *fork()* y que pueden encontrarse en nodos diferentes.

Estas clases son llamadas InputChannel y OutputChannel e identifican la localización de los procesos, cuando tienen cada uno de los procesos localizados definen la forma de comunicación entre ellos y ejecutan la comunicación, su función es hallar la localización del proceso compañero en la comunicación. Si la forma de comunicación es Local – Local, como se dijo anteriormente, utiliza la funcionalidad de Chanin y Chanout, si es una forma de comunicación Local – Distribuida y Distribuida – Distribuida, utiliza la funcionalidad del API extendido.

Se pretende que para el programador sea transparente la forma de comunicación que se está utilizando, *local – local, local – distribuida o distribuida – distribuida.*

Al momento de realizar el diseño de la aplicación, sólo se necesitan definir los procesos y los canales presentes, sin necesidad de especificar qué forma de comunicación se desea manejar.

- *Limitantes*

A continuación se van a listar las limitantes que presentan los canales al momento de manejar comunicaciones distribuidas:

- Los canales envían únicamente datos primitivos: si el programador desea enviar objetos a través del canal, se debe encargar él mismo de realizar el *marshalling/unmarshalling*.

- Los canales después de ser inicializados, no pueden ser pasados a otro proceso, debido a que, cuando un canal es inicializado, se establece su localización, y, si el canal se envía a otro proceso que puede ser distribuido, su localización cambia y puede generar un error de comunicación.

7.1.3 Manejo de la Sincronización.

Barreras

Al igual que en C++CSP, una barrera es un evento que se utiliza para sincronizar procesos. Los procesos que pertenecen a una barrera son bloqueados hasta que todos los procesos que pertenecen a la barrera terminan su ejecución.

Tanto procesos locales como procesos distribuidos pueden pertenecer a una barrera. Si se tienen 2 procesos LA (local) y DB (distribuido) que se encuentran en una barrera X, si LA finaliza su ejecución y DB aún no finaliza, entonces LA se bloquea hasta que DB finalice.

Alternativas

Permite a un proceso esperar a que se cumpla un determinado evento (*guard).* Como se vio en el capítulo 6, un ALT es una primitiva que toma una lista de condiciones o eventos (guardas) y retorna cuando una de esas condiciones está lista. Típicamente, se puede tomar como guardas *inputs* a canales o *timeouts*.

7.2 INTERACCION DE LOS ELEMENTOS EN EL MODELO

Después de haber definido los elementos principales del modelo extendido de *CSP* es importante explicar la interacción existente entre ellos, por lo tanto se va a presentar un escenario donde se ilustre la interacción de los elementos del modelo.

Considere la aplicación definida en el capítulo anterior, en el que se tienen 3 procesos, dos procesos que leen y escriben de un canal (*OneIOneO y OneOOneI)* y otro proceso que hace una ejecución secuencial y paralela de los dos procesos anteriores (*CompositeIO)*.

Hacemos una variante y ahora se desea que el proceso *OneOOneI* sea un proceso distribuido, por lo tanto la definición de dicho proceso en la clase *CompositeIO* se hace de la siguiente forma:

Tabla 8. Ejemplo modelo extendido CSP

```
class CompositeIO : public CSProcess
{
        int id;
        protected:void run()
        {
                One2OneChannel<int> c0, c1;

                Parallel (
                        NEW OneIOneO(c0.reader(), c1.writer(), id),
                                CSProcessDistributed    (NEW    OneOOneI(c1.reader(),
                                c0.writer(), id), NULL);

        }
```

Al pasar el proceso OneOOneI por el constructor de *CSProcesessDistributed,* se crea una copia del proceso: la cual puede migrar a otro nodo del cluster, por lo tanto es necesario que, en el momento de querer comunicarse, se haga la localización de los procesos.

Además se deben definir nuevos canales y nuevos puertos de entrada y salida para manejar las diferentes formas de comunicación. En el siguiente capítulo se explican estas nuevas definiciones.

Pueden existir diversos escenarios posibles al momento de realizar la programación de una aplicación utilizando el modelo extendido de *CSP*, sin embargo, lo más importante para tener en cuenta es la localización de los procesos. Cuando los procesos se encuentran en nodos diferentes del *cluster* o son dos procesos diferentes del sistema, utilizan comunicación distribuida, pero, si se encuentran en el mismo nodo, se comunican localmente usando el API de C++CSP.

En el siguiente capítulo se estudiará el diseño utilizado para el modelo extendido de *CSP*, la forma de expresar las comunicaciones locales y distribuidas, los proceso locales y distribuidos entre otros componentes pertenecientes al modelo.

8. DISEÑO E IMPLEMENTACION DEL MODELO EXTENDIDO

En el capitulo anterior se observaron las principales características del modelo extendido de *CSP* para poder crear aplicaciones concurrentes y distribuidas y se explicó la forma de interacción entre los componentes. A continuación se explicará de una forma más detallada cada uno de los componentes creados a partir de este modelo.

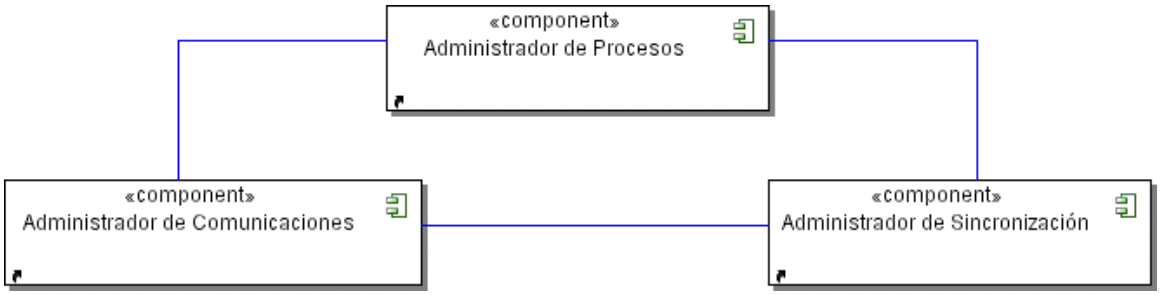
Al momento de realizar el diseño del modelo se tuvieron en cuenta los elementos explicados en los capítulos anteriores: se tomaron funcionalidades de *OpenMosix* y *PVM* y se desarrolló un diseño basado en el modelo CSP expuesto en el capitulo anterior. Se tomaron las principales características de los elementos mencionados anteriormente y se integraron para que interactuaran entre sí y ofrecer una solución de más alto nivel para realizar programación sobre clusters de alto desempeño.

8.1 COMPONENTES DEL API

En el modelo general, podemos tomar 3 componentes principales, estos son: Administrador de procesos, Administrador de comunicaciones, y Administrador de sincronización.

En la Figura 15 se puede observar la forma como se relacionan los componentes nombrados anteriormente.

Figura 15. Componentes del API extendido de CSP



Los tres componentes interactúan entre sí para lograr la definición del API que permita al programador la transparencia en la localización de procesos, ejecución de sentencias para procesos locales y distribuidos entre otras.

Según lo explicado en el capítulo anterior, se necesitan componentes que definan los procesos distribuidos, componentes que comuniquen los procesos, tanto distribuidos y locales y componentes que manejen la sincronización de los procesos.

Para lograr la definición de estos componentes, se tomaron las principales funcionalidades del API de C++CSP y se agregaron clases que permitieran cumplir estas tareas. Además estas nuevas clases van a utilizar PVM para el envío de mensajes entre procesos, comunicarlos cuando la forma de comunicación sea *local – distribuida* y *distribuida – distribuida* y para la localización de dichos procesos.

La filosofía de la definición del nuevo API se centra en:

1. Se crean los procesos.

2. Se crea el canal con un Identificador, en el constructor del canal se crean los puertos de lectura y escritura, se les envía por constructor el identificador y se registran en una clase construida llamada *PortDirectory*.

3. En el constructor de cada uno de los puertos, se les asigna un identificador (este identificador es dado por *PVM* y lo guarda en una variable *parentTID*) y se guarda en una variable el identificador del canal al que pertenece (viene por el constructor), además se inicializan variables que le van a permitir identificar donde se encuentran localizados los procesos (*myTID*, *otherTID*, entre otras).

4. Cuando los procesos crean el canal y piden los puertos de lectura y escritura, el canal los devuelve.

5. Se definen si estos procesos son locales o distribuidos (si el proceso es distribuido se hace un *fork()*). Si es distribuido, es necesario pasarlo por el constructor de la clase *CSProcessDistributed.*

6. Cuando se realiza un *Input* o un *Output*, se debe realizar una localización de los procesos (con el fin de saber si son locales o distribuidos), el puerto que esté trabajando en ese momento, ejecuta el método llamado *validar*() el cual asigna a la variable *myTID* un identificador de *PVM*, si el proceso es distribuido, el *TID* es diferente al guardado anteriormente en la variable *parentTID.*

7. Este método *validar()*, lo que hace es asignar los valores a cada una de las variables de los puertos de entrada y salida para tener la localización de los procesos. Si la variable *myTID* es diferente de *parentTID*, significa que se hizo un *fork()*.

8. Si los procesos son locales, se usan las sentencias de C++CSP, de lo contrario se usan las sentencias del nuevo API.

9. Si los procesos van a utilizar el nuevo API para la comunicación, se usan los TIDs asignados para el envío y recepción de los mensajes.

A continuación se explica con más detalle cada uno de los componentes del API definido y la definición de las clases creadas.

8.1.1 Administrador de Procesos.

Este componente es utilizado para manejar los procesos; contiene las clases encargadas de hacer la creación de los procesos.

Como se vio en el capitulo anterior, en la extensión del modelo, se definen 2 tipos de procesos: *locales y distribuidos.*

Los procesos locales son manejados y creados usando el API de C++CSP y los procesos distribuidos son creados a partir de una nueva clase llamada *CSProcessDistribuited*.

En este componente se encuentran las siguientes clases:

CSProcess

Una de las clases principales del API de C++CSP, explicada con más detalle en el capítulo 6. Es utilizada para crear los procesos.

CSProcessDistributed

Define los procesos *distribuidos.*

Es un proceso CSP, por lo tanto según lo explicado en el capítulo anterior hereda de *CSProcess*. En la implantación del método *run(),* se hace un *fork()* para crear un nuevo proceso del sistema y así permitir que dicho proceso pueda ser o no migrado a otro nodo del cluster.

En *PVM* cuando se realiza un *fork*(), se crea un nuevo identificador (*TID*), por lo tanto si a la aplicación que se está corriendo se le pregunta el *TID* antes y después de realizar el *fork*(), estos valores serán diferentes. Este valor es el que identifica si un proceso es *local* o *distribuido*.

- *Uso:*
  Para usar esta clase, es necesario pasar el proceso que se desea sea distribuido por el constructor.
  Según el ejemplo que se encuentra en la tabla 7 del capitulo 6, la utilización de esta clase se hace de la siguiente forma:

Tabla 9. Definición de CSProcessDistributed

```
class CompositeIO : public CSProcess
{
       int id;
       protected:void run()
       {
              One2OneChannel<int> c0, c1;

              Parallel (
                     NEW OneIOneO(c0.reader(), c1.writer(), id),
                            CSProcessDistributed(NEW              OneOOneI(c1.reader(),
                            c0.writer(), id), NULL);

       }
}
```

Previamente, se creó el proceso *OneOOneI* y éste es pasado por el constructor de *CSProcessDistributed*, al pasar el proceso, ya se están pasando los canales que quieren ser utilizados.

Cuando *CSProcessDistributed* recibe el proceso, en su método *run*() ejecuta un *fork*() así:

Tabla 10. Método run() de CSProcessDistributed

```
private void run() {

       if ( fork() == 0 ) {
              /*  Es el hijo y sigue haciendo lo que el papá tenía que hacer */
       }
       else {
              /* es el padre y espera */
              wait();
       }
```

Después de realizar el *fork(),* el proceso que se crea puede ser migrado a otro nodo del cluster por *OpenMosix*. Y, cuando finaliza, vuelve a su nodo *home* y el proceso padre finaliza también.

El *wait(),* lo que hace es bloquear al padre hasta que la ejecución del hijo finaliza, en este momento el padre se desbloquea y la ejecución finaliza.

- *Componentes de la clase*

Tabla 11. Componentes de la clase CSPDistributed

MÉTODO	DESCRIPCIÓN
CSProcessDistributed (CSProcess process)	El constructor recibe como parámetro el proceso que se

	desea que sea distribuido.
virtual void run()	Este método contiene la funcionalidad del proceso. Duplica el proceso a ser distribuido.

8.1.2 Administrador de Comunicaciones.

Este componente es utilizado para manejar las comunicaciones entre los procesos, contiene las clases encargadas de la creación de canales y envío de mensajes entre otras.

Como se explicó en el capitulo anterior, existen 3 forma de comunicación definidas en el nuevo modelo*, Local – Local, Local – Distribuido, Distribuido - Distribuido*

En la comunicación *Local – Local* se usan las clases originales del API de C++CSP, estas clases son:

One2OneChannel, One2AnyChannel, Any2OneChannel, Any2AnyChannel, Chanin y Chanout, explicadas en el capitulo 6.

En la definición del nuevo *API*, estas clases son utilizadas para heredar de ellas y escribir nuevas clases que puedan ser utilizadas por el programador para crear aplicaciones distribuidas y concurrentes sobre el cluster. Estas nuevas clases se encargarán de manejar los canales y puertos utilizados para la comunicación.

Las clases creadas a partir de las clases anteriores son:

OneToOneChannel, OneToAnyChannel, AnyToAnyChannel, AnyToOneChannel, InputChannel, OutputChannel y PortDirectory.

PortDirectory

Es una clase utilizada para guardar las direcciones de la localización de los puertos de entrada y salida del canal. Tiene dos estructuras (MAP) que guardan una referencia al puerto y el identificador del canal.

Además esta clase va a estar recibiendo constantemente mensajes *PVM* con el fin de devolver la ubicación de los puertos en el momento que se le pregunte.

- *Componentes de la clase*

Tabla 12. Componentes de la clase PortDirectory

MÉTODO	DESCRIPCIÓN

PortDirectory()	Es el constructor de la clase y hace un llamado al método *run().*
PortDirectory &getInstance ()	Este método devuelve una instancia de la clase. Es una clase Singleton.
void bind(int idChannel, InputChannel<DATA_TYPE> &in , OutputChannel<DATA_TYPE> &out)	Este método crea 2 estructuras MAP, una para el puerto de entrada y otra para el puerto de salida del canal. Y les asigna el identificador del canal y la referencia in y out.
void run()	Este método es llamado en el constructor de la clase, constantemente está recibiendo mensajes PVM. Cuando recibe los mensajes (que son enviados por los puertos de entrada y salida del canal), toma los valores que vienen en el mensaje y actualiza las variables de los puertos de entrada y salida, esto para que los puertos tengan el valor de su TID y el TID del otro proceso.

En el método *run()*, cuando recibe el mensaje, toma los siguientes valores:

id : Identificador del Canal.
type: Tipo de puerto que envía el mensaje.
tid: Identificador PVM del puerto que envía el mensaje.

El identificador del canal lo utiliza para sacar las referencias in y out de las estructuras MAP creadas.

El tipo de puerto lo utiliza para asignar los valores de otherTID en el otro puerto e identificar que puerto envió el mensaje.

El Identificador PVM es utilizado para enviar respuesta PVM al puerto que envió el mensaje.

El código de esta clase se encuentra a continuación:

Tabla 13. Código de la clase PortDirectory

```
#ifndef PORTDIRECTORY_H
#define PORTDIRECTORY_H
#include<stdio.h>
#include <map>
using namespace std;
#include "InputChannel.cpp"
#include "OutputChannel.cpp"

namespace csp {
  template <class DATA_TYPE>
    class PortDirectory {

    public:

        ~PortDirectory();

        static PortDirectory &getInstance()
        {
              if(instance==NULL) instance= new PortDirectory();
              return instance;
        }

    void bind(int idChannel, InputChannel<DATA> &in , OutputChannel<DATA> &out)
    {
       portsIn[idChannel]=&in;
       portsOut[idChannel]=&out;
    }


    private:
        void run()
       {
                char msg[256];
                int arrived = pvm_nrecv ( -1, -1 );
                if ( arrived > 0 )
                     pvm_upkstr ( msg );
                char id[200], type[5000], tid[100];
                strcpy(id,strtok( msg, "|" ));
                strcpy(type,strtok( msg, "|" ));
                strcpy(tid,strtok( msg, "|" ));
                InputChannel<DATA_TYPE> in = *portsIn [atoi ( id )];
                OutputChannel<DATA_TYPE> out = *portsOut[atoi ( id )];
                int parentTID = in.getParentTID ( );
                int tidTmp = atoi ( tid );
                if ( strcmp ( type, "in" ) == 0 ) {
                     if ( parentTID != tidTmp )
                            in.setMyTID ( tidTmp );
                     out.setOtherTID ( tidTmp );
                     if ( in.getOtherTID()!= -1 && out.getMyTID()!= -1 ) {
                            if ( parentTID != in.getMyTID())
                                pvm_send(in.getMyTID(),in.getOtherTID((;
                            if ( parentTID != out.getMyTID ( ) )
                                pvm_send(out.getMyTID(),out.getOtherTID());
                            in.setSynchronized ( true );
                            out.setSynchronized ( true );
                     }
                }
```

```
                else if ( strcmp ( type, "out" ) == 0 ) {
                     if ( parentTID != tidTmp )
                            out.setMyTID ( tidTmp );
                     in.setOtherTID ( tidTmp );
                     if (out.getOtherTID() != -1 && in.getMyTID() != -1) {
                            if ( parentTID != in.getMyTID ( ) )
                                pvm_send(in.getMyTID(),in.getOtherTID());
                            if ( parentTID != out.getMyTID ( ) )
                                pvm_send(out.getMyTID(),out.getOtherTID());
                            in.setSynchronized ( true );
                            out.setSynchronized ( true );
                     }
                }
        }

        PortDirectory()
        {
            run ( );
        }


        static PortDirectory *instance;
        map<int, InputChannel<DATA_TYPE>*> portsIn;
        map<int, OutputChannel<DATA_TYPE>*> portsOut;

    };
}
#endif //PORTDIRECTORY_H
```

OneToOneChannel

Esta clase hereda de la clase *One2OneChannel* del API de C++CSP. Define un canal uno a uno en el nuevo API, cuando devuelve el *reader* (puerto de lectura del canal) o el *writer* (puerto de escritura del canal), debe registrarlo en el *PortDirectory*.

- *Componentes de la clase*

Tabla 14. Componentes de la clase OneToOneChannel

MÉTODO	DESCRIPCIÓN
OneToOneChannel()	Constructor de la clase. Crea un identificador para el canal, crea los puertos de entrada y salida del canal y los registra en el PortDirectory.
Chanin <DATA_TYPE> reader()	Obtiene un puerto de lectura para el canal.
Chanout <DATA_TYPE> writer()	Obtiene un puerto de escritura para el canal.

int getId()	Devuelve el identificador del canal.
void setId()	Asigna un identificador al canal.

El código de esta clase se encuentra a continuación:

Tabla 15. Código de la clase OneToOneChannel

```
#ifndef ONETOONECHANNEL_H
#define ONETOONECHANNEL_H

#include "PortDirectory.cpp"


namespace csp
{
    template <class DATA_TYPE>
    class OneToOneChannel : public One2OneChannel<DATA_TYPE>
       {
    public:

        OneToOneChannel()
              {
                     setId ( getPid ( ) * 100 + ++consecutivo );
                     in = new InputChannel<DATA_TYPE> ( this, true, getId ( ) );
                     out = new OutputChannel<DATA_TYPE>(this,true, getId ( ) );
                     PortDirectory<DATA_TYPE>::getInstance().bind(getId(), o );
              }

        OutputChannel<DATA_TYPE> writer()
              {
                     return out;
              }

        InputChannel<DATA_TYPE> reader()
              {
                     return in;
              }

        int getId()
              {
                     return id;
              }

        void setId(int _id)
              {
                     id=_id;
              }

    private:
        int id;
        static int consecutivo = 0;
        InputChannel<DATA_TYPE> in;
        OutputChannel<DATA_TYPE> out;
    };
```

```
}

#endif //ONETOONECHANNEL_H
```

InputChannel y OutputChannel

Son los puertos de lectura y escritura de un canal. Heredan de las clases *Chanin* y *Chanout* respectivamente.

Al momento de usarlas, se usas de igual forma que en el API de C++CSP.

```
OneToOneChan<int> chan;
InputChannel<int> in = chan.reader();
int a,b;
in.input(&a);
in.input(&b);

One2OneChannel<int> chan;
Chanout<int> out = chan.writer();
int a,b;
out.output(&a);
out.output(&b);
```

Se debe definir primero el tipo de canal, luego traer su *reader* o *writer* y realizar el *input* o el *output.*

En el momento que se hace el *input / output*, la clase verifica el valor de una variable llamada *synchronized,* esta variable es *boolean* y define si el puerto tiene inicializadas las variables myTID y otherTID que guardan los valores de los TID de si mismo y del otro puerto del canal. Si el valor de esta variable es *false*, entonces ejecuta un método llamado *validate()* que se encarga de inicializar estas variables. Cuando ya se encuentran inicializadas entonces dependiendo de la localización de los procesos, utiliza el API de C++CSP o sentencias PVM de envío y recepción de mensajes. Si myTID y otherTID tienen el mismo valor, entonces utiliza el API C++CSP, de otra forma utiliza PVM.

- *Componentes de la clase*

Tabla 16. Componentes de las clases InputChannel y OutputChannel

MÉTODO	DESCRIPCIÓN
InputChannel /OutputChannel(int id)	Constructor de la clase. Recibe el identificador para el canal y se lo asigna a una variable. Inicializa los valores de *parentTID*, *myTID*, *otherTID*, *synchronized* y el tipo del canal.
void validate()	Inicializa la variable *myTID* y

	envía un mensaje a *parentTID*, en este caso lo recibe *PortDirectory* y este le envía un mensaje devuelta con el valor para asignar a *otherTID.* Después cambia el estado de la variable synchronized a trae.
int getParentTID()	Método que devuelve el valor de *parentTID.* Esta variable guarda el TID devuelto por *pvm_mytid().*
void setParentTID(int pTID)	Método que asigna valor a la variable *parentTID*.
int getMyTID()	Método que devuelve el valor de la variable *myTID*. Cuando se inicializa el puerto, esta variable tiene valor -1, cambia su valor cuando se hace un *input* / *output* y toma el valor de *pvm_mytid().*

El código de *InputChannel* se encuentra a continuación:

Tabla 17. Código de la clase InputChannel

```
#ifndef INPUTCHANNEL_H
#define INPUTCHANNEL_H
#include <string>

#include "csp.h"
#include </root/pvm3/include/pvm3.h>

/*#ifndef CHANNEL_ENDS.H
  #include "csp/channel_ends.h"
#endif*/
using namespace std;

namespace csp {

    template<class DATA_TYPE>
    class InputChannel : public csp::Chanin<DATA_TYPE>
       {
    public:

        InputChannel (int id )
              {
                     setParentTID ( pvm_mytid ( ) );
                     setMyTID ( -1 );
                     setOtherTID ( -1 );
                     setSynchronized ( false );
                     setIdChannel ( id );
```

```
            setType ( "in" );
      }

~InputChannel()
      {
      }

int getParentTID()
      {
            return parentTID;
      }

void setParentTID(int pTID)
      {
            parentTID = pTID;
      }

int getMyTID()
      {
            return myTID;
      }

void setMyTID(int _myTID)
      {
            myTID=_myTID;
      }

int getOtherTID()
      {
            return otherTID;
      }

void setOtherTID(int _otherTID)
      {
            otherTID=_otherTID;
      }

bool getSynchronized()
      {
            return synchronized;
      }

void setSynchronized(bool _synchronized)
      {
            synchronized=_synchronized;
      }

int getIdChannel()
      {
            return idChannel;
      }

void setIdChannel(int _idChannel)
      {
            idChannel=_idChannel;
      }

string getType()
      {
            return type;
      }
```

```
        void setType(string _type)
              {
                    type=_type;
              }
    private:

        void validate()
              {
                    if ( !getSynchronized ( ) )
                    {
                          int valor;
                          setMyTID ( pvm_mytid ( ) );
                          char temp[256];
                          itoa(getIdChannel(),temp, 10);
                          string msg;
                          msg= temp + "|" + getType() + "|";
                          itoa(getMyTID(),temp, 10);
                          msg += temp;
                          pvm_send(getParentID(), msg.c_str());

                          if ( getParentTID ( ) != getMyTID ( ) ) {
                                pvm_recv ( getParentTID ( ), &valor );
                                setOtherTID ( valor );
                                setSynchronized ( true );
                          }
                    }
              }

    void input(DATA_TYPE* const dest) const
    {
        if ( !getSynchronized ( ) )
              validate()
        else
              if (getParentTID() == getMyTID())
                    super();
              else
                    comunicate();
    };

    public:

    private:
        int parentTID;
        int myTID;
        int otherTID;
        bool synchronized;
        int idChannel;
        string type;
    };
}

#endif
```

El código de *OutputChannel* se encuentra a continuación:

Tabla 18. Código de la clase OutputChannel

```
#ifndef OUTPUTCHANNEL_H
#define OUTPUTCHANNEL_H
#include <string>

#include "csp.h"
#include </root/pvm3/include/pvm3.h>

/*#ifndef CHANNEL_ENDS.H
  #include "csp/channel_ends.h"
#endif*/
using namespace std;


namespace csp {
    template <class DATA_TYPE>
       class OutputChannel : public csp::Chanout<DATA_TYPE>
       {
    public:

        OutputChannel(int id){
                      setParentTID ( pvm_mytid ( ) );
                       setMyTID ( -1 );
                       setOtherTID ( -1 );
                       setSynchronized ( false );
                       setIdChannel ( id );
                       setType ( "in" );
       }

        ~OutputChannel(){
       }

        int getParentTID()
              {
                     return parentTID;
              }

        void setParentTID(int pTID)
              {
                     parentTID = pTID;
              }

        int getMyTID()
              {
                     return myTID;
              }

        void setMyTID(int _myTID)
              {
                     myTID=_myTID;
              }

        int getOtherTID()
              {
                     return otherTID;
              }

        void setOtherTID(int _otherTID)
              {
                     otherTID=_otherTID;
```

```
        }

    bool getSynchronized()
        {
                return synchronized;
        }

    void setSynchronized(bool _synchronized)
        {
                synchronized=_synchronized;
        }

    int getIdChannel()
        {
                return idChannel;
        }

    void setIdChannel(int _idChannel)
        {
                idChannel=_idChannel;
        }

    string getType()
        {
                return type;
        }

    void setType(string _type)
        {
                type=_type;
        }
private:

    void validate()
        {
                if ( !getSynchronized ( ) )
                {
                        int valor;
                        setMyTID ( pvm_mytid ( ) );
                        char temp[256];
                        itoa(getIdChannel(),temp, 10);
                        string msg;
                        msg= temp + "|" + getType() + "|";
                        itoa(getMyTID(),temp, 10);
                        msg += temp;
                        pvm_send(getParentID(), msg.c_str());

                        if ( getParentTID ( ) != getMyTID ( ) ) {
                                pvm_recv ( getParentTID ( ), &valor );
                                setOtherTID ( valor );
                                setSynchronized ( true );
                        }
                }
        }

void output(DATA_TYPE* const dest) const
{
    if ( !getSynchronized ( ) )
            validate()
    else
            if (getParentTID() == getMyTID())
```

```
                        super();
                else
                        comunicate();
        };

        public:

        private:
            int parentTID;
            int myTID;
            int otherTID;
            bool synchronized;
            int idChannel;
            string type;
        };
}

#endif
```

8.1.3 Administrador de Sincronización.

Para el administrador de sincronización, se tienen las clases de barreras y alternativas. Estas clases fueron tomadas del API de C++CSP.

La interacción de los componentes definidos anteriormente, permiten al programador definir las aplicaciones concurrentes y distribuidas.

Sin embargo es importante resaltar que no solo la definición del API permite que se creen estas aplicaciones, al interior del API se tienen elementos muy importantes como PVM y la interacción con *OpenMosix* que hacen que estas aplicaciones sean concurrentes y distribuidas.

8. CONCLUSIONES

El principal aporte que brinda el presente trabajo de investigación, consistió en crear y especificar un modelo de programación paralela y concurrente basado en un modelo existente, el cual permite a los programadores de aplicaciones sobre cluster escribir el código de sus aplicaciones sin necesidad de preocuparse por la arquitectura del cluster, número de nodos existentes y localización de cada uno de ellos.

Fue necesario estudiar diferentes herramientas y programas existentes que permiten el trabajo sobre clusters y tomar las principales características de algunos de ellos para definir el API de programación que facilitara el trabajo y fuera de alto nivel.

Se tomaron 3 herramientas para poder realizar la definición del API de programación, estas herramientas fueron: OpenMosix, PVM y C++CSP. Cada una de ellas aportaba a la definición del API una parte importante, OpenMosix se encargaría del balanceo de carga, PVM se encargaría de comunicar los procesos que se encontraran en diferentes nodos del cluster o procesos que estuvieran en el mismo nodo del cluster pero que fueran diferentes procesos del sistema y C++CSP sería la base para realizar la programación del API.

El presente trabajo no solo aporta la definición del API de programación sobre clusters, sino además el estudio de herramientas con diferentes características y diferentes funcionalidades que juntas pueden ser utilizadas para definir y realizar nuevas herramientas.

Durante el desarrollo del presente trabajo de programación, se presentaron algunos problemas al momento de definir las herramientas con las que se trabajaría, ya que entre las desventajas que tiene OpenMosix, se encuentra que no migra JVM (maquina virtual de java) y fue necesario buscar una alternativa de lenguaje de programación diferente a Java, herramienta con la que se esperaba realizar el API. Además fue necesario realizar cambios en el código fuente de PVM para que cuando en al API se hiciera un *fork*(), PVM le asignara un nuevo TID al proceso creado y así poder identificarlo como un proceso diferente y enviarle mensajes.

A pesar que se realizó la definición del API de programación y se realizaron algunas pruebas individuales de cada una de las herramientas seleccionadas, al momento de hacer la implementación del modelo definido surgieron problemas inesperados que atrasaron el desarrollo del trabajo y no permitieron la finalización exitosa del API. Durante esta etapa, cabe resaltar que se tuvo la ayuda de la persona que creó el API de C++CSP Neil Brown, esta persona a través de E–mails, estuvo ayudando en la instalación y configuración de C++CSP, ya que al momento de instalar dicho API se presentaron problemas de configuración.

9. TRABAJO FUTURO

- **Finalización de Implementación y pruebas del API.**

En trabajos futuros, se puede finalizar la codificación del API y realizar pruebas que permitan determinar si la definición del modelo extendido cumple la funcionalidad que pretende cumplir.

- **Inclusión Objetos en el envío de mensajes a través de los canales.**

Según la definición actual del API, éste no permite el envío de Objetos a través de los canales, el programador si desea hacer este envío, debe hacer el *marshalling/unmarshalling,* en una siguiente versión, se puede incluir que el API realice estas tareas.

- **Permitir paso de canales ya inicializados.**

Los canales después de ser inicializados, no pueden ser pasados a otro proceso, debido a que, cuando un canal es inicializado, se establece su localización, y, si el canal se envía a otro proceso que puede ser *distribuido,* su localización cambia y puede generar un error de comunicación. Por lo tanto en una siguiente versión se puede implementar esto ya que sería bastante funcional para el programador tener esta facilidad.

REFERENCIAS

[1] Tomado de la presentación en PowerPoint
www.fisica.uson.mx/~antoniom/clusters2.ppt
Noviembre 10 de 2004.

[2] THOMAS STERLING. Una Introducción a los Clusters de PC's para Cómputo de Alto Rendimiento. Disponible en: http://clusters.unam.mx/Proyectos/white_paper/cap1.html
Noviembre 10 de 2004.

[3] SACHA KRAKOWIAK. Patterns and Frameworks for Middleware construction. - Université Joseph Fourier, Grenoble.

[4] ROSA MARÍA YÁÑEZ GÓMEZ. Introducción a las tecnologías de Clustering en GNU/Linux.

[5] MOSHE BAR. The openMosix Internals. Disponible en:
http://es.tldp.org/almacen/Manuales-LuCAS/doc-manual-openMosix-1.0/doc-manual-openMosix_html-1.0/node30_ct.html.
Enero de 2005.

[6] GIAN PAOLO GHILARDI. Consideration on Openmosix. Universidad de Milano.
Disponible en: http://www.democritos.it/events/openMosix/papers/crema.pdf.
Enero de 2005.

[7] MATT RECHENBURG. The openMosix-API. Disponible en:
http://www.openmosixview.com/docs/openMosixAPI.html#general.
Febrero 2005.

[8] C.A.R. HOARE. Communicating Sequential Processes. The Queen's University Belfast, Northern Ireland
Disponible en: http://www.cs.virginia.edu/~evans/crab/hoare1978csp.pdf
Marzo 2005.

[9] NEIL BROWN. PETER WELCH. An Introduction to the Kent C++CSP Library. Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, England.
Marzo 2005.

[10] Documentation for The Kent C++ CSP Library. Disponible en:
http://www.twistedsquare.com/cppcsp/docs/classcsp_1_1CSProcess.html
Marzo 2005.

[11] Documentation for The Kent C++ CSP Library. Disponible en: http://www.twistedsquare.com/cppcsp/docs/classcsp_1_1One2OneChannel.html Marzo 2005.

[12] Documentation for The Kent C++ CSP Library. Disponible en: http://www.twistedsquare.com/cppcsp/docs/classcsp_1_1Chanin.html Marzo 2005.

[13] Documentation for The Kent C++ CSP Library. Disponible en: http://www.twistedsquare.com/cppcsp/docs/classcsp_1_1Chanout.html Marzo 2005.

[14] Documentation for The Kent C++ CSP Library. Disponible en: http://www.twistedsquare.com/cppcsp/docs/classcsp_1_1Barrier.html Marzo 2005.

[15] Documentation for The Kent C++ CSP Library. Disponible en: http://www.twistedsquare.com/cppcsp/docs/classcsp_1_1Alternative.html Marzo 2005.