INSTITUTO SUPERIOR MINERO METALÚRGICO DE MOA
"DR. ANTONIO NUÑEZ JIMENEZ"
FACULTAD DE METALURGIA - ELECTROMECÁNICA

HYDRA:
Prototipo de plataforma para la computación distribuida

Tesis presentada en opción al título de Ingeniero Informático

AUTOR:

Amed E. Sheriff López.

TUTORES:

Lic. Yiezenia Rosario Ferrer.

Lic. Jhonlier Suárez Molina.

Moa - Holguín

Julio 2007

…"Casi siempre caminan los hombres por sendas trilladas; pocos obran por sí mismos, sino por el espíritu de imitación; pero como este espíritu no puede ser exacto en todo, ni suele ser posible llegar a la altura de aquellos que se toman como modelos, el hombre advertido debe únicamente seguir los caminos que abrieron otros, tenidos por superiores, e imitar bien a los que han sobresalido, a fin de que si no consigue igualarles, tengan al menos sus acciones alguna semejanza con las suyas. Cada uno deberá portarse como el ballestero prudente, que cuando ve que el blanco a que dirige sus tiros se halla demasiado distante, considera la fuerza de su arco y apunta más alto, no para dar en el blanco, sino para tocarle.."

Nicolás Maquiavelo

Dedicatoria

Es muy difícil no dar crédito al ser más sublime que existe y se merece todo lo que podamos ofrendarle, es por eso que te dedico a ti este trabajo madre mía…

Padre, consejos y apoyo incondicional, ejemplo de hombría, forjador de mi carácter, es por eso, papá que te dedico a ti también este pequeño gran paso…, orgulloso de ser SHERIFF... Con todo cariño, y más...su hijo…

Amed Elías Sheriff López.

Agradecimientos

A Dios por el bienestar de mi familia y el mío, por proveerme siempre lo necesario para cumplir con mis metas, por darme raciocinio ,por mantenerme firme, por perdonarme las fallas que he tenido, por mantener mi familia unida.

A mí mamá Moira, por darme siempre el apoyo que he necesitado en los tiempos difíciles, por acompañarme en mis buenos y malos tiempos, por sufrir junto a mi las angustias de algún mal momento, por ser la madre ejemplar, dedicada, abnegada, cariñosa, bella, tan dulce y llena de amor.

A mí papá Héctor, el amigo más fiel que tengo, el mejor compañero, la imagen a seguir, mi consejero, mi viejo Pá.

A mis hermanos, los dos, Jasil y Faride que con sus palabras de cariño y admiración, me han hecho superar algunas circunstancias difíciles para mí.

A mí querida abuela Yolanda, por sus oraciones y pensamientos, por los ánimos que me da siempre.

A toda mi familia boliviana por creer en mi, es mucho lo que tengo que agradecerles…….

A mí otra familia cubana, Inalvis, Luís, Nuri, Ángel Luís, Yeleynis, Iraide, Israel, Mircia, Pedrito, Alexey, Geomani, Eric, Ever, gracias por tantas cosas que han hecho por mi, que no se pueden enumerar, tantas muestras de afecto y paciencia.

A Xenia, por siempre acompañarme, ayudarme, entenderme, consolarme, quererme, cuidarme, soportarme, mimarme, hacerme reír, protegerme, aconsejarme, es que no puedo simplemente decirte gracias, esa palabra no tiene significado para el sentimiento que me llena al pensar en ti…

A mis amigos Idalberto, Yaritza, Jhonlier, Ismael, Yudith, Isidro, Yadira, Roger, Eglys, Javier, Yiezenia, Virgen, Royki, si no he nombrado alguno pido disculpas.

Y por ultimo, quiero agradecer a todas aquellas personas que de una u otra forma han hecho un tanto difícil el convivir en paz, o han intentado hacerme perder la cordura frente a algunas situaciones, dentro o fuera de la residencia estudiantil, lo cual me ha servido de mucho para forjar más mi carácter, y de esta manera crecer espiritualmente, y darme cuenta que hay personas que no merecen ni que se los tomen en cuenta……

Gracias……

Amed E. Sheriff López.

Resumen

La necesidad de realizar cálculos y algoritmos complejos, proyectos vinculados con la simulación a gran escala y la creciente necesidad de potencia computacional para las aplicaciones que puedan resolver estos problemas demanda que las Ciencias Informáticas utilicen computación distribuida (CD) para resolverlas, en muchos de los casos como vía más factible y menos costosa.

El presente trabajo de diploma propone al acercamiento inicial a la implementación de una plataforma para la computación distribuida, que permita el desarrollo de algoritmos paralelos.

A lo largo de las siguientes páginas se exponen en detalle los elementos del análisis y diseño de la aplicación *Hydra*, prototipo inicial, así como los cálculos de factibilidad, estudio de precedentes y principales conceptos de la CD en la actualidad.

Summary

The need to realize calculations and complex algorithms, projects linked with large scale simulations and the growing necessity of computational power for the applications that can solve these problems demands that computer sciences use distributed programming to solve them, in most cases it is the more feasible and less expensive way.

The present diploma work proposes an initial approach to development of a platform for distributed programming, that allow the development of parallel algorithms.

Through out the following pages the elements of the analysis and design of the application, *Hydra*, are exposed in detail, as well as the calculations of feasibility and study of precedents and the main concepts of the DP at the present time.

TABLA DE CONTENIDOS

LISTAS DE TABLAS

Introducción

Hoy por hoy, en pleno siglo XXI la necesidad de realizar cálculos computacionales complejos, grandes proyectos de simulación, la explotación de servicios de Internet, así como el aumento en la complejidad en tiempo y espacio, e incluso la realización de películas con gráficos 3D, conlleva a adquirir máquinas cada vez más potentes, con mayor cantidad de procesadores y por tanto, mucho más costosas. Sin embargo existen soluciones igual de factibles y que consumen *menos* recursos.

Por ejemplo, la industria cinematográfica acude al uso de varias estaciones de trabajo *Sun Microsystem*, en lugar de una supercomputadora, para realizar sus efectos digitales en 3D; *Google* brinda sus eficientes servicios mediante *grids* de computadoras en vez de carísimos *mainframes*[1]; los algoritmos que por su complejidad se consideraban irrealizables, hablando en términos de vida humana, hoy encuentran solución mediante el trabajo cooperado de varios ordenadores. En todos los casos se percibe la utilización de Computación Distribuida (CD) para resolver los problemas.

Computación distribuida (CD) se refiere a dividir una tarea grande en otras más pequeñas, donde cada cliente puede trabajar según su potencialidad, contribuyendo a alcanzar la meta común. El resultado final es la potencia de cómputo que iguala a lo que pueden producir los supercomputadores más grandes.

Los países en vías de desarrollo, pueden aprovechar las facilidades que otorga la CD en vista de que las dificultades económicas les impiden contar con supercomputadores, por lo que el cómputo distribuido de alto rendimiento juega hoy en día, un papel importante en la solución de problemas de las ciencias, las ingenierías y del comercio moderno.

Las universidades del país que forjan profesionales, en diferentes ramas de la informática o la computación, necesitan de herramientas que sirvan como ejemplo para el aprendizaje de técnicas de computación, que colaboren de alguna manera al desarrollo tecnológico en vías de utilizar los recursos que tenemos, optimizarlos en cuanto a rendimiento sin necesitar de *hardware* costoso. Para llegar a este objetivo, la solución al problema es: crear un sistema que logre combinar virtualmente toda la potencialidad de un grupo de máquinas de manera que

[1] Ver glosario de téminos

parezca que se dispone de una supercomputadora muy potente, trabajando y resolviendo cualquier tipo de problema de cálculo y/o cómputo que se le presente, en un tiempo mínimo de ejecución.

En nuestro centro, problema no es solo la existencia de un *Cluster*[2], pues tampoco podemos contar con uno, sino aprovechar los recursos que se tienen (cualquier tipo de máquina con la que se cuente) y crear un ambiente de desarrollo multiplataforma para todos aquellos programadores ó usuarios interesados en la CD.

Como se ha observado, la idea de la obtención de un *Cluster* es muy remota; por lo tanto se plantea la utilización de los recursos disponible, para ejemplificar un pequeño *Cluster* heterogéneo. Se requiere de una aplicación que corra sobre cualquier plataforma, capaz de controlar todas las formas de interacción entre sus nodos, pues no se trata simplemente de interconectar máquinas en una red, sino también de garantizar el comienzo y el final de la comunicación entre ellas, y que el envío ó recibo de mensajes sea fiable. De igual manera, debe detectar la incorporación de nuevas máquinas, distribuir las tareas de acuerdo a la capacidad de cada nodo, y por supuesto garantizar la disponibilidad del sistema.

Actualmente, en el territorio cubano, son pocas las instituciones educacionales que disponen de *Clusters* a mediana o gran escala, pero claro, sin dejar de mencionar a algunas de ellas que trabajan con este paradigma de computación a pesar de que son pocas, por ejemplo el Instituto Superior Politécnico José Antonio Echeverría (ISPJAE), la Universidad Central de las Villas (UVCL), la Universidad de Ciencias Informáticas (UCI), todas estas utilizan la CD para la solución a sus diferentes problemas, que a diferencia de otras empresas y fábricas las cuales optan por el equipamiento de máquinas sumamente caras.

El ISMM no cuenta con ningún sistema semejante al de los centros de educación superior antes nombrados, que resuelva el persistente problema de la falta de computadores de potencia para efectuar proyectos que necesiten de muchos requerimientos, tales como la simulación a gran escala, principalmente basándose en el hecho de que el centro tiene como base de prestigio carreras ligadas al trabajo en terrenos de diferente constitución y forma, ó involucrados con sistemas eléctricos y mecánicos u orientados a las ramas de la minería, la metalurgia y la geología, donde una buena simulación de los mismos, sería de gran ayuda a

[2] Ver glosario de términos

futuros profesionales y podría dar una herramienta y solución desde lo académico a cuestiones prácticas de esas especialidades.

El presente trabajo de diploma pretende dar solución a la situación problémica expuesta, por lo que como **problema** se define la inexistencia de plataformas potentes para dar respuesta a las crecientes demandas potenciales de computadores con mayores prestaciones para soluciones de programación en el centro.

Se impone la necesidad de incursar en la CD para la implementación de una aplicación que imite la filosofía de un *cluster* de computadoras, utilizando las computadoras existentes, sin que estas dejen de brindar los servicios que tradicionalmente brindan (ofimática, programación tradicional, etc.), para demostrar la gran utilidad de esta técnica, y que el Instituto Superior Minero Metalúrgico de Moa sea uno más de los centros que optimiza sus recursos informáticos.

En el centro no existen antecedentes de un sistema como el que se propone. Con el presente proyecto se pretende la obtención de un producto de software acorde con las metas que se ha propuesto la sociedad cubana en cuanto a optimización de recursos informáticos, que siga las mejores prácticas de desarrollo de aplicaciones de este tipo y permita introducir en el ISMM la enseñanza curricular de la Computación Distribuida.

Entonces, se propone la implementación de una aplicación que sea multiplataforma, sin cambios sustanciales en los recursos informáticos ya antes dispuestos para esas máquinas. La aplicación deberá aportar, a todo aquel que lo requiera, mayor potencia de cálculo y ejecución para dar respuesta a problemas que necesiten altas prestaciones. De esta forma, se podría contar con una herramienta interesante e indispensable para el óptimo aprovechamiento de, al menos, una parte de nuestras pocas máquinas.

La presente investigación tiene como **objeto de estudio** las plataformas para la computación distribuida.

El **objetivo general** del trabajo de diploma es el diseño e implementación de una plataforma para la programación distribuida.

De acuerdo a esto se plantean los siguientes **objetivos específicos:**

1. Selección de la tecnología para la comunicación eficaz entre servidores.

2. Implementación una librería en Python que permita realizar las funciones básicas de la aplicación.

A partir del problema planteado, el objeto de estudio y los objetivos propuestos se establece el **campo de acción** siguiente: desarrollo de plataformas para la computación distribuida.

El desarrollo de esta investigación se sustenta en la **idea a defender** siguiente: Se puede obtener una versión funcional de una plataforma para el cómputo distribuido con la utilización de los recursos computacionales disponibles en la Universidad.

Para cumplir los objetivos y resolver la situación problémica planteada, se ejecutaron las siguientes tareas:

1. Revisión del estado de arte de la teoría y la tecnología de la computación distribuida.
2. Análisis y diseño de la plataforma para la programación distribuida.
3. Implementación de la aplicación.

El presente documento se estructura en 4 capítulos:

Capítulo 1: Computación Distribuida, se profundiza en los fundamentos teóricos de este tema, las distintas aproximaciones, y los aspectos necesarios para el entendimiento del tema, que ayudarán a la rápida comprensión del negocio y de la propuesta de solución.

Capítulo 2: Análisis y diseño de la solución, se presenta un acercamiento al análisis y diseño de la plataforma para la CD, se muestran los artefactos de ingeniería de software obtenidos como resultado del proceso.

Capítulo 3: Detalles de la implementación, en este capítulo se describe qué es Hydra, cómo funciona, en que lenguaje fue implementado, y por que se escogió el mismo, entre otras prestaciones del sistema.

Capítulo 4: Estudio de factibilidad, se analizan en función del modelo COCOMO II los costos del desarrollo de la aplicación propuesta, los beneficios tangibles e intangibles. Se presentan los cálculos realizados, así como las tablas con datos de la aplicación y de los resultados finales.

Finalmente se presentan las conclusiones y recomendaciones del trabajo, la bibliografía y los anexos.

Capítulo 1: Acercamiento a la Computación Distribuida

Sumario

- Introducción.
- Evolución histórica de la Computación Distribuida.
- Computación distribuida.
- *Clusters* y *Grids*.
- Computación paralela.
- Taxonomía de Flynn.
- Sistemas de Memoria Compartida.
- Sistemas de Memoria Distribuida.
- Herramientas y plataformas para sistemas distribuidos.

1.1 Introducción

Obedeciendo al creciente desarrollo tecnológico en la informática, hoy en día mucha gente cuenta con máquinas relativamente potentes; no obstante, la mayoría del potencial de cada máquina no es utilizado en totalidad, y no se ha pensado que ese potencial enorme de procesadores subutilizados podrían ocuparse en otros fines más científicos entre los cuales se encuentra la Computación Distribuida.

La Computación Distribuida (CD) encuentra sus bases en aspectos conceptuales que han ido sirviendo de modelos ideales y propuestas de solución para las actuales plataformas que soportan CD.

A lo largo de este capítulo se desglosan los fundamentos teóricos más cercanos a la aplicación a desarrollar, los principales paradigmas de plataformas existentes para la CD, herramientas, arquitecturas, clasificación de las mismas, así como también cuestiones sobre el alcance que debe tener una aplicación de esta índole.

1.2 Evolución histórica de la Computación Distribuida.

La computación distribuida, comunitaria o colectiva surge a finales de los 80's como estrategia para descomponer grandes cifras en factores primos. Este proceso, aparentemente

sencillo e inverso a la multiplicación, exige a los procesadores un esfuerzo de cálculo enorme. Arjen Lenstra y Mark Massanem, dos científicos del laboratorio DEC en Palo Alto (EEUU) pensaron que sería más eficaz repartir la tarea de factorización en una red local de estaciones de trabajo.

En la primera mitad de los 90's, existieron muchos proyectos de investigación en la comunidad académica y de investigación, enfocados hacia la computación distribuida. Un área clave de investigación enfocada en el desarrollo de herramientas que permitan un alto rendimiento de los sistemas de computación distribuidos para funcionar como una gran computadora.

En 1995 se realiza la conferencia de Súper Computación en San Diego en la IEEE/ACM, 11 redes de alta velocidad fueron usadas para conectar 17 sitios con grandes terminales de recursos computacionales para la demostración de la creación de una súper "meta computadora". Esta demostración fue llamada I-Way y fue dirigido por Ian Foster de Argonne National Labs (ANL) y la Universidad de Chicago)[3].Sobre esta red de demostración fueron desarrolladas y ejecutadas sesenta aplicaciones diferentes, abarcando varias ramas de la ciencia e ingeniería

El éxito de la demostración de I-Way obligó a la agencia DARPA del gobierno de los Estados Unidos, en Octubre de 1996, a fundar una organización para crear un proyecto para herramientas de computación distribuida. El proyecto de investigación fue dirigido por Ian Foster de la ANL y Carl Kesselman de la Universidad del Sur de California. El proyecto fue nombrado Globus y el equipo creó una colección de herramientas que dejó la fundación para las actividades de la computación en *Grid* en las comunidades académicas e investigativas. Como en la Conferencia de Súper Computación en 1997, 80 sitios de la red mundial corrieron software basado en el Kit de herramientas del Globus fueron conectadas juntas.

La computación en *Grid*, podría hacer disponible un tremendo poder computacional para cualquiera, a cualquier hora, y de una manera realmente transparente, igual que, la red de energía eléctrica habilita la energía a billones de terminales eléctricas.

La computación en *Grid*, en las comunidades académicas y de investigación permaneció enfocada en crear una estructura eficiente para influenciar el alto rendimiento distribuido en

[3] Foster, Ian, 2000, "Internet Computing and the Emerging Grid," *Nature,*12/2002.

los sistemas de computación. Pero con la explosión de la Internet y el incremento del poder de las computadoras de escritorio durante el mismo periodo, se han realizado muchos esfuerzos para crear sistemas distribuidos poderosos conectando PC's juntos en una red. En 1997, Entropía se lanzó para recolectar las computadoras inactivas de la red para resolver problemas de interés científico. La red Entropia creció a 30,000 computadoras con una ganancia de velocidad por encima de un teraflop (10^{12} FLOPS) por segundo.

Hoy en día, muchas corporaciones como la IBM, Sun Microsystems, Intel, Hewlett Packard, y algunas pequeñas compañías como Platform Computing, Avaki, Entropia, DataSynapse, y United Devices están invirtiendo sus dólares para el buen uso y creación de la nueva generación de mentes líderes sobre la computación en *Grid* que esta enfocada preferiblemente en las aplicaciones de negocios mas que las académicas e investigativas.

1981	Bruce J. Nelson, de Xerox PARC y la universidad de Carnegie-Mellon, describe y nombra llamadas a procedimiento remotos (PRC). RPC es después la base para muchos sistemas programados paralelos y distribuidos.
1993	Los ordenadores personales de seiscientos voluntarios reclutados por Lenstra y Massane trabajaron en red hasta descifrar uno de estos puzzles criptográficos (RSA-129).
1995	Se realiza la conferencia de Súper Computación en San Diego, 11 redes de alta velocidad fueron usadas para conectar 17 sitios con grandes terminales de recursos computacionales para la demostración de la creación de una súper "meta computadora". Sesenta aplicaciones diferentes, abarcando varias facultades de la ciencia e ingeniería, fueron desarrolladas y ejecutadas sobre esta red de demostración.
1996	Dan Werthimer y sus colegas de la Universidad de Berkeley, presentaron la idea en la V Conferencia Internacional de Bioastronomía., la computación distribuida ya había cosechado

	sus primeras victorias: en la última década, todos los récords de cálculo (mayor número primo, obtención de PI con más dígitos) son pulverizados por redes de computación integradas por aficionados.
1997	Se funda Entropía, una compañía que "activa y recicla los recursos desaprovechados de las PC's para convertirlos en poder creativo e incorporarlos a redes de computación". Hasta el momento, los retos científicos aparentemente altruistas son los únicos que han logrado una respuesta masiva de la Red. Como SETI@Home que es un proyecto desarrollado por la Universidad de California Berkeley e implica el procesar los datos recopilados por el telescopio de radio de Arecibo en Puerto Rico. Este proyecto es el "1er " proyecto de computación distribuida lo que se lo podría considerar la madre de todo los demás.
2000	Los artículos sobre la computación en Grid dejan el campo comercial y se popularizan dramáticamente, apareciendo los artículos en diarios y revistas tales como, The New York Times, Economist, Business 2.0, Red Herring, Washington Post, Financial Times, Yomiuri Shimbum, The Herald de Glasgow, Jakarta Post, Dawn de Karachi, etc.
2001	Un análisis estadístico realizado por Lexus-Nexis muestra que las referencias de la computación en Grid en medios populares de Estados unidos se incrementaron de 100 a 500 citaciones en un cuarto del presente año.
2005	Muchas corporaciones como la IBM, Sun Microsystems, Intel, Hewlett Packard, y algunas pequeñas compañías como Platform Computing, Avaki, Entropia, DataSynapse, y United Devices comienzan a invertir su dinero para la nueva generación de mentes lideres sobre la computación en Grid que esta enfocada

	preferiblemente en las aplicaciones de negocios más que las académicas y de investigación.

Antes de que finalice esta década, la velocidad del súper procesador integrado por todos los equipos en red superará la cifra mítica del Petaflop (mil billones de operaciones por segundo) acercándose a las capacidades de la mente humana: 1011 neuronas x 104 conexiones/neurona x 100 Hz frecuencia de disparo x 1 bit/disparo = 1017 bits/s. La mayor parte de este potencial seguirá perdiéndose en ciclos vacíos mientras no se popularicen las aplicaciones de computación distribuida. En cualquier caso, es sólo cuestión de tiempo conseguir una trama tan compleja y eficaz como el tejido neuronal.

Los proyectos de Computación Distribuida recientes se han diseñado para utilizar las computadoras de centenares de millares de voluntarios todo sobre el mundo, vía el Internet, para buscar señales de radio extraterrestres, buscar los números primos tan grandes que tienen más de diez millones de dígitos, y de encontrar drogas más eficaces para luchar el virus del SIDA, analizar la respuesta de los pacientes a la quimioterapia, curas potenciales para el ántrax, etc.

Por ejemplo, en el mundo existen ya varias compañías que trabajan con estos modelos, entre ellas están, el grandísimo motor de búsqueda de GOOGLE, Yahoo!, y considerado el más grande del mundo SETI@Home[4].

En Cuba, el 13 de febrero de 2007 se planteó en una mesa redonda referente a la Bioinformática y la tecnología *Grid* cómo introducir esta última en nuestro territorio. El problema fundamental es el poco conocimiento de esta tecnología, por lo que no se ha implantado aún. Existen centros particulares y solo usuarios seleccionados dentro de cada centro que utilizan esta tecnología, por lo que unos están subutilizados y otros sobrecargados. La idea es implementar en estos centros la tecnología *Grid* para aumentar la capacidad de cálculo. Se planteó que las principales dificultades existentes para lograr esto son: desconocimiento de la existencia de la tecnología, falta de confianza en ella, pocas aplicaciones preparadas para su uso y la falta de personal adiestrado. Sin embargo, uno de los

[4] Ref.[PP, 2006]

pilares de la computación en *Grid* es la seguridad y privacidad sobre los recursos y la información. Los objetivos de este debate fueron presentar la tecnología a la comunidad científica y promover las investigaciones para su implementación. En fin, el desarrollo de las redes internacionales se mueven hacía la computación en *Grid*; el área de bioinformática es una de las llamadas a ser de punta de lanza en el uso de los *Grids* de computadoras.

A pesar del desarrollo y las bondades de esta tecnología nuestro país anda a la saga en su investigación y uso, puesto que no sea materializado ningún proyecto de envergadura en el que se haya aplicado esta tecnología.

Como se puede apreciar, la computación distribuida ha ido evolucionando de manera agigantada en las últimas décadas, en busca de soluciones más factibles, económicas, con recursos más estándar, pero ya hace algunos años más atrás, la filosofía de esta rama de la informática tenia bases teóricas, como veremos a continuación.

1.3 Computación Distribuida

Computación Distribuida (CD) es una técnica de la informática que soluciona un gran problema dando las partes pequeñas del problema a muchas computadoras para solucionar y después combinando las soluciones de las piezas en una única solución para el gran problema. En este caso la ejecución simultánea de una tarea ocurre en varios procesadores de máquinas distintas o sea, cada procesador tiene su propio módulo de memoria, los computadores no necesariamente tienen que estar cerca, ni con el mismo sistema operativo.

Este método de distribución del trabajo para la ejecución de las tareas, garantiza un increíble aumento en el rendimiento, no solo por tener cada parte de la respuesta un bus propio por donde hacer llegar la petición, sino que todo el conjunto de ordenadores trabajan como uno solo superpotente, combinando los recursos de memoria RAM, velocidad de microprocesador, convirtiéndose en una supercomputadora.

Sin embargo, incorporarse a una red de computación distribuida no implica ceder ni compartir la capacidad del ordenador. Es más parecido a una labor de reciclaje: El 99% de la potencia de nuestros procesadores no se utiliza de continuo, aunque permanece disponible para operaciones puntuales. El software de computación distribuida "recicla" la potencia sobrante

para llevar a cabo tareas en red, sin que el usuario vea mermada la velocidad o memoria de su equipo.

El siguiente gráfico ilustra la distribución del trabajo en los distintos procesadores para al final combinarse cada pequeña tarea en la única solución.

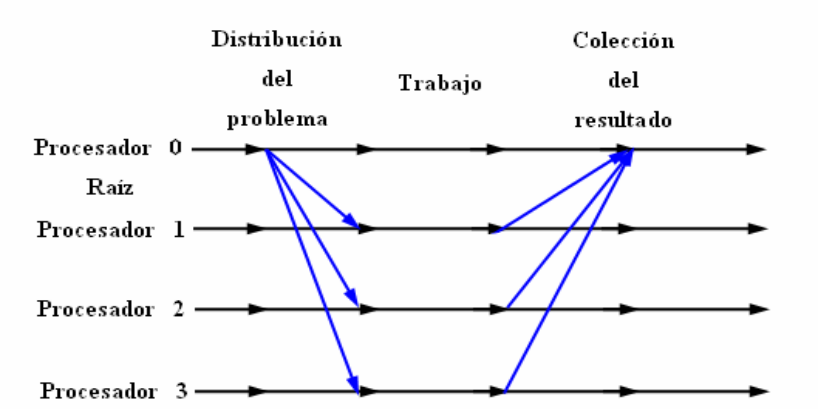

Figura 1. Modelo de ejecución CD

.

Este y otros modelos de computación en red, se denominan comúnmente por el término en inglés Net Computing, consisten básicamente en interconectar sistemas distribuidos para aprovechar de forma conjunta y coordinada sus recursos. Los diferentes tipos de sistemas distribuidos se pueden agrupar en:

- Cluster Computing. No es más que un grupo de ordenadores independientes interconectados entre sí. Los elementos conectantes son cables y un software especial de cluster.
- Intranet Computing. Busca explotar los ciclos subutilizados de un conglomerado de máquinas integrantes de una red LAN.
- Internet Computing. Busca explotar de manera diferente estaciones de trabajo ociosas y PC's para crear un sistema computacional distribuido poderoso con un alcance global y con capacidades de supercomputadoras.
- Peer-to-peer Computing. El termino "peer-to-peer" (P2P) se refiere a los tipos de sistemas y aplicaciones que emplean recursos distribuidos para realizar una función de una manera descentralizada. Algunos de los beneficios del aprovechamiento de P2P

son: la mejorar la escalabilidad evitando la dependencia en puntos centralizados; eliminar la necesidad de infraestructura costosa por el ensamblado directo de entre clientes; y permitir la agregación de recursos.

- Collaborative & Computing Portals. Es una iniciativa Web que pretende promover y distribuir toda la información relacionada con la súper computación entre los interesados en esta rama informática y de forma completamente gratuita. Esto incluye la publicación de noticias relacionadas con dicho tema, ya sean de carácter empresarial, software libre, proyectos universitarios, etc. En este modelo, las desventajas que se presentan son pocas y solamente tiene que ver con la velocidad en la interacción de los nodos, que esta determinada por el tipo de red que se va a utilizar, las distancias entre ellos, y la aplicación que los controle.

1.4 Cluster

Cluster es un grupo de múltiples ordenadores unidos mediante una red de alta velocidad, de tal forma que el conjunto es visto como un único ordenador, más potente que los comunes de escritorio. De un cluster se espera que presente combinaciones de los siguientes servicios:

- Alto rendimiento: Un cluster de alto rendimiento es un conjunto de ordenadores que está diseñado para dar altas prestaciones en cuanto a capacidad de cálculo. Los motivos para utilizar un cluster de alto rendimiento son:
    - el tamaño del problema por resolver y
    - el precio de la máquina necesaria para resolverlo.
- Alta disponibilidad.- es la calidad de estar presente, listo para su uso, a mano, accesible.
- Equilibrio de carga.- distribución de las tareas según las características potenciales, de los procesadores, como también de los recursos existentes en la rejilla.
- Fiabilidad.- es la alta probabilidad de un funcionamiento correcto.
- Escalabilidad.- es la capacidad de un equipo para hacer frente a volúmenes de trabajo cada vez mayores sin, por ello, dejar de prestar un nivel de rendimiento aceptable. Existen dos tipos de escalabilidad:

1. Escalabilidad del hardware (también denominada «escalamiento vertical»). Se basa en la utilización de un gran equipo cuya capacidad se aumenta a medida que lo exige la carga de trabajo existente.
2. Escalabilidad del software (también denominada «escalamiento horizontal»). Se basa, en cambio, en la utilización de un cluster compuesto de varios equipos de mediana potencia de modo que se pueden añadir nodos a un cluster para aumentar su rendimiento.

1.5 Grid

La computación en *Grid* o en malla es un nuevo paradigma de **computación distribuida** en el cual todos los recursos de un número indeterminado de computadores son englobados para ser tratados como un único supercomputador de manera transparente.

Estas computadoras englobadas no están conectadas o enlazadas firmemente, es decir no tienen porque estar en el mismo lugar geográfico. Se puede tomar como ejemplo el proyecto SETI@Home, en el cual trabajan computadoras alrededor de todo el planeta para buscar vida extraterrestre.

1.5.1 Tipos de Grids

La computación en *Grid* puede ser utilizada de varias maneras para tratar diferentes tipos de requerimientos de aplicación. A menudo, los *Grids* son categorizados por el tipo de soluciones que estos mejor manejan. Los tres tipos primarios de *Grids* son descritos a continuación, obviamente entre estos tipos no existen diferencias marcadas y a menudo los *Grids* pueden ser una combinación de dos o más de estos tipos. De cualquier forma, cuando se considera desarrollar aplicaciones que puedan correr en un ambiente *Grid*, es necesario recordar el tipo de ambiente *Grid* que será utilizado ya que este puede afectar varias de nuestras decisiones.

- ***Grid* computacional**

  Un *Grid* computacional esta enfocado específicamente hacia el lado de los recursos de potencia. En este tipo de *Grid*, muchas de las maquinas son servidores de gran rendimiento.

- ***Grid* de rebúsqueda**

Un *Grid* de este tipo es el más usado comúnmente con grandes cantidades de máquinas de escritorio. Las máquinas son rebuscadas para los ciclos ociosos disponibles del CPU y otros recursos. Los usuarios de las máquinas de escritorio generalmente dieron el control sobre sus recursos cuando estos estaban disponibles para participar en el Grid.

- ***Grid* de datos**

Un *Grid* de datos es responsable del almacenamiento y de proveer acceso a los datos a través de las múltiples organizaciones. Los usuarios no están concientes de donde se encuentran los datos hasta donde se les permita el acceso a estos. Por ejemplo, uno puede tener dos universidades haciendo búsquedas sobre la ciencia de la vida, cada una con sus propios datos. Un *Grid* de datos puede permitir a ambos compartir sus datos, gestionar los datos, y hacerse cargo de los asuntos de la seguridad tales como quién tiene acceso a qué.

1.6 Programación Paralela.

La Programación Paralela (PP) es una técnica basada en la ejecución simultánea de procesos o tareas en un mismo ordenador (con varios procesadores). Esta técnica enfatiza la verdadera simultaneidad en el tiempo de la ejecución de las tareas, garantiza un aumento en el rendimiento, ya que en los computadores mono procesador, el CPU es compartido por varios procesos al mismo tiempo (multiplexación ó multiprogramación). Una arquitectura de computadora paralela usa un simple espacio de dirección.

Los sistemas basados en este concepto, conocidos como multiprocesadores de memoria-compartida, permiten la comunicación entre procesadores a través de variables almacenadas y comparten el mismo espacio de memoria en este caso el mayor problema radica en sincronizar unas tareas con otras, ya sea mediante secciones críticas, semáforos ó paso de mensajes, esto para garantizar la exclusión mutua donde sea necesario. A continuación se muestra un gráfico del comportamiento de hilos siguiendo la técnica de PP a través de de un solo procesador.

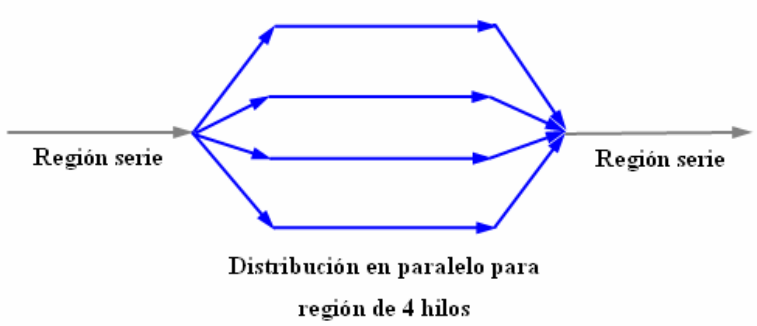

Figura 2. Flujo de un programa en un modelo de ejecución bajo PP

El paralelismo puede estar visto como un problema, ya que una máquina paralela es muy costosa. Pero, si tenemos disponibilidad de un conjunto de máquinas heterogéneas de pequeño o mediano porte, cuya potencia computacional sumada sea considerable, eso permitiría generar sistemas distribuidos de muy bajo costo y gran potencia computacional.

1.7 Taxonomía de Flynn.

La **taxonomía de Flynn** es una clasificación de arquitecturas de computador propuesta por Michael J. Flynn en 1972. Las cuatro clasificaciones definidas por Flynn se basan en el cardinal de instrucciones concurrentes (control) y en el de los flujos de datos disponibles en la arquitectura de la siguiente manera:

- Instrucción Simple / Dato simple (SISD).- computador secuencial que no explota el paralelismo en las instrucciones ni en flujos de datos. Ejemplos de arquitecturas SIS son las máquinas monoprocesador tradicionales como el PC o los antiguos mainframes.
- Instrucción Simple / Múltiples Datos (SIMD).- un computador que explota varios flujos de datos dentro de un único flujo de instrucciones para realizar operaciones que pueden ser paralelizadas de manera natural. Por ejemplo, un procesador vectorial.
- Instrucción Múltiple / Dato simple (MISD).- poco común debido al hecho de que la efectividad de los múltiples flujos de instrucciones suele precisar de múltiples flujos de datos. Sin embargo, este tipo se usa en situaciones de paralelismo redundante, como por ejemplo en navegación aérea, donde se necesitan varios sistemas de respaldo en

caso de que uno falle. También se han propuesto algunas arquitecturas teóricas que hacen uso de MISD, pero ninguna llegó a producirse en masa.

- Instrucción Múltiple / Múltiples Datos (MIMD).- varios procesadores autónomos que ejecutan simultáneamente instrucciones diferentes sobre datos diferentes. Los sistemas distribuidos suelen clasificarse como arquitecturas MIMD; bien sea explotando un único espacio compartido de memoria, o uno distribuido.

De los cuales nos interesa analizar el último caso, que se descompone en:

- Sistemas de Memoria Compartida.
- Sistemas de Memoria Distribuida.
- Sistemas de Memoria Compartida/Distribuida.

1.7.1 Sistemas de Memoria Compartida.

En este tipo de sistemas cada procesador tiene acceso a toda la memoria, es decir hay un espacio de direccionamiento compartido. Se tienen tiempos de acceso a memoria uniformes, ya que todos los procesadores se encuentran igualmente comunicados con la memoria principal y las lecturas y escrituras de todos los procesadores tienen exactamente las mismas latencias; y además el acceso a memoria es por medio de un ducto común. En esta configuración, debe asegurarse que los procesadores no tengan acceso simultáneamente a regiones de memoria de una manera en la que pueda ocurrir algún error.

Ventaja:

- La facilidad de la programación. Es mucho más fácil programar en estos sistemas que en sistemas de memoria distribuida.

Desventajas:

- El acceso simultáneo a memoria es un problema (conflictos de concurrencia).
- Poca escabilidad de procesadores, debido a que se puede generar un cuello de botella al incrementar el numero de CPU's.

1.7.2 Sistemas de Memoria Distribuida.

Estos sistemas tienen su propia memoria local. Los procesadores pueden compartir información solamente enviando mensajes, es decir, si un procesador requiere los datos

contenidos en la memoria de otro procesador, deberá enviar un mensaje solicitándolos. Esta comunicación se le conoce como *Paso de Mensajes* (MPI).

Ventaja:

- La escalabilidad. Las computadoras con sistemas de memoria distribuida son fáciles de escalar, mientras que la demanda de los recursos crece, se puede agregar más memoria y procesadores.

Desventajas:

- El acceso remoto a memoria es lento.
- La programación puede ser complicada.

Las computadoras MIMD de memoria distribuida son conocidas como *sistemas de procesamiento en paralelo masivo* (MPP) donde múltiples procesadores trabajan en diferentes partes de un programa, usando su propio sistema operativo y memoria. Además se les llama multicomputadoras, máquinas libremente juntas o *cluster*.

1.8 Balance de carga (Load Balancing)

Una de las cuestiones que se levantan en cualquier sistema de multiprocesamiento es el "load balancing". **Load balancing** es el problema de distribuir una tarea a un conjunto de procesadores en donde cada procesador tiene aproximadamente la misma cantidad de trabajo para ejecutar.

Como una analogía, considere un grupo de personas tratando de sacar el agua de un bote con baldes. Hay dos tamaños de baldes, un es el doble de grande que el otro. Cada uno saca el mismo número de baldes de agua del bote, pero los baldes grandes toman el doble de tiempo en vaciarse, porque son mucho mas pesados. Las personas que cargan los baldes mas pequeños quedarán libres durante bastante tiempo. En el tiempo que le toma a las personas con baldes grandes vaciarlos, las personas que cargan baldes chicos, podrá vaciar dos baldes. Claramente sacar el agua podría ser mucho más eficiente.

1.9 Herramientas y plataformas para sistemas distribuidos

1.9.1 MPI (Message Passing Interface)

MPI es un estándar para la comunicación inter-proceso en un esquema de multiprocesador de memoria-distribuida. Es una especificación para librerías de paso de mensajes, designada a ser un estándar para memoria distribuida, paso de mensajes, computación paralela.

El fin de MPI simplemente es proveer un amplio estándar usado para escribir programas de paso de mensajes. La interfaz intenta establecer un práctico, portátil, eficiente, y flexible estándar para el paso de mensajes.

MPI es el resultado de los esfuerzos de numerosos individuos y grupos en el transcurso de dos años.

La implementación del estándar es usualmente dejada para los diseñadores de sistemas en el cual MPI corre, pero una implementación de dominio público está disponible. MPI es un conjunto de rutinas de librería para C/C++, FORTRAN y Java. MPI es una interfaz estándar, por lo tanto el código escrito para un sistema puede fácilmente ser transportado hacia otro sistema con aquellas librerías.

Cuando un programa bajo MPI comienza, el programa se descompone en un número de procesos especificados por el usuario. Cada proceso corre y se comunica con otras instancias del programa, posiblemente corriendo en el mismo procesador o en diferentes. Lo mejor ocurre cuando los procesos están en medio de procesadores. La comunicación básica consiste en enviar y recibir datos de un proceso a otro y no como en OpenMP en donde la comunicación entre hilos ocurre mediante variables compartidas. Esta comunicación toma lugar en una red de trabajo muy rápida que conecta a los procesadores en un sistema de memoria-distribuida.

Un paquete de datos enviados con MPI requiere bastantes piezas de información: el proceso de envío, el proceso de recepción, la dirección de comienzo en memoria a ser enviados, el número de datos que han sido enviados, un identificador de mensajes, y el grupo de todos los procesos que puede recibir el mensaje. Todos estos objetos están disponibles para ser enviados por el programador. Por ejemplo, uno puede definir un grupo de procesos, y luego enviar mensajes solamente a ese grupo.

Algunas rutinas colectivas de comunicación no requieren todos los objetos. Por ejemplo, una rutina que permite a un proceso comunicarse con otro en un grupo, cuando es llamada por aquellos procesos, podría no requerir la especificación de recepción sino hasta que todos los procesos en el grupo, dispongan de un receptor.

En uno de los programas más simples en MPI, un proceso maestro despacha trabajo a los procesos trabajadores. Esos procesos reciben el dato, realizan tareas en él, y envían los resultados al proceso maestro, el que combinará los resultados.

Razones para el uso de MPI:

Estandarización: MPI es la única librería de paso de mensajes la cual puede ser considerada como estándar. Esta soportada sobre todas las plataformas HPC.

Portabilidad: Esto significa que no es necesario modificar la fuente del código cuando se importan las aplicaciones a otras plataformas diferentes que soportan MPI.

Rendimiento: Las implementaciones del vendedor deben ser capaces de explotar las cualidades del hardware propio para optimizar el rendimiento.

Funcionalidad: Más de 115 rutinas.

Disponibilidad: Una variedad de implementaciones están disponibles, para el vendedor como para el dominio público.

Uno de los grandes retos en la programación de esquemas de multiprocesamiento de memoria-distribuida es implementar una eficiente comunicación inter-proceso. La comunicación no está limitada por la simple relación maestro-trabajador. Podría ser muy bueno para el caso de que los procesos requieran datos o computar resultados desde cualquier proceso durante la ejecución. Puede, incluso, ser el caso de que cada proceso requiera el mismo dato de todos los demás procesos. Asegurando la sincronización de los procesos, es esos casos agrega un nivel de complejidad a programas desarrollados en un esquema de memoria-distribuida. Haciendo la comunicación eficiente, es de este modo, minimizar la sobrecarga involucrada en el pasaje de mensajes, agrega mucha mas complejidad.

MPI provee varias rutinas de comunicación para socorrer al programador en un ambiente de comunicación inter-proceso. Estas rutinas incluyen:

- **barredores.** puntos dentro del programa donde cada proceso espera hasta que todos los procesos se encuentren allí. Cuando este ocurre, cada proceso reinicia la ejecución
- **atasco de envíos y recepción.** las rutinas de paso de mensajes causan que un proceso espere hasta que un mensaje es enviado o recibido antes de continuar la ejecución
- **no-atasco de envíos y recepción.** las rutinas de paso de mensajes no causarán que un proceso espere hasta que un mensaje sea recibido o enviado antes de continuar con la ejecución
- **comunicaciones colectivas.** los mensajes pueden ser enviados de tres maneras diferentes: un proceso envía a otros procesos un mensaje, en un mensaje colectivo a un grupo de procesos, todos los procesos del grupo envían mensajes a todos los procesos en el mismo grupo, y todos los procesos en el grupo envían un mensaje a uno de los procesos.

El objetivo de la plataforma es un sistema de memoria distribuida incluyendo masivamente máquinas paralelas, *clusters* SMP (Simple Mesage Passing), redes de *clusters* y redes de maquinas heterogéneas (*grid*).
Todo paralelismo es explícito: el programador es responsable por la identificación correcta del paralelismo y de la implementación del algoritmo resultante utilizando constructores MPI.
El número de tareas dedicadas a correr un programa es estático. Nuevas tareas no pueden ser dinámicamente engendradas durante el tiempo de ejecución. (MPI-2 esta intentando tratar este tema).

1.9.1.1 Balance de carga en MPI

Al igual que programas de esquema multiprocesador de memoria-compartida, los programas bajo un esquema de memoria-distribuida debe resolver el problema de load balancing. El objetivo es mantener todos los procesos ocupados computando los resultados útiles, minimizando el costo de sobrecarga en comunicación. Le permitiré volver a la analogía de los baldes de agua. Esta vez, asuma que cada persona que tiene un balde, puede arrojar la misma cantidad de agua en el mismo lapso de tiempo (usando los mismos baldes, en cuanto a longitud se refiera) y cada persona con un balde se encuentra en diferentes compartimientos del barco, y los mismos están totalmente aislados entre ellos. Podríamos querer que el agua

sea distribuida de igual manera entre los compartimientos, de manera que el trabajo sea más rápido y eficiente.

1.9.2 PVM (Parallel Virtual Machine)[5]

Máquina Virtual Paralela es una librería para el cómputo paralelo en un sistema distribuido de computadoras. Está diseñado para permitir que una red de computadoras heterogénea comparta sus recursos de cómputo (como el procesador y la memoria RAM) con el fin de aprovechar esto para disminuir el tiempo de ejecución de un programa al distribuir la carga de trabajo en varias computadoras. Provee una estructura unificada al alcance de los programas paralelos que pueden ser desarrollados de una manera eficiente y simple utilizando hardware existente.

Esta herramienta crea una nueva abstracción, que es la máquina paralela virtual, empleando los recursos computacionales libres de todas las máquinas de la red que pongamos a disposición de la biblioteca. Es decir, dispone de todas las ventajas económicas asociadas a la programación distribuida, ya que se emplea los recursos hardware de dicho paradigma; pero programando el conjunto de máquinas como si se tratara de una sola máquina paralela.

PVM permite que una colección de sistemas de computadoras heterogéneas sea vista como una sola maquina virtual. PVM maneja transparentemente todos los mensajes de rutina, conversión de datos, y la programación de tareas a través de una red de arquitecturas computacionales incompatibles.

El modelo de computación PVM es simple, todavía muy general, y satisface una gran variedad de estructuras de programas de aplicación. La interfase de programación es deliberadamente simple, permitiendo así estructuras de programas simples para ser implementadas de una manera intuitiva. El usuario escribe su aplicación como una colección de tareas colaborativas. Las tareas de acceso a recursos PVM a través de una librería de una interfaz de rutinas estándar. Estas rutinas permiten la inicialización y finalización de tareas a través de la red así como la comunicación y sincronización entre tareas. Las primitivas de paso de mensajes en PVM están orientadas hacia operaciones heterogéneas, involucrando tipos de estructuras fuertes para el almacenamiento en memoria y la transmisión. Las

[5] Ref. [Ismael, 2004]

estructuras de comunicación incluyen el envío y recibo de estructuras de datos así como primitivas de alto nivel como transmisión, barreras de sincronización, y la suma global.
Las tareas PVM pueden poseer estructuras de control y dependencia arbitrarias. En otras palabras, cualquier punto en la ejecución de una aplicación concurrente, cualquier tarea en existencia pueden comenzar o detener otras tareas, o añadir o quitar computadoras de la maquina virtual. Cualquier proceso puede comunicarse y/o sincronizarse con cualquier otro. Cualquier estructura de dependencia y control puede ser implementada bajo el sistema de PVM con el uso apropiado de los constructores de PVM y las declaraciones de control propias lenguaje del servidor.
La máquina paralela virtual es una máquina que no existe, pero un API apropiado permite programar como si existiese. El modelo abstracto que permite usar el API de la PVM consiste en una máquina multiprocesador completamente escalable (es decir, que podemos aumentar y disminuir el número de procesadores *en caliente*). Para ello, va a ocultar la red que se esté empleando para conectar nuestras propias máquinas, así como las máquinas de la red y sus características específicas. Este planteamiento tiene numerosas ventajas respecto a emplear un supercomputador, de las cuales, las más destacadas son:

- **Precio.** Así como es mucho más barato un computador paralelo que el computador tradicional equivalente, un conjunto de ordenadores de mediana o baja potencia es muchísimo más barato que el computador paralelo de potencia equivalente. Además, al no ser la PVM una solución que necesite de máquinas dedicadas (es decir, el *daemon* de PVM corre como un proceso más), podemos emplear en el proceso los tiempos muertos de los procesadores de todas las máquinas de nuestra red a las que tengamos acceso. Es decir, si ya tenemos una red montada, el costo de tener un supercomputador paralelo va a ser cero ya que disponemos de las máquinas, no tendremos que comprar nada nuevo, y además la biblioteca PVM es software libre, por lo que no hay que pagar para usarla.

- **Disponibilidad**. Se refiere en cuanto a que los componentes son fáciles de adquirir, por ejemplo: todo centro de cálculo tiene una cantidad mínima de máquinas obsoletas,

y que no están siendo usadas. Con esa determinada cantidad de máquinas "inservibles" que por las características de hardware no pueden correr ni aplicaciones livianas como un Word para Windows, se puede instalar Linux, la PVM y añadirlo al supercomputador paralelo virtual que conforma las máquinas que ya tendríamos en red.

- **Tolerancia a fallos**. Si por cualquier razón fallara uno de los ordenadores que conforman la PVM y el programa que la usa está razonablemente bien implementado. La aplicación puede seguir funcionando sin problemas. Siempre hay alguna razón por la que alguna máquina puede fallar, y la aplicación debe continuar haciendo los cálculos con aquel hardware que continúe disponible.
- **Heterogeneidad**. Se puede crear una máquina paralela virtual a partir de ordenadores de cualquier tipo. La PVM nos va a abstraer la topología de la red, la tecnología de la red, la cantidad de memoria de cada máquina, el tipo de procesador y la forma de almacenar los datos.

1.9.3 Globus Toolkit

Es un proyecto fundado en Octubre de 1996 por la agencia DARPA del gobierno de los Estados Unidos con ese nombre, para crear herramientas de computación distribuida. Este equipo creó una colección de herramientas para las actividades de la computación en *Grid* en las comunidades académicas e investigativas.

1.9.3.1 Python Globus (PyGlobus)

Es un proyecto, que tiene la finalidad de hacer accesible el uso completo del kit de herramientas Globus para Python. SWING es utilizado para generar el código de interfaz necesario. Actualmente han sido incluidas la mayoría de las versiones 2.4 y superiores del kit de herramientas Globus.

Los objetivos propuestos por este proyecto son:

- Proveer de una interfase amigable orientada a objeto para el kit de herramientas Globus.
- Proveer un rendimiento similar utilizando lo menos posible código C.

- Minimizar el número de cambios necesarios cuando algunos aspectos de Globus cambien.
- Hacer que Globus sea tan natural de utilizarlo desde Python, como sea posible. Por ejemplo, el módulo io permite la manipulación de GSITcpSocket como Python los objetos socket.

Capítulo 2: Análisis y Diseño de la solución.

Sumario

- Introducción.
- Descripción del problema y solución propuesta.
- Modelo de dominio.
- Requerimientos funcionales del sistema
- Requerimientos no funcionales del sistema.
- Requisitos de Hardware y Software.
- Requisitos de usabilidad.
- Requisitos de rendimiento.
- Requisitos de diseño e implementación.
- Modelos y diagramas.

2.1 Introducción

En este capítulo, se propone el diseño para el desarrollo de una plataforma para el trabajo distribuido, tomando en cuenta la necesidad de máquinas potentes, no solo en el instituto si no también a niveles nacionales; capaces de realizar cálculos muy complejos, o alguna otra tarea que requiera gran potencia computacional, o simplemente agilizar los procesos utilizados cotidianamente.

Con el fin de obtener una solución para este problema, se propone la creación de un sistema para el cómputo distribuido sobre Python, capaz de combinar recursos subutilizados de máquinas pequeñas y económicas, dando como resultado un computador mucho mas potente y veloz, y lo más importante es que estamos utilizando nuestros propios medios para lograrlo.

Seguidamente, se realiza el modelado del proceso que tiene que ver con el objeto de estudio, se enumeran los requisitos funcionales y no funcionales que debe tener el sistema propuesto, lo que permitirá tener una concepción general del mismo.

2.2 Descripción del problema y solución propuesta

Una de las necesidades más grandes que tiene el ISMM, es la adquisición de máquinas relativamente poderosas en cuanto a cálculo y procesamiento de datos, a raíz de no contar con los recursos económicos suficientes para este fin; el problema se centra en que la mayoría de las especialidades que oferta esta institución tiene como base de prestigio carreras ligadas al trabajo en terrenos de diferente constitución y forma, ó involucrados con sistemas eléctricos donde la simulación de ciertas situaciones es esencial para el desarrollo profesional, y los actuales equipos no dan abasto a estas exigencias, por lo tanto se resuelve crear una aplicación capaz de combinar las características individuales de cada computador, de esta manera aprovechar al cien por ciento los recursos con que contamos y lograr un entorno distribuido que de la sensación al usuario que está trabajando con una máquina mucho más potente y veloz de las que hasta ahora a podido utilizar.

2.3 Modelo de dominio

Teniendo en cuenta que el negocio tiene un bajo nivel de estructuración, nos basaremos en un modelo de dominio.

En el modelo de dominio se describen las clases más importantes en el contexto del sistema, a la vez que permite mostrar los principales conceptos que se manejan en el dominio del sistema en desarrollo mediante el lenguaje UML.

El elemento fundamental del sistema es el servidor (Medulla), todo depende de él. El servidor (Medulla) interactúa con las peticiones del cliente (Heracles) realizadas por el usuario del sistema. Esto ocurre en todas las conexiones con el resto de los Medulla-hermanos, donde:

1. Medulla, deberá conectarse a la Hydrandad (conjunto de todos los servidores Medulla activos), obtener la lista de los servidores activos que conforman el *Grid*, difundir dependencias con los ficheros predefinidos por el usuario y será el responsable de ejecutar otras aplicaciones Hydra.
2. Heracles por su parte, es encargado de conectar el servicio con otro hermano, así mismo activar el servicio Hydra, encargarse de la transferencia de ficheros y la ejecución de módulos.

3. hydra_egg.py, esta es una librería Python para el desarrollo de aplicaciones de cómputo distribuido sobre Hydra, es decir el soporte para el programador.

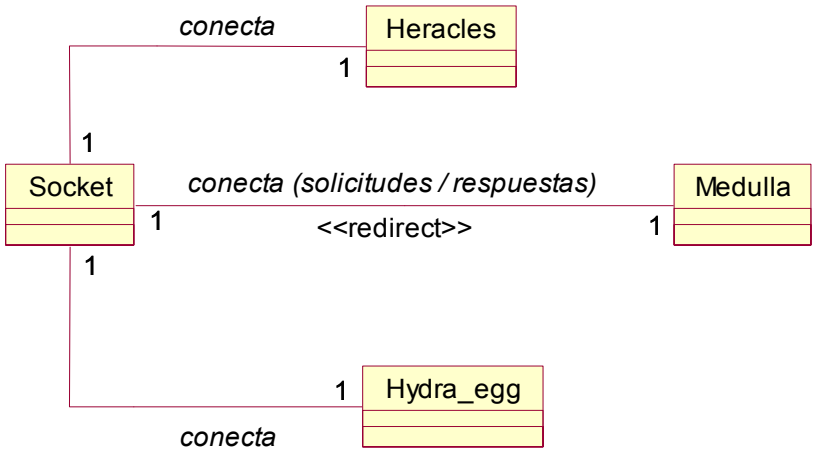

Figura 3. Modelo de dominio del sistema.

2.4 Requerimientos funcionales del sistema

Los requerimientos funcionales definen las responsabilidades del sistema, o sea, las funciones que éste será capaz de realizar.

R1. (Sostener y alcanzar la conectividad) Reconocimiento de los nodos que integrarán el *grid*.

R2. Garantizar la distribución y ejecución de tareas, la difusión de las dependencias entre los ficheros en el gris, así como el paso de mensajes entre los nodos del mismo.

Por su parte cada componente responde individualmente a sus propios requerimientos funciones, así como:

Medulla: Servidor de aplicación.

1. Conectarse a la Hydrandad.
2. Obtener la lista de hermanos.
    - Se envía un acuse de recibo a las máquinas existentes en un listado de nodos.
    - En caso de encontrarse más nodos, la lista se va actualizando manualmente y ocurre lo anterior.

- En caso contrario, se redistribuye las tareas asignadas teniendo en cuenta sobre todo la disponibilidad del sistema.

3. Difundir dependencias con los ficheros predefinidos por el usuario.
4. Ejecutar una aplicación Hydra.
    - Garantizar el paso de mensajes.
    - Asignar Id's.
    - Obtener estados.

Heracles: Cliente para el control del sistema Hydra.

1. Conectar el servicio con otro hermano.
2. Activa el servicio Hydra.
3. Transferencia de ficheros.
4. Ejecución de módulos.

hydra_egg: Librería Python para el desarrollo de aplicaciones de cómputo distribuido sobre Hydra.

1. Iniciación de la aplicación.
2. Asignación de Id's a los procesadores.
3. Paso de mensajes a los procesadores.
4. Recibo de mensajes de los procesadores.
5. Control del número de integrantes del *Grid*.
6. Control del estado del procesador (terminó de procesar, procesando, desocupado).
7. Cerrar conexiones.

2.5 Requerimientos no funcionales del sistema

Del sistema las partes que interactuarán con el usuario tendrán una interfaz amistosa que favorezcan a la configuración del sistema.

Para el caso de Hydra, la cual no cuenta con interfaz visual específica, no son necesarias hacer caracterizaciones de la misma. Sin embargo en el proceso de diseño e implementación deberían responderse a requerimientos favorables tales como:

2.5.1 Requisitos de software y de hardware

- Para la correcta ejecución de esta aplicación no se necesita de ningún tipo de software en específico, salvo que el sistema soporte la versión 2.4 de Python, que es la que se utilizará para el desarrollo de la plataforma; tomando en cuenta que este módulo es disponible sobre todos los modernos sistemas Unix, Windows, MaCos, BeOS, OS/2, y probablemente en plataformas adicionales.
- Los requisitos de hardware que Python necesita, no son específicos para cada uno de los casos en todas sus versiones, sino que se resume solamente en plantear que Python es soportado por todo tipo de procesadores, incluyendo la familia de x86. Obviamente la velocidad de respuesta de cada petición, la interfaz gráfica que uno quiera montar y no solo en este caso el tipo de utilización que se le dará al *software* dependerá de las características propias de cada computador.
- Hydra, necesita que las maquinas que conformarán el Grid tengan algún tipo de salida de red, para la comunicación entre ellas, debido a que este tipo de colaboración inter-nodo se puede realizar no solo en una red LAN sino también vía Internet, por que el protocolo que se utiliza es el TCP/IP, no obstante los nodos de un Cluster pueden conectarse mediante una simple red Ethernet con placas comunes (network adapters o NIC'S) , o utilizarse tecnologías especiales de alta velocidad como Fast Ethernet, Gigabit Ethernet, Myrinet, Infiniband, SCI, etc. Requisitos de usabilidad

- La aplicación será capaz de correr sin problema en los distintos tipos de sistemas operativos en que la versión 2.4 de Python sea soportada; ya que un número de características no ha sido soportado desde la versión Python 2.3, y el código para soportarlas ha sido quitado en la versión de Python 2.4. Para más información refiérase a www.python.org o si tiene una versión instalada de Python lea el contenido equivalente en **readme.txt**.
- Los Sockets y sus módulos están disponibles en todos los sistemas modernos Unix, Windows, MacOS, BeOS, OS/2, y probablemente en plataformas adicionales. **Nota:** algunos comportamientos pueden depender de la plataforma, desde que las llamadas se realizan a los sockets de las API's de los sistemas operativos. La interfase de Python es una simple transliteración de las llamadas de los sistemas Unix y la librería de interfase para el estilo objeto-orientado de Python: la función socket() devuelve un objeto socket tales métodos implementan las varias llamadas al sistemas de sockets. Los tipos parámetro son un tanto más de alto nivel que las interfaces en C: como las operaciones read() y write()en los archivos de Python, la asignación de buffer en las operaciones de recibir es automático, y el tamaño del buffer esta implícito en las operaciones de envío.

2.5.2 Requisitos de Rendimiento

- El tiempo de respuesta dependerá de la programación, del SO, de la plataforma de hardware donde corre el sistema y del tipo de conexión de red.
- Las altas prestaciones en cuanto a capacidad de cálculo, dependerán de la eficiencia de los algoritmos de implementación de esta plataforma.

2.5.3 Requisitos de diseño e implementación

Como lenguaje de programación se utiliza Python por las facilidades que brinda para este tipo de aplicaciones[6].

---

[6] *Especificaciones del lenguaje Python, Cap. III,* Lenguaje de implementación

2.5.4 Requisitos de disponibilidad

- Referente a este requisito, el sistema se puede concebir en máquinas estándar, con las que ya se cuenta, ó se pueden adquirir a costos muy bajos comparados con el computador paralelo de potencia equivalente.
- Por otro lado el software que se está utilizando es libre, bajo licencia GNU, disponible en http://www.python.org .

2.6 Modelo de Casos de Uso del Sistema

Durante el proceso de análisis de los requisitos funcionales del sistema se extrajeron posibles actores, responsabilidades, funcionalidades e interacciones entre estos, descritos a continuación mediante diagramas de casos de uso.

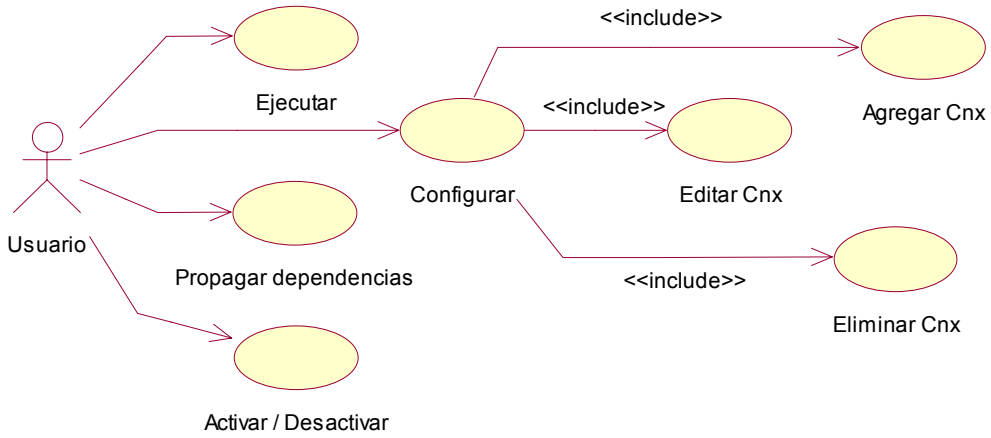

Figura 4. Diagrama de casos de uso del sistema.

En el diagrama de la Figura 5 se detalla explícitamente las responsabilidades del cliente (Heracles), el cual prácticamente realiza una copia de las solicitudes del usuario hacia el servidor.

Del mismo modo en el diagrama de la Figura 6 se describen las diferentes funciones que realiza el servidor (Medulla) después de conocer las peticiones del cliente.

2.7 Definición de los actores de los casos de uso del sistema

Actores	**Justificación**
Usuario	Será el que ejecute el cliente (Heracles) y a partir de ahí, todas las funciones del mismo.

A continuación se presentan los casos de uso determinados para satisfacer los requerimientos funcionales de sistema:

CU-1	Ejecutar
Actores	Usuario
Descripción	Ejecuta Hydra, esta a su vez automáticamente realiza las conexiones entre nodos para la utilización necesaria de recursos.
Referencia	R1

CU-2	Configurar
Actores	Usuario
Descripción	Se configuran 3 procesos fundamentales entre los nodos, es decir agregar nodo, editar la información referente a un nodo, eliminar un nodo ó ignorarlo.
Referencia	R1

CU-3	Propagar dependencias
Actores	Usuario
Descripción	Se copian en cada nodo que lo necesite los módulos necesarios para el funcionamiento adecuado de los *scripts*.
Referencia	R2

CU-4	Activar / Desactivar
Actores	Usuario
Descripción	Se refiere a activar o desactivar el servidor Hydra de la máquina.
Referencia	R2

2.8 Diseño del Sistema.

2.8.1 Diagrama de Iteración del diseño

Para la explicación gráfica del comportamiento interno de Hydra, se utilizará un diagrama de **secuencia**, por ser en este caso capaz de plasmar las relaciones entre clases, iteración de estas con el usuario, y entre las instancias de las mismas. El mismo para su mayor comprensión será desglosado en partes, empezando por lo que se plantea la primera iteración general entre el usuario y el cliente, para más adelante comprender cada una de las interacciones existentes entre este y el servidor.

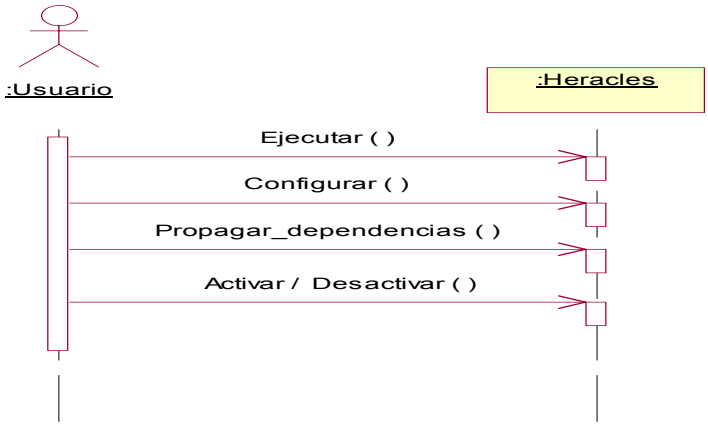

Figura 7 Diagrama de secuencia "general".

En este diagrama se muestran las diferentes acciones que el usuario puede solicitar al cliente, descritas anteriormente en los casos de uso.

En las figuras 8, 9, 10 y 11 se muestran los diagramas referentes al desglosamiento del diagrama de secuencia "general".

2.8.2 Diagrama de Clases del diseño

Este diagrama describe gráficamente las especificaciones de las clases de software y de las interfaces en Hydra.

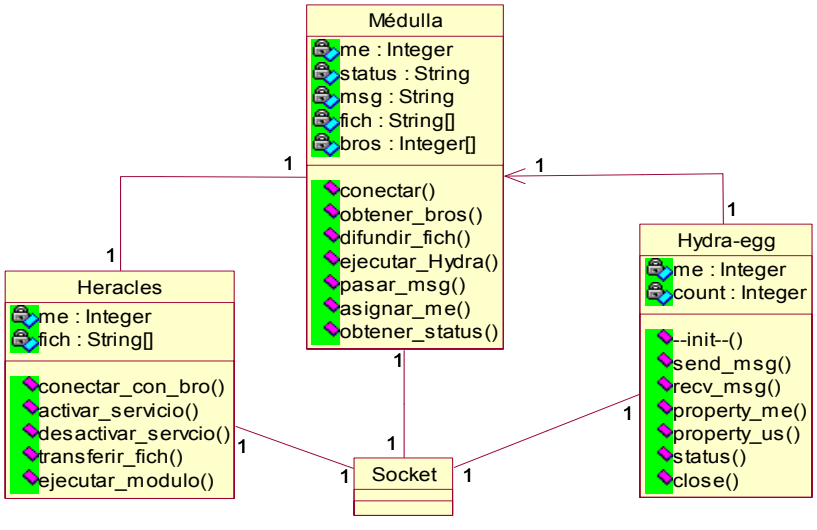

Figura 8 Diagrama de Clases del diseño.

2.9 Descripción general de las clases

Nombre: hydra_egg	
Tipo de clase: Controladora	
Atributo	**Tipo**
Me	Integer
Us	Integer
Para cada responsabilidad:	
Nombre:	**Descripción**
--init--()	Iniciación de la aplicación.
send_msg()	Pasa una cadena a la dirección (id) del procesador; de ser un id negativo el mensaje (id remitente, información) es pasado a todos los nodos del Grid, caso contrario el mensaje llega al procesador específico.
Recv_msg()	Recibe una cadena de la dirección (id) del procesador, si no contiene ningún argumento o el id es negativo, se espera mensaje de todos los integrantes del Grid, caso contrario se espera mensaje de un destinatario en específico; para cualquier caso, se devuelve una dupla [origen, mensaje].
property_me()	Hace uso del id propio asignado a cada nodo, dado por un número hasta N-1
property_us()	Obtiene la cantidad de nodos que integran en Grid.
Status()	Se le pasa el estado de otro procesador de interés: (terminó de procesar, esta procesando, desocupado).
close()	Cierra la conexión.

Nombre: Medulla	
Tipo de clase: Controladora	
Atributo	**Tipo**
Me	Integer
Status	String
Msg	String
Fich	String []
Bros	Strings []
Para cada responsabilidad:	
Nombre:	**Descripción**
conectar()	Realiza la ejecución del método para la conexión a la "hydrandad".
obtener_BROS()	Obtiene la lista de los hermanos existentes dentro del Grid.
difundir_fich()	Esta operación está asociada a Ejecutar. Se difunden por todas las "hermanas" ejecutantes los ficheros de los que depende la aplicación Python a distribuir.
ejecutar_Hydra()	Ejecuta una aplicación Hydra.
pasar_msg()	Garantiza el correcto paso de menajes entre los procesadores.
asignar_me()	Asigna identificadores propios a cada procesador.
obtener_status	Obtiene el estado actual de cada procesador.

Nombre: Heracles	
Tipo de clase: Controladora	
Atributo	**Tipo**
Me	Integer
Fich	String[]
Para cada responsabilidad:	
Nombre:	**Descripción**
conectar_con_bro()	Se encarga de conectar el servicio con otro hermano.
activar_servicio()	Activa el servicio Hydra.
desactivar_servicio()	Desactiva el servicio Hydra.
transferir_fich()	Se encarga de la transferencia segura de ficheros.
ejecutar_modulo()	Ejecuta el módulo importado por el usuario.

2.10 Diagrama de despliegue

El modelo de despliegue es un modelo de objetos que describe la distribución física del sistema en términos de cómo se distribuye la funcionalidad entre los nodos de cómputo.

Representa la arquitectura física de un sistema de cómputo. Puede mostrar cada equipo de cómputo y dispositivo en el sistema y los componentes que en ellas residen.

UML ilustra la forma en que luce un sistema físicamente pues un sistema consta de nodos, donde cada nodo se representa por un cubo. Una línea asocia a dos cubos y simboliza la conexión entre ellos.

El modelo de despliegue está compuesto por:

1. Nodos, características y conexiones
2. Clases activas sobre los nodos
3. Vista de la arquitectura del modelo de despliegue

La arquitectura física se compone de un nodo Cliente/Servidor (C/S) que aloja la aplicación interactuando con otro u otros nodos donde se crea una copia de la aplicación convirtiendo a este ultimo en otro nodo C/S, para de esta forma obtener la interconectabilidad entre estos, el número de nodos que pueden integrar el sistema no es finito. Para desarrollar toda esta funcionalidad, los nodos se interconectan a través de TCP/IP. La figura siguiente muestra el diagrama de despliegue correspondiente.

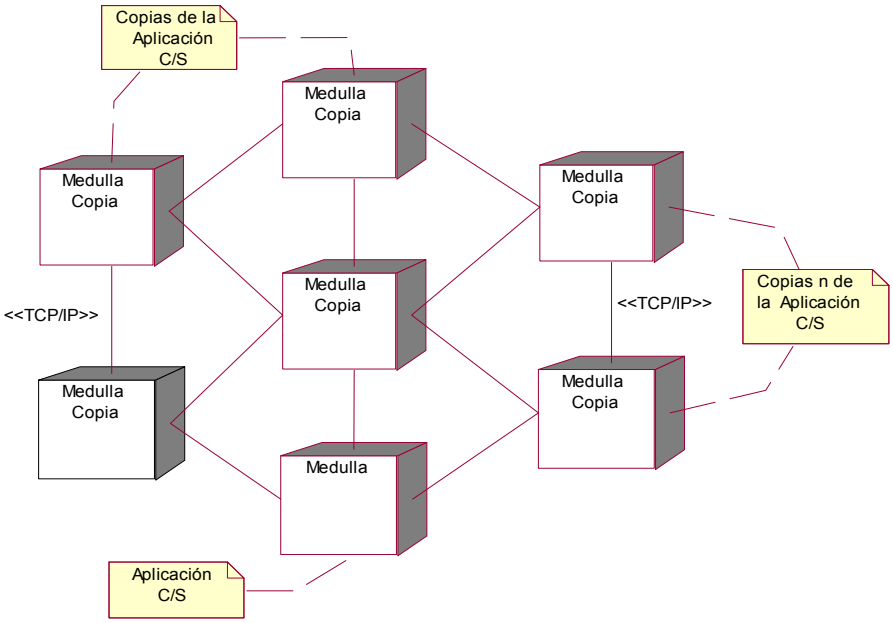

Figura 9 Diagrama de Despliegue.

Capítulo 3: Detalles de implementación

Sumario

- Introducción
- Arquitectura general de la *Hydra*
    - Comunicaciones sobre *Hydra*
- Ejecutantes
    - Objetos Hydra
    - Esquema básico
- El servidor *Medulla*
- El cliente *Heracles*
- El producto final

3.1 Introducción

Como ya se ha mencionado, la *Hydra* es un sistema de tres partes: el servidor de aplicaciones *Medulla* los clientes *Heracles* y los clientes ejecutantes derivados de los módulos *hydra_egg.py* que en conjunto constituyen la plataforma para la ejecución distribuida de tareas de programación. También se ha mencionado que al igual que sistemas como MPI, *Hydra* es una plataforma de procesamiento y memoria distribuidos según el esquema de el mismo código que alterna según la identidad del ejecutante (provista por la *Medulla* regente) y que la comunicación entre procesos se logra mediante el pase de mensajes con los *sockets* de la librería *socket.py*.

A continuación se hará una descripción de los distintos componentes de la plataforma, así como de los recursos asociados a los mismos y se darán detalles relativos a su funcionamiento y estructuración.

3.2 Arquitectura general de la Hydra

La plataforma *Hydra* establece un canal de comunicación con protocolo propio (SAVIA) entre computadoras conectadas por una red LAN, de modo que cada máquina con un servidor

Medulla constituye un nodo y a una rejilla de nodos *Hydra* se les conoce por *visión* (figura 10). El centro de mando de una visión viene dada por una *medulla regente* quien dirigirá la ejecución de tareas, así como llevará el control del paso de mensajes entre ejecutantes.

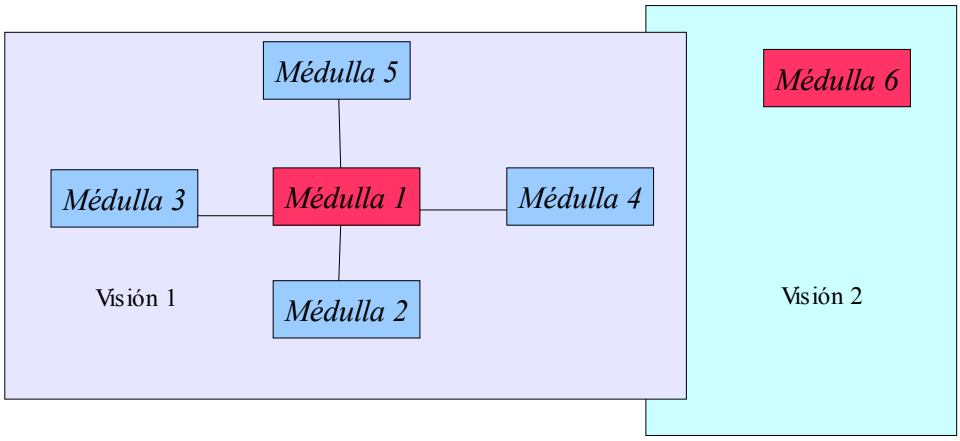

Figura 10 Visiones en una red Hydra. En rojo las medullas regentes.

La medulla regente es siempre la primera de la visión (por defecto al iniciarse una medulla se carga como regente a no ser que se especifique otra cosa). La segunda medulla en conectarse a ésta será la primera sucesora y así sucesivamente. De modo que puede haber visiones de un solo nodo, de dos o más en dependencia de cuántas medullas se hayan conectado a la regente de una visión determinada. El hecho de la existencia de una cadena de sucesión en la regencia garantiza que mientras queden medullas, la visión se sostendrá, difundiéndose desde la regente a las subordinadas los cambios en la visión; no así la ejecución de tareas, que se verá afectada por una desconexión repentina de la regente.

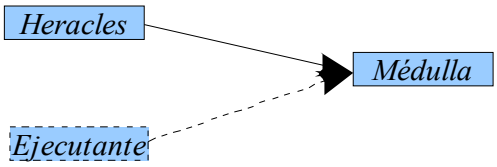

Figura 11Componentes de la Hydra en una máquina

Haciéndole una ampliación a un nodo, puede apreciarse que está compuesto por el servidor y el cliente *Heracles,* y eventualmente podrá contar con un ejecutante derivado del módulo *hydra_egg.py* (figura 11). Este último caso depende de la ejecución de alguna tarea en el nodo. En cada caso habrá otros ficheros de configuración o temporales, específicos a cada componente y según las acciones que estos realicen. No se preveen ficheros temporales del lado del desarrollador *Hydra* pues es evidente que eso depende de la implementación específica de cada tarea.

3.2.1 Las comunicaciones en *Hydra*

Toda la comunicación entre las partes se realiza mediante *sockets* de flujo de cadenas (*STREAM sockets*), desarrollados en el módulo *sockets.py* (ver la documentación de Python 2.4), lo cual conlleva a la la emisión en forma de representaciones de cadena de todos los tipos de datos que se intercambian.

Este hecho no debe perderse de vista, en especial es cuando se vayan a difundir archivos binarios mediante el comando **broadcast,** pues debe indicarse que son binarios para que se realice su conversión a cadena y puedan transferirse y reconstruirse correctamente, utilizando para ello los métodos que implementa la librería *uu.py*, especializada en codificación y decodificación de archivos no textuales. Tampoco puede ser desconocido por parte del desarrollador de aplicaciones sobre *Hydra*, en tanto deberá, una vez "recibido" un mensaje, saber con exactitud qué tipo de datos fue enviado.

3.2.1.1 SAVIA

La comunicación inter e intranodos se rige según las regulaciones del protocolo SAVIA cuyos comandos siempre se encierran entre paréntesis angulares y un par de "~":

<~LETRA_INSTRUCCIÓN~> PARÁMETROS

Donde los parámetros varían según cada instrucción.

Debe conocerse que hay dos grupos de instrucciones: las de respuesta y las de solicitud de consulta. Las primeras corresponden a indicaciones de confirmación, error o aceptación de una

instrucción de solicitud de consulta. El segundo caso es fácilmente reconocible pues va siempre encerrado entre los pares <~*l*~> y <~*k*~>, que indican la petición de apertura de comunicación entre un cliente y el servidor y la llamada de cierre del enlace para que el servidor continúe atendiendo a otras llamadas. Visto a manera de esquema, toda solicitud de consulta sigue la sintaxis:

<~l~> tcliente host instrucción parámetros
...
...Datos e instrucciones que se intercambian
...
<~k~>

donde *tcliente* representa el tipo de cliente que está haciendo la petición de enlace y puede tomar tres valores: *heracles, medulla* y *performer*. El *host* es el 2o parámetro y el mismo debe contener el nombre o número IP de la máquina o valor del *me* del ejecutante, según el caso que sea. Después siguen la *instrucción* o consulta (estado de un ejecutante determinado, envío o recepción de mensajes, ejecución de algún módulo, entre otras) y sus parámetros. Normalmente, después de la solicitud de enlace le sigue una respuesta afirmativa o negativa del servidor y así sucesivamente se produce el intercambio de datos o se procesa el cierre de conexión.

Cada tipo de cliente tiene un juego propio de instrucciones SAVIA, asociadas con las funcionalidades de cada uno y las respuestas del servidor. En el anexo C se muestra la especificación general del protocolo.

3.2.1.2 STREAM-SOCKETS en *Hydra*

Algo que debe tenerse en cuenta en todo momento es que los sockets que utiliza Hydra son STREAM-SOCKETS, o sea de flujo de cadenas de caracteres y de hecho los datos que se pasan por los métodos *send* y *recv* son de tipo cadena, por lo que los desarrolladores no pueden perder este hecho de vista y deben preparan formas de recuperación y conversión de los datos enviados como mensajes durante la ejecución. Es decir, que es el desarrollador de

aplicaciones para Hydra quien determina qué datos se envían por mensajes y cómo se procesan de cada lado (emisor/receptor).

Por ejemplo, si se envía la representación de cadena de un valor entero, el receptor debe, si quiere usarlo como entero, recurrir a algún método de reconversión (*int* por ejemplo). Por supuesto que pueden implementarse procesamientos más complicados, en dependencia de los deseos del programador.

Otro detalle a tener en cuenta es qué cantidad de información fluye en cada momento por la plataforma, pues debe lograrse un balance para que ni la cantidad de información sea insuficiente para la solución de la tarea dada, ni tampoco que llegue a saturar la red (ni la capacidad en memoria del servidor) con constantes paquetes que pudieron ahorrarse con una mejor implementación de la tarea a distribuir.

3.2.1.3 Seguridad en las comunicaciones de Hydra

El tema de la seguridad en la comunicación de los sistemas de cómputo distribuido normalmente se desvía hacia dos tendencias: dejarle todo al desarrollador o brindar algunos esquemas de seguridad (encriptación/desencriptación del paso de mensajes, uso de claves públicas y privadas, etc.). También *Hydra* debe garantizar, aparte de la seguridad, la confiabilidad del paso de mensajes: es decir, que el mensaje saliente llegue a su destino y que además llegue correctamente, sin errores, ni pérdidas.

El esquema de seguridad del primer prototipo de Hydra, como más bien es una versión experimental, deja a un lado la encriptación y supone que el desarrollador controlará la seguridad y corrección de la información puesta en los mensajes entre ejecutantes. Tampoco protege las partes del sistema de comunicaciones SAVIA, aunque se prevén en versiones futuras formas de encriptación de los mensajes y extensiones seguras de SAVIA.

3.2.2 Arquitectura Cliente/Servidor

Como ya se ha visto la plataforma está diseñada para un comportamiento de tipo cliente servidor; pero con la particularidad de que mientras la relación entre Heracles y Medulla es la típica relación de dos capas donde el primero viene apenas a ser una interfaz del otro quien tiene a su cargo la mayor parte del procesamiento; mientras que entre un ejecutante y la

medulla regente la relación existente cambia pues en general el primero tiene prácticamente todo el peso del procesamiento y utiliza al servidor para obtener pequeños datos estaduales de la ejecución y poner y extraer los mensajes entre los ejecutantes. Pero de cualquier manera, como es imprevisible la forma y acciones que llevará a cabo un ejecutante, pues dependen de la voluntad de los desarrolladores de CD, se llega a notar la presencia de tres niveles de comportamiento: interfaz (bien sea como Heracles o Ejecutante), lógica o de procesamiento y una para las comunicaciones.

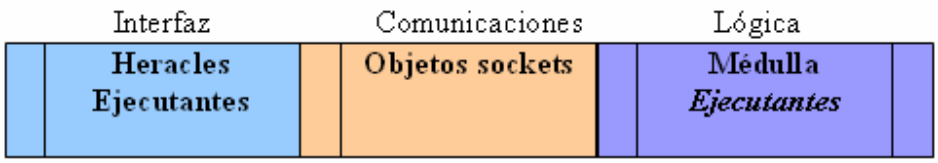

Figura 12 Diseño de 3 capas de la plataforma Hydra.

3.3 Los Ejecutantes

Desde el punto de vista del desarrollador de aplicaciones sobre *Hydra,* es vital conocer algunas características del modelo según el cual se deben regir estas aplicaciones. En primer lugar, los ejecutantes deben seguir la idea de "un único código, varias alternativas", que consiste en lograr varios comportamientos utilizando el mismo código mediante condicionales alimentadas con el identificador único (el *me*) que le es asignado, de modo que cada ejecutante pueda realizar una parte de la tarea total y comunicarse con el resto de sus compañeros para alimentar sus entradas de datos.

3.3.1 La clase *Hydra* de *hydra_egg.py*

Como se verá más adelante en el esquema básico de aplicación para CD que se sugiere a los desarrolladores, en la librería *hydra_egg* se garantiza una forma para obtener la información básica del ejecutante, así como permitir el envío y recepción de mensajes, acciones que están previstas en el comportamiento de las instancias de la clase *Hydra,* declarada en dicha librería. Aunque ya el esquema de clases ha sido visto y descrito es bueno volver sobre él y esclarecer algunos detalles en función del desarrollador de aplicaciones.

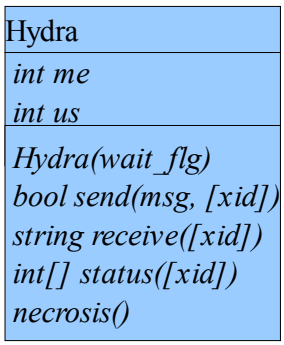

Figura 13 Diagrama de la clase Hydra

Los atributos de las instancias de Hydra son principalmente el *me* y el *us* (con sus respectivos métodos de acceso), quienes indican el identificador de ejecutante y la cantidad de ejecutantes invocados para dicha ejecución. En este apartado se conocen además como atributos, aunque no sean visibles al programador: el *minimum* cuyo valor proviene de la cantidad mínima necesaria de ejecutantes para llevar a cabo la tarea; el valor del puerto del *socket* y el nombre o IP del regente; una conexión a dicho *socket*; la cantidad de bytes que se han de trasmitir máximo en cada ocasión; entre otros.

Los métodos de la clase contienen un repertorio de comportamientos que garantizan los principales requisitos funcionales para el desarrollador:

- **Hydra()**: Activación del ejecutante en el registro de ejecución del regente, carga de los atributos a partir del fichero de iniciación *exe_seed.hyd*, espera (o no espera si se envía un valor falso como parámetro del constructor de la clase) a que se active la cantidad mínima de ejecutantes propuestos para esa ejecución.
- **me()**: Devuelve el valor numérico 0..*us*-1 identificador del ejecutante.
- **us()**: Devuelve el valor del *us*.
- **necrosis()**: Le avisa al regente que el ejecutante correspondiente ha concluido el procesamiento, que no se recibirán más mensajes con destino a él, por lo que los que están en su bandeja de entrada son automáticamente borrados y a cualquier otro ejecutante que haga una solicitud de estado se le indicará que este ya ha concluido.

- **send(msg, destinatario)**: Pone la cadena de caracteres indicada por *msg* en la lista de mensajes salientes del regente con destino al ejecutante cuyo identificador *me* coincide con *destinatario.* Si se le indica un valor negativo de destino (por defecto vale -1) el mensaje se le envía a todos los ejecutantes excepto a sí mismo. Devuelve verdadero si pudo enviar el mensaje.
- **receive(remitente)**: Espera a la entrega de un mensaje proveniente del ejecutante con identificador igual al valor del remitente y cuyo destino sea el ejecutante en cuestión. Devuelve una tupla formada por el identificador del remitente y el mensaje en forma de cadena. **Nota:** si el valor del remitente es negativo se esperará al primer mensaje de que vaya con destino el ejecutante que invocó el procedimiento.
- **status(identificador)**: Devuelve una séxtupla de valores enteros cuyo significado es el que sigue:

  (id, estado, me, mer, ms, msr)

  id: representa el identificador del ejecutante.
  estado: tiene tres valores: *undefined,* si no se ha activado el ejecutante; *active:* en ejecución y *finished:* en caso de que ya ese ejecutante haya realizado la *necrosis.*
  me: se corresponde con la cantidad de mensajes que van destino al ejecutante *id.*
  mer: se corresponde con la cantidad de mensajes entregados ya al ejecutante *id.*
  ms: indica cuántos mensajes han salido de id y están en espera de ser entregados.
  msr: denota la cantidad de mensajes emitidos por *id* que han sido recibidos.

Nota: Se prevé en futuras versiones de la plataforma agregar otras informaciones de corte general a la tupla de estado.

El desarrollador avezado puede sobrecargar estos métodos, en dependencia de lo que desee lograr, siempre que conozca las órdenes correctas de SAVIA y los pasos para realizar cada una

de ellas. También debe documentarse acerca del funcionamiento del proceso de ejecución desde el lado del servidor.

3.3.3 El fichero de iniciación *exe_seed.hyd*

Una vez que se ha gestionado del lado del servidor la operación de ejecución de una tarea distribuida sobre *Hydra,* en cada PC se almacena alguna información de utilidad para realizar cabalmente las acciones de un *script* ejecutante. Esa información indica el nombre o dirección IP de la medulla regente y puerto al que han de conectarse por STREAM-SOCKET; el valor del identificador del ejecutante y la cantidad de miembros que tiene su comunidad de ejecución y la cantidad mínima de arrancada. Esa información viene declarada en el archivo de texto *exe_seed.hyd*, en forma de líneas de texto de la forma:

atributo valor

El orden de los atributos no es significativo, pues el constructor de la clase está preparado para leerlos sin importar el orden en que aparezcan.
Es importante conocer además que la medulla local es la encargada de la creación del fichero, una vez que recibe la solicitud de ejecución por parte de la regente y es posterior a la difusión de todas las dependencias de la aplicación a correr. De hecho, es el último paso antes de mandar a ejecutar el módulo de CD por vía del intérprete Python.

3.3.2 Esquema básico de los ejecutantes

Según el modelo de CD escogido para el desarrollo de la *Hydra* "un mismo código, varios comportamientos", cualquier programador que conozca las peculiaridades de la plataforma y la implementación de los *sockets* hecha para Python, el protocolo SAVIA y algunos aspectos de lo que se espera en materia de comportamiento por parte de los ejecutantes respecto a su intercambio con el servidor, puede hacer su propia versión de un ejecutante de CD. No obstante, igual es útil brindar un esquema básico de lo que puede ser un ejecutante estándar, de manera que todos los desarrolladores tengan un punto de partida a la hora de crear sus aplicaciones.

```
#carga de la librería con la definición de la clase Hydra
import hydra_egg

#Se crea una nueva instancia de la Hydra
hydra = Hydra()

#Se determina cuántos en la visión están ejecutando el presente script
size = hydra.us()

#Se detecta qué ejecutante soy yo
id = hydra.me()
#id es un entero desde 0 hasta hydra.us()-1

#En esta parte se escribe el código común previo a la especialización
.
.
.
#Fin

#Mediante condicionales, realizo la acción que me corresponde
if size >= Cantidad_de_Nodos_Deseados:
      if id == 0:
            #acciones si soy la hydra 0
            # Escriba su cógido acá
            #fin
      elif id == 1:
```

```
            #acciones si soy la hydra 1
            # Escriba su cógido acá
            #fin
      elif id == 2:
            #acciones si soy la hydra 2
            # Escriba su cógido acá
            #fin
      .
      .
      .
      elif id == k:
            #acciones si soy la hydra k-ésima
            # Escriba su cógido acá
            #fin
      .
      .
      .
      else:
            #acciones si no fuera ninguna de las anteriores
            # Escriba su cógido acá
            #fin

#Región para acciones comunes al final
#Escriba su código acá
#fin

#Se termina el trabajo
```

```
hydra.necrosis()
```

…

Sustituya código de cada ejecutante en los bloques dados para cada valor del *me.* También pueden personalizarse el comportamiento de las regiones previas a la activación del ejecutante en el servidor con el llamado a la construcción de la instancia de la *Hydra.*
Véase en los anexos algunos ejemplos.

3.4 Medulla

El servidor de la *Hydra* se conoce como *Medulla,* pues constituye el centro de comunicaciones y control de la plataforma. El mismo resuelve las solicitudes de los clientes y garantiza la ejecución y la comunicación intra e internodos; además de gestionar las acciones dentro de la visión: altas, bajas, cambios de regencia, etc.

3.4.1 Algoritmo de funcionamiento

El funcionamiento de la medulla es simple al igual que sus estructuras de datos: un algoritmo de tres partes básicas:

1. Iniciación.
    a) Activación del servicio de comunicaciones.
2. Ciclo de procesamiento de las consultas en lenguaje SAVIA.
3. Terminación.

3.4.1.1 Estructuras de datos básicas para el funcionamiento de la medulla

Como entre los diversos cometidos que tiene el servidor de la plataforma es conocer quiénes son sus compañeros y cuál es su jerarquía entre ellos, es decir tener conocimiento exacto de la línea de sucesión; en la Medulla se guarda una copia de la visión en la variable *vission,* la cual es una lista cuyos registros constan de una estructura muy sencilla:

(host, puerto, contador de fallos de conexión, ...)

El orden que tiene cada nodo en la lista (copia que poseen todas las máquinas que comparten la misma visión) indicará la posición en la línea de sucesión. Dicha lista se guarda luego en el fichero de iniciación del servidor.

Otra responsabilidad básica del servidor es gestionar la ejecución de aplicaciones. Para esta operación, la más compleja a su cargo, se cuenta con una serie de estructuras, banderas y variables de control que permiten llevar a cabo esa operación. A continuación se describen algunas de ellas:

running_flg: un valor verdadero en esta bandera indica que se está llevando a cabo una ejecución, de él depende en general la existencia del resto de los objetos relacionados con la ejecución.

runners_minimun_quantity: es el valor entrado como de mínimo en la orden de ejecución solicitada, en la mayoría de los casos, los ejecutantes se cuelgan mientras no se satisface este valor.

runners_limit: su valor equivale a la cantidad de ejecutantes máxima solicitada.

runners_vission: es la lista ordenada de tuplas que contienen el estado de cada ejecutante (vea en la sección dedicada a los objetos Hydra la descripción del método *status*).

msgs: contiene la cola de mensajes donde cada índice es una tupla de la forma:

(origen, destino, mensaje)

donde se encolan y extraen los mensajes en la medida que se resuelven consultas del tipo *send* y *receive* provenientes de los ejecutantes.

Un detalle no declarado hasta el momento es que para el correcto funcionamiento de algunos pasos en los clientes (acciones que no siempre pueden resolverse en el mismo momento de su solicitud, sino que han de esperar al completamiento de otras tareas, por lo general desempeñadas fuera del servidor) a veces estos activan servicios de escucha tipo servidor y se quedan colgados en espera de la respuesta del servidor que hará entonces las veces de un mero cliente (de ahí el hecho que se hayan contemplado al realizarse la acción de enlace o *link* con alguna medulla, debe identificarse el tipo de cliente que lo está efectuando: *heracles, medulla*

o *performer*). El caso es que el servidor debe tener conocimiento de quién está colgado en espera de respuesta y cuál es la causa; por ejemplo, un ejecutante que espera terminar su activación hasta que se termine de completar la activación de la cantidad mínima de ejecutantes o un ejecutante en espera de un mensaje. Para ello en la medulla la información de los ejecutantes en espera se guarda en una lista de clientes en espera o *halted_clients* cuyos elementos siguen la estructura:

(tipo de cliente, id, puerto, motivo)

donde el *tipo de cliente* se corresponde con los tres tipos que pueden abrir un enlace; el *id* es un valor entero que puede ser en dependencia del tipo de cliente el índice de un equipo en la *vission* o el identificador del ejecutante; *puerto* se refiere al puerto por donde el cliente está escuchando y *motivo* es la causa de la espera: *'minimum', 'incomming msg',* etc...
Por supuesto, que existen otras muchas variables, métodos y estructuras de datos que no pueden ser cubiertas acá por razones de espacio, en cambio el interesado puede referirse al anexo B donde viene el código de la *Medulla* y por ende, todos sus detalles funcionales comentados.

3.4.1.2 Iniciación

En el algoritmo de la Medulla se producen, en ese mismo orden, la creación de las estructuras de datos elementales del servidor (*vission, runners_vission, msgs, halted_clients,* etc.), algunas variables de control (*runners_minimum_quantity, runners_limit*, entre otras) y banderas de estado de la aplicación (*running_flg, client_mode_flg,* etc.); las acciones de carga de los valores de arranque a partir del fichero *hydra_seed.hyd.*
El fichero *hydra_seed.hyd* comparte muchas similitudes, en cuanto al lenguaje de definición usado para asignarle valores a sus atributos, con *exe_seed.hyd*; pero, por supuesto, tiene especificidades relacionadas con el funcionamiento del servidor.
En este archivo principalmente se almacena una copia del último estado de la visión, que le permite conocer a la medulla local su estatus dentro de la misma.

De perderse el fichero o de no existir en el momento de la activación del servidor, éste crea una copia por defecto que lo convierte en una medulla regente, hasta que se especifique lo contrario. Este método puede utilizarse para llevar a cabo un proceso de separación de la visión, pero tiene el inconveniente de que si la medulla regente antigua le envía un mensaje u otra de la visión, se actualizará la información del servidor.
Más detalles de la sintaxis válida de los archivos semilla en los anexos al capítulo.

3.4.1.3 Activación del servicio de comunicaciones

El montaje del servicio de escucha de los clientes sigue los pasos utilizados en los ejemplos de la Referencia de Python 2.4 (veáse el tema 17.2.3 de la Referencia de Librerías de Python), donde se intenta activar un STREAM-SOCKET local por el puerto 50010.

3.4.1.4 Ciclo de procesamiento de instrucciones entrantes

Las medullas están constantemente escuchando en pos de posibles peticiones por parte de los clientes, peticiones que son hechas según las normas de SAVIA, o sea una solicitud de enlace seguido de la consulta que desea realizar el cliente; el servidor procesa esa consulta, si es correcta, emite las respuestas o pide más información. Esos pasos son ejecutados en el ciclo de la medulla que es su alma.
En dicho ciclo convive un intérprete sencillo para decodificar cada instrucción SAVIA y ejecutar la acción equivalente mediante una serie de condicionales relacionadas con el tipo de cliente y la solicitud hecha.
Aunque puede forzarse el cierre del servidor mediante los recursos del sistema operativo, lo recomendable es usar la instrucción que tiene el cliente Heracles para esos efectos, pues permite, en caso de que fuese un regente, que la segunda medulla en la línea de sucesión tome el control.
En la sección de los anexos pueden verse algunos esquemas de las comunicaciones entre los clientes y el servidor.

3.5 Heracles

Una vez instalada la plataforma, toda la interacción del usuario se concentra en el cliente Heracles, pues el mismo cuenta con un staff de instrucciones que le permiten interactuar con el

servidor local y el regente para llevar a cabo las funcionalidades que supone realice el sistema: solicitar entrada en la visión, difundir archivos, correr algún ejecutante, etc.

La interfaz, pensada en el futuro con un aspecto más amistoso e interactivo, por ahora tiene presencia de consola en la que se introducen las órdenes y se emiten las correspondientes respuestas y mensajes de error (Figura 14).

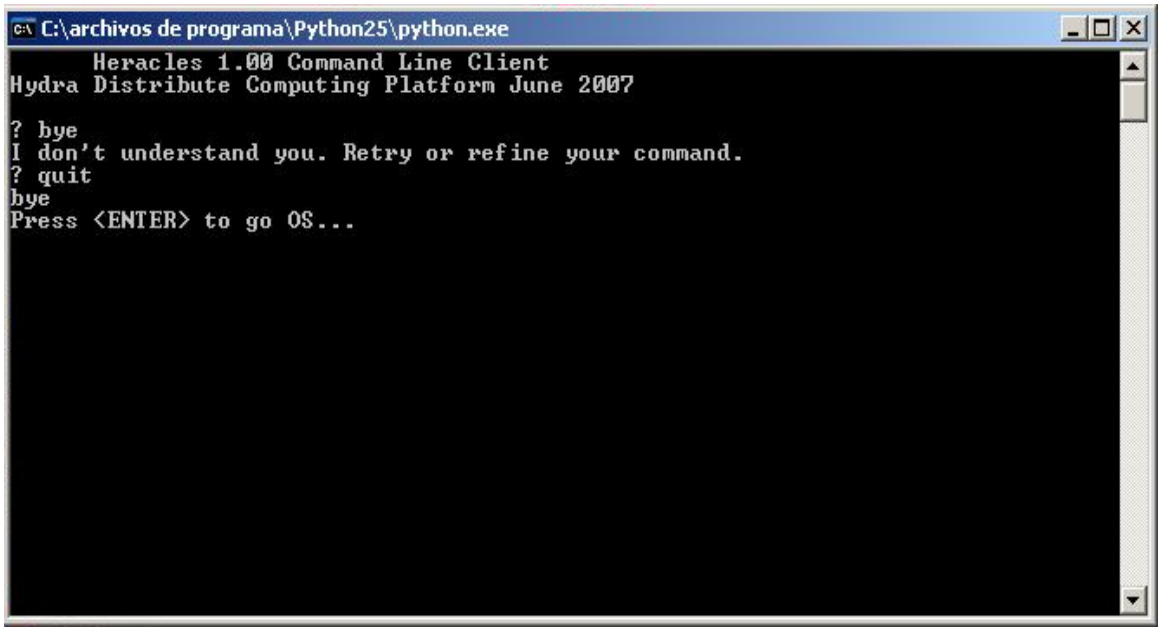

Figura 14 Captura de sesión del cliente de consola Heracles.

Algunos de los comandos que pueden utilizarse son:

`help [instrucción]`

Devuelve información de ayuda de la instrucción en cuestión o si no se escribe ninguna instrucción se muestran todas las instrucciones hábiles del cliente.

`quit`

Cierra el cliente.

```
test [cnx|regent|vission]
```

Prueba la comunicación (ping) con la conexión por defecto, la medulla regente o la visión entera.

```
run [max] min ejecutante [dependencia1 dependencia2 dependencia3 ...]
```

Solicitud de ejecución de una tarea llamada *ejecutante* en *max* procesadores con un cantidad mínima de arranque de *min* procesadores que necesitan los ficheros *dependencia1, dependencia2, ...,* etc.

```
reclute host [port]
```

Agrega la máquina actual a la visión de hidrandad que tenga el nodo con nombre o dirección IP *host* que escucha por el puerto *port.*

En el anexo A se describe la lista de instrucciones hábiles para el cliente *Heracles* de línea de comando, su sintaxis y algunos ejemplos.

3.5.1 El algoritmo de funcionamiento

Al igual que el servidor, *Heracles* consta en lo básico de las siguientes regiones:

1. Iniciación.
2. Ciclo de resolución de consultas.
3. Terminación.

3.5.1.1 Iniciación

El proceso de iniciación del *Heracles* consiste en cargar, si existe el fichero *heracle_seed.hyd*, los valores por defecto del host local y el regente, los puertos de comunicación de ambos, etc; en caso contrario creará uno con los valores por defecto.
El archivo de iniciación *heracle_seed.hyd* estará en el mismo directorio que el cliente y su estructura interna es similar a la de los otros ficheros semilla ya vistos.
Véase la estructura y los comandos permitidos en cada tipo en el anexo A y anexo B del documento.

3.5.1.2 Ciclo de procesamiento de consultas

En este ciclo se procesa la línea de comando entrada por el usuario en busca de instrucciones válidas para convertirlas en comandos SAVIA comprensibles al servidor; de lo contrario, emite un mensaje de error para que el usuario refine su consulta.
Es evidente la necesidad de un intérprete de consultas encargado de la traducción de órdenes entradas por el usuario al lenguaje del protocolo de comunicaciones y de la ejecución de las acciones correspondientes a cada comando, así como de la salida de respuestas provenientes de las medullas y de los mensajes de error.
Este ciclo concluye una vez que se introduce la orden *'quit'* o se aborta con un CTRL+C o CTRL+BREAK, solo que la última acción no se recomienda pues puede interrumpirse algún proceso que se esté realizando.

3.6 Producto final

El desarrollo del prototipo (o la versión final de la aplicación) concluye en tres script Python (o dos aplicaciones y un script Python) correspondientes a *Medulla*, *Heracles* e *hydra_egg.py* quienes cumplen funciones diferentes y por tanto, siguen pasos diferentes de instalación.

Un acceso directo al servidor *Medulla* debe colocarse, si está en Windows, en el Inicio del Menú de Inicio para que se cargue al iniciar cada sesión, igual puede solicitarse su ejecución desde el registro como parte de los servicios del sistema, lo cual sería mucho mejor.

Por otro lado, el módulo *hydra_egg.py* se copiará en los directorios que están en el camino de las librerías del Python para que puedan ser importadas desde cualquier ejecutante (veáse Python Path). Por ejemplo puede copiarse en la carpeta que contiene las librerías básicas del sistema.

El *Heracles* es libre de ser copiado en cualquier lugar. Pero recuérdese que una vez ejecutado el mismo crea algunos ficheros temporales y así como su fichero semilla.

En la versión definitiva se entregará un instalador que lleve a cabo todos los procesos antes descritos de forma automática.

Capítulo 4: Estudio de factibilidad

Sumario

- Introducción
- Beneficios del proyecto
- Estimación de costos del proyecto
    - Estimación de líneas de código fuente
    - Estimación de esfuerzo
    - Estimación de tiempo de desarrollo
    - Estimación de hombres a tiempo completo
- Costo del software
- Análisis de costo-beneficio

4.1 Introducción

Para poner en marcha cualquier proyecto de software Es necesario realizar una estimación del trabajo a realizar, de los recursos necesarios y del tiempo que transcurrirá desde el comienzo hasta el final de su realización.

La mayor parte de los proyectos informáticos presentan carencias de recursos y las fechas de entrega no se corresponden con la realidad.

Por el alto costo que tienen la mayoría de los sistemas y productos basados en computadoras y a pesar del grado de inseguridad que pueda existir en cuanto a la exactitud de los resultados obtenidos, es razonable desarrollar y estimar antes de empezar a construir el software.

No se puede permitir en un proyecto pérdida de recursos, esfuerzo, tiempo y crédito profesional, es por eso que se hace necesario evaluar la posibilidad real de la realización de un proyecto cuanto antes y determinar si este está o no dentro de las perspectivas financieras.

En este capítulo se presentan cada una de las actividades asociadas a la factibilidad del proyecto, puesto que cuanto mas se sepa mejor será la estimación, la misma realizada con el método de puntos de función del modelo de COCOMO II en la etapa de diseño temprano. Se muestran los

beneficios tangibles e intangibles que representan para el sistema propuesto, un análisis de costos y beneficios que permiten valorar si el sistema es factible.

4.2 Beneficios de proyecto.

Los beneficios de este proyecto como se han descrito a lo largo de los anteriores capítulos están relacionados al aprovechamiento de recursos computacionales (máquinas obsoletas, RAM, dispositivos de almacenamiento, CPU, etc.) con que contamos actualmente, como vía más factible y menos costosa de contar con una herramienta que nos permita la realización no solo de cálculos y algoritmos complejos, si no también de proyectos vinculados con la bioinformática y la creciente necesidad de potencia computacional para las aplicaciones que puedan resolver estos y otros problemas.

Es decir la virtualización de estos recursos informáticos para "crear" la ilusión de que se trabaja con una supercomputadora dispuesta a la mayor de nuestras exigencias. Utilizando nuestros propios medios para lograrlo, sin necesidad de algún hardware costoso.

4.3 Estimación de Costos del proyecto

La estimación del proyecto se realizó mediante los puntos de función desajustados, los cuales se utilizan para el cálculo de las instrucciones fuentes. Para así obtener un enfoque a la magnitud del sistema, además de obtener indicadores como la cantidad de hombres necesarios, esfuerzo, el tiempo de duración y el costo del proyecto.

Se trabajó con los requerimientos funcionales del sistema que es lo más cercano a los datos que se van a manipular en el sistema; debido a no tener bien definido el Modelo de Análisis para el Caso de Estudio.

4.3.1 Requerimientos funcionales del sistema

R1. (Sostener y alcanzar la conectividad) Reconocimiento de los nodos que integrarán el *grid*.
R2. Garantizar la distribución y ejecución de tareas, la difusión de las dependencias entre los ficheros en el gris, así como el paso de mensajes entre los nodos del mismo.

Por su parte cada componente responde individualmente a sus propios requerimientos funciones, así como:

Medulla: Servidor de aplicación.

1. Conectarse a la Hydrandad.
2. Obtener la lista de hermanos.
    a. Se envía un acuse de recibo a las maquinas existentes en un listado de nodos.
    b. En caso de encontrarse más nodos, la lista se va actualizando manualmente y ocurre lo anterior.
    c. En caso contrario, se redistribuye las tareas asignadas teniendo en cuenta sobre todo la disponibilidad del sistema.
3. Difundir dependencias con los ficheros predefinidos por el usuario.
4. Ejecutar una aplicación Hydra.
    a. Garantizar el paso de mensajes.
    b. Asignar Id's.
    c. Obtener estados.

Heracles: Cliente para el control del sistema Hydra.

1. Conectar el servicio con otro hermano.
2. Activa el servicio Hydra.
3. Transferencia de ficheros.
4. Ejecución de módulos.

hydra_egg: Librería Python para el desarrollo de aplicaciones de cómputo distribuido sobre Hydra.

1. Iniciación de la aplicación.
2. Asignación de Id's a los procesadores.
3. Paso de mensajes a los procesadores.

4. Recibo de mensajes de los procesadores.
5. Control del número de integrantes del *Grid*.
6. Control del estado del procesador (terminó de procesar, procesando, desocupado).
7. Cerrar conexiones.

Cada requerimiento analizado en el paso anterior se clasificó y se le asignó un peso.

4.3.2 Cálculo de puntos de función desajustados

A continuación se agrupan los requerimientos funcionales del sistema en: Entradas externas, Salidas externas, Peticiones, Ficheros internos, e Interfaces externas. Clasifican do cada uno de ellos por su nivel de complejidad en: Simple, Media, Compleja.

Se cuenta la cantidad de transacciones por cada Nivel de complejidad bajo y se multiplica por el peso asociado en la tabla 4[7]. Todos estos productos se suman y se obtienen los puntos de función desajustados (UFP).
Entonces para este caso contamos con:

Tabla 1 Puntos de función desajustados

Características	**Complejidad**			
	Baja	Media	Alta	Aporte
Entradas externas	**4 * 3**			**12**
Salidas externas	**3 *4**		**5 * 7**	**47**
Consultas externas		**2 *4**		**8**
Archivos lógicos internos		**1 *10**		**10**
Archivos de interfaz externos				
Puntos de Función Desajustados UFP Total				**77**

[7] Ver anexo D, tablas 1-4

4.3.3 Estimar la cantidad de instrucciones fuente (SLOC).

Para el cálculo de las instrucciones fuentes (SLOC) se utilizó la fórmula siguiente:

SLOC = UFP * ratio.

Luego

SLOC = 77 * 60

SLOC ≈ 4620 líneas de código fuente

Por tratarse de un lenguaje de muy alto nivel, orientado a objeto y que cuenta con una gran variedad de librerías, con un gran número de funcionalidades implementadas, se ha acordado considerar el 30% de reutilización de código fuente, el cual se descuenta, por lo tanto:

SLOC ≈ 3234 líneas de código fuente

KSLOC= 3,234 (Miles de líneas de código)

Donde UFP es el total de puntos de función desajustados, y ratio es una constante para las SLOC de cada lenguaje de programación en este caso tiene un valor para Python de 60.

4.3.4 Estimación de esfuerzo

El esfuerzo es la cantidad de tiempo que una persona invierte trabajando en el desarrollo de un proyecto durante un mes. La sigla que lo representa es PM.

$$PM_{NS} = A * Size^{E} * \prod_{i-1}^{n} EM_i \quad \textbf{Donde:} \quad E = B + 0.01 * \sum_{j=1}^{5} SF_j$$

Donde:

Size: Tamaño estimado (KSLOC).

A = 2.94, B = 0.91,

EM_i: Multiplicadores de esfuerzo.

Se tiene el tamaño (KSLOC) faltaría calcular E el cual depende de los factores de escala:
Son cinco factores que afectan E (exponente del TAMAÑO).

Factores de Escala

- PREC: Precedencia.
- FLEX: Flexibilidad.
- RESL: Riesgos.
- TEAM: Cohesión del Equipo.
- PMAT: Madurez de las Capacidades.

Tabla 2 Constantes y fórmulas para el cálculo del esfuerzo

Siglas	**Indicador**	**Valor o fórmula**
PM	Esfuerzo	$A * (Size)^{E} * \prod EMi$
A	Constante	2.94
Size	Miles de instrucciones fuentes	0.54
E	Agregado de 5 factores de escala	$B + 0.01 * \Sigma SFi$
EM	Multiplicadores de esfuerzo	Se muestran en la tabla 7.
B	Constante	0.91
SF	Factores de escala	Se muestran en la tabla 8.

Para obtener los resultados de las fórmulas anteriormente expuestas, se calcularon los valores de cada factor de escala (SF_j) y de cada multiplicador de esfuerzo (EM_i).

Tabla 3 Multiplicadores de esfuerzo

Multiplicador	**Valor**	**Justificación**
PERS	0.83	Los desarrolladores tienen en general alto conocimiento en la programación de sistemas, se considera alta las

		capacidades de los analistas y de los programadores. No se esperan cambios significativos en el personal del equipo de desarrollo.
RCPX	1.00	El producto tiene una moderada complejidad, existe una alta confiabilidad de la documentación.
RUSE	1.00	En la implementación del sistema existe una alta reusabilidad de códigos, con vistas a la construcción de componentes a través del proyecto.
PDIF	1.29	El sistema operativo a utilizar es Windows que cambia aproximadamente cada año, por lo que puede considerarse en alguna medida volátil, aunque las características del lenguaje de programación permite la ejecución multiplataforma.
PREX	1.00	Basta experiencia en cuanto al lenguaje, se conoce el tipo de software y herramientas para el desarrollo de aplicaciones de este tipo. Por tanto se valora como nominal.
FCIL	0.73	Se utilizan herramientas modernas de programación como Python. Así como para la documentación se utilizó la notación UML y para su modelado visual se empleó la herramienta Rational Rose.

Tabla 4 Factores de escala

Factor de Escala	**Valor**	**Justificación**
PREC	2.48	Resulta algo familiar para los desarrolladores el tipo de aplicación.
FLEX	1.01	Hubo cierto acuerdo de forma general en cuanto a las

		interfaces de diseño y los requisitos del software.
RESL	1.41	Se tomó ciertas estrategias para tener el mínimo de riesgos en el entorno de la aplicación.
TEAM	1.1	Bastas experiencias en el trabajo en equipo. Buen acoplamiento de forma general a la hora de trabajo.
PMAT	3.12	Existe gran madurez en cuanto a la complejidad del software.

Producto de los multiplicadores de esfuerzo:

ΠEM = RCPX * RUSE * PDIF * PERS * PREX * FCIL
ΠEM= 1.00 * 1.00 * 1.00 * 0.83 * 1.00 * 0.73
ΠEM= 0, 6059

Sumatoria de los factores de escala:

ΣSF = PREC + FLEX + RESL + TEAM + PMAT
ΣSF = 2.48 + 1.01 + 1.41 + 1.1 + 3.12
ΣSF= 9, 12

Cálculo de esfuerzo

$E= B + 0.01 * \Sigma SF_i$
E= 0.91 + 0.01 * 9,12= 1,0012

$PM = A * (Size)^{E} * \prod EM_i$
$PM= 2.94 * 3{,}234^{1,0012} * 0{,}6059 = 5{,}76$ hombres-mes

Se necesitan 6 personas para realizar el software en un mes.

4.3.5 Estimación del tiempo de desarrollo

Al conocer el valor del esfuerzo se puede calcular el tiempo de desarrollo (TDEV) estimado del software, es decir, cantidad de meses necesarios para desarrollar el software.

$$TDEV_{NS} = C * \left(PM_{NS}\right)^{F}$$ **Donde:** $$F = D + 0.2 * 0.01 * \sum_{j=1}^{5} SF_j$$

$$F = D + 0.2 * \left(E - B\right)$$

Tabla 5 Constantes y fórmulas para el cálculo del tiempo de desarrollo

Siglas	**Indicador**	**Valor o fórmula**
TDES	Tiempo de desarrollo	C * (PM) ^{F}
C	Constante	3.67
PM	Esfuerzo	5,76hombre-mes
F	Exponente de escala	D + 0.2 * (E - B)
D	Exponente base para la ecuación del cronograma (constante)	0.28
E	Agregado de 5 factores de escala	B + 0.01 * Σ SFi
B	Exponente de base escalado para la ecuación de esfuerzo que puede ser calibrado (constante)	0.91
ΣSF	Factores de escala	9.12

F= D + 0.2 * (E - B)
F = 0.28 + 0.2 (1,0012- 0.91) = 0,29824

TDES= C * (PM) ^{F}
TDES = 3.67 * (5.76) ^{0,29824}= 6,18 meses

EL tiempo necesario para desarrollar el proyecto es de 6.18 meses.

4.3.6 Cantidad de hombres a tiempo completo

Teniendo en cuenta el tiempo de desarrollo se calcula la cantidad de personas (CH) necesarios para desarrollar el software, se obtiene la tabla 6:

Tabla 6 Constantes y fórmulas para el cálculo de la cantidad de personas

Siglas	Indicador	Valor o fórmula
CH	Cantidad de hombres por mes	PM/TDES
PM	Esfuerzo	5,76hombre-mes
TDES	Tiempo de desarrollo	6.19 meses

CH = 5.76 / 6.19 = 0,93 personas

Es necesaria 1 persona para realizar el software en 6.19 meses.

4.4 Costo del software

El costo del software depende del salario promedio de las personas que lo desarrollan y del esfuerzo que ellas realizan para la ejecución del mismo y se calcula a través de la fórmula representada en la tabla 7.

Tabla 7 Constantes y fórmulas para el cálculo del costo del software

Siglas	Indicador	Valor o fórmula
C	Costo del proyecto	CHM *PM
CHM	Costo de hombres por mes	CH * SP
SP	Salario básico de un Ingeniero	$ 225.00
PM	Esfuerzo	5.76 hombre-mes

Costo del software
El salario medio es de $ 225.00

C = 1 *225 * 5.76 = $ 1296

El software cuesta $ 1296

Tabla 8 Resultados de las estimaciones de esfuerzo, tiempo de desarrollo, cantidad de hombres y costo del proyecto

Cálculo de:	Valor	Justificación
Esfuerzo	5.76 hombres-mes	Cantidad de tiempo que una persona invierte trabajando en el desarrollo de un proyecto
Tiempo de desarrollo	6 meses	Cantidad de meses para terminar el proyecto.
Cantidad de personas	1	Cantidad de personas necesarias para terminar el proyecto en 6 meses.
Costo nominal	$ 1296	Cantidad de dinero que cuesta el proyecto después de terminado.
Salario medio	$ 225.00	Salario básico de un ingeniero

4.5 Análisis de costos y beneficios

El desarrollo del software no reportó gran consumo de recursos. Puesto que la tecnología utilizada es totalmente libre y gratuita, por lo que no fue necesario incurrir en gastos referentes al pago de licencias, no obstante no se han tomado en cuenta gastos como: la energía eléctrica utilizada en el periodo de tiempo de desarrollo, así como la depreciación de los equipos de cómputo utilizados, etc. En cuanto a la implantación, la aplicación puede ser ejecutada en cualquier tipo de máquina garantizándose el funcionamiento satisfactorio de la misma. Según los resultados obtenidos en el estudio de la factibilidad del proyecto se ha logrado evaluar y determinar rigurosamente los recursos necesarios para la realización de cada una de las funciones de la aplicación a implantar, lo que da una alta importancia al momento de tomar decisiones referentes a la planificación y desarrollo de la misma, también es importante señalar que un balance entre los costos y beneficios se inclina incuestionablemente a la aplicación de la de solución propuesta.

Conclusiones

En este trabajo se abordaron temas relacionados con las tendencias actuales de la Computación Distribuida, y la importancia que tiene desarrollar esta técnica de programación en nuestro centro y en el país, teniendo en cuenta el poco conocimiento de esta tecnología. Con la realización de esta investigación se arriba a las siguientes conclusiones:

- Existen potencial de cómputo y recursos informáticos subutilizados, los cuales se pueden aprovechar ventajosamente mediante uso de la computación distribuida.
- Una plataforma para computación distribuida soluciona el problema de la falta de computadoras de potencia, entre otras prestaciones, sin necesidad de hardware costoso.
- Hydra, es la primera aproximación real de una plataforma para sistemas distribuidos en nuestro centro, dando una solución al problema de la carencia de máquinas potentes.
- Se implementó una librería que permite la ejecución de scripts Python creados por el usuario, o importados de algún sitio.
- La implementación de la aplicación se realizó con software libre.
- El lenguaje de programación permite que el software sea multiplataforma.

Recomendaciones

Para el desarrollo de futuras versiones de esta aplicación, se recomienda:

- Realizar las pruebas concernientes a la compatibilidad de la aplicación con el sistema operativo Linux, entre otros que soporten Python.
- Revisar asuntos correspondientes al aumento de la seguridad referente a la encriptación de información, utilización de capas de seguridad (SSL), etc.
- Utilización de funciones de constructores para cadenas.
- Probar la plataforma con la versión 2.5 de Python y sus nuevas mejorías.

Bibliografía

1. [Bart,2003] Bart Jacob, Grid computing: What are the key components?, 2003-6-1, Disponible en: www.ibm.com (ultima actualización 2006-6-27).
2. [Comparex,2003]Comparex,Cluster Computing-Uno para todos y todos para uno, 2003, Disponible en: http://www.comparex.es/es/es/solutions/high_availability/cluster_computing.html [Duncan,1990] Duncan, Ralph, "A Survey of Parallel Computer Architectures", IEEE Computer. Febrero 1990, pp. 5-16.
3. [Flynn,1972] Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol.-21, pp.948, 1972. Disponible en: http://es.wikipedia.org/wiki/Taxonom%C3%ADa_de_Flynn
4. [Grid, 2007] José M Fernández, La tecnología GRID y la Bioinformática en Cuba, 13 de febrero de 2007, Disponible en: http://www.informatica2007.sld.cu/ploneparticlemultipage.20070215.8489400644/resumen-del-dia-13-de-febrero
5. [Iluvatar, 2004] Iluvatar & amigos, Proyectos de Computación Distribuida, 2004, Disponible en: http://www.overclockers.cl/
6. [Ismael, 2004] Ismael Olea , Introducción PVM, 2004-03-22 , Disponible en: ftp://netlib2.cs.utk.edu [Ismael, 2004] Ismael Olea , ¿Qué es un Cluster de computadoras?, 2004-03-22 , Disponible en: http://es.tldp.org/Manuales-LuCAS/doc-cluster-computadoras/doc-cluster-computadoras-html/node7.html
7. [Ivar,2004]Ivar Jacobson, Grady Booch, James Rumbaugh, El Proceso Unificado de Desarrollo de Software,2004, Disponible en: http://www.awl.com
8. [Lucas, 1997] Lucas Morea, Sistemas Distribuidos, 1997, Disponible en: http://monografias.com
9. [Peer,2003] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, Zhichen Xu, Peer-to-peer Computing, Julio 3-2003, Disponible en: http://www.webzinemaker.com/admi/m10/page.php3?num_web=5518&rubr=4&id=34698

10. [PP, 2006] Wikipedia.org, Programación Paralela, 2006, Disponible en: http://es.wikipedia.org/w/index.php?title=programacion&paralela=no(modificada por última vez el 01:11, 16 sep 2006).
11. [Pressman,2002] Roger S. Pressman, Ingeniería de Software, Editorial Félix Varela, Ciudad de La Habana – Cuba, 2002.
12. [PyGlobus, 2002] Laboratorio Nacional Lawrence Berkeley, División de Investigación Computacional, Departamento de Computación Distribuida, PyGlobus, Febrero 17, 2002, Toronto Canadá. Disponible en: http://wwwitg.lbl.gov/gtg/projects/pyGlobus/
13. [vpm,2006] Wikipedia.org, Máquina Virtual Paralela, 2006, Disponible en: http://es.wikipedia.org/w/index.php?title=PVM&redirect=no(modificada por última vez el 09:09, 14 oct 2006.)
14. [Yadira, 2006] Yadira Romero, Eglys Vázquez. Trabajo de diploma "Spider".ISMM Moa-Holguín, 2006.

Glosario de Términos

1. **Actor:** Alguien o algo, fuera del sistema o negocio que interactúa con el sistema o negocio.
2. **Atributo:** Es un valor lógico de un estado de un objet**o.**
3. **Cluster de computadoras.-** Un cluster es un grupo de equipos independientes que ejecutan una serie de aplicaciones de forma conjunta y aparecen ante clientes y aplicaciones como un solo sistema; es un conjunto de computadoras interconectadas con dispositivos de alta velocidad que actúan en conjunto usando el poder cómputo de varios CPU en combinación para resolver ciertos problemas dados.
4. **Computación Distribuida (CD)-** Es una técnica de la ciencia informática que soluciona un problema grande dando las partes pequeñas del problema a muchas computadoras para solucionar y después combinando las soluciones para las piezas en una solución para el gran problema.
5. **Disponibilidad.-** Calidad del sistema de estar presente, listo para su uso, a mano, accesible.
6. **Fiabilidad.-** La probabilidad de un funcionamiento correcto.
7. **FLOPS.-** Es el acrónimo de Floating point Operations Per Second (operaciones de punto flotante por segundo). Se usa como una medida del rendimiento de una computadora, especialmente en cálculos científicos que requieren un gran uso de operaciones de punto flotante. FLOPS, al ser un acrónimo, no debe nombrarse en singular como FLOP, ya que la S final alude a second (o segundo) y no al plural.
8. **Grid.-** Red de computadoras heterogéneas, que comparten sus recursos informáticos, de cómputo ó dispositivos específicos, etc.

9. **Mainframe.-** O Computador Central, es una computadora grande, potente y costosa usada principalmente por una gran compañía para le procesamiento de una gan cantidad de datos; por ejemplo, para el procesamiento de transacciones bancarias.

10. **Máquina Virtual Paralela**.- Es una librería para el cómputo paralelo en un sistema distribuido de computadoras. Está diseñado para permitir que una red de computadoras heterogénea comparta sus recursos de cómputo (como el procesador y la memoria RAM) con el fin de aprovechar esto para disminuir el tiempo de ejecución de un programa al distribuir la carga de trabajo en varias computadoras.

11. **Programación Paralela(PP)** .- La programación paralela es una técnica de programación basada en la ejecución simultánea, bien sea en un mismo ordenador (con varios procesadores) o en un cluster de ordenadores, en cuyo caso se denomina computación distribuida.

12. **Python:** Es un lenguaje de programación interpretado e interactivo, permite escribir programas muy compactos y legibles.

13. **RUP:** El Proceso Unificado Rational (RUP) es una metodología de desarrollo para la programación orientada a objetos. Según Rational (diseñadores de Rose Rational y el Idioma Modelado Unificado), RUP está como un mentor en línea que mantiene pautas, plantillas, y ejemplos de todos los aspectos y fases de desarrollo del programa. RUP es un software comprensivo que diseña herramientas que combinan los aspectos procesales de desarrollo (como las fases definidas, técnicas, y prácticas) con otros componentes de desarrollo (como los documentos, modelos, manuales, el código, y así sucesivamente) dentro de un armazón unificándose.

14. **Software:**(Componentes lógicos, programas, software).—Programas o elementos lógicos que hacen funcionar un ordenador o una red, o que se ejecutan en ellos, en contraposición con los componentes físicos del ordenador o la red.
15. **Súper computadora.-** es una computadora con capacidades de cálculo muy superiores a las comúnmente disponibles máquinas de escritorio de la misma época en que fue construida.
16. **TCP/IP:** (Transmission Control Protocol/Internet Protocol) Es el conjunto de protocolos que definen a Internet. Originalmente diseñado para el sistema operativo UNIX, hoy en día existe software TCP/IP disponible para la mayoría de los sistemas operativos. Para poder utilizar la Internet, su computador debe tener software TCP/IP.
17. **UML**: Lenguaje unificado de modelado -Unified Modeling Language para construir modelos; no guía al desarrollador en la forma de realizar el análisis y diseño ni le indica cual proceso de desarrollo adoptar.

Anexo A: Comandos del *Heracles*

Help

Sintaxis:

help [comando]

Descripción:

Por defecto muestra la lista de comandos válidos del cliente *Heracles*. En caso de que aparezca seguido del nombre de un comando, muestra la información de ayuda de éste: sintaxis y una breve descripción.

Quit

Sintaxis:

quit

Descripción:

Cierra el cliente. Debe recibir un mensaje del tipo "Presione una tecla para continuar" antes de salir al sistema.

Run

Sintaxis:

run [max] min ejecutante [dependencia1 [dependencia2 [dependencia3…]]

Descripción:

Solicita la ejecución de al menos *min* módulos *ejecutantes* pero antes copia además los archivos o *dependencias* que usarán éstos para cumplimentar su tarea. Una vez que se hayan activado al menos *min* ejecutantes se les avisará si están colgados en espera para que prosigan pues ya hay "suficientes" procesadores para ellos. Si se especifica el valor *max* el regente limitará la solicitud de equipos ejecutantes a ese valor.

Test

Sintaxis:

test [cnx | regent | vission]

Descripción:

Emite una especie de ping a la conexión con la medulla local, la regente o con todos los que pueda localizar en la visión. Las dos últimas acciones son coordinadas y resueltas por la medulla regente. Por defecto prueba la conexión local.

See

Sintaxis:

see host [port]

Descripción:

Incorpora, de no estar, el *host* dado a la visión en el último lugar de la línea de sucesión.

Anexo B: Ejemplos de ejecutantes

Saludos

Copie el siguiente código como un script de Python (.py)

```
import hydra_egg


hidra = Hydra() #se crea una nueva instancia
mi_id = hidra.me()
if hidra.us() >= 2:
      if mi_id:
            #Si no soy el ejecutante 0
            #se envia un mensaje con la identificacion
            hidra.send("Hola! Soy el ejecutante " + `mi_id`)
      else:
            #si soy el ejecutante colector
            print "Hola! Soy el ejecutante colector"
            cnt = hydra.us()
            while cnt:
                  ejecutante, msg = hidra.receive()
                  print ejecutante, msg
                  cnt -= 1
hidra.necrosis()
```

Anexo C: Protocolo de comunicaciones SAVIA

El sistema de comunicaciones es específico a cada componente de la plataforma y se divide en dos tipos: las instrucciones de respuesta y las de consulta o de solicitud de acción.

Instrucciones de respuesta

!

Sintaxis:

<~!~>

Descripción:

Indica que el proceso se ha terminado correctamente o que la acción que estaba detenida en pos de una respuesta ya puede proseguir.

?

Sintaxis:

<~?~> [codigo]

Descripción:

El proceso ha fallado o falta algún dato. En caso de venir seguido de un valor numérico el mismo equivale a la descripción del tipo de error sucedido.

#

Sintaxis:

<~#~>

Descripción:

Indica que vuelva a reintentarlo después o que se quede en espera.

Instrucciones de consulta

Comunes

Petición de Enlace

Sintaxis:

<l> tcliente host instrucción parámetros

Descripción:

El cliente de tipo *tcliente* radicado en el *host* (o el identificador de ejecución si es un ejecutante) solicita abrir una conexión en un servidor *Medulla* para resolver la consultada *instrucción* con los *parámetros* dados.

Desconexión

Sintaxis:

<~k~>

Descripción:

Cierra la conexión abierta por la instrucción *l*.

SAVIA para ejecutantes

Activación

Sintaxis:

<~i|w~>

Descripción:

Cambia el estado de ejecutante dado en la instrucción de enlace a *activo* e incrementa el contador de ejecutantes activos en la medulla regente. La diferencia entre usar *i* ó *w* es que el segundo bloquea al ejecutante hasta recibir el aviso de que la condición "contador de ejecutantes igual a *minimum*" se ha satisfecho; además de que si usa *w*, se esperan cualquiera de las 3 respuestas posibles (**!** *ok*, **?** *error* y **#** *espera*). La instrucción *i* simplemente prosigue con el resto de los pasos del ejecutante y admite solo dos tipos de respuesta: *ok* y *error*.

Desactivación

Sintaxis:

<~q~>

Descripción:

Es solicitada en el método *necrosis* de la clase *Hydra* y recibe por respuesta *ok* o *error*. Al recibir esta señal, la medulla regente marca al ejecutante como *finalizado* y a partir de entonces, ignora todo tipo de mensajes que van hacia dicho ejecutante.

Estado

Sintaxis:

<~s~>[id_ejecutante]

Descripción:

Solicitud del estado de un ejecutante dado su *id*. El resultado es una tupla que contiene información acerca de el estado de ejecución, la cantidad de mensajes en cola (entrantes y salientes) y recibidos y enviados, etc.

Mensajería

Sintaxis:

<~m~> acción id_ejecutante longitud

Descripción:

Envía o recibe mensajes según la acción que se especifique: *in* o *out* hacia o desde *id_ejecutante* con una *longitud* total en caracteres. La primera recibe mensajes provenientes de *id_ejecutante* o si no hay ninguna, espera a que lo haya; mientras la segunda acción pone en cola el mensaje con destino *id_ejecutante*. Si el valor del *id* falta recibe o emite de/a cualquier ejecutante. Inmediatamente después se emite una señal de cualquiera de los tipos vistos: error, ok o espera. En el tercer caso se supone que se monte un servidor en espera de la recepción del mensaje.

SAVIA para clientes Heracles

Alta de la medulla local en una visión

Sintaxis:

<~a~> host [port]

Descripción:

Solicita que la medulla local se encole en la visión a la que pertenece *host* y si se especifica la comunicación la hará por el puerto *port.*

Ejecución de tarea

Sintaxis:

<~r~> min [max] tarea [dependencia1 [dependencia2 [dependencia3 ...]]]

Descripción:

Invoca la ejecución de al menos *min* módulos *tarea* basados en **hydra_egg.py** que dependen de los archivos *dependencia1*, *dependencia2*…

SAVIA para Medullas

Alta de la medulla local a una visión

Sintaxis:

<~a~>

Descripción:

Solicita que la agreguen a la visión. La acción devuelve la visión si se comunica con un regente o el nombre de la regente de la visión si el nodo contactado no es el dominante.

Anexo D: Clasificación de las Características según complejidad (COCOMO II)

Tabla 9 Ficheros lógicos internos (ILF), Ficheros de interfaz externa (ELF)

ELF - ILF			
Elementos de datos			
Ficheros	1 - 19	20 - 60	61+
1	Bajo	Bajo	Medio
2 - 5	Bajo	Medio	Alto
6+	Medio	Alto	Alto

Tabla 10 Salidas externas (EO) Consultas externas (EQ)

EO - EQ			
Elementos de datos			
Ficheros	1 - 5	6 -19	20+
O,1	Bajo	Bajo	Medio
2 - 3	Bajo	Medio	Alto
4+	Medio	Alto	Alto

Tabla 11 Entradas externas (EI)

EI			
Elementos de datos			
Ficheros	1 - 4	5 - 15	16+
0,1	Bajo	Bajo	Medio
2 - 3	Bajo	Medio	Alto
4+	Medio	Alto	Alto

Tabla 12 Pesos según nivel de complejidad

	Nivel de complejidad		
Características	Bajo	Medio	Alto
ILF	7	10	15
ELF	5	7	10
EI	3	4	6
EO	4	5	7
EQ	3	4	6