Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

PROYECTO DE GRADO

Cloud Computing sobre infraestructuras de software libre y su aplicación al estudio del desarrollo embrionario

Mathias Escobar
mathiescobar@gmail.com

Santiago Falero
sanfalero@gmail.com

Jennifer Martín
jennifermartinferreira@gmail.com

Guillermo Urrutia
guilleup@gmail.com

Diciembre de 2015

Tutor:
Sergio Nesmachnow

Cloud computing sobre infraestructuras de software libre y su aplicación al estudio del desarrollo embrionario

Resumen

La computación científica requiere de una gran cantidad de recursos informáticos para realizar experimentos a gran escala. Tradicionalmente, este requerimiento se ha abordado mediante el uso de infraestructuras de computación de alto rendimiento dedicadas (e.g. clusters y supercomputadoras), que suelen ser difíciles de instalar, mantener y operar. La computación distribuida en la nube (Cloud Computing) proporciona un modelo completamente nuevo de utilización de la infraestructura informática, que permite acceder a los recursos de forma dinámica con un esfuerzo mínimo de configuración e interacción del proveedor.

En este informe se presenta el diseño e implementación de una plataforma distribuida y tolerante a fallos, construida bajo el paradigma de Cloud Computing utilizando herramientas y tecnologías de código abierto. Esta plataforma funciona como entorno de ejecución para simulaciones que permiten explorar la dinámica de procesos biológicos mediante el análisis cuantitativo del movimiento de moléculas en células vivas. Estas simulaciones contribuyen a la investigación del complejo proceso que ocurre en los estadios iniciales del desarrollo embrionario.

La plataforma fue construida integrando recursos computacionales de la Facultad de Ingeniería (FING) y la Universidad de Buenos Aires (UBA). Para la gestión de los recursos se utilizó OpenNebula, un software de código abierto que permite el desarrollo de plataformas cloud flexibles, extensibles y escalables. Se implementó un mecanismo que permite aumentar el poder de cómputo de la plataforma, integrando recursos provistos por otras instancias de OpenNebula de manera dinámica. Para el control y coordinación de los diversos componentes de la plataforma se utilizó ZooKeeper, un software de código abierto que ofrece una alta confiabilidad en la coordinación de aplicaciones distribuidas.

La plataforma construida permitió la ejecución paralela y eficiente de simulaciones, realizando una contribución práctica al estudio de fenómenos biológicos complejos.

Palabras clave: computación científica, computación paralela, cloud computing, opennebula, zookeeper, desarrollo embrionario

Índice general

Capítulo 1

Introducción

La computación científica consiste en la construcción de modelos matemáticos y métodos numéricos para resolver problemas científicos y de ingeniería. Una de las principales aplicaciones de la computación científica es la simulación numérica, que permite el estudio de sistemas complejos y fenómenos naturales que serían demasiado costosos, peligrosos o incluso imposibles de analizar mediante la experimentación directa [24].

En general, la computación científica requiere una gran cantidad de recursos informáticos para realizar experimentos a gran escala. Tradicionalmente, este requerimiento se ha abordado mediante el uso de infraestructuras de computación de alto rendimiento dedicadas (e.g. clusters y supercomputadoras), que suelen ser difíciles de instalar, mantener y operar. La computación distribuida en la nube (Cloud Computing) proporciona un modelo completamente nuevo de utilización de la infraestructura informática, que permite acceder a los recursos de forma dinámica [25]. El Instituto Nacional de Estándares y Tecnología (National Institute of Standards and Technology, NIST) de Estados Unidos, define Cloud Computing como un modelo de tecnología de la información que permite el acceso bajo demanda a recursos informáticos como redes, servidores, almacenamiento, aplicaciones y servicios, que pueden ser provistos rápidamente y liberados con un esfuerzo mínimo de configuración o interacción del proveedor [17].

La Espectroscopia de Correlación de Fluorescencia (Fluorescence Correlation Spectroscopy, FCS) y las diversas técnicas de imagenología de células vivas han ampliado notablemente la comprensión del complejo proceso que ocurre en los estadios iniciales del desarrollo embrionario, contribuyendo a la realización de estudios cuantitativos de alta fidelidad. La FCS es una técnica utilizada para explorar la dinámica de procesos biológicos mediante la obtención de información cuantitativa acerca del movimiento de moléculas en células vivas. Esta técnica consiste en el análisis de la intensidad de las fluctuaciones causadas por moléculas marcadas con fluorescencia, que se desplazan a través de un pequeño volumen de observación de un microscopio [7].

En escenarios simples (e.g. moléculas moviéndose de forma pasiva en un medio homogéneo), los datos experimentales obtenidos de la FCS pueden ajustarse a funciones analíticas derivadas del análisis teórico. Sin embargo, muchos procesos dinámicos a nivel celular constituyen escenarios más complejos donde el análisis experimental puede ser combinado con simulaciones para ayudar a la interpretación de los datos.

La combinación de las herramientas Monte Carlo Cell (MCell) [20] y Fluorescence Emission Recipes and Numerical routines Toolkit (FERNET) [2] permite simular las interacciones moleculares que ocurren en escenarios realistas y aplicar diversas técnicas basadas en FCS. MCell es un software basado en algoritmos de Monte Carlo para simular el comportamiento molecular dentro y entre células. FERNET es un software desarrollado en la Universidad de Buenos Aires (UBA), que permite generar trazas de intensidad de fluorescencia a partir de simulaciones realizadas con MCell. Los datos generados por FERNET pueden ser comparados con los datos experimentales para dar soporte a la validación de modelos biológicos.

En muchos casos resulta interesante realizar un conjunto de simulaciones a partir de un mismo modelo variando determinados parámetros (técnica conocida como barrido paramétrico). En este

tipo de experimentos, los resultados de las simulaciones son analizados en conjunto para validar hipótesis planteadas sobre el modelo propuesto. La ejecución de MCell y FERNET demanda una alta capacidad de cómputo debido a que el movimiento y las reacciones de cada molécula debe ser calculado en cada paso de tiempo, limitando las posibilidades de ejecutar grandes cantidades de simulaciones en una computadora personal. Dadas las características de MCell y FERNET, no es posible aplicar paralelismo en la ejecución de una simulación, pero sí en la ejecución de un conjunto de simulaciones independientes. En este contexto, sería interesante contar con un sistema distribuido para ejecutar conjuntos de simulaciones de forma paralela y a gran escala.

El objetivo de este proyecto es diseñar e implementar un sistema escalable que permita ejecutar grandes cantidades de simulaciones en forma paralela y eficiente, basándose en el paradigma de Cloud Computing y utilizando tecnologías de código abierto. Para alcanzar este objetivo se propone configurar un cloud privado integrando los recursos computacionales provistos por la Facultad de Ingeniería de la UdelaR (Uruguay) y la Facultad de Ciencias Exactas y Naturales de la UBA (Argentina). Para la creación del cloud se propone utilizar OpenNebula [18], un software de código abierto dedicado a la gestión integral de centros de datos y servicios de Cloud Computing. Para ampliar el poder de cómputo del sistema, se propone implementar un mecanismo que permita integrar recursos provistos por otras instancias de OpenNebula de forma dinámica y con esfuerzo mínimo de configuración.

Sobre los recursos virtualizados provistos por OpenNebula se propone ejecutar diversos componentes de software del sistema, utilizando un esquema de trabajo maestro-esclavo. Estos componentes se comunican entre sí a través de ZooKeeper [13], un software de código abierto desarrollado por The Apache Software Foundation que ofrece un servicio centralizado para la coordinación y configuración de aplicaciones distribuidas.

El sistema propuesto permite brindar soporte a la ejecución de simulaciones solicitadas por los usuarios a través de un portal web. Las solicitudes se almacenan en una base de datos y posteriormente se distribuyen para su ejecución entre un conjunto de procesos esclavos, que operan bajo la supervisión y coordinación de un proceso maestro. Para ejecutar las simulaciones solicitadas de manera eficiente, el proceso maestro aplica algoritmos de balanceo de carga que tienen en cuenta las características de cada simulación y el poder de cómputo de los recursos disponibles, con el objetivo de minimizar el tiempo total de ejecución de la carga de trabajo.

En los sistemas distribuidos, los fallos pueden ocurrir más a menudo que en los sistemas centralizados. Por este motivo, el sistema a desarrollar cuenta con mecanismos que permiten la detección y recuperación ante fallos en los principales componentes del sistema. Estos mecanismos son implementados a partir del servicio de notificaciones que ofrece Zookeeper y la capacidad de replicación del servidor central de coordinación.

Este proyecto se enmarca en el contexto de colaboración existente entre la Facultad de Ciencias Exactas y Naturales de la UBA (Argentina), la Facultad de Ingeniería de la UdelaR (Uruguay) y el Australian Regenerative Medicine Institute (Australia). En este contexto, la principal contribución de este proyecto es el estudio y la creación de un mecanismo para integrar centros de datos gestionados por OpenNebula, así como su análisis y validación sobre un sistema distribuido orientado a la computación científica.

El trabajo se divide en 6 capítulos. La introducción presenta brevemente la motivación del trabajo y los aspectos tratados en el resto del documento. El marco teórico describe en profundidad los conceptos, el software y las herramientas utilizadas en el proyecto. El tercer capítulo describe el proceso de instalación y configuración de OpenNebula en el cluster FING, la creación de las máquinas virtuales donde se ejecutarán los principales componentes que conforman el sistema, y el mecanismo de integración con otras instancias de OpenNebula (en particular con la instancia instalada en la UBA). En el cuarto capítulo se presenta la arquitectura general, la distribución física y el funcionamiento interno de los distintos componentes. En el quinto capítulo se presenta el análisis experimental que evalúa el sistema en funcionamiento y se realizan experimentos de rendimiento y tolerancia a fallos. Finalmente, en el sexto capítulo se presentan las conclusiones del proyecto y las principales líneas de trabajo futuro.

Capítulo 2

Marco teórico

En este capítulo se desarrollan los principales conceptos vinculados al proyecto. Se comienza con una introducción al concepto de Cloud Computing, detallando los distintos modelos de servicio y de implementación (clouds públicos, privados, comunitarios e híbridos). Luego, se describen las herramientas utilizadas para la implementación del sistema: OpenNebula como gestor de infraestructura y virtualización, Zookeeper como coordinador de procesos, MCell como motor de ejecución de simulaciones moleculares y FERNET como herramienta para el análisis e interpretación de las simulaciones.

2.1. Cloud computing

El Instituto Nacional de Estándares y Tecnología (National Institute of Standards and Technology, NIST) de Estados Unidos, define Cloud Computing como un modelo de tecnología de la información que permite el acceso bajo demanda a recursos informáticos como redes, servidores, almacenamiento, aplicaciones y servicios, que pueden ser provistos rápidamente y liberados con un esfuerzo mínimo de configuración o interacción del proveedor [17].

El cloud se caracteriza por ofrecer un servicio bajo demanda y con elasticidad, lo que permite a los clientes obtener los recursos que requieren del proveedor de forma dinámica según sus necesidades y pagando lo que realmente consumen. Un ejemplo claro es un pico de demanda de servicio, que puede ser afrontado rápidamente y sin mayores inconvenientes gracias a la capacidad de auto escalar del cloud, evitando que el servicio ofrecido deje de funcionar por incapacidad de afrontar la demanda impredecible. Los proveedores de infraestructuras cloud utilizan la virtualización para ofrecer los servicios, ya que es ideal para afrontar picos de demanda por la agilidad y flexibilidad que permite.

Una importante funcionalidad del cloud es que los recursos informáticos del proveedor se combinan para servir a múltiples consumidores mediante un modelo de tenencia múltiple (un principio de arquitectura de software donde una sola instancia de la aplicación se ejecuta, pero sirviendo a múltiples clientes), con diferentes recursos físicos y virtuales dinámicamente asignados y reasignados de acuerdo a la demanda del consumidor. Existe independencia de la ubicación, ya que el cliente generalmente no tiene control o conocimiento sobre la ubicación exacta de los recursos proporcionados. Sin embargo, el cliente generalmente puede especificar la ubicación en un nivel de abstracción más alto (e.g. un país o un estado).

2.1.1. Modelos de servicio del cloud

Los modelos de servicio del cloud [27] ofrecen a los usuarios distintas características, como puede verse en la figura 2.1. Por ejemplo, un usuario puede adquirir servicios a nivel de infraestructura y obtener recursos computacionales para sus aplicaciones, también puede acceder a nivel de plataforma para utilizar como ambiente de su desarrollo, o acceder al cloud a nivel de

software, permitiéndole utilizar aplicaciones ofrecidas por el proveedor. Las diferentes jerarquías y modelos de servicio se describen a continuación.

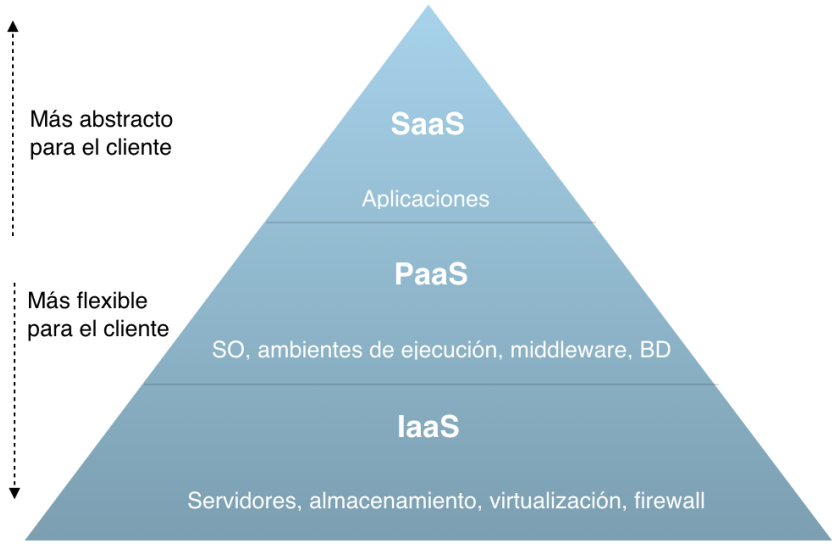


Figura 2.1: Modelos de servicio del cloud

Infraestructura como servicio (Infrastructure as a Service, IaaS). IaaS es un modelo donde un proveedor ofrece servicios de computación y almacenamiento, haciéndose cargo de su alojamiento y mantenimiento. En general, se ofrecen recursos en un ambiente virtualizado que incluye máquinas virtuales, almacenamiento, direcciones IP, firewalls y balanceadores de carga. El consumidor alquila recursos de hardware en vez de comprarlos. Este modelo permite ir variando el consumo de los recursos en función de las necesidades del consumidor, aprovechando los beneficios que ofrece la virtualización.

Plataforma como servicio (Platform as a Service, PaaS). PaaS es un modelo que está por encima del IaaS en cuanto al nivel de abstracción que ofrece al cliente. En el modelo PaaS, el proveedor ofrece una plataforma de computación que incluye un sistema operativo, un ambiente de ejecución, bases de datos, servidores de aplicación, sistemas de mensajería, etc. Estos componentes están instalados y configurados con el fin de brindar un ambiente acorde a las necesidades de los desarrolladores, facilitando el proceso de construcción y puesta en funcionamiento de aplicaciones.

Software como servicio (Software as a Service, SaaS). SaaS es el modelo de servicio de abstracción más alta respecto a los modelos IaaS y PaaS. En el modelo SaaS se provee a los usuarios acceso a aplicaciones y bases de datos a través de Internet. Las aplicaciones son instaladas en el cloud por el proveedor y son accedidas a demanda por los clientes (mediante suscripción al servicio o pago por uso). En el modelo SaaS, el manejo de la infraestructura y las plataformas es transparente para los clientes, que no requieren instalar o ejecutar las aplicaciones del cloud en sus computadoras ni preocuparse por su mantenimiento. Las aplicaciones en el cloud pueden soportar un gran número de usuarios, ya que pueden adaptarse a la variabilidad de demanda mediante la clonación de tareas en máquinas virtuales y balanceadores de carga de manera transparente para el usuario.

2.1.2. Modelos de implementación del cloud

Los modelos de implementación determinan el modo operativo, el ambiente, el tamaño y el acceso al cloud. Las organizaciones pueden optar por el modelo que más se adapte a sus necesidades para llevar a cabo sus negocios.

Clouds públicos. Los clouds públicos son ofrecidos por los proveedores de servicios a través de Internet y están disponibles para el uso del público en general. En los cloud públicos el consumidor contrata el servicio (que puede ser gratis en algunos casos) y utiliza los recursos para sus necesidades, generalmente desconociendo el lugar en donde están ubicados y si están siendo compartidos con otros usuarios. Este mecanismo implica que la seguridad no sea la característica principal en un cloud público, en especial cuando se trabaja en redes no confiables.

Una de las ventajas que presentan los clouds públicos es que pueden operar a demanda, teniendo la flexibilidad de escalar cuando el cliente lo requiera y pagando solamente por los recursos que se utilizan en ese momento, lo que suele ser una alternativa más económica a la de ampliar la infraestructura de la propia organización. Algunos de los proveedores de clouds públicos más conocidos son Amazon Web Services (AWS), Google y Microsoft.

Clouds privados. Los clouds privados se basan en infraestructura dedicada al uso de una organización. Esta infraestructura puede ser manejada por la propia organización o por terceros. Por lo general, tanto el proveedor como los consumidores de un cloud privado son departamentos de la propia organización.

En el modelo de implementación privado existen variantes dependiendo de quien se encarga del alojamiento y/o del mantenimiento de la infraestructura del cloud. Cuando se requiere mayor control sobre los datos y mayor seguridad, el cloud puede ser alojado en las instalaciones de la propia organización (en inglés se conoce como *on-premises*). Un cloud construido en las instalaciones de la propia organización requiere un esfuerzo adicional de mantenimiento, espacio físico, disponibilidad del hardware, y costos derivados de la construcción de la infraestructura subyacente. Por otro lado, el cloud puede ser alojado en organizaciones externas que se especializan en la creación y gestión de clouds privados, ofreciendo sus instalaciones y manteniendo el concepto de privacidad, asegurando recursos físicos exclusivos para la organización cliente.

Clouds comunitarios. Los clouds comunitarios son infraestructuras compartidas entre organizaciones con intereses en común (e.g. políticas, requerimientos de seguridad, jurisdicción). El cloud comunitario puede ser gestionado por un subconjunto de las organizaciones participantes o por una organización ajena a la comunidad.

El objetivo de los clouds comunitarios es que las organizaciones tengan los beneficios de escalabilidad de un cloud público, adicionando el nivel de privacidad, seguridad y políticas que usualmente se manejan en clouds privados.

Clouds híbridos. Un cloud híbrido es una composición entre dos o más clouds de tipo privado, público o comunitario, que si bien son distintos, están unidos por una tecnología (propietaria o estandarizada) para explotar los beneficios de cada uno de los modelos. El objetivo principal es mantener el ambiente operativo lo más eficiente posible para el negocio de la organización que lo utiliza. Como contrapartida, un cloud híbrido requiere mantener las distintas plataformas que lo integran y la comunicación entre estas plataformas.

Un ejemplo de cloud híbrido se presenta cuando una organización propietaria de un cloud privado contrata temporalmente los servicios de un cloud público para satisfacer picos de demanda que puedan surgir en el cloud de la organización (modelo conocido como *cloud bursting*) [16]. Otro ejemplo de cloud híbrido se da cuando una organización quiere ofrecer servicios a sus clientes a través de un cloud público, pero prefiere mantener algunos datos sensibles de la organización como servicios internos en un cloud privado.

2.2. OpenNebula

OpenNebula [18] es un software de código abierto orientado al desarrollo de clouds privados, públicos e híbridos. Este software comenzó como un proyecto de investigación llevado a cabo por Ignacio M. Llorente y Rubén S. Montero en 2005 y fue lanzado públicamente bajo licencia Apache 2 en marzo de 2008. Actualmente, OpenNebula es desarrollado y mantenido por The OpenNebula Project [21].

El software OpenNebula se presenta como plataforma para la gestión integral de centros de datos (*data centers*) y la exposición de servicios de Cloud Computing. OpenNebula incluye herramientas para la creación, administración y gestión del ciclo de vida de máquinas virtuales, permitiendo automatizar su proceso de instanciación y configuración (que puede comprender preparar imágenes de disco, configurar interfaces de red, iniciar determinados servicios, etc.). Estas máquinas virtuales pueden ser instanciadas sobre infraestructura local o sobre proveedores de cloud computing externos (actualmente Amazon EC2, IBM SoftLayer y Microsoft Azure son soportados) permitiendo la implementación de un modelo *cloud bursting*.

2.2.1. Arquitectura general

Una instalación estándar de OpenNebula está conformada por un conjunto de nodos físicos (*hosts*) sobre los cuales se crean las máquinas virtuales y un nodo principal (*front-end*) que ejecuta el proceso principal (demonio) y los diferentes servicios que componen OpenNebula. Las imágenes de disco y los archivos utilizados por las máquinas virtuales son almacenados en *datastores*. Un datastore es cualquier medio de almacenamiento que pueda actuar como repositorio centralizado de imágenes de disco y archivos. Los datastores son típicamente montados sobre arquitecturas de tipo NAS (*Network Attached Storage*) o SAN (*Storage Area Network*). Debe existir al menos una red física que comunique el front-end, los hosts y los datastores. La figura 2.2 ilustra la estructura general de una instalación típica de OpenNebula.

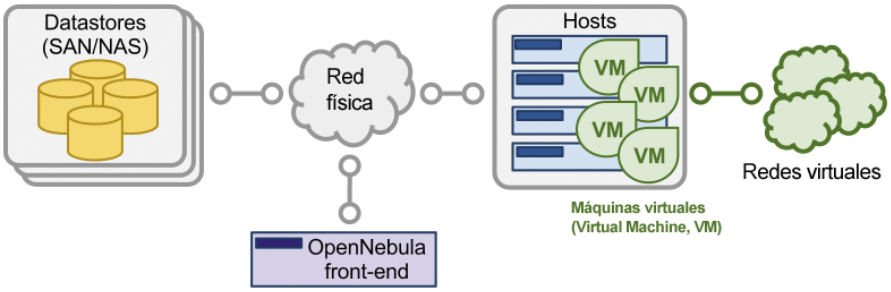


Figura 2.2: Componentes de una instalación estándar de OpenNebula.

2.2.2. Subsistemas

OpenNebula se descompone funcionalmente en varios subsistemas (monitoreo, red, almacenamiento, virtualización, autorización, etc.). Cada subsistema expone sus funcionalidades a través de interfaces estándar e independientes de las tecnologías subyacentes. La comunicación con las tecnologías subyacentes se establece mediante drivers específicos, dándole flexibilidad suficiente al sistema para evitar fuertes dependencias con estas tecnologías y altos costos de cambio, fenómeno que se conoce como bloqueo del proveedor (*vendor lock-in*). OpenNebula incluye drivers para un conjunto de tecnologías (ver figura 2.3) y cuenta con un repositorio de drivers implementados por la comunidad. OpenNebula provee documentación para desarrolladores que quieran implementar nuevos drivers.

La comunicación con los hipervisores instalados en los hosts es necesaria para la ejecución de las acciones requeridas durante el ciclo de vida de las máquinas virtuales. Esta comunicación

se realiza a través del subsistema de virtualización. Los hipervisores soportados actualmente son Xen, KVM y VMWare.

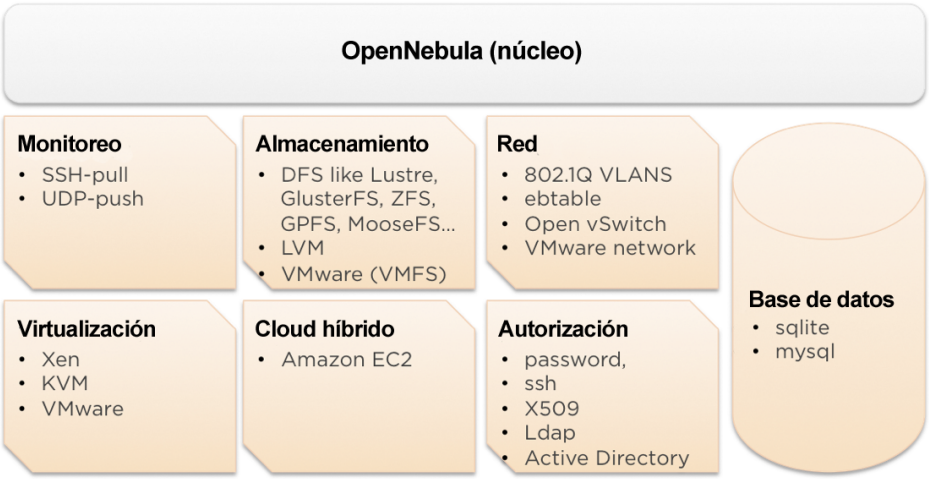


Figura 2.3: Subsistemas y tecnologías actualmente soportadas.

La información de los hosts y máquinas virtuales existentes (e.g. estado, indicadores básicos de rendimiento, consumo de recursos) es recolectada por el subsistema de monitoreo, que se comunica con los hosts a través de UDP o SSH y con los hipervisores a través del subsistema de virtualización. La información recolectada por el subsistema de monitoreo es accesible y utilizada por el resto de los subsistemas. OpenNebula provee su propia implementación para el monitoreo, compuesta por un conjunto de scripts (denominados *probes*) que se ejecutan en los hosts para recolectar la información y métricas necesarias. La salida de estos scripts es enviada a OpenNebula usando un modelo UDP-PUSH en el cual cada host es el encargado de ejecutarlos periódicamente y enviar la salida vía UDP al front-end, o bien un modelo SSH-PULL donde el front-end es quien se encarga de acceder a cada host vía ssh, ejecutar los scripts y obtener la salida. La implementación del monitoreo es extensible permitiendo la incorporación de probes que invoquen sistemas de monitoreo externos y recolecten métricas adicionales.

El subsistema de red permite que las máquinas virtuales distribuidas puedan interconectarse entre sí a través de sus interfaces de red, por medio de segmentos de red (VLANs) independientes, permitiendo al mismo tiempo su conexión con el exterior. Esto permite que cada máquina virtual pueda tener acceso a varias redes, públicas o privadas, definidas mediante templates de red.

El subsistema de almacenamiento se encarga de la gestión de las imágenes de disco y archivos utilizados por las máquinas virtuales. Los datastores se clasifican en tres tipos: system, images y files. Los datastores de tipo system se encargan de almacenar las imágenes de las máquinas virtuales en ejecución, que pueden ser copias o enlaces a imágenes almacenadas en datastores de tipo images, que funcionan como repositorio central. Los archivos que no son imágenes de disco como *kernels*, discos RAM o archivos de contexto, se almacenan en datastores de tipo files. Los archivos de contexto suelen contener metadatos que le permiten a las máquinas virtuales configurar sus interfaces de red y servicios específicos al iniciar, lo que se conoce como proceso de contextualización (se presenta en detalle en la sección 2.2.3).

OpenNebula proporciona sistemas de escalado híbrido, que permiten que los recursos asignados por la infraestructura local (privada) del data center puedan llegar a escalar sobre infraestructura pública de red. El escalado híbrido resulta particularmente útil en condiciones de picos de carga y se implementa a través de la comunicación con interfaces de programación estándar (Application Programming Interface, API) como la API de Amazon EC2, permitiendo establecer la arquitectura de un cloud híbrido. Los metadatos generados y utilizados por OpenNebula para la gestión de la plataforma (hosts, máquinas y redes virtuales, imágenes, datastores, usuarios,

etc.) son persistidos en una base de datos relacional (SQLite es soportada por defecto y se incluye soporte para MySQL). La base de datos MySQL puede ser utilizada en entornos donde se requiera alta disponibilidad del front-end o se pretenda conformar una federación con otras instancias de OpenNebula, lo cual requiere replicar la base de datos (el proceso se detalla en la sección 2.2.6).

El mecanismo de validación de usuarios instalado por defecto en OpenNebula es basado en nombres de usuario y contraseñas, que son almacenados en archivos de configuración con formato de texto plano. Es posible utilizar otros mecanismos como ssh, certificados x509, LDAP o Active Directory.

2.2.3. Contextualización de máquinas virtuales

En OpenNebula, las máquinas virtuales son instanciadas a partir de archivos de base denominados *templates*, que definen las características principales de las máquinas a crear (e.g. CPU, memoria, almacenamiento). Cada máquina virtual instanciada a partir de un template necesita determinados parámetros de configuración particulares, por ejemplo una dirección IP única. Estos parámetros particulares son denominados parámetros de contexto. Los parámetros de contexto son determinados en forma dinámica durante la creación de la máquina virtual y son aplicados generalmente durante el arranque. El proceso de aplicación de los parámetros de contexto se denomina contextualización.

La contextualización es utilizada principalmente para la asignación automática de direcciones IP, permitiendo automatizar completamente el proceso de creación de máquinas virtuales a partir de templates genéricos. Para llevar adelante la contextualización, es necesario que los parámetros de contexto sean accesibles dentro de la máquina virtual, y que se ejecute durante el arranque un servicio capaz de leer estos parámetros y configurar la máquina virtual.

OpenNebula genera una imagen en formato ISO para cada máquina virtual previo a su creación, denominada imagen de contexto. En la imagen de contexto se incluye por defecto un archivo de nombre context.sh, generado por OpenNebula con los parámetros de contexto básicos (parámetros de configuración de red, claves ssh, etc.). Es posible incluir otros archivos de contexto particulares como certificados, archivos de configuración, scripts de inicialización, etc., indicándolo en el template.

La imagen de contexto es montada como un CD-ROM en la máquina virtual. Este CD-ROM de contexto es accesible dentro de la máquina virtual, y por lo tanto cualquier proceso que ejecute con privilegios suficientes puede acceder al contenido y configurar la máquina virtual en consecuencia. En la figura 2.4 se ilustra el acceso desde la máquina virtual al contenido de la imagen de disco y la de contexto.

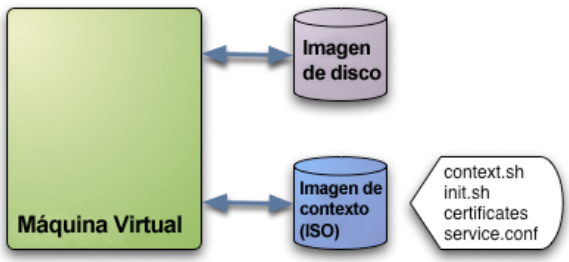


Figura 2.4: Imágenes de disco y de contexto montadas por OpenNebula en una máquina virtual.

El proceso de configuración de la máquina virtual puede incluir definición de interfaces de red, modificación de la lista de claves ssh autorizadas, establecimiento del nombre del host, etc. OpenNebula provee scripts que realizan estas tareas de configuración básicas, adaptados para algunas distribuciones de Linux. Estos scripts deben ser pre-instalados en la imagen de disco sobre la cual se crearán las máquinas virtuales.

El mecanismo de contextualización basado en imágenes que utiliza OpenNebula es el recomendado por el estándar OVF (Open Virtualization Format). Una de las principales ventajas de este mecanismo es que no necesita que la máquina virtual esté conectada a la red, y por lo tanto puede ser utilizado para configurarla.

2.2.4. Interfaz XML-RPC

El demonio de OpenNebula (proceso central instalado en el front-end) ofrece las funcionalidades necesarias para la administración general de la plataforma y de sus recursos (hosts, datastores, máquinas virtuales, redes virtuales, imágenes, templates, usuarios, etc.). Estas funcionalidades son expuestas a través de una API de tipo XML-RPC, permitiendo que sean invocadas de forma remota. El protocolo XML-RPC se caracteriza por utilizar XML para codificar los datos y HTTP como protocolo de transmisión de mensajes.

La comunicación entre el software desarrollado en el marco de este proyecto y el entorno de OpenNebula es realizada mediante la API XML-RPC de OpenNebula. La creación y destrucción programática de máquinas virtuales a través de la API es una de las las funcionalidades más relevantes para el proyecto, ya que uno de los objetivos del proyecto es administrar el uso de los recursos de manera dinámica en función de la carga de trabajo. La API también es invocada desde el sistema implementado para obtener información sobre la capacidad de cómputo de los hosts (frecuencia de reloj del procesador, capacidad de memoria RAM, etc). La información de los hosts es utilizada para determinar la capacidad y las características de las máquinas virtuales a crear, con el objetivo de distribuir la carga de trabajo en forma óptima (este proceso se detalla en la sección 4.2.4).

2.2.5. Interfaz web Sunstone

Sunstone es una aplicación web que permite acceder a las principales funcionalidades del demonio de OpenNebula. Esta aplicación permite administrar y realizar operaciones básicas sobre todos los recursos de la plataforma OpenNebula. Todas las acciones realizadas desde Sunstone son resueltas invocando los servicios expuestos por el demonio a través de la API XML-RPC. En la figura 2.5 se muestra la página principal de la interfaz de administración de Sunstone.

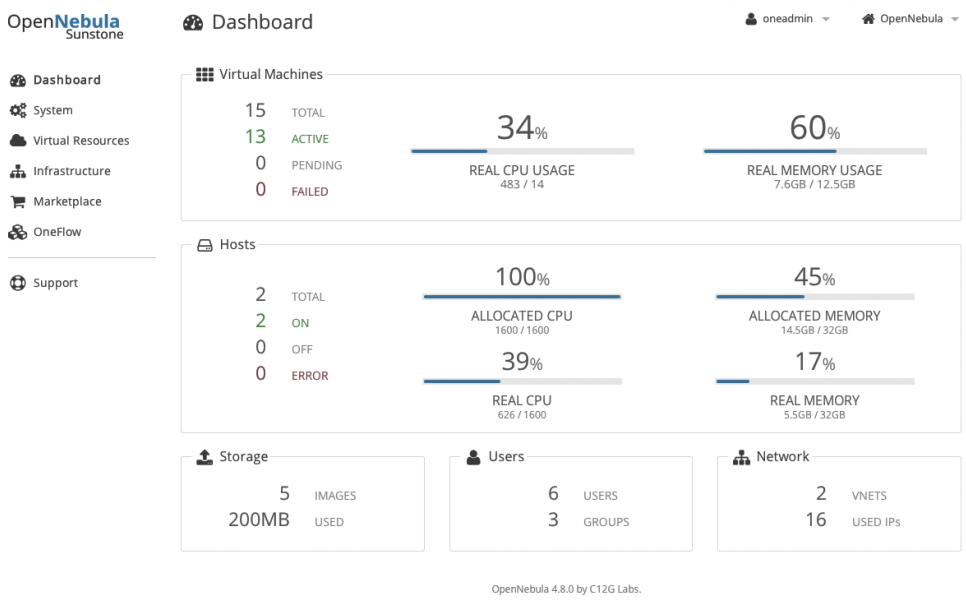


Figura 2.5: Interfaz de administración web Sunstone.

2.2.6. Federación de data centers

OpenNebula soporta la integración de instalaciones independientes (también denominadas instancias o zonas) mediante la federación de data centers. Una federación está compuesta por una instancia principal (zona maestra) y una o más instancias secundarias (zonas esclavas) que se fusionan para compartir los recursos y la administración de la infraestructura.

La federación implica una integración a bajo nivel entre las instancias, resuelta por OpenNebula mediante la replicación de un subconjunto de tablas de la base de datos y la invocación a los demonios de cada instancia a través de la API XML-RPC. Para mantener la consistencia de los datos, la zona maestra es la única habilitada para escribir en las tablas compartidas. Cualquier acción de escritura realizada desde otra zona debe ser solicitada a la zona maestra. En la figura 2.6 se muestra la comunicación entre instancias que conforman una federación.

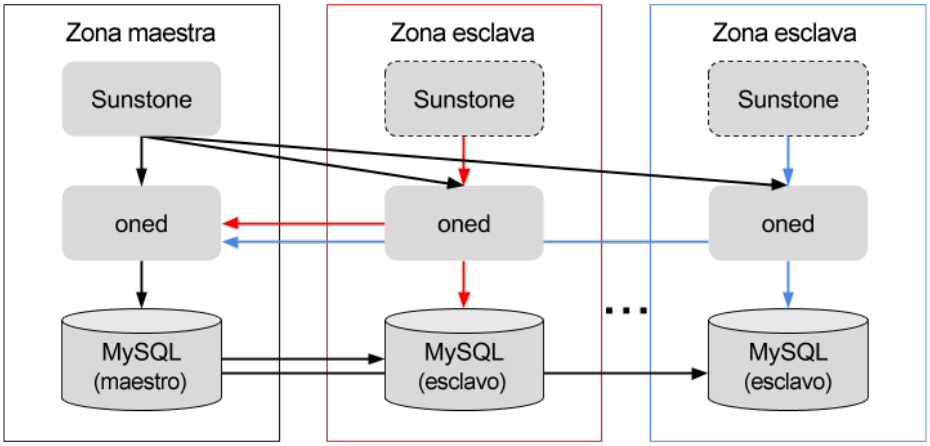


Figura 2.6: Comunicación entre instancias de una federación.

2.3. Apache ZooKeeper

Apache ZooKeeper [13] es un software de código abierto desarrollado por The Apache Software Foundation que ofrece un servicio centralizado para la coordinación y configuración de aplicaciones distribuidas. El software fue creado por el grupo de investigación Yahoo! Research y nació como un sub-proyecto de Hadoop. Actualmente ZooKeeper es un proyecto con vida propia y se ha vuelto muy utilizado en sistemas distribuidos (Apache HBase, Apache Kafka, Apache Solr, Yahoo! Fetching Service, Facebook Messages y Netflix son algunos de los proyectos que lo utilizan).

El servicio de coordinación de aplicaciones distribuidas (denominadas clientes) ofrecido por ZooKeeper se basa en una estructura jerárquica compartida de registros de datos (accesible para lectura y escritura por todos los clientes) y un sistema de notificaciones que permite que los clientes se suscriban a registros específicos para ser notificados de cambios realizados por otros clientes. Los registros pueden ser utilizados para almacenar información de coordinación (e.g. datos de estado, ubicación y configuración de los clientes).

ZooKeeper está implementado en Java. El software se debe instalar en un conjunto de servidores que serán los encargados de mantener la estructura de datos, ejecutar el sistema de notificaciones y responder a las peticiones de las aplicaciones cliente. ZooKeeper provee APIs en Java y C para las aplicaciones cliente. Estas APIs facilitan la implementación de primitivas de sincronización típicas (e.g. semáforos) y algoritmos de coordinación de procesos (e.g. elección de líder, colas distribuidas, bloqueos compartidos).

2.3.1. Modelo de datos

ZooKeeper almacena los datos en una estructura jerárquica de registros que puede verse como un sistema de archivos. En lugar de guardar archivos o directorios, ZooKeeper almacena *znodes* (nodos ZooKeeper). Estos znodes pueden ser registros de datos y padres de otros znodes al mismo tiempo. Existe un único znode raíz a partir del cual se construye la estructura. En la figura 2.7 se presenta un árbol de znodes de ejemplo.

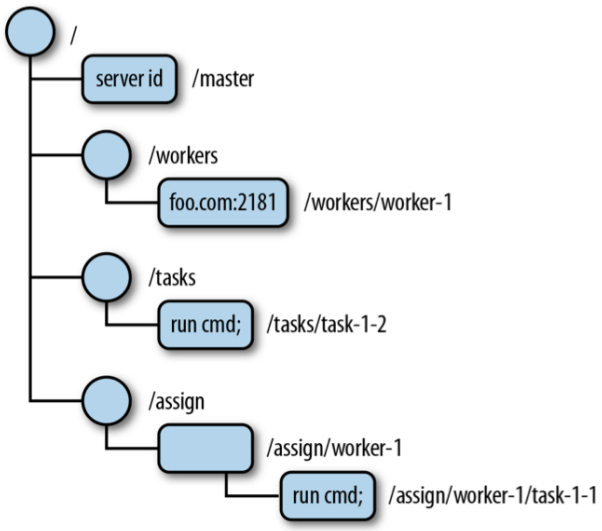


Figura 2.7: Estructura de znodes en ZooKeeper. [13]

El servicio ZooKeeper puede estar replicado en un conjunto de servidores (denominado *ensemble*) para aumentar su disponibilidad. Los servidores del ensemble mantienen en memoria el árbol de znodes, y almacenan en disco el log de transacciones y las capturas del estado del sistema (*snapshots*). Dado que el árbol de znodes se mantiene en memoria, el servicio puede alcanzar alto rendimiento y baja latencia. Como contrapartida, el tamaño del árbol está limitado por la cantidad de memoria de cada servidor. Además, ZooKeeper limita a 1MB el tamaño de los datos de cada znode.

Los znodes presentan un número de versión y una marca de tiempo (*timestep*), y son mantenidos en una estructura llamada *stat* que es actualizada cuando se realiza un cambio. Estas características permiten realizar operaciones de actualización sujetas a una versión particular de un znode. En ZooKeeper existen tres clasificaciones de znodes que se describen a continuación.

Persistentes Los znodes persistentes sólo pueden ser eliminados mediante un borrado explícito por parte de un cliente. Usualmente los znodes persistentes son utilizados para almacenar datos que deben ser preservados a lo largo del tiempo, independientemente de la existencia o no del cliente que los creó.

Efímeros Los znodes efímeros son eliminados por ZooKeeper si el cliente que los creó pierde la comunicación con los servidores (para ZooKeeper este cliente deja de existir). Los znodes efímeros, al igual que los znodes persistentes, pueden ser eliminados mediante un borrado explícito. Los znodes efímeros no pueden tener znodes hijos.

Secuenciales ZooKeeper mantiene internamente una secuencia asociada a cada znode. Cada znode creado en modo secuencial toma el siguiente valor de la secuencia asociada al znode padre. El valor tomado es concatenado al final del nombre asignado al znode. Por ejemplo, el primer znode secuencial creado bajo el znode "/system" será "/system/<nombre-asignado>-0000000001", el segundo será "/system/<nombre-asignado>-0000000002" y así sucesivamente. Los znodes secuenciales pueden ser znodes persistentes o efímeros.

ZooKeeper permite manipular el árbol de znodes mediante las siguientes operaciones básicas:

- create */path data*: crea un znode en la ruta */path* con el contenido *data*
- delete */path*: borra el znode en la ruta */path*
- exists */path*: verifica la existencia del znode identificado con la ruta */path*
- setData */path data*: escribe los datos *data* en el contenido del znode */path*
- getData */path*: retorna los datos en el znode identificado por */path*
- getChildren */path*: retorna una lista con los nombres de los znodes hijos del znode */path*

Las operaciones de lectura y escritura sobre el contenido de los znodes no son parciales. Esto significa que una modificación del contenido de un znode reemplaza todo el contenido existente, y la lectura de los datos de un znode implica obtener todo su contenido.

2.3.2. Eventos y notificaciones

ZooKeeper es un servicio orientado a sistemas distribuidos. Realizar operaciones innecesarias en ZooKeeper puede producir un aumento de la latencia en la comunicación entre los componentes del sistema y el ensemble de servidores. ZooKeeper cuenta con un mecanismo de suscripción a eventos generados en los znodes (e.g. creación, borrado, modificaciones). Los clientes ZooKeeper se suscriben a los eventos por medio de un vigilante (*watch*) y son notificados cuando el evento se produce. Esta funcionalidad evita que los clientes realicen accesos innecesarios al servicio (e.g. *polling*) para verificar si la estructura de znodes sufrió algún cambio.

Un watch es una función con un contexto, que se ejecuta una única vez cuando el evento al que fue suscrito por el cliente se dispara. Si un cliente ZooKeeper pretende ser notificado de un evento a lo largo del tiempo, debe registrar un nuevo watch cada vez que el evento de interés es disparado. ZooKeeper permite registrar watches a los siguientes eventos: *i*) creación y/o eliminación de un znode (watch de existencia), *ii*) cambios de contenido de un znode y *iii*) cambios en los hijos de un znode.

ZooKeeper ofrece las siguientes garantías sobre los watches:

- Los watches se ordenan respecto a otros eventos, otros watches y respuestas asincrónicas.
- Un cliente será notificado de un evento antes de que pueda visualizar los cambios que produjo el evento en el modelo de datos.
- El orden de los eventos desde ZooKeeper, corresponde con el orden de los cambios que ha visto el servicio.

Los clientes ZooKeeper no reciben notificaciones mientras están desconectados del servicio. Cuando un cliente se conecta nuevamente al servicio ZooKeeper, todos los watches que se habían registrado previo a la desconexión son registrados nuevamente y disparados si es necesario. En ZooKeeper existe un escenario en donde el cliente puede llegar a perder el evento al que está suscrito. Si el cliente registra un watch de existencia sobre un znode que aún no ha sido creado, y posteriormente el cliente pierde la conexión, entonces puede perder el evento si el znode es creado y borrado durante el periodo de desconexión.

En ZooKeeper, el cliente debe manejar la latencia entre el evento y el envío de un nuevo watch del servicio. Si el cliente ZooKeeper es notificado de un evento y no registra un watch inmediatamente, puede perder los eventos sucesores. Para evitar la pérdida de eventos descrita, ZooKeeper provee integrado a las operaciones de su API, la opción de registrar un nuevo watch inmediatamente que el evento es disparado.

2.3.3. Arquitectura

El servicio ZooKeeper funciona con un único servidor en modo *standalone* o con más de un servidor en modo *quorum*. En modo quorum, todos los servidores mantienen el modelo de datos replicado y responden las peticiones de las aplicaciones cliente.

Cuando ZooKeeper ejecuta en modo quorum y recibe una petición que modifica el modelo de datos, la modificación debe efectuarse en una cantidad mínima de servidores (denominada quorum) para responder satisfactoriamente la petición. Este mecanismo evita que los clientes deban esperar a que los datos se encuentren replicados en todos los servidores para obtener una respuesta del servicio. En su lugar, los datos deben estar replicados al menos en los servidores del quorum.

El quorum es definido por el administrador del servicio, y debe ser al menos la mitad más uno de los servidores del ensemble (deben ser mayoría). Si el quorum es minoría y sus servidores quedan incomunicados con el resto de los servidores, ZooKeeper seguirá funcionando (ya que cuenta con mayoría de servidores activos) pero el resto de los servidores continuará con un modelo de datos desactualizado.

En el ensemble de ZooKeeper uno de los servidores asume el rol de líder y el resto de seguidores. El líder es elegido cuando el ensemble inicia su ejecución o cuando el líder falla (reelección de líder). El proceso de elección de líder finaliza cuando la mayoría de los seguidores (el quorum) sincronizó su estado. Todas las actualizaciones de datos que realizan los clientes sobre el servicio ZooKeeper son direccionadas al líder, y posteriormente, replicadas a los seguidores mediante un proceso llamado emisión atómica (*atomic broadcast*). Una vez que el quorum persiste los datos del atomic broadcast en disco y memoria, el líder salva los cambios y notifica satisfactoriamente al cliente. La figura 2.8 muestra un ejemplo de flujo de replicación de datos en los servidores ZooKeeper, iniciado cuando un cliente actualiza un dato del modelo.

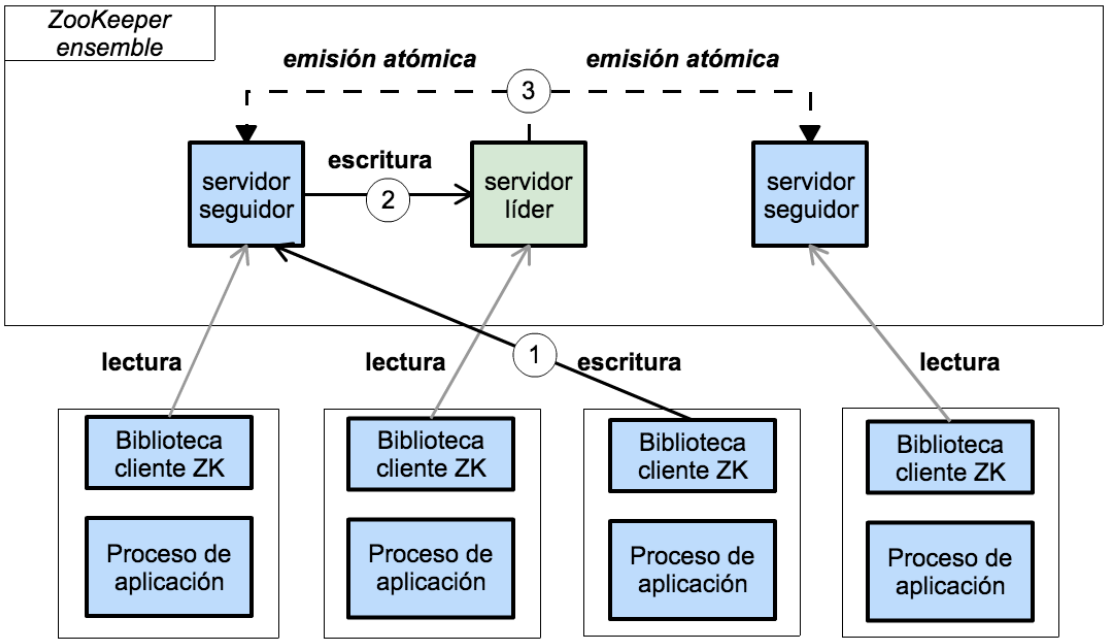


Figura 2.8: Diagrama de replicación de datos en servidores ZooKeeper.

Agregar servidores al ensemble ZooKeeper aumenta la distribución de las peticiones entre servidores y reduce la latencia en lecturas de datos para los clientes. En el caso de las actualizaciones de datos, la latencia no mejora a medida que se agregan servidores en ZooKeeper porque todas las actualizaciones deben ser aprobadas por el líder.

ZooKeeper ofrece garantías sobre las actualizaciones y visualizaciones de los datos que observan los clientes que se detallan a continuación.

Consistencia secuencial Las actualizaciones que realiza un cliente se aplican en el orden que fueron enviadas al servicio ZooKeeper.

Atomicidad Las actualizaciones de datos (sean o no satisfactorias) no son resultados parciales.

Imagen única del sistema Un cliente conectado al servicio mantiene la misma visión del modelo, independientemente del servidor ZooKeeper al que se encuentre conectado.

Confiabilidad En el momento que se actualiza el modelo de datos, los cambios permanecen hasta que el modelo vuelva a ser actualizado. De esta garantía se derivan dos corolarios:

- Si el cliente obtiene un código de retorno exitoso al efectuar una actualización, los cambios serán aplicados. En caso de producirse un fallo (e.g. errores de comunicación, timeout), el cliente no debe asumir que la actualización no fue realizada.
- Una actualización observada por un cliente mediante el retorno de un código exitoso o peticiones de lectura, no es revertida ante una recuperación de una falla del servidor.

Puntualidad Se garantiza que el estado del servicio que observan los clientes ZooKeeper, estará con las últimas actualizaciones dentro de un lapso de tiempo del orden de 10 segundos. Cualquier cambio en el sistema es observado por el cliente en ese lapso, o el cliente detecta un corte del servicio.

Las actualizaciones son persistentes en el tiempo y se aplican en el orden en el que fueron efectuadas, pero no necesariamente son visualizadas por todos los clientes del servicio al mismo tiempo (consecuencia de que los servidores no se actualizan simultáneamente). Los clientes pueden utilizar un watch para estar sincronizados con el estado del servicio.

2.3.4. Sesiones

Para comenzar a realizar operaciones en ZooKeeper, un cliente debe conectarse al servicio indicando IPs y puertos de los servidores ZooKeeper. La API ZooKeeper selecciona uno de los servidores al azar e intenta iniciar una conexión, si no se puede establecer la conexión o se pierde en algún momento, el mecanismo se repite con otro servidor hasta que la sesión quede establecida.

Cuando la conexión se establece con el servicio ZooKeeper, se crea un identificador y una clave para la sesión. El identificador y la clave son válidos ante todos los servidores del ensemble. La sesión es establecida con un único servidor a la vez, pero puede ser migrada a otro servidor en caso de no tener respuesta en el servidor al que está vinculada.

Las sesiones en ZooKeeper mantienen un tiempo de vida que culmina cuando la conexión es cerrada explícitamente por el cliente o por falta de respuesta del servicio (timeout). Para determinar si una sesión expiró, el cliente envía periódicamente paquetes de solicitud/respuesta (ping) a los servidores. Este mecanismo de control se conoce como *heartbeat*, y permite a un servidor ZooKeeper determinar si la sesión sigue activa y al cliente determinar si el servidor sigue activo.

El timeout de sesión se produce cuando los servidores del ensemble no reciben heartbeat del cliente. Al momento del timeout, los servidores ZooKeeper expiran la sesión (invalidando el identificador) y eliminan los watches y znodes efímeros del cliente. Cuando los znodes efímeros son eliminados por ZooKeeper, son disparadas las notificaciones a los clientes que habían registrado un watch en esos znodes.

En la figura 2.9 se muestran los estados y transiciones por las que puede pasar una sesión ZooKeeper durante su ciclo de vida. Inicialmente la sesión está en el estado *No conectado* y pasa a estado *Conectando* durante el proceso de inicialización. Normalmente cuando la conexión es establecida, se pasa al estado *Conectado*. Cuando el cliente pierde la conexión y no puede obtener respuesta del servidor, se vuelve al estado *Conectando* mientras se intenta buscar otro

de los servidores ZooKeeper para conectarse. Si el cliente se puede conectar con otro servidor o conectar nuevamente con el servidor original, se cambia a estado *Conectado* nuevamente (este ciclo puede producirse en varias ocasiones durante el tiempo de vida de la sesión). En otro caso, ZooKeeper declara la sesión expirada y se pasa a estado *Cerrada*. Un cliente también puede cerrar la sesión explícitamente y pasar de estado *Conectado* a *Cerrada*.

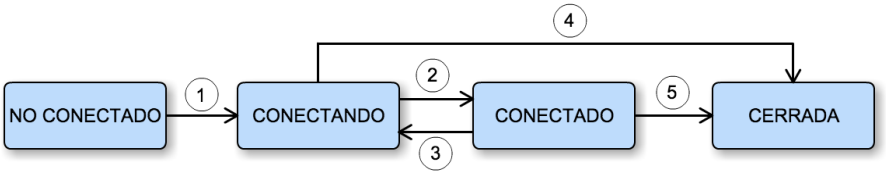


Figura 2.9: Estado de sesiones y transiciones

2.4. Espectroscopía de Correlación de Fluorescencia

La Espectroscopía de Correlación de Fluorescencia (Fluorescence Correlation Spectroscopy, FCS) es una técnica utilizada para explorar la dinámica de procesos biológicos. Esta técnica permite la detección y el análisis de fluctuaciones temporales de pequeños conjuntos de moléculas, combinando una extremada sensibilidad con una elevada confianza estadística. La FCS permite analizar el movimiento de moléculas en células u órganos vivos e intactos (experimentos *in vivo* o *in situ*).

La FCS consiste en el análisis de la intensidad de las fluctuaciones causadas por las moléculas marcadas con fluorescencia. Las moléculas se desplazan a través de un pequeño volumen de observación de un microscopio confocal o de dos fotones de excitación. La FCS es una herramienta muy útil para determinar concentraciones locales, coeficientes de difusión y cambios en la conformación de biomoléculas fluorescentes en condiciones fisiológicas [7].

Para comprender en detalle el complejo mecanismo mediante el cual se forma un organismo se requieren estudios cuantitativos de alta fidelidad en el espacio y el tiempo. La FCS y las diversas técnicas de imagenología de células vivas han ampliado notablemente la comprensión de los detalles moleculares y celulares del desarrollo embrionario.

En escenarios simples (e.g. moléculas moviéndose de forma pasiva en un medio homogéneo) el análisis de FCS se realiza mediante funciones analíticas, que pueden corresponderse con los datos experimentales para recuperar la tasa fenomenológica de los parámetros (e.g, coeficientes de difusión). Sin embargo, muchos procesos dinámicos a nivel celular no siguen modelos simples, y en muchos casos no es posible obtener una función analítica a través del análisis teórico de modelos más complejos. En escenarios más complejos, el análisis experimental puede ser combinado con simulaciones de Monte Carlo para ayudar a la interpretación de la recuperación de los datos. La comparación entre un modelo reducido simulado y los datos experimentales puede revelar pistas importantes acerca de los procesos dinámicos ocultos en los datos obtenidos de FCS.

En la figura 2.10 se muestra el proceso de medición de la FCS, que consiste en los siguientes procedimientos:

1. Representación esquemática de la configuración experimental (figura 2.10a). La muestra (células que expresan una proteína fluorescente) se coloca en la parte superior de la platina de un microscopio confocal o de dos fotones de excitación. El láser de excitación se centra en la muestra en una zona de difracción limitada y los fotones fluorescentes producidos en el pequeño volumen de observación son recogidos como una función de tiempo.

2. Detección de moléculas etiquetadas (figura 2.10b). Las moléculas fluorescentes (rojas) se difunden a través del pequeño volumen de observación (gris) definido en estos microscopios. Las moléculas pueden, por ejemplo, interactuar con otros componentes celulares (azules y verdes).

3. Obtención de una traza representativa de la intensidad de fluorescencia (figura 2.10c). La traza muestra las fluctuaciones originadas por el movimiento de las moléculas fluorescentes dentro y fuera del volumen de observación. Se puede demostrar que la amplitud de las fluctuaciones está inversamente relacionada con el número de moléculas en el volumen de observación, mientras que la duración es determinada por la dinámica de estas moléculas. Esta información puede ser recuperada calculando la función de autocorrelación (ver ecuación 2.1) donde $\delta I(t) = I(t) - < I(t) >$ representa la fluctuación de la intensidad, los paréntesis angulares indican el tiempo medio y τ es el tiempo de retraso.

$$G(\tau) = \frac{< \delta I(t) \cdot \delta I(t+\tau) >}{< I(t) >^2} \tag{2.1}$$

4. Gráfico de ejemplo de la función de autocorrelación obtenida a partir de la traza de la intensidad de fluorescencia (figura 2.10d).

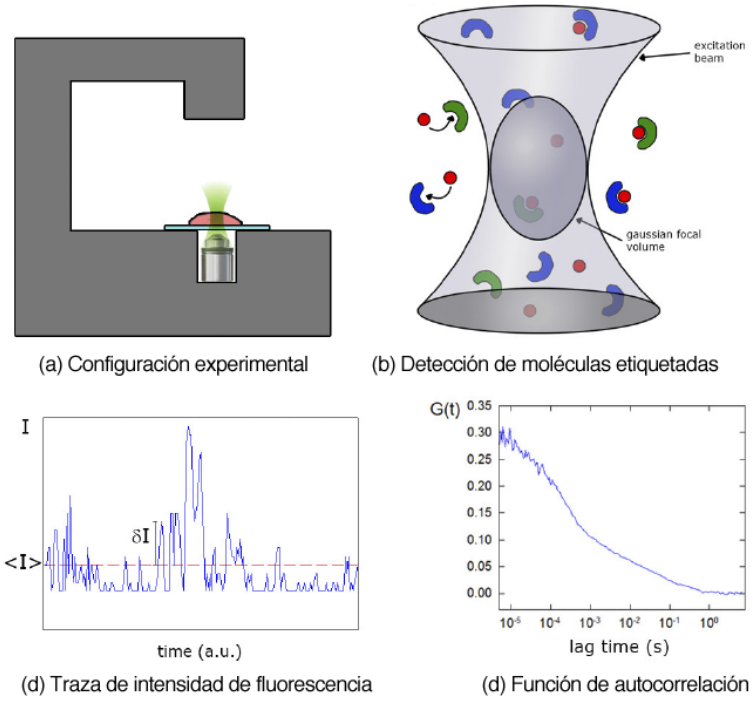


Figura 2.10: Representación del proceso de medición de la espectroscopía de correlación de fluorescencia [7].

2.5. MCell y FERNET: software para FCS

MCell es una herramienta de software que permite la creación de escenarios realistas en 3D para representar modelos celulares y realizar, a través de algoritmos de Monte Carlo, simulaciones del movimiento y reacciones de moléculas dentro y entre células.

MCell es utilizado como motor de simulación del movimiento y comportamiento molecular. La salida de MCell consiste en la posición de cada molécula en cada paso de tiempo de la simulación. Estas posiciones son utilizadas como parámetro de entrada para la ejecución de FERNET. La ejecución de FERNET genera la traza de fluorescencia para cada tiempo de muestreo definido en la configuración del modelo utilizado en la simulación. Luego los resultados son comparados con los datos experimentales y sirven para dar soporte a la validación del modelo biológico propuesto. La figura 2.11 muestra el flujo de trabajo entre las aplicaciones MCell y FERNET.

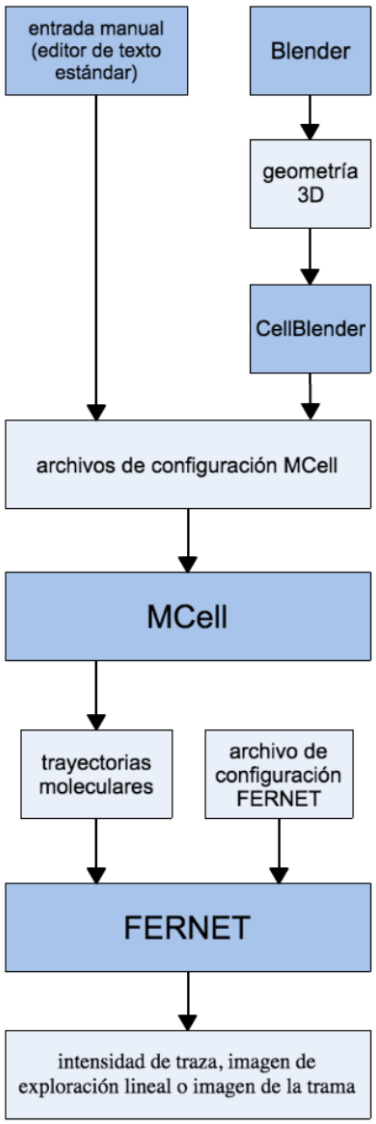


Figura 2.11: Representación esquemática del flujo de trabajo de MCell-FERNET [3].

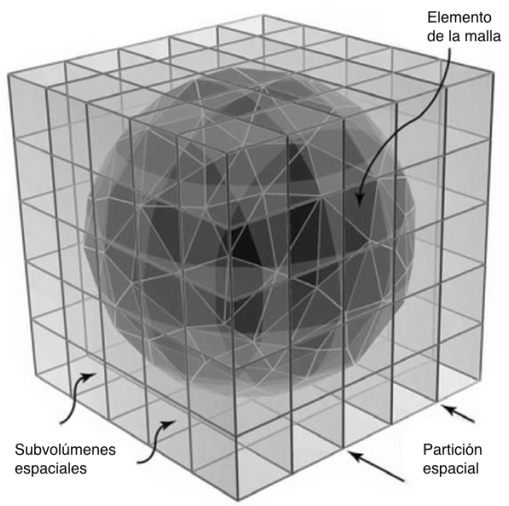
2.5.1. MCell

En MCell se pueden definir mallas triangulares que permiten representar figuras curvas complejas simulando estructuras subcelulares (figura 2.12a). Las mallas están divididas en superficies triangulares, y a su vez, estas superficies se subdividen en pequeños triángulos (llamados baldosas efectoras) que representan moléculas estacionarias. La difusión de moléculas se produce cuando las moléculas saltan de baldosa a baldosa, eventualmente reaccionando con otras moléculas vecinas con una probabilidad configurada. La difusión individual de una molécula es llamada movimiento aleatorio de pasos (*random walk movements*).

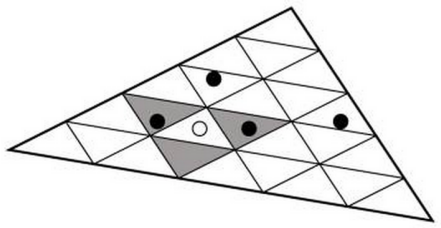
Las simulaciones transcurren en iteraciones espaciadas por un paso de tiempo configurado en el modelo de MCell. Las moléculas o ligandos son liberadas en sitios específicos de la malla (*release sites*) durante la simulación. En cada paso de tiempo las moléculas se mueven difusamente en el espacio, en direcciones y distancias calculadas probabilísticamente (la distancia de difusión máxima puede ser acotada en el modelo). MCell soporta moléculas difusas reactivas en soluciones (moléculas volumétricas) y en membranas (moléculas superficiales). Las moléculas volumétricas

son definidas en el modelo MCell con un nombre y tres coeficientes de difusión dimensionales, y las moléculas superficiales se definen con dos coeficientes dimensionales y se ubican en las baldosas efectoras. Durante una simulación pueden producirse transiciones de estados unimoleculares o bimoleculares mientras los ligandos reaccionan luego de una colisión. Las reacciones también son definidas en el modelo MCell de la simulación.

Para acelerar la búsqueda de intersecciones entre elementos de la malla y moléculas, MCell subdivide el volumen computacional en subvolúmenes espaciales (SSV). La figura 2.12a muestra la representación de una estructura subcelular mediante una malla de elementos triangulares y la subdivisión en subvolúmenes que realiza MCell. La partición en subvolúmenes se ajusta de modo que cada SSV contiene un pequeño número de triángulos y también un pequeño número de moléculas. La búsqueda de colisiones se puede limitar a los SSV, y a sus vecinos cuando la molécula pasa la frontera del SSV. La figura 2.12b representa moléculas superficiales ocupando una baldosa en una subdivisión baricéntrica de un triángulo de la malla. La molécula que se evalúa por el algoritmo de colisiones (circulo abierto), encuentra otras moléculas (círculos negros) en los tres elementos adyacentes (triángulos compartidos).



(a) Malla en MCell [5].



(b) Moléculas en triangulo que compone una malla en MCell [14].

Figura 2.12: Representación de estructuras subcelulares en MCell.

Para simular estocásticamente los eventos complejos que suceden en los ambientes celulares reactivos con un costo computacional razonable, MCell introduce rutinas dinámicas de movimiento browniano para modelar la difusión de moléculas individuales, *ray-tracing* para detectar colisiones entre moléculas, *ray-marching* para propagar rayos después de las colisiones, y probabilidades de Monte Carlo para decidir qué colisiones conducen a eventos de reacción [14]. MCell utiliza un lenguaje descriptivo de modelado (Model Description Language, MDL) como una interfaz de usuario que permite diseñar y controlar las simulaciones.

El desarrollo de MCell es un proyecto colaborativo entre el Departamento de Cómputo y Sistemas Biológicos en la Universidad de Pittsburgh, el Centro de Supercomputo de Pittsburgh, el Laboratorio de Computo de Neurobiología en el Instituto Salk y la Universidad Carnegie Mellon.

2.5.2. FERNET

FERNET (Fluorescence Emission Recipes and Numerical routines Toolkit) es un software desarrollado con el objetivo de asistir en las simulaciones e interpretaciones de experimentos de microscopía confocal. Fue desarrollado y es actualmente mantenido por la Facultad de Ciencias

Exactas y Naturales de la Universidad de Buenos Aires, Argentina, por Juan Francisco Angiolini y Esteban Mocskos [2].

FERNET genera trazas de intensidad de fluorescencia e imágenes a partir de las simulaciones MCell. En cada paso de tiempo de la simulación, FERNET evalúa la posición de cada molécula y calcula la probabilidad de emisión de fotones con la ecuación 2.2. En la ecuación 2.2, ω_{xy} y ω_z corresponden a la cintura radial y axial de la función de dispersión de punto (PSF) y (x_0, y_0, z_0) es el centro del volumen confocal.

$$g(x,y,z) = exp\left(\frac{-2\left((x-x_0)^2+(y-y_0)^2\right)}{\omega_{xy}^2} + \frac{-2(z-z_0)^2}{\omega_z^2}\right) \tag{2.2}$$

Para determinar si la molécula está excitada, FERNET compara el valor $g(x, y, z)$ con un número aleatorio entre 0 y 1, si $g(x, y, z)$ es el mayor, la molécula es considerada para ser promovida a un estado de excitación. Para saber si la molécula excitada emite fotones, FERNET compara el valor aleatorio entre 0 y 1 con q, dónde $q = \varepsilon' \cdot \kappa$, siendo ε' el brillo molecular (cantidad por segundo por molécula) y κ es la permanencia entre los eventos de emisión. La molécula emite fotones si el número aleatorio es mayor a q. Los valores ε' y κ pueden ser configurados por el usuario, sus valores por defecto son 105 cpsm y 200 ns, respectivamente.

Para determinar la cantidad de fotones emitidos por la molécula durante el lapso de tiempo configurado por el usuario (t_s), FERNET repite el procedimiento que determina si emite fotones t_s/κ veces, asumiendo que la posición de la molécula simulada es aproximadamente constante durante t_s. Esta afirmación puede no ser cierta para las moléculas que se mueven rápidamente, por lo tanto, FERNET compara t_s con el tiempo de residencia media para la molécula que se mueve más rápido (τ_D). Si $\tau_D < 10t_s$, el programa sugiere modificar t_s para generar una muestra correcta del sistema simulado.

FERNET repite el procedimiento para calcular la emisión de fotones con todas las moléculas en cada paso de tiempo de la simulación. La traza de intensidad de fluorescencia es calculada sumando los fotones emitidos por las moléculas cada paso de tiempo [3].

2.5.3. Modos de ejecución de MCell+FERNET

FERNET utiliza como datos de entrada la salida de MCell. Para lograr la comunicación entre los dos componentes de software, se utiliza una versión modificada de MCell. La modificación realizada a MCell consiste en introducir dos nuevos modos de ejecución llamados *JOINED* y *PIPED*. La ejecución de MCell en cualquiera de estos modos genera un archivo de salida que contiene el nombre de la molécula y su posición a lo largo del tiempo. Este archivo de salida de MCell es utilizado como entrada al programa FERNET.

En el modo JOINED, el archivo de salida es físicamente creado una vez que MCell finaliza la simulación. Por este motivo, el tamaño del archivo puede alcanzar valores muy grandes (en el orden de gigabytes). Sin embargo, si el usuario no desea analizar las trayectorias moleculares simuladas, puede optar por el modo de ejecución PIPED, que busca reducir el tiempo total de las simulaciones. El modo PIPED no crea un archivo físico con las trayectorias. En su lugar, utiliza un canal de datos (*pipe*) que conecta la salida de MCell con la entrada de FERNET. Si MCell es iniciado en modo PIPED se pausará hasta que FERNET sea iniciado con el pipe correspondiente como entrada. Una vez que MCell y FERNET están conectados, el resultado de la ejecución de simulaciones será la información de fluorescencia.

En el apéndice A se describen los archivos de configuración de MCell y FERNET.

Capítulo 3

Plataforma de desarrollo

La solución desarrollada brinda soporte a la ejecución de simulaciones solicitadas por los usuarios a través de un portal web. Estas solicitudes son almacenadas en una base de datos relacional y posteriormente distribuidas para su ejecución entre un conjunto de procesos denominados Workers, que operan bajo la supervisión y coordinación de un proceso maestro denominado Master-WMS. Estos procesos trabajan bajo un patrón maestro-esclavo y son coordinados a través de Apache ZooKeeper, un software de código abierto que ofrece alta confiabilidad en la coordinación de procesos distribuidos.

Los Workers, el Master-WMS y los diferentes procesos que componen el sistema son ejecutados en máquinas virtuales creadas y gestionadas a través de OpenNebula, una plataforma de código abierto orientada a la creación de clouds privados mediante la virtualización de infraestructura.

Para este proyecto se instaló OpenNebula en uno de los nodos del cluster FING [19] (Universidad de la República) y se trabajó además con la instalación de OpenNebula realizada por la Universidad de Buenos Aires (UBA) sobre el cluster TUPAC [8]. Este capítulo describe el proceso de instalación y configuración de OpenNebula en el cluster FING, la creación de las máquinas virtuales donde se ejecutarán los procesos principales y el mecanismo de integración con otras instalaciones de OpenNebula (en particular con la instalación de la UBA).

3.1. Recursos disponibles

Esta sección describe las características de los recursos de hardware utilizados para el despliegue del sistema.

3.1.1. Cluster FING

La Universidad de la República (UdelaR) cuenta con una infraestructura computacional de alto desempeño montada en sus propias instalaciones, dedicada a proveer soporte para la implementación de proyectos de investigación enfocados en la resolución de problemas complejos. Por más información sobre datos de uso, infraestructura, objetivo y demás sobre el cluster FING ver [19]. Para este proyecto se asignó un equipo de uso compartido denominado node38 (nodo 38 del cluster) que cuenta con las siguientes especificaciones:

- Modelo: HP Proliant DL385 G7
- Procesador: AMD Opteron 6172 2.10GHz
- Número de Procesadores: 2
- Núcleos por Procesador: 12
- Memoria RAM: 72 GB
- Almacenamiento local: 250 GB

En las etapas iniciales del proyecto el equipo node38 se encontraba en la red interna del cluster. Esta ubicación limitaba la conectividad de máquinas externas hacia el nodo, debido a las características del firewall del cluster. Esta conectividad es necesaria para la comunicación entre los procesos remotos que se ejecutan en máquinas virtuales creadas en la UBA con los creados en FING. Las características de la comunicación necesarias para la integración entre ambas instancias se detallan en la sección 3.6.

Para resolver los problemas de conectividad se decidió extraer el nodo de la red e incluirlo en una red DMZ (Demilitarized Zone [23]), que tiene políticas de acceso menos restrictivas. Si bien con esta disposición del nodo en la red se pierde la conectividad con el resto de los recursos del cluster, los cuales eran candidatos a extender la capacidad de cómputo del sistema, la integración entre ambas universidades es un gran punto a favor por esta opción. Esta integración permite una mayor escalabilidad del sistema de cara a la integración con recursos informáticos existentes por fuera del cluster FING.

Se comenzó a trabajar sobre la última versión de OpenNebula (4.6.2) disponible a la fecha de inicio del proyecto (mayo de 2014) para poder investigar y hacer uso de todas las funcionalidades relevantes para la investigación. Al usar la versión 4.6.2 de OpenNebula se asegura la compatibilidad con la instalación de OpenNebula existente en la UBA (también versión 4.6.2). Esta compatibilidad es un requisito para utilizar las funcionalidades avanzadas de federación de data centers que ofrece OpenNebula (este tema se aborda en la sección 3.6). La versión 4.6.2 de OpenNebula requiere como mínimo CentOS versión 6 para su correcto funcionamiento, esto presentó un contratiempo debido a que en el nodo se contaba con la versión de CentOS 5.5. El pedido de actualización se elevó a la URI (Unidad de Recursos Informáticos de la Facultad de Ingeniería) para su ejecución. La actualización fue llevada a cabo posteriormente con éxito.

3.1.2. Cluster TUPAC

Para este proyecto se contó con recursos provistos por la UBA. Se asignaron cuatro nodos del cluster TUPAC, un cluster dirigido a la resolución de modelos de simulación utilizando técnicas de computación de alto rendimiento (High Performance Computing, HPC) ubicado en el Centro de Simulación Computacional para Aplicaciones Tecnológicas.

Los cuatro nodos del cluster TUPAC asignados para este proyecto cuentan, cada uno, con dos procesadores AMD Opteron 6320 de 8 núcleos, funcionando a una frecuencia de 2.8 GHz. Estos nodos son utilizados a través de una instalación de OpenNebula 4.6.2 realizada en la UBA.

3.2. Instalación de OpenNebula en el cluster FING

Esta sección describe el procedimiento de instalación de OpenNebula en el cluster FING y su puesta en funcionamiento.

3.2.1. Modalidades de instalación

Una instalación típica de OpenNebula se realiza utilizando el gestor de paquetes del sistema operativo, instalando un conjunto de paquetes provisto por OpenNebula para cada distribución. Este es el mecanismo de instalación sugerido por OpenNebula, ya que asegura la instalación de la última versión disponible de los paquetes e instala automáticamente las dependencias que sean necesarias, evitando conflictos de versiones. Esta modalidad de instalación es denominada *system wide*. En una instalación system wide, tanto OpenNebula como las dependencias que sean necesarias son instaladas en los directorios de uso general del sistema (e.g. /etc, /var, /usr). Por este motivo, una instalación system wide requiere permisos de administrador.

Debido a que el usuario provisto para este proyecto no cuenta con permisos de administrador en el nodo del cluster FING asignado, se optó por instalar OpenNebula en modalidad *standalone*. Esta modalidad instala OpenNebula bajo un directorio determinado por el usuario, evitando los conflictos de permisos y manteniendo la instalación accesible únicamente por el usuario. Como contrapartida, una instalación en modalidad standalone requiere instalar el producto en forma

manual, instalando cada dependencia en una versión compatible, compilando el código fuente, ejecutando un script de instalación y ajustando las variables de entorno del sistema.

3.2.2. Instalación de dependencias

OpenNebula establece un conjunto de dependencias para su correcto funcionamiento. Estas dependencias se dividen entre paquetes de sistema y gemas de ruby. Todo pedido de instalación de paquetes de sistema debió ser elevado a la URI dado que el usuario de sistema operativo asignado para el proyecto no cuenta con permisos de administrador. Si bien OpenNebula ofrece un script que simplifica la tarea de instalar las gemas, este script requiere permisos de administrador para ejecutarlo. Por lo tanto se optó por instalar las gemas localmente para el usuario del proyecto, realizando individualmente la instalación de cada gema. En los listados 3.1 y 3.2 se detallan los paquetes de sistema y las gemas de ruby instaladas respectivamente.

```
ruby (1.8.7)            sqlite3 (3.6.20)        xmlrpc-c (1.16.24)
openssl (1.0.1)         mysql-devel (5.1.6)     ruby-devel (1.8.7)
curl-devel (7.19.7)     libxml2-devel (2.7.6)   libxslt-devel (1.1.26)
expat-devel (2.0.1)
```

Listado 3.1: Paquetes de sistema instalados.

```
curb (0.8.6)            daemons (1.1.9)         eventmachine (1.0.3)
json (1.8.1)            mini_portile (0.6.1)    mysql (2.9.1)
net-ldap (0.2.2)        nokogiri (1.5.10)       opennebula (4.6.1)
polyglot (0.3.5)        rack (1.5.2)            rack-protection (1.5.3)
rake (10.3.2)           sequel (4.17.0)         sinatra (1.4.5)
sqlite3-ruby (1.2.4)    thin (1.6.3)            tilt (1.4.1)
treetop (1.6.3)         uuidtools (2.1.5)       xml-simple (1.1.4)
```

Listado 3.2: Gemas de ruby instaladas.

3.2.3. Compilación e instalación de OpenNebula

OpenNebula utiliza una base de datos relacional para el almacenamiento de sus estructuras de control y administración de imágenes de máquinas virtuales, templates de maquinas virtuales, usuarios, roles, redes virtuales, etc. La versión utilizada de OpenNebula (4.6.2) soporta dos proveedores de almacenamiento: SQLite y MySQL. El proveedor instalado por defecto es SQLite.

Utilizar SQLite como proveedor de almacenamiento limita ciertas funcionalidades de OpenNebula que solo se pueden activar si se utiliza MySQL como proveedor. Una de estas funcionalidades es la de establecer una federación entre instancias OpenNebula, debido a que OpenNebula requiere características de replicación para implementar esta funcionalidad que solo MySQL le ofrece. Por estas limitaciones, en el marco de este proyecto se optó por utilizar MySQL.

La compilación de OpenNebula se realiza a través del comando `scons mysql=yes` en el directorio que contiene el código fuente (extraído de [22]). El argumento `mysql=yes` indica que se utilizará MySQL como proveedor de base de datos. La instalación de OpenNebula se realiza invocando el script *install.sh*, incluido en el código fuente. Este script, entre otras funcionalidades, permite indicar el directorio donde se instalará OpenNebula.

Cuando se instala OpenNebula en modalidad standalone, es necesario agregar un conjunto de variables de entorno en el archivo *.bashrc* ubicado en el home del usuario. Estas variables se presentan en el listado 3.3.

```
export ONE_LOCATION=~/opennebula
export ONE_AUTH=$ONE_LOCATION/auth/one_auth
export PATH=~/.gem/ruby/1.8/bin:$ONE_LOCATION/bin:~/bin:$PATH
export LD_LIBRARY_PATH=~/libs/
export ONE_XMLRPC=http://localhost:2634/RPC2
```

Listado 3.3: Variables de entorno de OpenNebula

3.2.4. Configuración de KVM

OpenNebula utiliza la interfaz de virtualización libvirt [15] para interactuar con el hipervisor de máquinas virtuales KVM [10], por lo que es necesario realizar ciertas configuraciones en el host donde se instala OpenNebula para poner en funcionamiento el driver KVM.

En primer lugar, se debe modificar la configuración de qemu para que no cambie los permisos de propiedad de los archivos de las imágenes de discos de las maquinas virtuales, incluyendo `dynamic\_ownership = 0` en su archivo de configuración (*/etc/libvirt/qemu.conf*). Además, es necesario modificar el usuario y grupo bajo el cual libvirt va a ejecutar. Las modificaciones realizadas en el archivo de configuración de qemu para la instalación realizada en este proyecto se indican en el listado 3.4.

```
user = "pgccomp"
group = "kvm"
dynamic_ownership = 0
```

Listado 3.4: Modificaciones realizadas en el archivo de configuración de Qemu.

Para invocar los servicios de KVM y crear máquinas virtuales desde la instalación de OpenNebula realizada, es necesario agregar el usuario de la instalación a los grupos *libvirt* y *kvm*. Además, es necesario definir los permisos de los usuarios del grupo *libvirt*, para el manejo de las maquinas virtuales, en el archivo de configuración de libvirt (*/etc/libvirt/libvirtd.conf*). En el listado 3.5 se describen los comandos ejecutados para agregar el usuario del proyecto, y en el listado 3.6 se muestran las modificaciones realizadas al archivo de configuración de libvirt.

```
sudo groupadd libvirt
sudo usermod -a -G libvirt pgccomp
sudo usermod -a -G kvm pgccomp
```

Listado 3.5: Comandos para crear el grupo libvirt y agregar el usuario pgccomp a los grupos libvirt y kvm.

```
listen_tls = 0
unix_sock_group = "libvirt"
unix_sock_ro_perms = "0777"
unix_sock_rw_perms = "0777"
auth_unix_ro = "none"
auth_unix_rw = "none"
```

Listado 3.6: Modificaciones realizadas en el archivo de configuración de libvirt.

3.2.5. Configuración de la base de datos

Como parte importante dentro de la configuración general de OpenNebula, es necesario crear la base de datos que OpenNebula usará como almacenamiento para sus estructuras de control. A su vez, es necesario también crear el usuario con el cual se conectará OpenNebula y brindarle los permisos necesarios para el uso de esta base de datos. Esto se logra mediante las sentencias SQL presentadas en el listado 3.7.

```
CREATE USER oneadmin@localhost IDENTIFIED BY 'oneadmin';
CREATE DATABASE opennebula;
GRANT ALL PRIVILEGES ON opennebula.* TO 'oneadmin' IDENTIFIED BY 'oneadmin';
```

Listado 3.7: Sentencias SQL para la creación y configuración de la base de datos de OpenNebula.

3.2.6. Configuración e inicio de los servicios de OpenNebula

El archivo de configuración del demonio de OpenNebula se denomina *oned.conf*. Este archivo se encuentra alojado en /etc/one en instalaciones system wide y en $ONE_LOCATION/etc en instalaciones standalone (el formato del archivo *oned.conf* puede ser consultado en `http://docs.opennebula.org/4.6/administration/references/oned_conf.html`).

La interfaz web Sunstone se configura en el archivo *sunstone-server.conf*, ubicado en /etc/one/ en instalaciones *system wide* y en $ONE_LOCATION/etc en instalaciones *standalone* (el formato del archivo *sunstone-server.conf* puede ser consultado en `http://docs.opennebula.org/4.6/administration/sunstone_gui/sunstone.html`).

En el contexto de este proyecto, fue necesario cambiar el puerto de acceso a la API XML-RPC configurado por defecto (2633) ya que se encontraba ocupado. Como consecuencia de este cambio, fue necesario modificar el archivo de configuración de Sunstone. Además se cambió el sistema de monitoreo por defecto, pasando del modelo UDP-PULL al modelo SSH-PULL por simplicidad en la configuración (se explican los dos modelos en la sección 2.2.2). En los listados 3.8 y 3.9 se presentan los cambios realizados a los archivos *oned.conf* y *sunstone-server.conf* respectivamente.

Luego de configurar los dos principales servicios de OpenNebula (el demonio *oned* y la interfaz web *sunstone*) el sistema se encuentra en condiciones de iniciar estos servicios (se muestran los comandos correspondientes en el listado 3.10).

```
# OpenNebula Configuration file
PORT = 2634
#-------------------------------------------------------------------
#  KVM SSH-pull Information Driver Manager Configuration
#    -r number of retries when monitoring a host
#    -t number of threads, i.e. number of hosts monitored at the
#               same time
#-------------------------------------------------------------------
 IM_MAD = [
       name       = "kvm",
       executable = "one_im_ssh",
       arguments  = "-r 3 -t 15 kvm-probes" ]
#-------------------------------------------------------------------
```

Listado 3.8: Cambios realizados en el archivo oned.conf.

```
# OpenNebula sever contact information
:one_xmlrpc: http://localhost:2634/RPC2
```

Listado 3.9: Cambios realizados en el archivo sunstone-server.conf.

```
service opennebula start
service opennebula-sunstone start
```

Listado 3.10: Comandos de inicio para los servicios de OpenNebula.

Una vez iniciado el demonio de OpenNebula, es necesario registrar el nodo físico (host), mediante el comando `onehost create localhost -i kvm -v kvm -n dummy`. Los parámetros del comando `onehost` corresponden a:

- `-i`: driver de información utilizado para monitorear el host.
- `-v`: driver de virtualización para operar sobre las maquinas virtuales.
- `-n`: driver de red para el manejo de las redes virtuales.

La documentación detallada del proceso de registro de un host se puede consultar en `http://docs.opennebula.org/4.6/administration/hosts_and_clusters/host_guide.html`.

3.3. Definición de redes virtuales

OpenNebula permite definir redes virtuales sobre las interfaces de red existentes. Generalmente, se define la red virtual como un subconjunto del espacio de direcciones IP de la red física. Las máquinas virtuales pueden ser conectadas a una o más redes virtuales y OpenNebula se encarga de asignarles las direcciones IP correspondientes.

La definición de redes virtuales se puede hacer desde sunstone o en el front-end con el comando `onevnet create`. Este comando recibe como argumento el template de la red a crear (un archivo de texto conteniendo los parámetros necesarios). En el template se deben indicar ciertos parámetros como la interfaz de red, la dirección del DNS, Gateway, y el tipo de red, que puede ser FIXED (se especifican una a una las direcciones IP que pueden ser asignadas) o RANGED (se especifica un rango de direcciones IP).

Para este proyecto se definieron dos redes virtuales. La primera red (nombrada *fixed-virtual-network*) se definió para conectar las máquinas virtuales que ejecutan servicios centrales como los servidores Zookeeper, el servidor de base de datos y el servidor web. Esta red se definió de tipo FIXED ya que es utilizada por una cantidad fija de máquinas virtuales. La segunda red (nombrada *workers-virtual-network*) se definió para conectar las máquinas virtuales que ejecuten procesos Worker. Esta red se definió de tipo RANGED ya que la cantidad de máquinas virtuales conectadas es variable. Las máquinas virtuales que ejecuten procesos Worker son creadas y eliminadas a demanda según la carga de trabajo del sistema y no reciben conexiones entrantes, por lo tanto, pueden tomar cualquier dirección IP disponible. Una vez que estas maquinas virtuales son eliminadas, las direcciones IP son liberadas y pueden ser tomadas por nuevas máquinas virtuales.

En los listados 3.11 y 3.12 se muestran los templates de creación de ambas redes.

```
NAME    = "fixed-virtual-network"
TYPE    = FIXED
BRIDGE  = virbr0
LEASES  = [IP=192.168.122.2]
LEASES  = [IP=192.168.122.3]
LEASES  = [IP=192.168.122.4]
LEASES  = [IP=192.168.122.5]
LEASES  = [IP=192.168.122.6]
LEASES  = [IP=192.168.122.7]
NETWORK_ADDRESS = 192.168.122.0
NETWORK_MASK    = 255.255.255.0
GATEWAY         = 192.168.122.1
DNS             = 192.168.122.1
```

Listado 3.11: Template de la red virtual fixed-virtual-network.

```
NAME     = "workers-virtual-network"
TYPE     = RANGED
IP_START = 192.168.122.11
IP_END   = 192.168.122.254
BRIDGE   = virbr0
NETWORK_ADDRESS = 192.168.122.0
NETWORK_MASK    = 255.255.255.0
GATEWAY         = 192.168.122.1
DNS             = 192.168.122.1
```

Listado 3.12: Template de la red virtual workers-virtual-network.

Las redes virtuales se definieron sobre la interfaz virbr0, creada por KVM para conectar las máquinas virtuales utilizando NAT. Por lo tanto, las máquinas virtuales creadas toman direcciones IP en una subred del host (la 192.168.122.x) y no son accesibles directamente desde fuera del host. Esta limitación es aceptable en el escenario del proyecto ya que se cuenta con un único host en FING, y sólo una cantidad fija de máquinas virtuales reciben conexiones entrantes; estas máquinas virtuales son accedidas desde fuera del host utilizando redirección de puertos. Las redirecciones definidas en el proyecto se detallan en la sección 3.6.

En una instalación típica de OpenNebula, donde se cuente con varios hosts, las redes virtual pueden ser definidas sobre un bridge. De esta forma, las máquinas virtuales instanciadas serán conectadas a la misma red de los hosts, pudiendo comunicarse entre ellas y ser accedidas directamente desde cualquier otro host de la red.

3.4. Preparación de imágenes y contextualización

Los diferentes componentes que conforman la solución (procesos Master-WMS y Worker, servidores Zookeeper, servidor web, base de datos relacional, etc.) son ejecutados en máquinas virtuales gestionadas por OpenNebula. Estas máquinas virtuales son instanciadas en base a imágenes de disco que contienen el sistema operativo y el software necesario pre-instalados, de acuerdo a los procesos que ejecutarán.

Esta sección describe el proceso de creación de las imágenes necesarias y el mecanismo de contextualización de máquinas virtuales, que facilita la asignación automática de direcciones IP y el pasaje de parámetros a cada máquina virtual creada.

3.4.1. Creación de imágenes de disco

Las imágenes de disco fueron creadas a partir de una imagen de base con sistema operativo CentOS 7, una distribución de Linux mantenida por la comunidad, derivada de los paquetes liberados al público por Red Hat de su distribución comercial RHEL (Red Hat Enterprise Linux). Esta imagen de base es proporcionada por CentOS y contiene pre-instalado el paquete cloud-init [6], compuesto por un conjunto de scripts y herramientas para la contextualización de máquinas virtuales en entornos cloud. El paquete cloud-init es compatible con diferentes distribuciones de Linux y plataformas de virtualización (Amazon EC2, OpenNebula, OpenStack, CloudStack, entre otras). Cloud-init se encarga de configurar las interfaces de red y claves ssh permitidas de la máquina virtual de acuerdo a los metadatos cargados por OpenNebula en el archivo *context.sh* del CD-ROM de contexto. El CD-ROM de contexto es una unidad de CD-ROM creada por OpenNebula en cada máquina virtual. En esta unidad se incluyen metadatos y archivos para parametrizar la máquina virtual (el contenido del CD-ROM de contexto y el proceso de contextualización de máquinas virtuales se detalla en la sección 2.2.3).

En OpenNebula, las imágenes pueden ser creadas desde sunstone o bien con el comando `oneimage`. Este comando recibe un archivo con los datos necesarios para la creación de la imagen (template) donde se pueden especificar entre otros parámetros: el nombre, el tipo de imagen (por ejemplo, el tipo de imagen "OS" indica que se está creando una imagen de Sistema Operativo) y la ubicación (puede ser un link o una ruta en el filesystem). En el listado 3.13 se presenta un ejemplo de template de una imagen de disco, cuyo comando de creación corresponde a `oneimage create image-template.one --datastore default`.

```
NAME   = "centos7-cloud-image"
PATH   = "http://cloud.centos.org/.../CentOS7-GenericCloud.qcow2"
TYPE   = OS
DRIVER = qcow2
```

Listado 3.13: Template de ejemplo de una imagen de disco (archivo image-template.one).

Además del template, para crear una imagen se debe indicar el nombre o identificador del datastore donde residirá (mediante el parámetro `--datastore`). La imagen será copiada desde de la ubicación especificada en el template al datastore indicado. En caso de que la ubicación especificada en el template sea una ruta en el filesystem, la misma no puede estar contenida los directorios `$ONE_LOCATION/var/`, `$ONE_LOCATION/etc/` o `$HOME`. Esta restricción es impuesta por OpenNebula para evitar la copia al datastore de datos sensibles como ser certificados, claves ssh, la propia base de datos de OpenNebula, etc. Se pueden crear excepciones agregando el parámetro `SAFE_DIRS` a la configuración del datastore. En el caso de este proyecto se agregó el valor `SAFE_DIRS = "/export/home10/pgccomp/images"` mediante el comando `onedatastore update default`.

Una vez creada la imagen de base, se creó una copia de ésta para cada tipo de máquina virtual. Sobre estas copias se instaló el software adicional requerido en cada caso (por ejemplo, en la imagen para las máquinas virtuales que ejecuten procesos Worker se instaló MCell y FERNET). Para poder instalar software sobre una imagen, la imagen debe establecerse como persistente (utilizando el comando `oneimage persistent`), se debe instanciar una máquina virtual a partir de la imagen e instalar en ese máquina virtual el software necesario. Una imagen en modo persistente puede ser utilizada por una sola máquina virtual al mismo tiempo, y todo cambio que se realice dentro de la máquina virtual es persistido en la imagen cuando la máquina virtual se apaga.

En el listado 3.14 se muestran los comandos para clonar una imagen y establecer la copia como persistente.

```
oneimage clone centos7-cloud-image worker-image
oneimage persistent worker-image
```

Listado 3.14: Comandos para la clonación de una imagen de disco y establecimiento de la nueva imagen como persistente.

Para instanciar una máquina virtual se debe crear un template indicando la imagen que se utilizará y la red virtual a la cual conectarse (se indica una de las redes virtuales creadas en la sección 3.3). En el template creado se debe incluir la sección `CONTEXT` con el valor `NETWORK=YES` y la lista de claves públicas que podrán acceder a la máquina virtual vía ssh. La información de red (IP asignada, Gateway, etc.) y claves ssh, será colocada por OpenNebula en el archivo *context.sh* del CD-ROM de contexto, utilizado por cloud-init para configurar la máquina virtual durante el arranque. En el listado 3.15 se muestra un template de ejemplo para la creación de máquinas virtuales que ejecuten procesos Worker, y en el listado 3.16 se muestran los comandos de creación del template y de una máquina virtual a partir de ese template.

```
CONTEXT=[
  NETWORK="YES",
  SSH_PUBLIC_KEY="$USER[SSH_PUBLIC_KEY]"]
CPU="1.0"
DISK=[
  IMAGE="worker-image" ]
GRAPHICS=[
  LISTEN="0.0.0.0",
  TYPE="vnc" ]
MEMORY="1024"
NIC=[
  NETWORK="workers-virtual-network" ]
OS=[
  ARCH="x86_64" ]
VCPU="1"
```

Listado 3.15: Template de ejemplo para la creación de máquinas virtuales que ejecuten procesos Workers.

```
onetemplate create worker-template.one --name worker-template
onetemplate instantiate worker-template --name worker-vm
```

Listado 3.16: Comandos de creación de un template (definido en un archivo de nombre worker-template.one) y de una máquina virtual a partir de ese template.

Una vez creada la máquina virtual, se puede acceder vía ssh con usuario centos a la IP que le haya sido asignada por OpenNebula. La IP asignada puede verse ejecutando el comando `onevm show worker-vm`. Dentro de la máquina virtual se procede a instalar el software y realizar los cambios necesarios. Cuando la máquina virtual se apague (ejecutando el comando `onevm shutdown worker-vm`) los cambios realizados serán persistidos en la imagen. Luego de apagar la máquina virtual, se debe pasar la imagen a modo no persistente (ejecutando el comando `oneimage nonpersistent worker-image`) para que pueda comenzar a ser utilizada por múltiples máquinas virtuales.

Sobre todas las imágenes se instaló openjdk 1.8. Sobre las imágenes para servidores Zookeeper, servidor de base de datos y servidor web se instalaron además Zookeeper 3.4.6, MySQL 5.6.23 y JBoss wildfly 8.2.0 respectivamente. Sobre la imagen para Workers se instalaron MCell, FERNET, sus dependencias y un conjunto de scripts de contextualización provistos por OpenNebula, adaptados para que puedan funcionar en CentOS 7 y en conjunto con cloud-init, permitiendo la parametrización de las máquinas virtuales y los Workers que en ellas se ejecuten.

3.4.2. Instalación de MCell y FERNET

En la imagen de disco de las máquinas virtuales que ejecutan procesos Worker se instalaron MCell y FERNET. Para la compilación e instalación de MCell y FERNET se instalaron los paquetes que se muestran en el listado 3.17.

```
autoconf (2.69)          automake (1.13.4)        gcc-c++ (4.8.3)
bison (2.7)              flex (2.5.37)            byacc (1.9)
argtable (2.13)          libconfig (1.4.9)        libtiff (4.0.9)
```

Listado 3.17: Paquetes instalados para la compilacion e instalacion de MCell y FERNET.

Luego de instalar los paquetes mencionados, se compilaron a partir del código fuente: MCell en su versión 3.2.1 (revisión no oficial, adaptada para ejecutar en conjunto con FERNET) y FERNET en su versión 1.0a. Por último, se exportó la variable de entorno LD_LIBRARY_PATH utilizada por FERNET y se agregó al archivo /root/.bashrc para que sea exportada automáticamente en cada sesión (ver listado 3.18).

```
export LD_LIBRARY_PATH=/usr/local/lib
echo "export LD_LIBRARY_PATH=/usr/local/lib" >> /root/.bashrc
```

Listado 3.18: Variable de entorno LD_LIBRARY_PATH, utilizada por FERNET.

3.4.3. Instalación de scripts de contextualización

La contextualización juega un papel importante en las máquinas virtuales creadas para ejecutar Workers, ya que estas máquinas virtuales son creadas a demanda por el proceso Master-WMS, y en su CD-ROM de contexto se incluye: los metadatos cargados por OpenNebula (interfaces de red, claves ssh, etc.), los metadatos cargados por el Master-WMS (por ejemplo cantidad de Workers a ejecutar y parámetros específicos para cada uno), un conjunto de scripts encargados de iniciar los Workers en función de esta metadata, y el código que deben ejecutar (archivo JAR).

OpenNebula provee un paquete con scripts de contextualización que se instalan como servicio del sistema operativo en las imágenes y se encargan de configurar la máquina virtual en función del contenido del CD-ROM de contexto. La instalación del paquete mencionado genera un script principal en /etc/init.d (vmcontext) configurado como servicio (que se ejecuta durante el arranque de la máquina virtual, luego de iniciados los servicios de red) y un conjunto de scripts en /etc/one-context.d que son invocados por este servicio, encargados de realizar la configuración general de la máquina virtual (interfaces de red, calves ssh, hostname, DNS, etc.) y de invocar otros scripts de inicialización que estén incluidos en el CD-ROM de contexto. Específicamente el servicio vmcontext se encarga de:

1. Montar el CD-ROM de contexto en un directorio temporal.
2. Exportar como variables el contenido del archivo context.sh (generado por OpenNebula con los metadatos indicados en la sección CONTEXT del template de la máquina virtual). Esto permite que estas variables sean accesibles para el resto de los scripts que se ejecuten a continuación.
3. Ejecutar en orden alfabético los scripts ubicados en el directorio /etc/one-context.d.
4. Desmontar el CD-ROM de contexto.

Las tareas de configuración general realizadas por la mayoría de los scripts presentes en /etc/one-context.d ya son realizadas por cloud-init, por lo tanto el único archivo que se dejó en este directorio fue 99-execute-scripts.sh, que se encarga de invocar determinados scripts presentes en el CD-ROM de contexto, de acuerdo al siguiente criterio:

- Si se incluyó el parámetro INIT_SCRIPTS en la sección CONTEXT del template, se ejecutarán los scripts que este parámetro indique. El parámetro INIT_SCRIPTS debe contener una lista con las rutas de los scripts a ejecutar (relativas a la raíz del CD-ROM de contexto) separadas por espacio. Por ejemplo: INIT_SCRIPTS=tasks/copy-files.sh server/start.sh
- Si no se incluyó el parámetro INIT_SCRIPTS pero existe un archivo de nombre init.sh en la raíz del CD-ROM de contexto, se ejecutará ese archivo.

El script vmcontext es instalado por defecto en /etc/init.d y por lo tanto ejecutado como servicio por el sistema tradicional de gestión de servicios de Linux. En lugar de mantener este comportamiento por defecto, vmcontext se re-instaló como servicio del sistema operativo utilizando systemd [12], el nuevo sistema de gestión de servicios incluido en CentOS 7, que ofrece características avanzadas y facilidades como el registro automático de logs, el trackeo de los servicios iniciados y de sus procesos hijos, entre muchas otras.

Para re-instalar vmcontext como servicio gestionado por systemd, se movió el archivo vmcontext al directorio /usr/bin/, se creó el archivo /etc/systemd/system/vmcontext.service que representa al servicio en systemd (su contenido puede verse en el listado 3.19), y se habilitó su ejecución automática durante el arranque con el comando `sudo systemctl enable vmcontext`.

```
[Unit]
Description=OpenNebula Contextualization service
After=network.target

[Service]
Type=simple
User=root
ExecStart=/usr/bin/vmcontext start
RemainAfterExit=1

[Install]
WantedBy=multi-user.target
```

Listado 3.19: Contenido del archivo vmcontext.service.

El script 99-execute-scripts.sh fue modificado para que soporte la invocación a scripts ubicados en sub-directorios del CD-ROM de contexto, ya que la versión original de 99-execute-scripts.sh provista por OpenNebula requiere que los scripts a invocar estén ubicados en la raíz. Esta mejora es importante dado que los scripts de inicialización y el archivo JAR de Workers son incluidos por OpenNebula en un sub-directorio del CD-ROM de contexto. Las modificaciones realizadas a 99-execute-scripts.sh se detallan en el listado 3.20.

```
#Se determinan los scripts a ejecutar (init.sh o $INIT_SCRIPTS)
if [ -z "$INIT_SCRIPTS" ]; then
    if [ -f "$MOUNT_DIR/init.sh" ]; then
        INIT_SCRIPTS=init.sh
    fi
fi
...
#Se copian los script a ejecutar en un directorio temporal
#y se les da permisos de ejecucion
cd $MOUNT_DIR
for f in $INIT_SCRIPTS; do
    cp $f $TMP_DIR
    chmod +x $TMP_DIR/$(basename $f)
done
```

```
#Se invocan los scripts del directorio temporal
cd $TMP_DIR
for script in $TMP_DIR/*; do
    $script
done
```

Listado 3.20: Cambios realizados al script 99-execute-scripts.sh.

En síntesis, sobre la imagen para Workers se realizaron las siguientes modificaciones:

1. Instalación de MCell, FERNET y dependencias.
2. Instalación de vmcontext como servicio gestionado por systemd.
3. Versión modificada de 99-execute-scripts.sh en /etc/one-context.d.

Con las modificaciones realizadas la imagen queda preparada para que cualquier máquina virtual instanciada pueda auto-configurarse durante el arranque (tarea resuelta por cloud-init) en base a los metadatos cargados por OpenNebula en el archivo context.sh, e iniciar los Workers en base a los archivos y scripts cargados por el Master-WMS en el CD-ROM de contexto, invocados por 99-execute-scripts.sh.

3.4.4. Creación de imagen de contexto

Los archivos que se quieran incluir en el CD-ROM de contexto de las máquinas virtuales pueden ser empaquetados en una imagen de tipo CONTEXT. Las imágenes de tipo CONTEXT deben crearse a partir de un template que indique el nombre para la imagen y la ruta a un archivo comprimido que contenga los archivos que se quieren incluir (se muestra un template de ejemplo en el listado 3.21). El comando de creación de una imagen de tipo CONTEXT es `oneimage create context-files-image.one datastore files`, siendo context-files-image.one el template. El parámetro `--datastore` debe indicar el nombre o identificador de un datastore de tipo FILES (las imágenes de tipo CONTEXT solo pueden residir en datastores de tipo FILES). OpenNebula creará la imagen y la almacenará en el datastore indicado.

```
NAME  = "context-files"
TYPE  = CONTEXT
PATH  = /export/home10/pgccomp/images/context-files.tar.gz
```

Listado 3.21: Template para la creación de una imagen de tipo CONTEXT.

El uso de imágenes de tipo CONTEXT evita tener que incluir rutas relativas al front-end en los templates de creación de máquinas virtuales. Basta con indicar el identificador de una imagen de tipo CONTEXT en un template de creación de máquinas virtuales para que el contenido de la imagen sea incluido en el CD-ROM de contexto de las máquinas virtuales que se creen a partir del template (se detalla y se presenta un ejemplo en la sección 3.5).

El archivo JAR y los scripts de inicialización de Workers fueron empaquetados en una imagen de tipo CONTEXT para que sean incluidos en el CD-ROM de contexto de todas las máquinas virtuales que se creen para ejecutar Workers. En esta imagen se incluyeron los siguientes archivos:

- worker.jar: Archivo JAR con el código del proceso Worker.
- init-scripts/00-init-variables.sh: inicializa variables a ser utilizadas en los siguientes scripts.
- init-scripts/01-copy-context-files.sh: copia el contenido del CD-ROM de contexto a un directorio local.
- init-scripts/02-wait-for-network.sh: espera a que la red esté pronta, chequeando periódicamente que la IP asignada por OpenNebula sea accesible mediante ping, hasta recibir confirmación. Esto es necesario ya que el servicio de red es reiniciado por cloud-init luego de configurar las interfaces de red, y por lo tanto pueden pasar unos segundos hasta que quede operativo. Hasta ese momento no se pueden iniciar los Workers, ya que harán uso de la red.

- init-scripts/03-start-workers.sh: inicia los procesos Worker, invocando para cada uno el archivo worker.jar con los parámetros correspondientes. La cantidad de Workers a ejecutar y los parámetros para cada uno son visibles desde el script 03-start-workers.sh. Esta visibilidad se debe a que los parámetros para cada Worker fueron incluidos en la sección CONTEXT del template de creación de la máquina virtual, y por lo tanto incluidos como variables en el archivo context.sh, exportado por el servicio de contextualización vmcontext. El template de creación de la máquina virtual es generado por el proceso Master-WMS (se muestra un ejemplo en la sección 3.5).
- init-scripts/04-start-memory-control.sh: dispara un proceso (memory-control.sh) encargado de liberar periódicamente la memoria cache. Esto es necesario ya que durante el análisis experimental se detectó que para algunas simulaciones MCell hace uso intensivo de la cache, provocando el uso de la memoria swap. El uso del swap suele degradar significativamente el rendimiento de clientes y servidores Zookeeper, causando la pérdida de conexión y en efecto provocando la caída de los Workers que estén ejecutando en la máquina virtual.
- memory-control.sh: libera periódicamente la memoria cache, reduciendo la cantidad de memoria utilizada del sistema y evitando así el uso del swap. En CentOS, se logra con el comando `sync; echo 3 > /proc/sys/vm/drop_caches`.
- run-task.sh: ejecuta MCell y FERNET. Este script es invocado desde los Workers para la ejecución de una simulación puntual. Recibe por parámetro la ruta a los archivos que definen la simulación (.mdl para MCell y .cfg para FERNET) y el modo de ejecución de FERNET. En lugar de ejecutar MCell y luego FERNET, la salida de MCell es enviada a un pipe, que es usado a su vez como entrada para FERNET. De esta forma se ejecutan los dos programas en conjunto evitando la creación de archivos intermedios de gran tamaño (del orden de Gigabytes).

3.5. Creación y configuración inicial de máquinas virtuales

Una vez creadas las imágenes de disco, se instanciaron tres máquinas virtuales para ejecutar servidores Zookeeper, una para la base de datos, una para el portal web y una para el Master-WMS. Estas máquinas virtuales se conectaron a la red virtual *fixed-virtual-network* creada en la sección 3.3, tomando direcciones IP en el rango 192.168.122.[2-7]. En los templates se indica la red virtual y la imagen sobre la cual se instanciarán las máquinas virtuales. En el listado 3.22 se muestra el template utilizado para la creación de las máquinas virtuales que ejecutarán los tres servidores ZooKeeper. En el listado 3.23 se muestran los comandos de creación del template y de las tres máquinas virtuales.

```
CONTEXT=[
  NETWORK="YES",
  SSH_PUBLIC_KEY="$USER[SSH_PUBLIC_KEY]"]
CPU="1.0"
DISK=[
  IMAGE="zookeeper-image" ]
GRAPHICS=[
  LISTEN="0.0.0.0",
  TYPE="vnc" ]
MEMORY="2048"
NIC=[
  NETWORK="fixed-virtual-network" ]
OS=[
  ARCH="x86_64" ]
VCPU="1"
```

Listado 3.22: Template de ejemplo para los servidores ZooKeeper (archivo zk-template.one).

```
#Creacion del template
onetemplate create zk-template.one --name zk-template
#Creacion de las maquinas virtuales
onetemplate instantiate zk-template --name zk1
onetemplate instantiate zk-template --name zk2
onetemplate instantiate zk-template --name zk3
```

Listado 3.23: Comandos de creación del template y máquina virtuales para los servidores Zookeeper.

Previo a levantar los servidores Zookeeper, se debe realizar la configuración inicial en estas máquinas virtuales para que funcionen como ensemble. Deben setearse los parámetros descritos en el listado 3.24 en el archivo de configuración ubicado en zookeeper-3.4.6/conf/zoo.cfg.

```
#Unidad basica de medida de tiempo (ms)
tickTime = 3000
#Directorio donde se almacenaran los datos
dataDir = /var/lib/zookeeper
#Puerto de conexion para clientes
clientPort = 2181
#Timeout (en ticks) para la conexion inicial entre un servidor "follower"
#y el servidor "leader"
initLimit = 5
#Timeout (en ticks) para la conexiones posteriores (de sincronizacion)
#entre un servidor "follower" y el servidor "leader"
syncLimit = 2
#Direcciones de cada servidor del ensemble
#(los puertos 2888 y 3888 se utilizan para la comunicacion entre
#los servidores)
server.1 = 192.168.122.3:2888:3888
server.2 = 192.168.122.4:2888:3888
server.3 = 192.168.122.5:2888:3888
```

Listado 3.24: Archivo de configuración de los servidores Zookeeper (zookeeper-3.4.6/conf/zoo.cfg).

En cada servidor del ensemble debe crearse un archivo de nombre myid bajo el directorio configurado como dataDir, conteniendo únicamente un número que lo identifique en el ensemble (en concordancia con las propiedades server.X definidas anteriormente).

Por último, en cada máquina virtual se deben abrir los puertos para la comunicación entre servidores (2888 y 3888) y el puerto 2181 para aceptar conexiones de los clientes (se muestran los comandos utilizados en el listado 3.25). Luego se puede iniciar el servicio a través del comando `zookeeper-3.4.6/bin/zkServer.sh start`, y una vez iniciado en las tres máquinas virtuales, los servidores se comunicarán para conformar el ensemble. Las aplicaciones cliente deberán conectarse al ensemble indicando la dirección y puerto de los tres servidores. Zookeeper seleccionará un servidor al azar que será el encargado de responder a las peticiones de ese cliente, y si en algún momento este servidor fallara el cliente será re-conectado automáticamente con otro servidor del ensemble.

```
iptables -I INPUT -p tcp --dport 2181 --syn -j ACCEPT
iptables -I INPUT -p tcp --dport 2888 --syn -j ACCEPT
iptables -I INPUT -p tcp --dport 3888 --syn -j ACCEPT
```

Listado 3.25: Comandos para la apertura de los puertos utilizados por ZooKeeper.

De manera análoga, se crearon las máquinas virtuales para la base de datos, el portal web y el proceso Master-WMS. Fue necesario abrir los puertos 3306 para MySQL y 8080 para JBoss. En la tabla 3.1 se muestran las direcciones IP asignadas a cada máquina virtual y los servicios instanciados.

Máquina virtual	IP asignada	Servicio	Puerto
Servidor de Base de Datos	192.168.122.2	MySQL	3306
Servidor Zookeeper 1	192.168.122.3	Zookeeper	2181
Servidor Zookeeper 2	192.168.122.4	Zookeeper	2181
Servidor Zookeeper 3	192.168.122.5	Zookeeper	2181
Servidor Web	192.168.122.6	JBoss	8080
Master-WMS	192.168.122.7	Master-WMS	-

Tabla 3.1: Máquinas virtuales y servicios instanciados

Las máquinas virtuales que ejecutan Workers son creadas a demanda por el proceso Master-WMS en base a los recursos disponibles y la carga de trabajo dada (este proceso se detalla en 4.2.3). Para crear estas máquinas virtuales, el Master-WMS genera los templates en tiempo de ejecución y las instancia a través de la API para Java ofrecida por OpenNebula, que encapsula el acceso a los servicios web expuestos por el front-end (API XLM-RPC). En estos templates, además de la información básica como la cantidad de memoria y CPU, la imagen de disco, las redes virtuales, claves ssh, etc. se especifican bajo la sección CONTEXT:

- Imagen de contexto (creada en la sección 3.4.4, contiene el archivo worker.jar y los scripts de inicialización). Se indica el identificador de esta imagen en el atributo FILES_DS, para que su contenido sea incluido por OpenNebula en un directorio de nombre context-files (nombre que se le dió a la imagen) bajo la raíz del CD-ROM de contexto.
- Ruta a los scripts de inicialización incluidos en la imagen de contexto. Se indica la ruta en el atributo INIT_SCRIPTS, para que los scripts sean invocados durante el arranque por los scripts de contextualización instalados en la máquina virtual (específicamente por 99-execute-scripts.sh, como se detalla en la sección 3.4.3).
- Cantidad de Workers a ejecutar. Se indica en el parámetro WORKERINSTANCES. Este valor es utilizado por el script de inicialización 03-start-workers.sh, encargado de iniciar los procesos Worker invocando a worker.jar.
- Parámetros específicos para cada uno de los Workers a ejecutar. Se indican en los parámetros WORKERDATA{i}, con i entre 1 y WORKERINSTANCES. Estos valores son utilizados por 03-start-workers.sh en las invocaciones a worker.jar.
- Identificador de la maquina virtual asignado por OpenNebula e identificador de la zona donde se esta creando el Worker (los posibles valores son “fing” o “uba”). Se indican en los atributos VMID y ZONE respectivamente. Si bien no se ha creado la máquina virtual aún y por lo tanto no se conoce el identificador que le será asignado, se puede incluir el string “$VMID” y OpenNebula lo reemplazará por el valor adecuado. Estos valores son utilizados por 03-start-workers.sh en las invocaciones a worker.jar

En el listado 3.26 se muestra un ejemplo de template generado por el Master-WMS. Como producto de ese template, el CD-ROM de contexto de la máquina virtual instanciada contendrá la estructura de archivos que se muestra en la figura 3.1. El archivo context.sh será generado por OpenNebula con las variables que se muestran en el listado 3.27.

```
NAME      = fing-workervm
CPU       = 2.0
VCPU      = 2
MEMORY    = 1024
DISK      = [IMAGE="worker-image",
             READONLY="no",
             DRIVER="qcow2"]
NIC       = [NETWORK = "workers-virtual-network",
             NETWORK_UNAME="oneadmin"]
GRAPHICS  = [TYPE = "vnc",
             LISTEN  = "0.0.0.0"]
CONTEXT   = [NETWORK = "yes",
             SSH_PUBLIC_KEY = "$USER[SSH_PUBLIC_KEY]",

             FILES_DS="$FILE[IMAGE=\"context-files\"]",
             INIT_SCRIPTS="context-files/init-scripts/*",

             WORKERINSTANCES="2",
             WORKERDATA1="-1389945360@fing@37@2099@100.0",
             WORKERDATA2="-1523593202@fing@37@2099@100.0",

             ZONE="fing",
             VMID="$VMID"]
```

Listado 3.26: Template de creación para una máquina virtual que ejecute dos procesos Worker.

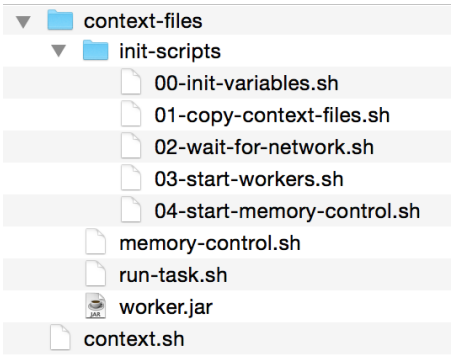


Figura 3.1: Estructura del CD-ROM de contexto generado para una máquina virtual que ejecute procesos Worker.

```
DISK_ID='1'
TARGET='hdb'
NETWORK='yes'
ETH0_DNS='192.168.122.1'
ETH0_GATEWAY='192.168.122.1'
ETH0_IP='192.168.122.204'
ETH0_MAC='02:00:c0:a8:7a:cc'
ETH0_MASK='255.255.255.0'
ETH0_NETWORK='192.168.122.0'
SSH_PUBLIC_KEY='ssh-dss AAAAB3N.. NZWg== pgccomp@cluster.fing.edu.uy

FILES_DS='/export/home10/pgccomp/opennebula/var//datastores/2/
53394e3562e894423bee651bb83d1974:'\''context-files'\'' '
INIT_SCRIPTS='context-files/init-scripts/*'
WORKERINSTANCES='2'
WORKERDATA1='-1389945360@fing@37@2099@100.0'
WORKERDATA2='-1523593202@fing@37@2099@100.0'

ZONE='fing'
VMID='907'
```

Listado 3.27: Archivo context.sh generado por OpenNebula.

En conclusión, para la creación de Workers, el Master-WMS genera los templates e invoca los servicios expuestos por el front-end para crear las máquinas virtuales, pasando estos templates como parámetro. OpenNebula crea las máquinas virtuales en base a la imagen de disco indicada, crea el CD-ROM de contexto para cada una y las instancia. Durante el arranque de cada máquina virtual se ejecutan los servicios de cloud-init (instalados en la imagen de disco), que leen el archivo context.sh y configuran la red en consecuencia. A continuación se ejecuta el servicio de contextualización de OpenNebula (vmcontext), que invocará los scripts de inicialización de Workers incluidos en el directorio init-scripts del CD-ROM de contexto. Una vez iniciados los procesos Worker, estos se conectarán al ensemble de servidores Zookeeper para tomar las tareas que le hayan sido asignadas y comenzar a ejecutar Mcell y FERNET, invocando al script runtaks.sh. El flujo completo de un proceso Worker se detalla en la sección 4.2.5.

3.6. Integración de instancias OpenNebula

OpenNebula cuenta con un mecanismo para la integración completa de instancias (i.e. instalaciones independientes), conformando una federación de centros de datos (data centers) distribuidos (este mecanismo se presenta en la sección 2.2.6). Se trata de una integración a bajo nivel y altamente acoplada que permite que el conjunto total de recursos, usuarios, grupos, roles, etc. pueda ser visto y administrado desde cualquiera de las instancias, dejando la lógica de comunicación entre instancias en manos de OpenNebula. Para establecer una federación se requiere configurar el demonio y la base de datos de OpenNebula en cada instancia, ya que la comunicación entre instancias se lleva a cabo por dos vías: 1) replicación de determinadas tablas de la base de datos y 2) invocaciones punto a punto entre demonios, utilizando los puntos de acceso (endpoints) a los servicios expuestos por cada uno (estos servicios se presentan en 2.2.4).

El sistema (específicamente el proceso Master-WMS) debe interactuar con OpenNebula para crear máquinas virtuales que ejecuten la carga de trabajo dada. Como se cuenta con acceso a dos instancias de OpenNebula (la primera instalada en FING como parte de este proyecto y la segunda instalada en la UBA) se manejó la posibilidad de conformar una federación entre ambas instancias. De esta forma, el proceso Master-WMS (que se ejecuta en FING) se comunicaría únicamente con la instancia de OpenNebula local para crear máquinas virtuales tanto en FING como en la UBA. Conformar esta federación simplificaría la lógica del Master-WMS, pero implica una integración a bajo nivel que además de los recursos integraría las instancias en un nivel administrativo que no resulta de interés en el marco de este proyecto. Además, conformar una federación implica un proceso de configuración no trivial en cada instancia, que requiere violar su autonomía y hace que la incorporación y desvinculación de instancias resulte poco flexible.

Tendiendo en cuenta los aspectos mencionados, se optó por realizar una implementación propia que resolviera la integración entre instancias limitándose a compartir recursos y haciendo uso únicamente de los servicios expuestos por cada una. De esta forma se facilita el proceso de configuración e incorporación de nuevas instancias, dado que la lógica de integración reside únicamente en el proceso Master-WMS. Los endpoints de cada instancia deben ser configurados en la base de datos central de la aplicación (la estructura de esta base de datos se describe en la sección 4.2.2) y el Master-WMS se encarga de realizar las invocaciones correspondientes a cada uno para obtener información acerca de los recursos disponibles e instanciar luego las máquinas virtuales necesarias. En la figura 3.2 se muestra de manera esquemática la integración de instancias mediante una federación en comparación al mecanismo de integración implementado, resuelto por el Master-WMS.

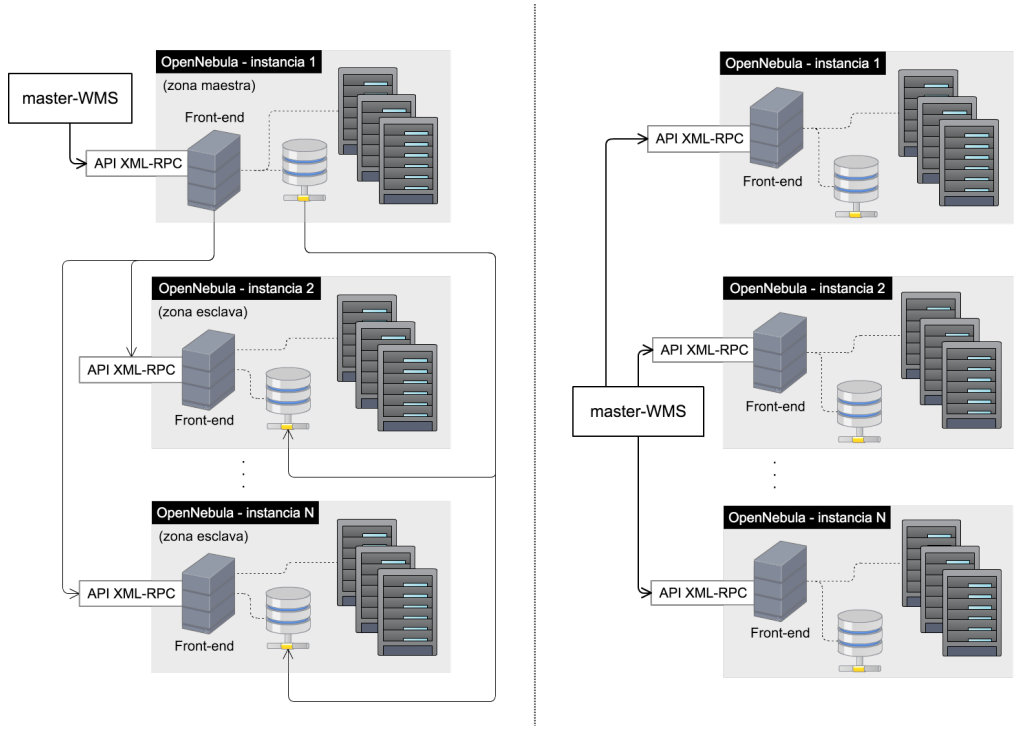


Figura 3.2: Integración completa de instancias mediante federación (izquierda). Integración a nivel de recursos, resuelta por proceso Master-WMS (derecha).

En la instancia de FING (configurada como zona principal) se despliegan los componentes centrales (Master-WMS, base de datos, servidores ZooKeeper, etc.), mientras que las demás instancias (zonas secundarias) se destinan únicamente a Workers. De esta forma se simplifica el proceso de incluir una nueva zona. Este proceso incluye:

1. Instalación de la imagen de disco para Workers en la nueva zona. Esta imagen es provista por FING, contiene MCell, FERNET y el software necesario para los procesos Workers (creada en la sección 3.4.1).

2. Instalación de la imagen de contexto en la nueva zona. Esta imagen es provista por FING (creada en la sección 3.4.4), contiene el código que debe ejecutar un proceso Worker y los scripts de inicialización que se ejecutarán en las máquinas virtuales.

3. Indicar a FING el endpoint y las credenciales de acceso (usuario/contraseña) a los servicios de la nueva zona.

4. Indicar a FING el Identificador de la red virtual a la cual deben conectarse las máquinas virtuales creadas en la nueva zona.

Dado que la comunicación entre Master-WMS y Workers se da únicamente a través de Zookeeper, los Workers no reciben conexiones entrantes y por lo tanto pueden estar conectados a una red local de la zona en la cual se ejecuten. Esta red debe tener conectividad hacia los servidores Zookeeper y la base de datos en FING.

Para establecer la instancia de OpenNebula de FING como zona principal, se configuró el firewall para habilitar cuatro puertos del host utilizado (donde se instanciaron los tres servidores Zookeeper y la base de datos) que acepten conexiones entrantes desde la UBA y redireccionarlos a la máquina virtual correspondiente. Las redirecciones definidas se muestran en la tabla 3.2.

Servicio	IP externa : puerto	IP local : puerto
Servidor de Base de Datos	164.73.47.238:10060	192.168.122.2:3306
Servidor Zookeeper 1	164.73.47.238:2888	192.168.122.3:2181
Servidor Zookeeper 2	164.73.47.238:2889	192.168.122.4:2181
Servidor Zookeeper 3	164.73.47.238:2890	192.168.122.5:2181

Tabla 3.2: Redirección de puertos desde el host (con IP externa 164.73.47.238) hacia las VMs creadas en la subred 192.168.122.x.

Desde la UBA se brindaron credenciales para acceder a los servicios de su OpenNebula y a la interfaz web de administración sunstone. Se instalaron las imágenes de disco y contexto para Workers (mediante sunstone, que permite hacer upload de las imágenes), y se configuró como zona secundaria. Un punto a destacar es que las credenciales brindadas por cada instancia pueden ser configuradas con las reglas específicas de una lista de control de acceso (Access control list, ACL) que permite definir OpenNebula, para limitar el acceso al subconjunto específico de recursos que se desee compartir (se puede consultar la lista completa de reglas para ACL soportadas por OpenNebula y la sintaxis requerida en `http://docs.opennebula.org/4.6/administration/users_and_groups/manage_acl.html`). En el contexto de este proyecto se comparten hosts, pero en general se comparten imágenes, medios de almacenamiento (datastores), redes, datos administrativos como usuarios y grupos, etc.

El Master-WMS debe conocer los recursos disponibles en cada una de las zonas configuradas y crear las máquinas virtuales necesarias en función de la carga de trabajo dada. Para esto, utiliza OpenNebula cloud API, un wrapper en Java para acceder a los servicios expuestos por cada OpenNebula. El Master-WMS itera sobre las zonas configuradas. En primer lugar, se crea una conexión utilizando el endpoint y las credenciales provistas, construyendo el objeto Client (las invocaciones sucesivas se realizarán incluyendo este objeto como parámetro). Luego, se obtiene la lista de hosts disponibles en la zona, obteniendo como respuesta un XML con los datos de cada host: identificador, nombre, estado, información de CPU, disco, memoria, procesador, hipervisor, etc. Se muestra un ejemplo de esta respuesta en el listado 3.28.

```
<HOST_POOL>
  <HOST>
    <ID>37</ID>
    <NAME>localhost</NAME>
    <STATE>1</STATE>
    <HOST_SHARE>
      <MAX_MEM>74347080</MAX_MEM>
      <MAX_DISK>8584541</MAX_DISK>
      <MAX_CPU>2400</MAX_CPU>
      <MEM_USAGE>12582912</MEM_USAGE>
      <CPU_USAGE>1900</CPU_USAGE>
      <RUNNING_VMS>16</RUNNING_VMS>
    </HOST_SHARE>
    <TEMPLATE>
      <ARCH><![CDATA[x86_64]]></ARCH>
      <CPUSPEED><![CDATA[2099]]></CPUSPEED>
      <HOSTNAME><![CDATA[nebula.fing.edu.uy]]></HOSTNAME>
      <HYPERVISOR><![CDATA[kvm]]></HYPERVISOR>
      <MODELNAME><![CDATA[AMD Opteron(tm) Processor 6172]]></MODELNAME>
    </TEMPLATE>
  </HOST>
  <HOST> ... </HOST>
  ...
</HOST_POOL>
```

Listado 3.28: Datos de los hosts.

Una vez recabada la información de los recursos disponibles en cada zona, se ejecuta un algoritmo que calcula la cantidad y características de los procesos Worker necesarios para la ejecución de la carga de trabajo dada, y la distribución de los mismos sobre los hosts disponibles (el algoritmo de asignación de tareas que se detalla en la sección 4.2.4). Con el resultado de este algoritmo se procede a crear las máquinas virtuales correspondientes en cada host, utilizando los métodos *allocate* y *deploy*. El método allocate crea la máquina virtual como entidad a nivel de OpenNebula (sin llegar a instanciarla en un host). Este método recibe como parámetro el template de creación, que contiene las características de la máquina virtual a crear (CPU, memoria, parámetros de contexto, etc.) y retorna el identificador asignado por OpenNebula. El método deploy instancia la máquina virtual en un host específico. Este método recibe como parámetros el identificador de la máquina virtual y el identificador del host.

Para poder invocar los métodos mencionados, se requiere que el usuario tenga permisos de lectura sobre hosts, imágenes, redes virtuales y permisos de administración sobre máquinas virtuales. Este último permiso es necesario para poder invocar el método deploy. Si no se invoca este método, la selección del host donde residirá la máquina virtual queda en manos del planificador interno de OpenNebula, impidiendo llevar a cabo la distribución específica definida por el algoritmo de asignación, que la define teniendo en cuenta el peso de cada tarea a ejecutar y la potencia de cada host, con el objetivo de minimizar el tiempo total de ejecución.

En términos de OpenNebula, los métodos para listar la información de los hosts, crear e instanciar máquinas virtuales invocados a través de Java corresponden a los métodos de la API one.hostpool.info, one.vm.allocate y one.vm.deploy respectivamente, y requieren los siguientes permisos, expresados con la sintaxis para reglas de ACL (la documentación completa de la API y los permisos requeridos por cada método se pueden consultar en `http://docs.opennebula.org/4.6/integration/system_interfaces/api.html`):

- one.hostpool.info: HOST:USE
- one.vm.allocate VM:CREATE, IMAGE:USE, NET:USE
- one.vm.deploy: VM:ADMIN, HOST:MANAGE

Cualquier instancia de OpenNebula que quiera compartir sus recursos para la ejecución de simulaciones deberá proporcionar un usuario con los privilegios mencionados sobre un conjunto de sus hosts, contar con una red local con conectividad hacia FING e instalar las imágenes de disco y de contexto para las máquinas virtuales que serán instanciadas. Este procedimiento se efectuó en la UBA, quedando configurada como zona secundaria.

Capítulo 4

Solución desarrollada

Este capítulo describe en detalle el funcionamiento interno de los distintos componentes que conforman el sistema, la arquitectura general y la distribución física de estos componentes. Se detalla la estrategia de asignación de tareas en función de la carga de trabajo del sistema y de los recursos disponibles, los mecanismos de tolerancia y recuperación ante fallos, y el flujo completo de actividad de los procesos Master-WMS y Worker, encargados de la coordinación y la ejecución de las tareas respectivamente.

4.1. Arquitectura del sistema

La arquitectura refleja la estructura del sistema en un alto nivel de abstracción, representando los componentes que dan soporte a los requisitos funcionales y no funcionales del sistema. Esta sección describe la interacción y la disposición de los componentes que conforman la solución.

4.1.1. Componentes de la arquitectura

La arquitectura adoptada fue diseñada para que el sistema opere sobre los recursos virtualizados que provee el cloud privado construido sobre la infraestructura del cluster FING y del cluster TUPAC. La arquitectura propuesta presenta un sistema heterogéneo y distribuido sobre máquinas virtuales, en donde cada uno de los subsistemas o componentes desempeña un rol específico e interactúa con otros componentes para satisfacer los requisitos del sistema. En la figura 4.1 se presenta el diagrama de componentes del sistema.

Uno de los componentes que brinda soporte a todo el sistema es el Data Storage. El Data Storage se encarga del almacenamiento no volátil relacionado a usuarios y simulaciones MCell+FERNET, siendo la fuente proveedora de información del resto de los componentes. El Data Storage es implementado mediante una base de datos relacional centralizada. Todos los componentes que se comunican con el Data Storage lo hacen utilizando las operaciones provistas por la Storage API, una biblioteca Java implementada en el marco de este proyecto para que todos los componentes del sistema manipulen el acceso a la información de forma estandarizada. La Storage API encapsula la generación de consultas SQL y la comunicación con la base de datos a través del driver JDBC.

El Web Portal es el punto de acceso para los usuarios finales del sistema. Los usuarios que requieren crear y administrar sus simulaciones se conectan a través de Internet al portal web, donde se autentican con su nombre de usuario y contraseña. El Web Portal es capaz de crear, procesar y monitorear las simulaciones ingresadas por los usuarios mediante la interacción con el Broker.

El Broker encapsula la lógica transaccional encargada de recibir, analizar, validar y almacenar las peticiones de los usuarios en el Data Storage para su posterior ejecución. Otra de las funciones del Broker es la consulta al Data Storage para obtener información de interés para el usuario, por ejemplo el estado de las simulaciones en ejecución y el resultado de las simulaciones finalizadas.

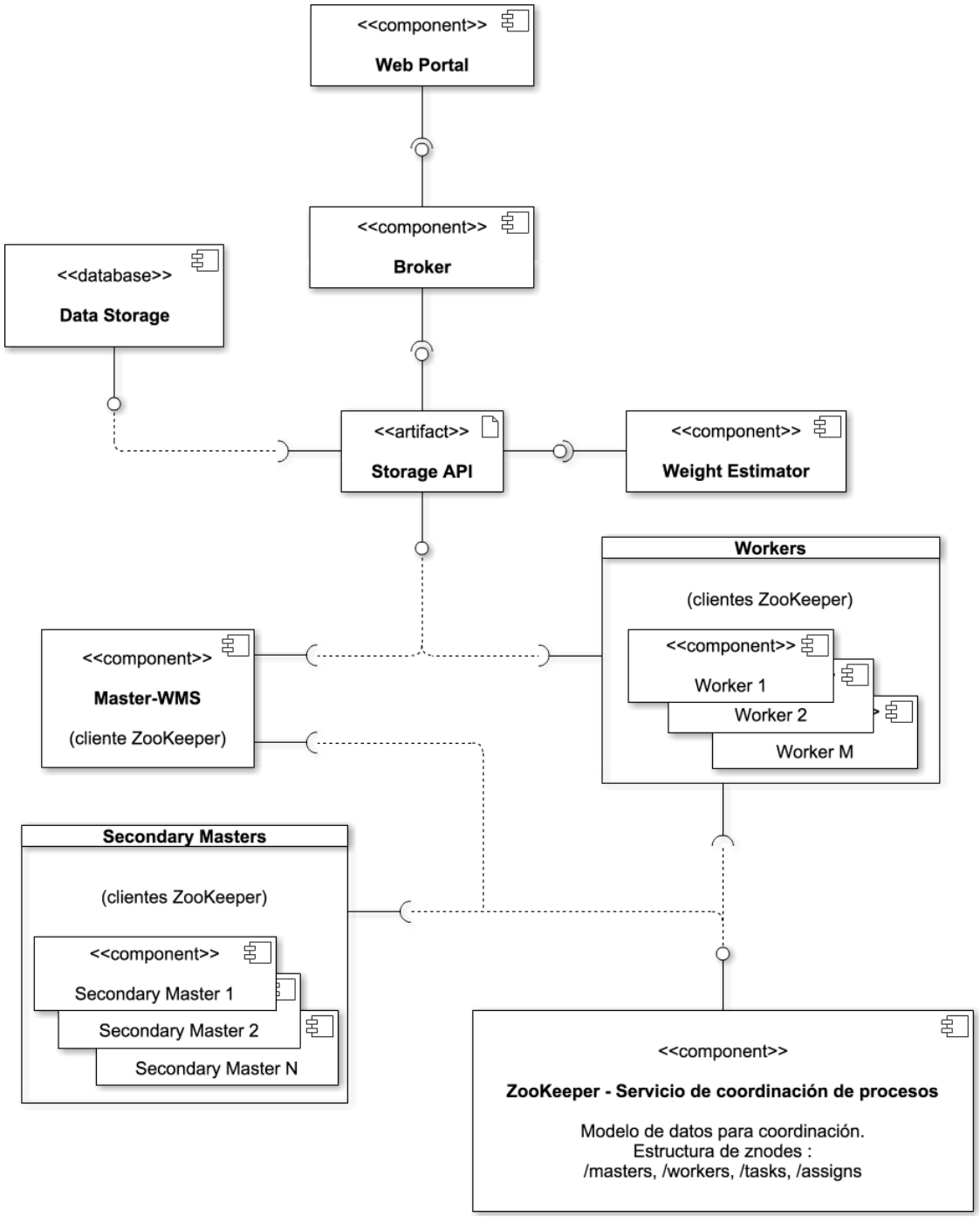


Figura 4.1: Diagrama de componentes del sistema

El Web Portal, el Broker y el Data Storage presentan un estilo arquitectónico en capas distribuido. Este estilo arquitectónico cliente-servidor en 3 niveles, separa presentación (Web Portal), lógica transaccional (Broker) y persistencia de datos (Data Storage) en tres capas lógicas.

Cuando las simulaciones son ingresadas al sistema y validadas aún no están listas para ser ejecutadas. Previo a su ejecución interviene el Weight Estimator, cuya función es tomar las tareas ingresadas por los usuarios, analizarlas y ponderarlas con un costo estimado de ejecución. Los costos estimados son utilizados como soporte para la posterior planificación de ejecuciones en paralelo de forma eficiente. El Weight Estimator es independiente del resto de los componentes, sólo se comunica con el Data Storage para registrar los costos estimados.

La ejecución de las simulaciones es realizada por los componentes Master-WMS y Worker, que ejecutan bajo un modelo maestro-esclavo de forma paralela y distribuida. El Master-WMS ejecuta con el rol de proceso maestro y los Workers como esclavos.

El Master-WMS es el componente encargado de realizar la planificación de las ejecuciones. La planificación realizada por el Master-WMS consiste en recuperar las tareas a ejecutar del Data Storage, crear los Workers con los recursos disponibles en el cloud, distribuir las tareas entre los Workers y monitorizar su ciclo de vida. La distribución de las tareas es realizada por el Master-WMS mediante un algoritmo de balanceo de carga. Este algoritmo se basa en la ponderación de las tareas realizada por el Weight Estimator y en las características de hardware de los Workers para determinar una distribución que minimice el tiempo total de ejecución del conjunto de tareas. Los detalles del algoritmo de asignación se detallan en 4.2.4. Cada Worker ejecuta las tareas que le fueron asignadas en forma secuencial y persiste el resultado de cada ejecución en el Data Storage para que pueda ser consultado por los usuarios.

Dentro de las funciones del Master-WMS se encuentra la detección y recuperación ante fallos en los Workers. Cuando se detecta un fallo en un Worker, el Master-WMS redistribuye las tareas que el Worker tenía asignadas entre los Workers que permanecen activos. A su vez, el sistema maneja un mecanismo de tolerancia a fallos del Master-WMS. Este mecanismo consiste en ejecutar en paralelo múltiples instancias de un proceso denominado Secondary Master. Cuando se produce un fallo en el Master-WMS, un Secondary Master asume el rol de proceso maestro, desarrollando las funciones del Master-WMS. Durante el lapso de tiempo entre que el Master-WMS falla y un Secondary Master ocupa su lugar, los fallos que ocurren en los Workers no son detectados. Por lo tanto, cuando un Secondary Master ocupa el rol de proceso maestro ejecuta una rutina de control que verifica si se produjeron fallos en los Workers. En caso de constatarse fallos se redistribuyen las tareas huérfanas (sin Worker asignado) entre los Workers activos. El mecanismo de tolerancia a fallos del Master-WMS se detalla en la sección 4.2.9.3.

La virtualización y la utilización de recursos a demanda son las características del cloud que dan soporte al modelo de ejecución paralela y distribuida planteado. Los Workers son ejecutados en máquinas virtuales creadas a demanda por el Master-WMS en función de los recursos disponibles en las instancias de OpenNebula configuradas en el sistema. Las máquinas virtuales son creadas a partir de una imagen de disco previamente configurada con el sistema operativo y software necesario para iniciar un conjunto de procesos Worker en el arranque de la máquina (se detalla en 3.4).

El Master-WMS, los Secondary Master y los Workers utilizan el servicio ZooKeeper para intercambiar y sincronizar información de control. El servicio ZooKeeper se ejecuta en uno o más servidores que mantienen el modelo de datos accesible para los clientes. La interacción entre clientes se realiza mediante el servicio de suscripciones a eventos que ofrece ZooKeeper (*watches* y notificaciones).

4.1.2. Distribución física de los componentes del sistema

Los componentes que conforman la arquitectura del sistema se encuentran distribuidos sobre las zonas OpenNebula instanciadas. Actualmente el sistema cuenta con dos zonas; una zona se encuentra ubicada en la infraestructura del cluster FING (zona FING) y la otra zona se encuentra ubicada en el cluster TUPAC (zona UBA). Los componentes del sistema trabajan sobre máquinas

virtuales con hardware y software específicos para realizar su función e interactuar a través de diversos mecanismos. La descripción del software utilizado para cada máquina virtual se especifica en la sección 3.4.1.

En la zona FING se distribuyen los siguientes componentes (figura 4.2a):

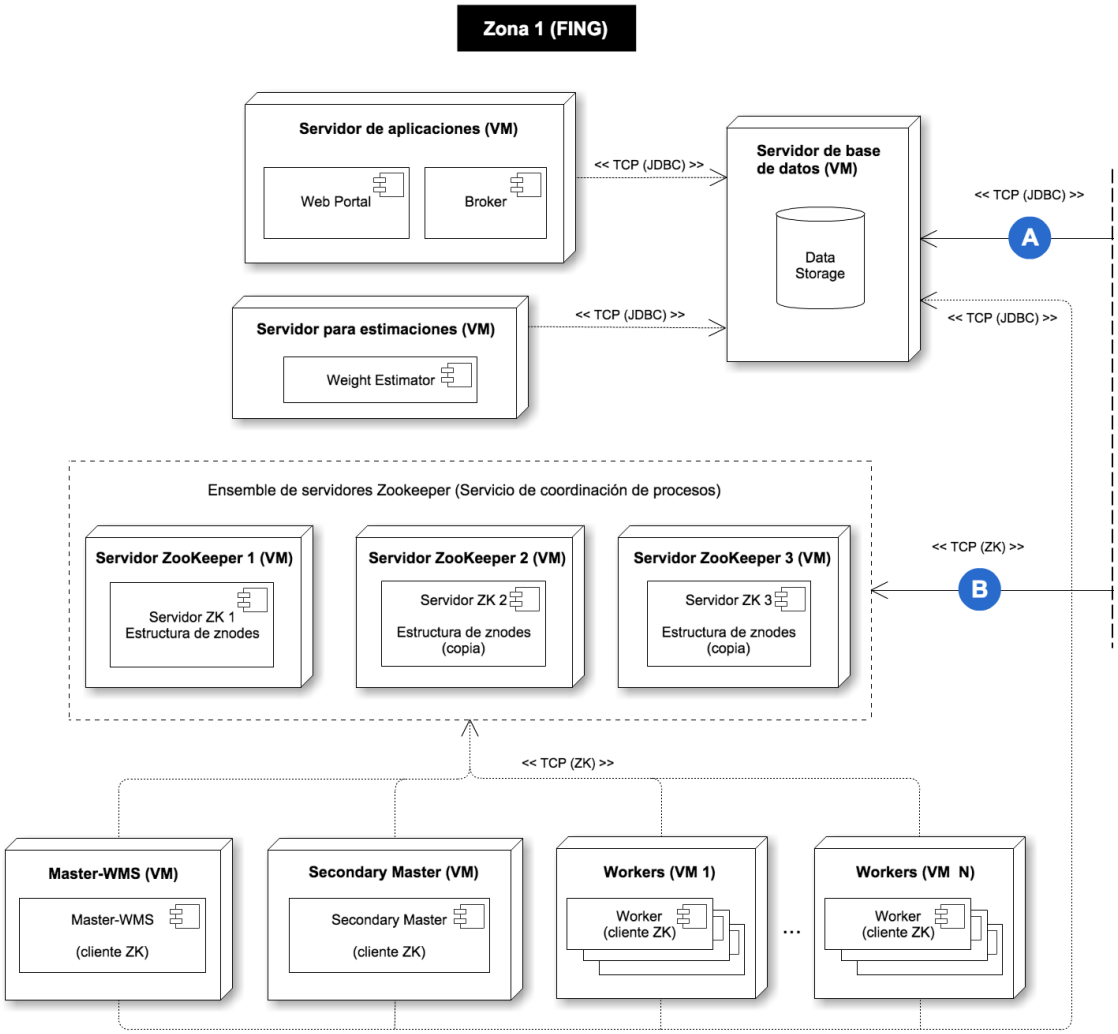
- Web Portal y Broker ejecutando en una única máquina virtual independiente.
- Weight Estimator ejecutando en una máquina virtual independiente.
- Data Storage ejecutando en una máquina virtual independiente.
- Tres servidores que conforman el ensemble Zookeeper y ejecutan en tres máquinas virtuales independientes.
- Workers ejecutando en máquinas virtuales que pueden contener más de un Worker.
- Master-WMS ejecutando en una máquina virtual independiente.
- Secondary Master ejecutando en máquinas virtuales independientes.

Dependiendo del grado de tolerancia a fallos en el Master-WMS que se quiere alcanzar se determina la cantidad de Secondary Masters a utilizar. La creación de Secondary Masters no es condición necesaria para que el sistema funcione. Los Secondary Masters pueden ejecutarse en una máquina virtual independiente o compartiendo máquina virtual con otro componente (e.g. Weight Estimator) para reutilizar recursos.

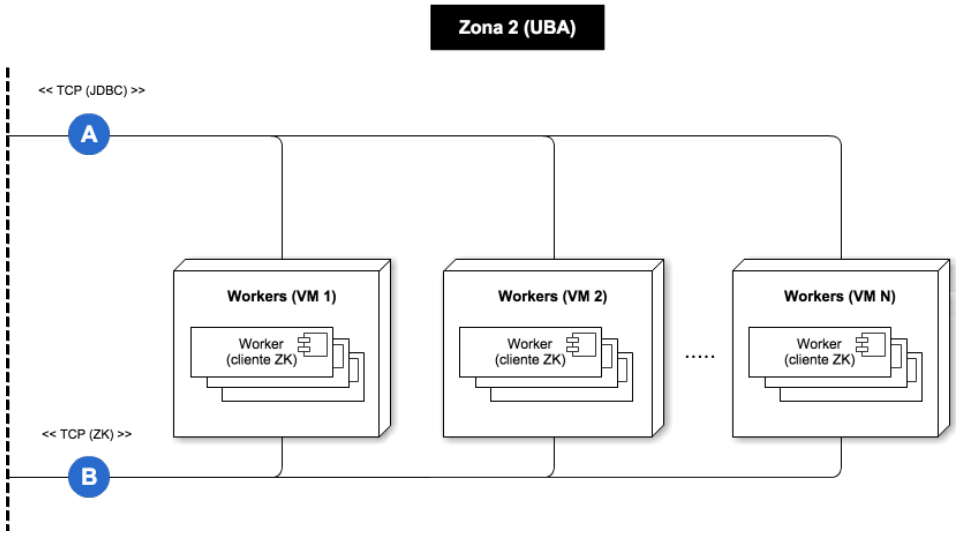
En la zona UBA los únicos componentes que ejecutan son los Workers (figura 4.2b). Los Workers son creados por el Master-WMS mediante los servicios de la instancia OpenNebula de la zona UBA. Los nodos físicos en la zona UBA sirven para incrementar el poder de computo del sistema y se utilizan de acuerdo a la demanda de recursos que requiere ejecutar una carga de trabajo (se detalla en la sección 3.6). La disposición de los componentes críticos en la zona FING permite tener mayor control sobre el sistema, ya que los permisos de gestión sobre infraestructura administrada por terceros (en el contexto de este proyecto la infraestructura de la zona UBA) suelen ser limitados.

La comunicación entre el Data Storage y el resto de los componentes del sistema se realiza directamente sobre TCP al igual que la comunicación de los clientes ZooKeeper (Master-WMS, Secondary Master y Workers) con el ensemble de servidores. Las conexiones con el Data Storage son gestionadas por la Storage API usando el conector a base de datos JDBC. Las conexiones con el ensemble de servidores ZooKeeper son gestionadas por la API para Java incluida en ZooKeeper.

El diseño arquitectónico del sistema facilita la incorporación de nuevas zonas secundarias (proceso detallado en la sección 3.6). El modo operativo del Master-WMS permite la vinculación y desvinculación de zonas de manera planificada o dinámica. La vinculación o desvinculación planificada se realiza mediante la configuración de las zonas disponibles en el Data Storage, sin necesidad de modificar componentes del sistema. La vinculación o desvinculación dinámica se realiza durante el proceso de ejecución del Master-WMS. Cuando existen fallos o problemas de comunicación con una zona, el Master-WMS redistribuye las tareas entre los Workers que permanecen en zonas activas (se detalla en la sección 4.2.9).



(a) FING. Zona principal, se distribuyen los componentes centrales. Las máquinas virtuales que ejecutan procesos Worker son creadas a demanda por el proceso Master-WMS.



(b) UBA. Zona secundaria destinada a alojar máquinas virtuales que ejecuten procesos Worker. Los Workers son creados a demanda por el proceso Master-WMS que se ejecuta en la zona principal.

Figura 4.2: Diagrama de distribución del sistema

4.2. Software

Esta sección describe el funcionamiento interno y los detalles de implementación de cada componente que integra la arquitectura del sistema descrita en la sección 4.1.1. En primer lugar se describe la estructura y la información que se almacena en el Data Storage y el ensemble de servidores Zookeeper. Luego se desarrollan aspectos relacionados al ciclo de vida de las tareas, la estimación de su costo de ejecución y el algoritmo de asignación implementado. Finalmente, se describe el flujo completo de los procesos Master-WMS y Worker, las funciones principales del Web Portal y Broker, y se introducen algunos aspectos relacionados con la tolerancia a fallos que serán retomados en la sección 4.2.9.

4.2.1. Servidores ZooKeeper

La comunicación y coordinación entre procesos es necesaria en sistemas distribuidos. En el sistema implementado en el marco de esta investigación, ZooKeeper es la herramienta encargada de gestionar la comunicación entre los procesos Worker, Master-WMS y Secondary Master. La coordinación con ZooKeeper es utilizada para la planificación de las cargas de trabajo y los mecanismos de tolerancia a fallos de los componentes del sistema.

El sistema implementado se configuró para funcionar con tres servidores ZooKeeper, distribuidos en tres máquinas virtuales alojadas en un host del cluster FING. El ensemble está configurado de manera estándar, pero ZooKeeper permite configurar los servidores para trabajar de manera más adecuada al contexto en el que operen. Algunos de los parámetros de configuración en ZooKeeper son [9]:

- Tamaño de la unidad básica de medida del tiempo (*tickTime*). Configurado por defecto en 3 segundos.
- Número máximo de solicitudes pendientes (*globalOutstandingLimit*). Debido a que los servidores pueden recibir solicitudes de clientes a una tasa mayor a la que son procesadas, deben mantener una cola de solicitudes. El tamaño de la cola está limitado por el parámetro *globalOutstandingLimit*, configurado por defecto en 1000 solicitudes.
- Límite de conexiones concurrentes por cliente (*maxClientCnxns*). Este parámetro permite limitar el número de conexiones concurrentes a nivel de socket que puede mantener un cliente con un servidor del ensemble. Se utiliza para prevenir ataques de negación de servicio (*Denial of Service, DoS*). El valor configurado por defecto es 60 conexiones.
- Timeout mínimo y máximo (medido en tickTimes) permitido por los servidores para las conexiones de los clientes (*minSessionTimeout* y *maxSessionTimeout*). Cada cliente que establece una conexión con los servidores indica el timeout para las peticiones que realice. El valor indicado debe estar en el rango $[minSessionTimeout, maxSessionTimeout]$ para que sea aceptado por los servidores, de lo contrario los servidores establecerán el timeout en *minSessionTimeout* o *maxSessionTimeout* (el valor que sea más cercano al solicitado por el cliente). Este procedimiento se conoce como timeout negociado (*negociated timeout*). Los valores por defecto para *minSessionTimeout* y *maxSessionTimeout* son 2 y 20 tickTimes respectivamente.
- El parámetro *leaderServes* indica si el servidor líder del ensemble acepta conexiones clientes. Por defecto el líder acepta conexiones.

4.2.1.1. Estructura de znodes

Los metadatos necesarios para dar soporte a la comunicación y sincronización de los procesos Worker, Master-WMS y Secondary Master son almacenados en los servidores ZooKeeper, bajo una estructura jerárquica de znodes. Los procesos Worker, Master-WMS y Secondary Master (i.e. clientes del ensemble) pueden leer, modificar y crear znodes. Las acciones efectuadas por un cliente sobre un znode, pueden disparar notificaciones a otros clientes que estén suscritos mediante el registro de un watch al znode (se detalla en la sección 2.3.2). Para el sistema implementado se definió la siguiente estructura de znodes (ilustrada en la figura 4.3):

/masters Los procesos dispuestos a tomar el rol de maestro (Secondary Masters) deben crear un znode efímero en modo secuencial bajo el znode */masters*, como parte del mecanismo de tolerancia a fallos del proceso Master-WMS basado en el algoritmo de elección de líder (*leader election*) propuesto en [13] y detallado en la sección 4.2.9.3.

/workers Representa a los procesos Worker activos en el sistema. Cada Worker crea al inicio de su ejecución, un znode efímero bajo */workers* para indicar que está disponible para ejecutar tareas. Cuando el Worker deja de estar disponible (i.e. caduca la sesión con el ensemble), el znode que lo representa es eliminado directamente por el servicio ZooKeeper y el proceso Master-WMS (que tiene un watch sobre este znode) es notificado para proceder con la redistribución de las tareas del Worker a otros Workers disponibles.

/tasks Mantiene znodes que contienen los datos básicos de las tareas que se deben ejecutar. Los datos completos de las tareas se encuentran almacenados en la base de datos. Cuando se finaliza la ejecución de una tarea, el znode correspondiente es removido de esta colección por el Worker que la ejecutó.

/assigns Representa las asignaciones de tareas a Workers. Cada znode hijo referencia a un Worker, y cada hijo de éste a las tareas asignadas al Worker. El znode */assigns* es cargado por el proceso Master-WMS en función de la salida del algoritmo de asignación descrito en la sección 4.2.4. Luego, cada Worker comenzará a ejecutar las tareas que estén bajo su znode en */assigns*.

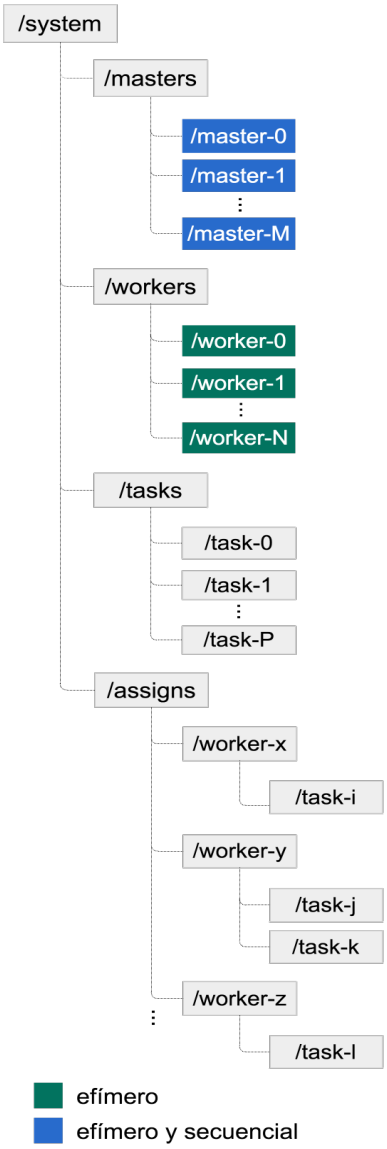


Figura 4.3: Estructura de znodes utilizada en el sistema implementado

4.2.2. Data Storage

El Data Storage es el componente contenedor de la información no volátil del sistema. Este componente es implementado en una base de datos relacional MySQL centralizada, sobre una máquina virtual gestionada por la instancia de OpenNebula en FING. El esquema de la base de datos del Data Storage se presenta en la figura 4.4.

En Data Storage contiene la información de los pedidos realizados por los usuarios del sistema y las tareas a ejecutar relacionadas a estos pedidos. Cada pedido define un conjunto de tareas a ejecutar, que comprenden archivos de configuración y parámetros de entrada para MCell y FERNET. Además, en el Data Storage se almacena la información de las instancias de OpenNebula (i.e. zonas) disponibles para distribuir procesos Worker. El Master-WMS utiliza la información de las zonas configuradas para poder consultar los recursos disponibles en cada una y crear las máquinas virtuales donde se ejecutarán los Workers.

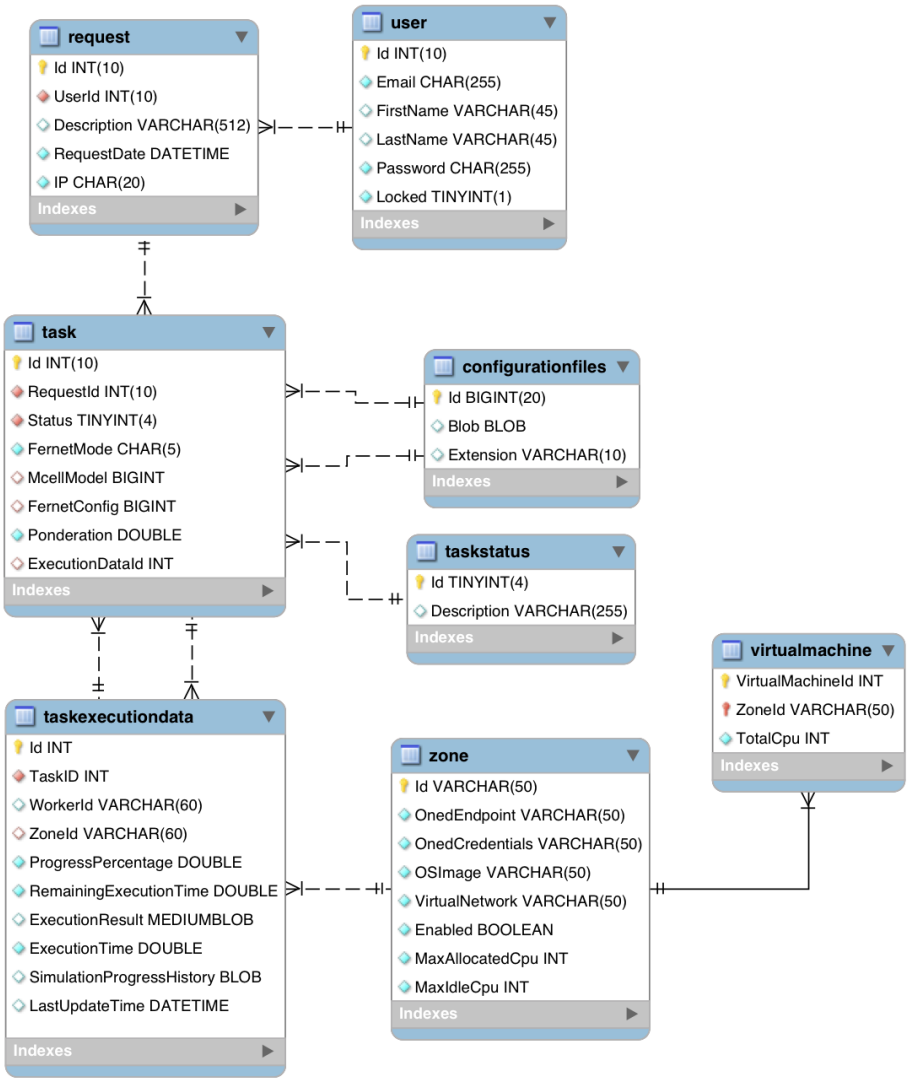


Figura 4.4: Esquema de la base de datos del sistema.

El esquema de la base de datos está compuesto por 8 tablas que se describen a continuación:

User Contiene la información de los usuarios finales del sistema.

Request Encapsula la información básica de los pedidos realizados por los usuarios (descripción, fecha de solicitud y dirección IP). Cada uno de estos pedidos puede comprender una o más tareas (i.e. simulaciones MCell+FERNET), que serán ejecutadas por los Workers.

ConfigurationFiles Almacena los archivos necesarios para ejecutar las simulaciones MCell+FERNET. Estos archivos son cargados en un campo de tipo BLOB. Debido al tamaño que suelen tener estos archivos (aproximadamente 1KB) resulta viable almacenarlos directamente en el DBMS, ya que no se genera un almacenamiento excesivo a corto plazo. De esta manera se evita el uso de sistemas de almacenamiento externos, y el sistema se beneficia de las características de integridad, atomicidad y seguridad que provee una base de datos.

Task Contiene y agrupa información relacionada con una simulación de un pedido concreto. Cada tarea referencia a dos archivos de configuración correspondientes a los archivos de entrada para MCell y FERNET. Esta tabla contiene campos para indicar el estado de la tarea, el modo de ejecución de FERNET, la ponderación de la tarea y los datos relacionados a su ejecución.

TaskStatus Almacena los posibles estados que pueden tener una tarea. Estos estados son:

- *Ingresada*: la tarea fue creada pero aún no ha sido estimado su costo, por lo tanto no está lista para ser ejecutada.
- *Pendiente*: la tarea fue creada y su costo ha sido estimado por el sistema, ya está lista para ejecutarse.
- *Ejecutando*: la tarea está siendo ejecutada por un Worker.
- *Finalizada*: la tarea se ejecutó satisfactoriamente.
- *Error*: la tarea finalizó con un error en la ejecución.
- *Cancelada*: el usuario canceló la ejecución de la tarea.
- *Demorada*: la tarea se está ejecutando pero con un progreso lento. En caso de persistir la lentitud el sistema la finalizará y le asignará el estado *Error*.
- *Enviada a re-ejecutar*: las tareas son cambiadas a este estado durante el proceso de redistribución de tareas, por ejemplo, ante la caída del Worker que la estaba ejecutando.

Los cambios de estado que pueden ocurrir durante el ciclo de vida de las tareas se muestran en la figura 4.5.

TaskExecutionData Almacena la información de la ejecución parcial o total de una tarea. A medida que una tarea ejecuta, el proceso Worker persiste información de control sobre el estado de ejecución con el objetivo de mostrar a los usuarios el progreso de sus simulaciones. Se almacena la zona y el Worker donde la tarea fue o está siendo ejecutada, el porcentaje de progreso y el tiempo de ejecución hasta el momento, el tiempo estimado de finalización, la información específica a la ejecución de MCell+FERNET (cantidad de iteraciones ejecutadas y cantidad de iteraciones ejecutadas por segundo) y un archivo .zip con el resultado de la simulación (almacenado en el campo ExecutionResult).

Zone Almacena la configuración correspondiente a las instancias de OpenNebula del sistema (zonas). Se almacena la URL de los servicios web de la zona (endpoint), las credenciales de acceso (usuario y password), las cantidad máxima de CPU para asignar a las máquinas virtuales que se creen (maxAllocatedCpu) y la cantidad máxima de CPU que puede queda ociosa una vez finalizadas las ejecuciones (maxIdleCpu). Una vez finalizada la ejecución de una carga de trabajo en el sistema, el proceso Master-WMS eliminará una determinada

cantidad de máquinas virtuales en cada zona si la CPU utilizada es mayor a maxIdleCpu. Las máquinas virtuales que no se eliminen, continuarán ejecutando los procesos Workers, que permanecerán ociosos hasta que le sean asignadas más tareas en la próxima ejecución del Master-WMS. Este mecanismo ayuda a reducir los tiempos de arranque del sistema, ya que no se deben crear todas las máquinas virtuales nuevamente en cada ejecución del Master-WMS.

VirtualMachine Contiene los datos de las máquinas virtuales creadas por el proceso Master-WMS para la ejecución de las simulaciones. Se almacena el identificador de la zona, el identificador de la máquina virtual y la cantidad de CPU asignada.

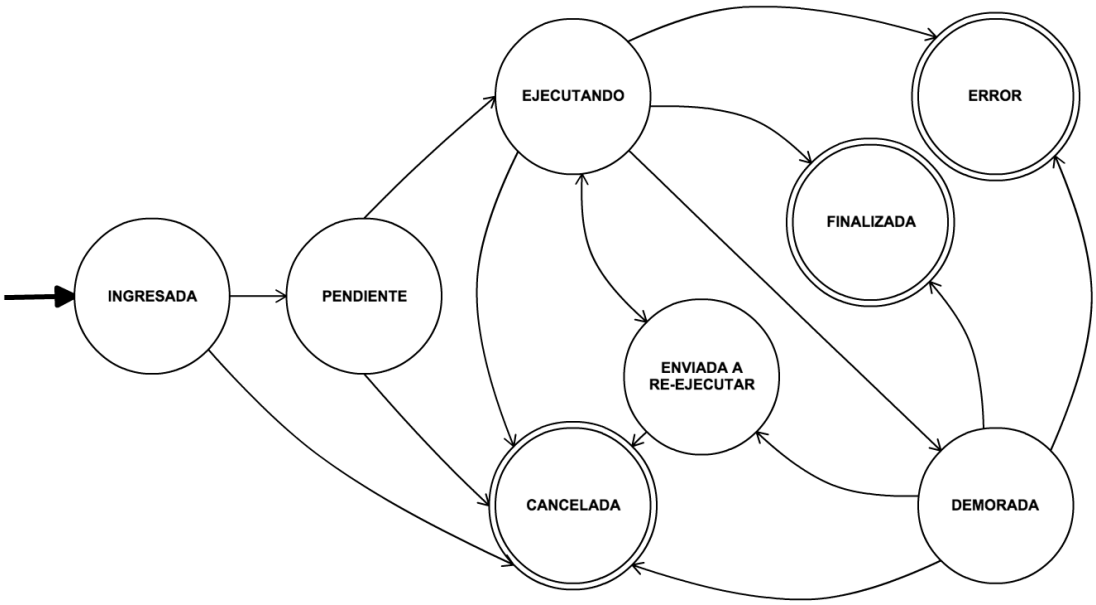


Figura 4.5: Posibles transiciones de estado durante el ciclo de vida de una tarea.

4.2.3. Proceso Master-WMS

El Master-WMS es un componente fundamental de la arquitectura del sistema. Como lo indica su nombre, cumple el rol de proceso maestro, controlando y comunicándose con todos los Workers a través de OpenNebula y ZooKeeper. Además, este proceso cumple el rol de manejador de carga de trabajo (*Workload Management System*, WMS) distribuyendo las tareas sobre los recursos disponibles del sistema y coordinando su ejecución, para lograr un mejor aprovechamiento de los recursos y reducir el tiempo total de ejecución de la carga de trabajo.

El sistema implementado cuenta con un único proceso ejecutando el rol Master-WMS. Este proceso ejecuta sobre una máquina virtual en la zona FING, como un cliente más del ensemble de servidores ZooKeeper. Pueden existir otros procesos ejecutándose en modo pasivo, que intervendrán ante una falla del Master-WMS para ocupar su lugar. Estos procesos de respaldo son los Secondary Masters. En la sección 4.2.9.3 se detalla el método de elección del proceso reemplazante.

El comportamiento típico que realiza el maestro en una ejecución involucra recuperar las simulaciones pendientes de la base de datos; crear los procesos Worker que sean necesarios; asignarles las tareas y finalmente realizar el seguimiento de la ejecución de estas tareas, interviniendo en caso de ser necesario para reasignar las tareas que por algún motivo no pudieron ejecutarse en el Worker que les fue asignado. La cantidad de Workers a crear es determinada en función de los recursos disponibles en las instancias OpenNebula configuradas, de la carga de trabajo a ejecutar y de los Workers ya existentes (aquellos que fueron creados en una ejecución anterior del Master-WMS y continuaron ejecutándose en modo pasivo una vez finalizada la ejecución de la carga de trabajo asignada). La ejecución de múltiples procesos Worker distribuidos implica que el Master-WMS deba responder ante fallos internos o de comunicación que pudiesen ocurrir en los Workers, supervisando y controlando que el sistema no quede en un estado inconsistente.

Las instancias OpenNebula son invocadas por el Master-WMS a través de la API para Java. Estas invocaciones permiten determinar los recursos disponibles en cada instancia y crear las máquinas virtuales necesarias para la ejecución de los nuevos Workers. La creación de máquinas virtuales es un proceso que puede demandar un tiempo considerable (minutos) dependiendo de la cantidad que vayan a ser creadas. Por este motivo, OpenNebula crea las máquinas virtuales utilizando operaciones que retornan el resultado de forma sincrónica pero la creación efectiva y el inicio de la máquina virtual se realiza de manera asincrónica. Para minimizar el tiempo de creación de las máquinas virtuales se implementaron dos mecanismos:

1. Reducir la cantidad de máquinas virtuales ha crear, permitiendo que se ejecute más de un Worker en cada una y así reducir el tiempo dedicado a la creación de máquinas virtuales. Debido a que los procesos Worker utilizan como máximo dos núcleos para realizar las simulaciones, es importante que la cantidad de Workers no supere la mitad de los núcleos de la máquina virtual si no se quiere perder rendimiento en la ejecución.

2. Mantener un número de máquinas virtuales activas entre ejecuciones independientes del sistema. Es posible definir una cantidad máxima de máquinas virtuales (especificando cantidad de CPU) que no serán destruidas al finalizar la ejecución del sistema, quedando disponibles para futuras ejecuciones.

Durante el ciclo de vida del Master-WMS se realizan operaciones asincrónicas en ZooKeeper para mejorar la performance general del sistema. Por ejemplo, las operaciones de lectura sobre el znode */workers* realizadas para obtener la lista de Workers activos del sistema. Además, el Master-WMS mantiene esta lista en memoria para minimizar la cantidad de consultas a ZooKeeper. Cada vez que se consulta el znode */workers* se coloca un *watch*; por lo tanto el Master-WMS es notificado ante cualquier cambio ocurrido en este znode (e.g. la eliminación de un znode hijo debido al fallo de un Worker) y actualiza la lista mantenida en memoria. Estas notificaciones son procesadas en un hilo de ejecución paralelo, evitando detener de forma innecesaria el hilo principal del maestro. El flujo de actividad completo del Master-WMS se muestra en la figura 4.6.

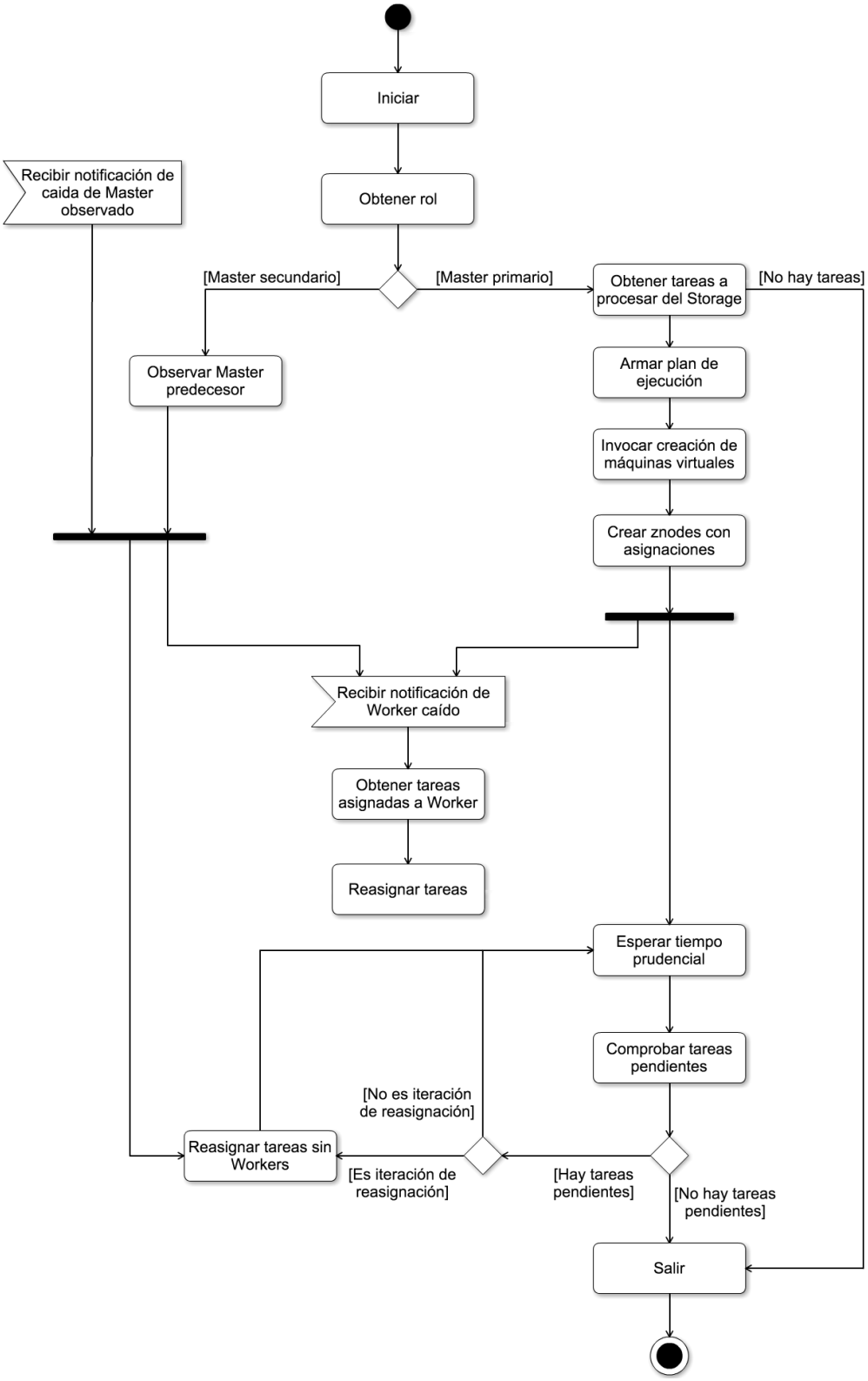


Figura 4.6: Diagrama de actividad del Master-WMS

A continuación se describen cada una de las actividades del flujo de ejecución del Master-WMS.

Iniciar El proceso Master-WMS comienza estableciendo una sesión con el ensemble de servidores ZooKeeper y creando la estructura de znodes necesaria para el funcionamiento del sistema. Se crea el znode raíz y sus hijos directos: */tasks* para agrupar las tareas a ejecutar, */workers* para mantener los Workers activos del sistema, */assigns* para mantener las asignaciones de tareas y */masters* para el manejo de tolerancia a fallos del Master-WMS. Todos los znodes creados son persistentes y no contienen datos.

Obtener rol En este punto es que el proceso determina si va a actuar como primario o como secundario. Dependiendo del rol que adquiere, el proceso maestro sigue uno de los dos caminos posibles del flujo de actividad. Un maestro secundario actúa como un respaldo y está en modo pasivo mientras no reciba una notificación de que el maestro primario falló. Esto se representa en las actividades "Observar Master predecesor" y "Recibir notificación de caída de Master observado" de la figura 4.6. En la sección 4.2.9.3 se profundiza en el mecanismo de tolerancia a fallos del Master-WMS.

Obtener tareas a procesar Cuando el proceso asume el rol de maestro primario, inicia el flujo de ejecución principal. La primera actividad de este rol es recuperar las tareas pendientes de la base de datos y crear los znodes correspondientes en ZooKeeper. Únicamente se recupera el identificador de cada tarea, que pasa a formar parte del identificador del znode. Los znodes son creados bajo el znode */tasks* en modo persistente (e.g. */tasks/task-id*) y contienen un string en formato json con la ponderación (i.e. costo estimado de ejecución) de la tarea (e.g. {ponderation: 1201.4}).

Armar plan de ejecución Uno de los objetivos principales del Master-WMS es planificar la ejecución de las simulaciones. Para cumplir con este objetivo, el maestro interactúa con cada una de las instancias de OpenNebula para reunir información acerca de sus recursos disponibles. En función de esta información, de la cantidad y del costo estimado de las tareas a ejecutar, se determina la cantidad de máquinas virtuales a crear en cada instancia, la cantidad de procesos Worker que se ejecutarán en cada máquina virtual y las tareas asignadas a cada uno de los Workers.

Luego de obtener las tareas a procesar, se consulta ZooKeeper para obtener los procesos Worker existentes. Estos Workers fueron creados en ejecuciones previas del Master-WMS y no fueron destruidos, por lo tanto continuaron ejecutándose en modo pasivo (manteniendo la conexión con ZooKeeper y su znode correspondiente bajo */workers*) una vez que finalizaron la ejecución de la carga de trabajo asignada.

Si la cantidad de Workers existentes es mayor o igual a la cantidad de tareas a procesar, se distribuirán las tareas entre los workers existentes. En caso de que la cantidad de tareas a ejecutar supere el número de Workers existentes, se crearán nuevos Workers si se cuenta con recursos adicionales.

Para determinar si se pueden crear nuevos Workers, el Master-WMS consulta en todas las zonas OpenNebula la CPU disponible de cada host para la creación de nuevas máquinas virtuales. A partir de esta información, se crea una colección de objetos que representan lógicamente los Workers candidatos a ser creados[1], asignándole dos núcleos de CPU a cada uno[2]. Por ejemplo, sobre un host con siete núcleos disponibles se podrán crear cuatro Workers, tres de ellos con dos núcleos cada uno y un cuarto Worker con un sólo núcleo. En el algoritmo 1 se muestra el pseudocódigo correspondiente a la obtención de las tareas, de los Workers existentes y de los candidatos a ser nuevos Workers.

[1] Se crea en memoria una entidad Worker, sin darlo de alta en una máquina virtual.

[2] La ejecución de MCell+FERNET utiliza a lo sumo dos núcleos de CPU, por lo tanto no tiene sentido asignarle a los Workers una cantidad mayor. De todas formas, este valor es configurable como MAX_CPUS_PER_WORKER.

Algoritmo 1 Algoritmo utilizado por el Master-WMS para determinar las tareas a ejecutar, los Workers existentes y los Workers que se podrían crear de acuerdo a los recursos disponibles.

```
1: tasks = storage.getPendingTasks()
2: existingWorkers = zookeeper.getExistingWorkers()
3: newWorkers = new Collection()
4:
5: if tasks.Size >existingWorkers.Size then
6:
7:   zones = openNebula.getZonesInfo(storage.getZonesEndpoints())
8:   for zone in zones do
9:
10:    for worker in existingWorkers do
11:      if worker.ZoneId = zone.Id then
12:        zoneCurrentCpu += worker.AssignedCpu
13:      end if
14:    end for
15:
16:    zoneUsableCpu = zone.MaxCpu - zoneCurrentCpu
17:    if zoneUsableCpu >0 then
18:      for host in zone.Hosts do
19:        host.AvailableCpu = min(host.AvailableCpu, zoneUsableCpu)
20:        while host.AvaiableCpu >0 do
21:          workerCpu = min(hostAvaiableCpu, MAX_CPUS_PER_WORKER)
22:          hostAvailableCpu -= workerCpu
23:          zoneUsableCpu -= workerCpu
24:          newWorker = new Worker(zone, host, workerCpu)
25:          newWorkers.add(newWorker)
26:        end while
27:      end for
28:    end if
29:  end for
30: end if
31:
32: allWorkers = existingWorkers.concat(newWorkers)
```

Una vez creados los objetos Worker, se invoca al algoritmo que asigna las tareas a Workers (se detalla en la sección 4.2.4) buscando una distribución que minimice el tiempo total de ejecución de la carga de trabajo. El algoritmo de asignación recibe como parámetros la colección de objetos Worker y la colección de tareas con sus respectivas ponderaciones, y retorna un diccionario con la lista de tareas asignadas a cada Worker.

Luego de generada las asignaciones, el Master-WMS crea los descriptores de las máquinas virtuales a instanciar, donde residirán los nuevos Workers que resultaron con tareas asignadas. Estos descriptores contienen la información necesaria para generar el template de creación de la máquina virtual: identificador del host, características de CPU y memoria, cantidad de Workers a ejecutar, etc. La cantidad de máquinas virtuales a crear para la ejecución, dependerá de la cantidad de Workers permitida por máquina virtual (configurable por el administrador del sistema). Las características de cada máquina virtual varían en función de la cantidad y las características de los Workers que ejecutará. Por ejemplo, una máquina virtual que deba ejecutar dos Workers con dos núcleos de CPU asignados a cada uno, ocupará cuatro núcleos del host y será creada con cuatro CPUs virtuales y 2GB de memoria RAM[3].

[3]MCELL y FERNET consumen una cantidad fija de memoria RAM durante su ejecución. Durante el análisis

Finalmente, se procede a crear las máquinas virtuales a través de las operaciones provistas por la API de OpenNebula. Para cada máquina virtual se genera un template que contiene los datos almacenados en el descriptor y la información de contexto, permitiendo que los servicios de contextualización instalados en la imagen de disco configuren la máquina virtual y disparen la ejecución de los Workers indicados (esto se conoce como proceso de contextualización, detallado en 3.4.3). En el algoritmo 2 se muestra el pseudocódigo correspondiente a la invocación del algoritmo de asignación, la creación de los descriptores y las nuevas máquinas virtuales.

Algoritmo 2 Algoritmo utilizado por el Master-WMS que invoca al algoritmo de asignación y crea las nuevas máquinas virtuales

```
1: {Llamada al algoritmo de asignación de tareas a Workers}
2: executionPlan = doExecutionPlan(tasks, allWorkers)
3:
4: {Creación de descriptores para las nuevas máquinas virtuales}
5: vmsToCreate = new Collection()
6: for zone in zones do
7:   for host in zone.Hosts do
8:     workersToCreate = getNewWorkersWithAssignments(executionPlan, zone, host)
9:     vm = new VirtualMachineDescriptor(host)
10:    n = 0
11:    for worker in workersToCreate do
12:      n = n+1
13:      vm.Cpu = vm.Cpu + worker.AssignedCpu
14:      vm.VCpu = vm.VCpu + worker.AssignedCpu
15:      vm.Memory = vm.Memory + 1024
16:      vm.Workers.add(worker)
17:
18:      if n >MAX_WORKERS_PER_VM then
19:        n = 0
20:        vmsToCreate.add(vm)
21:        vm = new VirtualMachineDescriptor(host)
22:      end if
23:    end for
24:  end for
25: end for
26:
27: {Creación de las máquinas virtuales}
28: for vm in vmsToCreate do
29:   vmTemplate = vm.generateTemplate()
30:   createdVm = openNebula.createVirtualMachine(zone.Id, host.Id, vmTemplate)
31: end for
```

Crear znodes de asignaciones Cuando el Master-WMS arma el plan de ejecución, lo modela en ZooKeeper creando los znodes con las asignaciones de cada Worker bajo el znode */assigns*. Una vez que los Workers son notificados por ZooKeeper, obtienen las tareas asignadas consultando su znode de tareas bajo */assigns*.

Al finalizar la creación de los znodes con las asignaciones, el Master-WMS coloca un *watch* en el znode */workers* para ser notificado cuando un znode es eliminado, lo que significa que un Worker falló.

experimental se comprobó que esta cantidad es inferior a 1GB, por lo tanto se le asigna 1GB de memoria RAM a cada Worker.

Recibir notificación de Worker caído y reasignar tareas Como se indicó en la sección 2.3.2, un evento ocurrido en los znodes de ZooKeeper dispara una notificación que es procesada en un nuevo hilo de ejecución en el cliente. Por lo tanto, las las notificaciones del fallo de un proceso Worker son procesadas por el Master-WMS en un hilo de ejecución independiente. La lógica de procesamiento de estas notificaciones consiste en recuperar todas las tareas asignadas al Worker que falló y reasignarlas al resto de los Workers activos del sistema.

El proceso de reasignación de tareas consiste en actualizar el estado de las tareas en la base de datos a *Enviada a re-ejecutar*. Posteriormente, se ejecuta el algoritmo de asignación de tareas con los Worker activos y las tareas pendientes del Worker que falló (se detalla en la sección 4.2.4). El resultado de las reasignaciones se modela creando los znodes correspondientes bajo el znode */assigns*.

Esperar tiempo prudencial y comprobar tareas pendientes En el hilo de ejecución principal, el maestro se dedica a monitorear el estado del sistema y corregir posibles inconsistencias que puedan surgir. Por ejemplo, en el lapso de tiempo en que el Master-WMS está inactivo (desde que falló hasta que un maestro secundario toma su lugar) pueden ocurrir fallos en Workers que no serán detectados. En este caso habrán tareas que no tengan un Worker asignado y deberán ser reasignadas.

En el momento que no existan tareas pendientes en el znode */tasks*, el Master-WMS finaliza su ejecución. El período del ciclo de verificación es configurado por el administrador del sistema.

Reasignar tareas sin Worker asignado Cuando el ciclo de verificación del Master-WMS detecta tareas pendientes no asignadas, se produce la reasignación de estas tareas. El mecanismo de reasignación que realiza el maestro es el mismo que el de asignación.

4.2.4. Algoritmo de asignación de tareas

Una vez relevados los recursos computacionales disponibles en OpenNebula y determinados los recursos necesarios para ejecutar la carga de trabajo, el Master-WMS debe asignar a cada recurso un subconjunto de tareas a ejecutar. El algoritmo de asignación de tareas presentado en esta sección busca optimizar la asignación con el objetivo de reducir el tiempo total de ejecución de la carga de trabajo.

Una carga de trabajo compuesta por tareas MCell+FERNET, puede contener simulaciones con diversos costos de procesamiento dependiendo de su configuración. Por ejemplo, en una misma carga de trabajo pueden existir tareas que insumen 10 minutos en ejecutar y otras que insumen 12 horas. A su vez, la capacidad de procesamiento de los Workers puede variar debido a las características del hardware (e.g. memoria y CPU) donde se ejecutan. Por este motivo, una tarea puede insumir mayor o menor tiempo de procesamiento dependiendo del Worker que la ejecute.

La distribución de la carga de trabajo podría realizarse dinámicamente si cada Worker toma una tarea pendiente, la ejecuta y repite el ciclo mientras existan tareas pendientes. Esta estrategia es efectiva en escenarios donde la capacidad de procesamiento de los Workers y el costo de las tareas sean razonablemente homogéneos, ya que es indiferente qué Worker toma cada tarea para el tiempo total de ejecución de la carga de trabajo.

En el escenario presente en esta investigación los costos de las tareas son diversos, por este motivo se optó por la implementación de un algoritmo de asignación que tuviese en cuenta los factores antes mencionados (variabilidad de costos de tareas y capacidad de ejecución de Workers) y retorne una asignación de tareas específica para cada Worker.

Como indicador de la capacidad de ejecución de un Worker, se tomó la frecuencia del procesador debido a que la velocidad de CPU es uno de los factores de hardware que más incide en el tiempo final de ejecución de una tarea. Esto se concluyó experimentalmente, observando los tiempos de ejecución de MCell y FERNET al variar los recursos de hardware (CPU y memoria). En los experimentos realizados, se comprobó que dos Workers que compartían CPU tardaban el doble de tiempo en realizar la ejecución de sus tareas, en relación a una ejecución sin compartir

CPU. Por otro lado, se comprobó que la memoria no es un factor influyente en el tiempo de ejecución de una simulación, a menos que sea muy acotada (inferior a 512MB). La velocidad de CPU de los recursos de hardware disponibles es consultada mediante la API de OpenNebula.

El algoritmo implementado para resolver la asignación, recibe como entrada una lista de Workers con su respectiva capacidad de ejecución y una lista de tareas con su respectiva ponderación. La ponderación del costo de una tarea se desligó del problema de asignación, y se resuelve de manera independiente a través del Weight Estimator (se detalla en la sección 4.2.6). El algoritmo de asignación implementado comienza por calcular el costo de ejecutar cada tarea en cada Worker, mediante el cociente entre el costo de la tarea y la capacidad del Worker. El resultado se almacena en una matriz de costos donde cada fila hace referencia a un Worker y cada columna a una tarea, y en cada entrada de la matriz se registra el costo de ejecutar la tarea en el Worker. Finalmente se ordenan las columnas de mayor a menor, de esta manera se tiene en las primeras columnas las tareas de mayor costo. El algoritmo 3 describe en alto nivel el procedimiento de asignación de tareas a Workers.

Algoritmo 3 Algoritmo de asignación de tareas

Entrada: lista de tareas: *tareas*, lista de Workers: *workers*, matriz de costos de asiganción worker-tarea: *costos*
Salida: diccionario worker-tareas asignadas: *asignaciones*

```
for worker in workers do
    worker.totalCostoAsignado = 0
end for
ordenarPorCosto(tareas) {Orden por costo descendente}
for tarea in tareas do
    totalCostoWorker = MAX_INT
    for worker in workers do
        totalAsignado = worker.totalCostoAsignado + costos[worker][tarea]
        if totalAsignado ≤ totalCostoWorker then
            totalCostoWorker = totalAsignado
            workerAsignado = worker
        end if
    end for
    workerAsignado.totalCostoAsignado = totalCostoWorker
    asignaciones.add(workerAsignado, tarea)
end for
```

El problema de asignación de recursos que se presenta en este informe, se conoce en investigación operativa como *Imbalanced Time Minimizing Assignment Problem* (ITMAP) o *Imbalanced Bottleneck Assignment Problem*. ITMAP es un problema de optimización donde se busca minimizar el cuello de botella, lo que significa que el Worker que más tiempo demore en completar su conjunto de tareas asignadas demore lo menos posible. El objetivo del algoritmo es minimizar el tiempo total de ejecución del conjunto de tareas que conforman la carga de trabajo.

La solución analítica al problema de asignación ITMAP puede ser expresada mediante un sistema de ecuaciones. Se define un conjunto de tareas J y un conjunto de Workers I (expresado en la ecuación (4.4)). Para una tarea $j \epsilon J$ y un Worker $i \epsilon I$, la asignación está dada por $x_{ij} = 1$, y el costo de la asignación está representado por la variable t_{ij}. La ecuación (4.1) es la función objetivo, y representa la minimización del cuello de botella. El modelo matemático se completa con las ecuaciones (4.2) y (4.3). La restricción de que una tarea puede ser ejecutada por un único Worker está representada en la ecuación (4.2) y la restricción de que cada Worker debe ejecutar al menos una tarea está representada en la ecuación (4.3).

$$min \quad T(X) = max_{i \epsilon I} \left(\sum_{j=1}^{n} (t_{ij} : x_{ij} > 0) \right) \tag{4.1}$$

$$sujeto\ a : \sum_{i \epsilon I} x_{ij} = 1, j \epsilon J \tag{4.2}$$

$$\sum_{j \epsilon J} x_{ij} \geq 1, i \epsilon I \tag{4.3}$$

$$x_{ij} \epsilon \{0, 1\}, (i, j)\ \epsilon\ I \times J, |I| = m, |J| = n \tag{4.4}$$

El problema de asignación ITMAP se encuentra desarrollado en profundidad en el artículo del departamento de matemáticas del *IIT Delhi* [4], donde se presenta una explicación más detallada del contexto del problema.

La implementación del algoritmo que resuelve el problema ITMAP quedó por fuera del alcance de este proyecto, pero es un buen punto de mejora que se considera como trabajo futuro.

Otra solución analizada fue la implementación de un algoritmo que resuelva el problema de asignación utilizando la técnica BackTracking [1]. En esta solución, el algoritmo realiza el BackTracking sobre las tareas, asignando en cada iteración un Worker diferente para ejecutarla. Cuando se alcanza el punto donde todas las tareas tienen un Worker asignado, se calcula el costo total de la asignación resultante. Si el costo de la solución encontrada es menor a la mejor solución encontrada hasta el momento, la nueva solución pasa a ser la mejor solución parcial del proceso de BackTracking. Cuando se finalizan todas las posibles asignaciones, la solución optima es la solución que minimiza la asignación.

El problema del algoritmo de BackTracking es el tiempo de resolución que presenta al escalar en la cantidad de Workers o tareas. En una ejecución del sistema con n Workers y m tareas, el orden del algoritmo es n^m. Por este motivo, el tiempo de ejecución crece exponencialmente a medida que la cantidad de Workers y tareas aumenta, siendo inviable la aplicación de esta técnica en el marco de este proyecto.

4.2.5. Procesos Worker

Los procesos Worker son clientes ZooKeeper que realizan el procesamiento de las simulaciones MCell+FERNET. Estas simulaciones son ejecutadas por los Workers en modo *pipe*; de esta forma la salida de MCell es consumida por FERNET a medida que es generada, evitando la generación de archivos intermedios de gran tamaño (se detalla en la sección 2.5.3). Los Workers ejecutan sobre máquinas virtuales en las instancias de OpenNebula configuradas. Una máquina virtual puede alojar más de un proceso Worker (la cantidad de Workers por máquina es configurable).

Los Workers son creados por el Master-WMS en función de los recursos de CPU disponibles y las tareas que se recuperan de la base de datos para ejecutar. El Master-WMS es el encargado de finalizar la ejecución de los Workers, destruyendo las máquinas virtuales donde residen una vez completada la ejecución de las tareas.

Los Workers pueden sufrir fallos internos que impidan que continúe su ejecución. A su vez, pueden existir fallos en la red que impidan la comunicación con alguno de los componentes del sistema (e.g. servidores ZooKeeper). En la sección 4.2.9.4 se explica cómo procede el Master-WMS cuando ocurren fallos en Workers.

En la figura 4.7 se presenta el flujo de actividad completo de un Worker activo del sistema. A continuación se describen cada una de las actividades.

Iniciar El Worker comienza estableciendo una sesión con los servidores ZooKeeper y creando un znode efímero con su identificador bajo el znode */workers*. Este znode representa el estado activo del Worker y es destruido por ZooKeeper cuando el proceso termina o caduca la sesión.

Recuperar tareas asignadas El Worker toma las tareas que le fueron asignadas por el Master-WMS del znode */assigns/worker-id* y las mantiene en memoria. Luego, coloca un *watch* en este znode para ser notificado en caso de recibir nuevas asignaciones.

Recibir notificación de nueva tarea Cada vez que una nueva tarea es asignada al Worker se dispara el *watch* previamente registrado en el znode */assigns/worker-id*. El procesamiento del *watch* consiste en recuperar las asignaciones y actualizar la lista mantenida en memoria.

Como los *watches* son disparados una única vez, es posible registrar un nuevo *watch* al mismo tiempo que una operación es invocada. El Worker utiliza este mecanismo para obtener las asignaciones y registrar un nuevo *watch* que le permita ser notificado de nuevas asignaciones.

Recuperar datos de tarea de Storage Una vez que el Worker obtuvo sus tareas asignadas, procede a tomar una de las tareas y recuperar los datos del Data Storage necesarios para la ejecución de la simulación (archivos de configuración para MCell y FERNET).

Ejecutar MCell+FERNET Cada tarea consiste en la ejecución MCell+FERNET con los archivos de configuración correspondientes. En paralelo a la ejecución de MCell+FERNET, el Worker ejecuta una rutina que persiste periódicamente los resultados parciales de la ejecución en el Data Storage (e.g. el progreso de la tarea).

En la configuración del Worker es posible definir la cantidad máxima de tiempo que puede permanecer una simulación sin mostrar actividad (i.e. la cantidad de tiempo que puede estar MCell sin escribir en la salida estándar). Una vez alcanzado este tiempo, el Worker cambia el estado de la tarea a *Demorada* e incrementa un contador de timeouts. Cuando este contador alcanza un valor determinado (también configurable) se detiene la ejecución de la tarea y se actualiza el estado a *Error*. Además, la ejecución de una tarea puede ser detenida debido a una cancelación explícita por parte del usuario o a un error interno de MCell o FERNET.

Persistir resultado en el Storage Una vez que la tarea finalizó su ejecución de manera satisfactoria el Worker persiste los archivos de salida generados por FERNET en el Data Storage,

elimina el znode de la tarea bajo */tasks* y elimina el znode de la asignación */assigns/worker-id/task-id*. Luego, comienza con la ejecución de la próxima tarea. En caso de no haber tareas pendientes, el proceso espera por nuevas asignaciones.

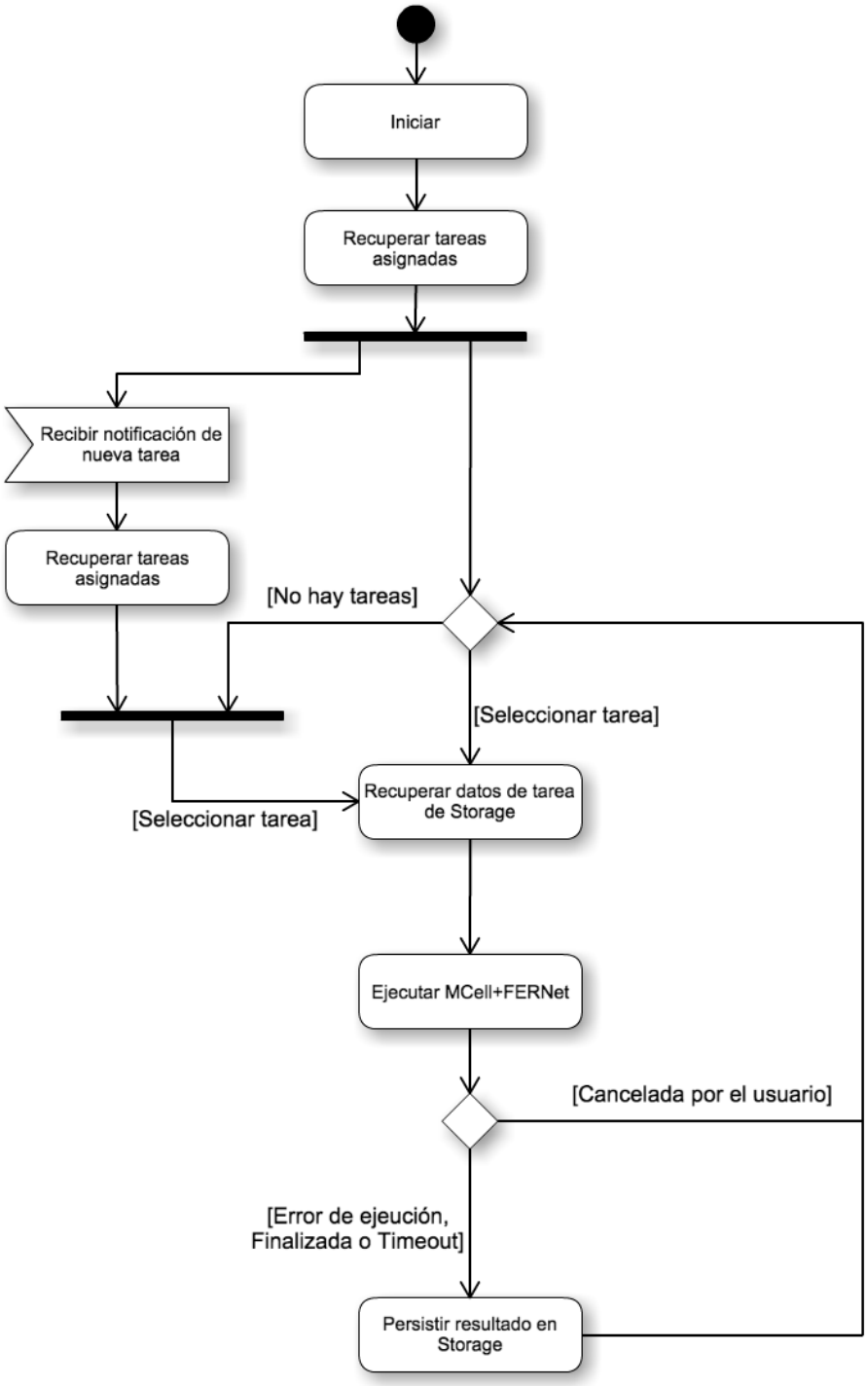


Figura 4.7: Diagrama de actividad del Worker

4.2.6. Weight Estimator

El algoritmo de asignación de simulaciones utilizado por el Master-WMS (se detalla en la sección 4.2.4) requiere que el tiempo de ejecución de las simulaciones haya sido estimado con anterioridad. El Weight Estimator es el componente del sistema encargado de estimar el tiempo de ejecución de las simulaciones.

La estimación del tiempo de ejecución de las simulaciones depende directamente de los valores ingresados en los parámetros del archivo de configuración de MCell. Durante la ejecución de una simulación la velocidad con la que FERNET consume datos del pipe es mayor a la que MCell los genera. Por lo tanto, el tiempo de procesamiento que añade FERNET al tiempo total de ejecución de la simulación es despreciable a los efectos de la estimación. Los parámetros de MCell que más afectan la duración de una simulación son el número de moléculas que se liberan, la cantidad de iteraciones definidas para la simulación, la geometría de los objetos contenedores de las moléculas y la cantidad de moléculas por subvolumen de cómputo (densidad) [26]. Experimentalmente se comprobó que la cantidad de iteraciones en una simulación afecta al tiempo de ejecución de manera proporcional en muchos casos (esto varía en función del resto de los parámetros de MCell).

El Weight Estimator ejecuta periódicamente un proceso encargado de recuperar las simulaciones ingresadas en la base de datos del sistema y ponderarlas estimando su tiempo de ejecución. Una vez que el proceso obtiene la ponderación actualiza el estado de la simulación a *Pendiente*, quedando lista para ser ejecutada.

Para estimar el tiempo de ejecución de una simulación, el Weight Estimator realiza la ejecución de una cantidad reducida (i_e) de iteraciones de la simulación. Al finalizar i_e iteraciones el tiempo de ejecución medido (t_e) es utilizado para estimar el tiempo de la simulación (T) en función de la cantidad real de iteraciones (I) mediante la ecuación 4.5. La ecuación 4.5 refleja una proporcionalidad lineal entre en el número de iteraciones de una simulación y el tiempo de ejecución.

$$T = \frac{t_e \times I}{i_e} \tag{4.5}$$

Durante el proceso de estimación, es posible que una simulación falle o tarde más de un tiempo máximo (configurable) en ejecutar el subconjunto de iteraciones definido. Cuando el Weight Estimator detecta una falla o se excede el limite de tiempo configurado asigna una ponderación por defecto a la simulación.

La estrategia de estimación del Weight Estimator no involucra todos los parámetros de la extensa configuración de MCell que pueden influir en el tiempo final de ejecución. Por lo tanto, no siempre se genera una estimación aproximada al tiempo real. Como se mencionó anteriormente, existen parámetros que pueden incidir en que la proyección lineal de la ecuación 4.5 no sea efectiva.

Se propone como trabajo futuro la incorporación de los parámetros más relevantes de MCell al proceso de estimación. Para mejorar la calidad de las estimaciones se podría utilizar un historial donde se registre los parámetros más relevantes y el tiempo de ejecución de cada simulación. Este historial puede ser utilizado para realizar interpolaciones multidimensionales o métodos de aproximación que permitan estimar (analíticamente) el tiempo de ejecución de futuras simulaciones.

4.2.7. Broker

El Broker es el componente dedicado a la administración de los pedidos de simulaciones realizados por los usuarios a través del Web Portal. En la arquitectura del sistema, el Broker se ubica en la capa de lógica transaccional.

4.2.7.1. Creación de simulaciones

Los usuarios acceden al sistema a través del Web Portal para ingresar las solicitudes de simulaciones MCell+FERNET. El ingreso de una simulación requiere que el usuario especifique los parámetros de configuración de MCell y FERNET para que el Broker genere los datos que se ingresarán al Data Storage. La configuración de una simulación se compone de los siguientes parámetros:

- Nombre de la simulación: el nombre para identificar el pedido en el listado de simulaciones.
- Time step: el valor de una unidad de paso de tiempo de MCell.
- Cantidad de iteraciones: la cantidad de pasos de tiempo que ejecutará MCell.
- Tamaño de la caja: la definición del espacio donde se desarrollará la simulación.
- Moléculas: los tipos de moléculas involucrados en la simulación y las constantes de difusión que regirán su comportamiento.
- Reacciones: las reacciones que pueden ocurrir en el marco de la simulación. Es posible definir un barrido paramétrico (se detalla en la sección 4.2.7.2) sobre el valor de la constante de difusión, especificando un valor inicial, un valor final y un valor de salto.
- Puntos de liberación: la ubicación en el espacio de la simulación donde se liberarán las moléculas y la cantidad de moléculas liberadas.
- Modo de ejecución de FERNET: el tipo de escaneo que realiza el microscopio simulado y su modo operativo. En el apéndice A.2 se explica en detalle en qué consiste cada modo de ejecución. Los valores posibles son: Point, Line, Multi, Image, Stack y SPIM.
- Configuración de FERNET: la configuración avanzada de FERNET. Cada modo de ejecución posee una configuración particular.

Una vez que los parámetros necesarios son ingresados por el usuario, el Broker valida que la especificación de las simulaciones definidas sea correcta, genera los archivos de configuración para MCell y FERNET e ingresa las simulaciones en el Data Storage con estado *Ingresada*.

4.2.7.2. Barridos paramétricos

Una de las funcionalidades interesantes del sistema es la definición de simulaciones por barrido paramétrico. Esta funcionalidad permite solicitar, con un sólo pedido, la ejecución de un conjunto de simulaciones que compartan la configuración general pero varíen en los coeficientes de difusión de las reacciones definidas.

Para definir un barrido paramétrico se debe indicar un valor inicial, un valor final y un valor de salto para cada constante de difusión. De esta forma, el Broker generará una simulación para cada combinación de constantes de difusión posible. El Broker generará los archivos de configuración MCell y FERNET de cada simulación e ingresará estas simulaciones al Data Storage, vinculándolas a un identificador de pedido único.

Un ejemplo de barrido paramétrico es la definición de una simulación con dos reacciones ($R1$ y $R2$), el rango de valores $[1, 3]$ para el coeficiente de difusión de $R1$ ($cdR1$) y el rango de valores $[1, 2]$ para el coeficiente de difusión de $R2$ ($cdR1$), con un salto de 1 en ambos rangos. Esta especificación generará 6 simulaciones, con $(cdR1, cdR2)$ tomando los valores (1,1), (1,2), (2,1), (2,2), (3,1) y (3,2).

4.2.8. Web Portal

Los pedidos de ejecución de simulaciones ingresan al sistema a través de una aplicación web, denominada Web Portal. Esta aplicación fue implementada utilizando el framework para desarrollo web Java Server Faces 2 (JSF2). Este framework permite desarrollar interfaces de usuario en aplicaciones Java Enterprise Edition. La aplicación web implementada se montó sobre un servidor de aplicaciones JBoss versión 8.2.0 (Wildfly) [11].

Los usuarios del sistema son autenticados mediante nombre de usuario y contraseña como se muestra en la figura 4.8. El portal web le permite a los usuarios autenticados definir simulaciones (y solicitar su ejecución) a través de una interfaz amigable. Al momento de definir una simulación, el usuario puede especificar barridos paramétricos sobre los coeficientes de difusión de las reacciones definidas. En las figuras 4.9 y 4.10 se muestran las pantallas de definición de simulaciones en el portal.

Figura 4.8: Pantalla de autenticación de usuarios.

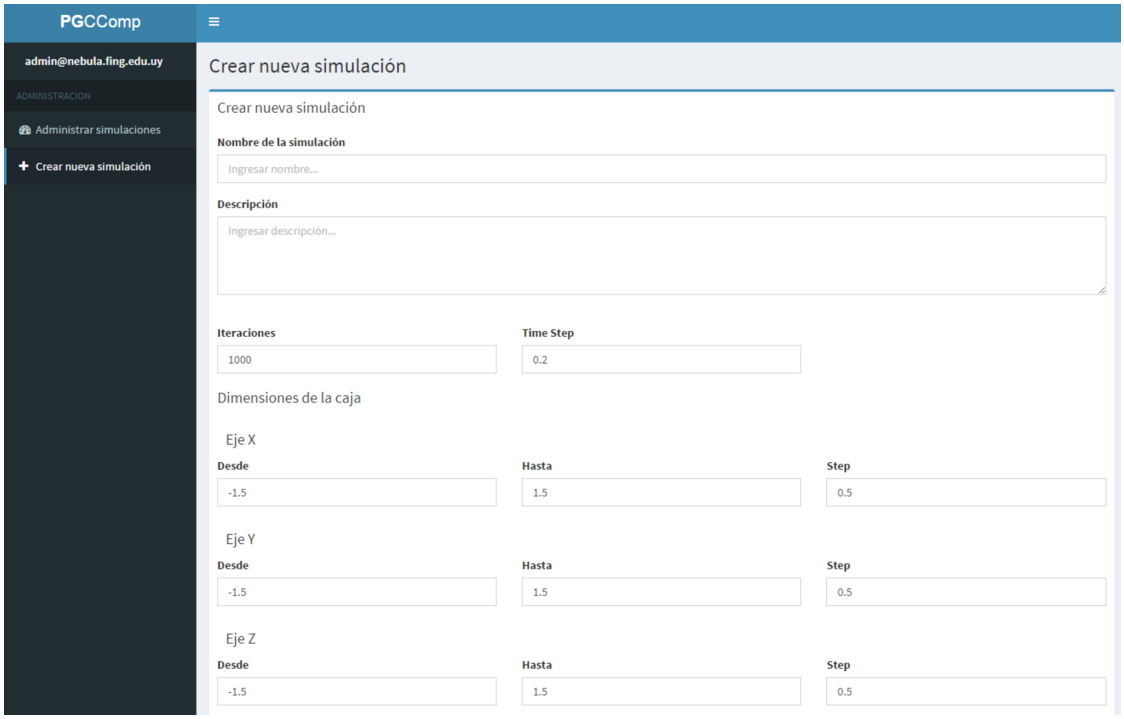


Figura 4.9: Formulario de definición de simulaciones (1/2).

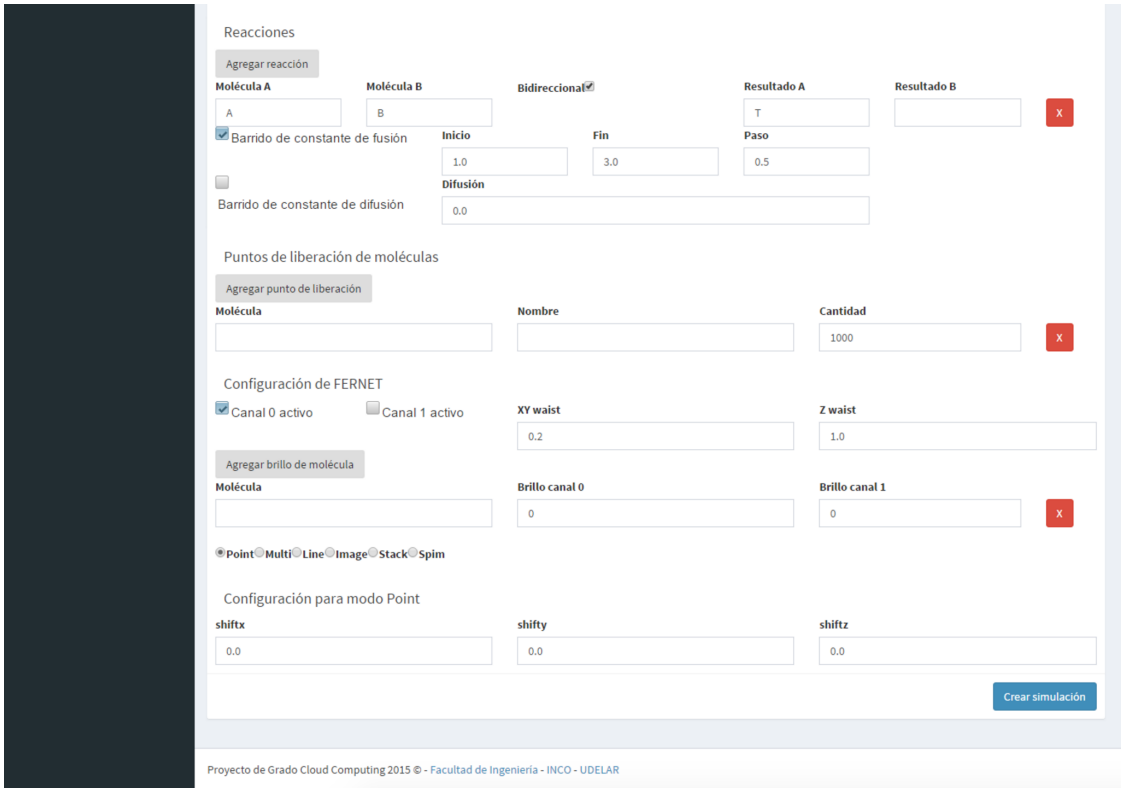


Figura 4.10: Formulario de definición de simulaciones (2/2).

4.2.8.1. Administración de simulaciones

Los usuarios autenticados pueden acceder a una pantalla de administración de las simulaciones definidas. En esta pantalla se listan las simulaciones definidas por el usuario, ordenadas por fecha de creación. Para cada simulación se muestran los siguientes datos:

- Estado (Pendiente, Completada, En ejecución, Finalizada, etc.)
- Fecha de creación
- Progreso
- Tiempo estimado restante (si se encuentra en ejecución)

Un barrido paramétrico es representado con una fila en el listado, al igual que una simulación simple. Las filas que representan barridos paramétricos pueden ser desplegadas para observar los datos de cada simulación generada.

Desde la pantalla de administración de simulaciones es posible descargar los resultados de las ejecuciones. Junto a cada simulación completada se muestra un enlace para descargar el resultado de su ejecución. En la figura 4.11 se muestra la pantalla de administración de simulaciones.

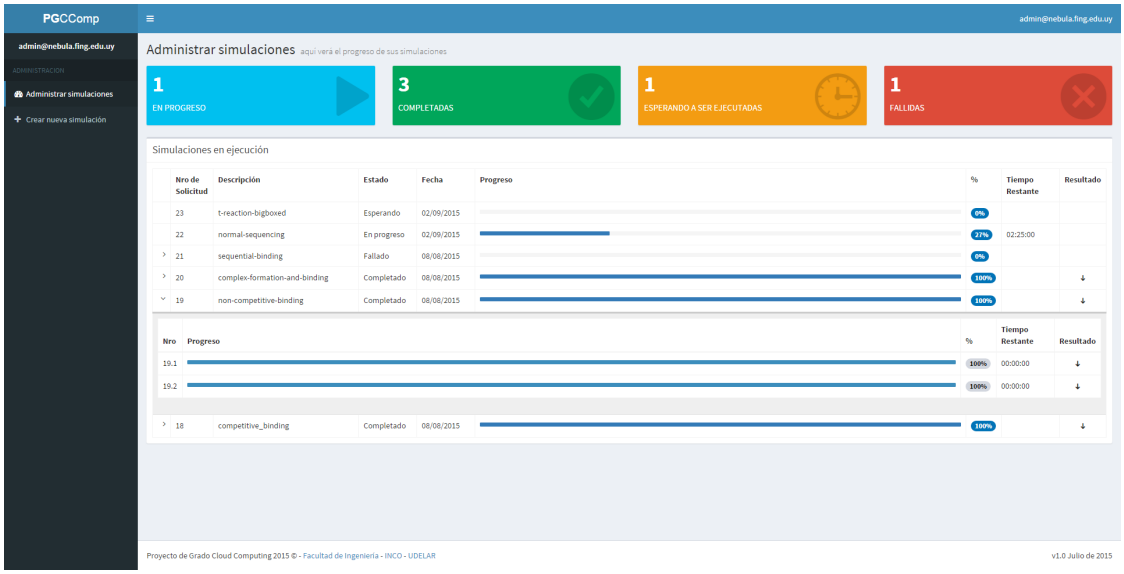


Figura 4.11: Pantalla de administración de simulaciones.

4.2.9. Tolerancia a fallos

En los sistemas distribuidos, los fallos pueden ocurrir más a menudo que en los sistemas centralizados. Tanto los componentes participantes de la solución como la red que se utiliza como canal de comunicación, suponen posibles puntos de falla a tener en cuenta a la hora de implementar un sistema distribuido. En esta sección se describen las técnicas y mecanismos utilizados para la detección y recuperación ante fallos que pueden ocurrir durante el proceso de ejecución en los diferentes componentes del sistema desarrollado.

4.2.9.1. Fallos en servidores ZooKeeper

Los servidores ZooKeeper dan soporte a la coordinación y sincronización de los componentes del sistema, en particular los Workers, el Master-WMS y los maestros secundarios. Cuando se cuenta con varios servidores (i.e. cuando se trabaja en modo quorum), la caída de un servidor

no implica la detención del servicio ZooKeeper. Como se mencionó en la sección 2.3.3, el tamaño del quorum debe superar la mitad de los servidores disponibles; por lo tanto el servicio admite f caídas en un ensemble de n servidores, con $f<n/2$. Para asegurar la continuidad del servicio, es necesario definir el tamaño del quorum y la distribución de los servidores de forma apropiada, basándose en pautas relativas al diseño de ZooKeeper y en la propia experiencia obtenida del sistema.

En este proyecto se resolvió mantener los servidores ZooKeeper en el cluster FING, priorizando la mejora de rendimiento que implica tener los servidores en la misma zona. Los recursos de la UBA se utilizaron únicamente para ejecutar Workers, ya que se dispone de menos control administrativo sobre estos recursos.

Al disponer de un único nodo físico en el cluster FING la tolerancia a fallos se ve disminuida. Si el nodo sufre algún problema de red o de infraestructura (e.g. pérdida de energía), los servidores ZooKeeper fallarán simultáneamente. De todas formas se decidió establecer un ensemble de 3 servidores, ejecutando cada uno en una máquina virtual independiente, para validar el modo quorum y poder redistribuir los servidores fácilmente en el caso de contar con más nodos.

4.2.9.2. Fallos en clientes ZooKeeper

La API para Java de ZooKeeper le permite a los clientes invocar las operaciones en forma sincrónica o asincrónica. Las invocaciones sincrónicas bloquean el hilo de ejecución del cliente hasta recibir una respuesta por parte de ZooKeeper. Las invocaciones asincrónicas son encoladas, liberando el hilo de ejecución del cliente, y enviadas a ZooKeeper por otro hilo de ejecución en el orden en que fueron realizadas (i.e. en orden FIFO). Las respuestas a estas invocaciones son procesadas en un hilo de ejecución independiente (denominado hilo de eventos) donde se ejecutan las operaciones de retorno definidas por el cliente.

En caso de una pérdida momentánea de conexión (i.e. no se alcanza el timeout de expiración de la sesión) las invocaciones asincrónicas son reenviadas automáticamente, sin necesidad de la intervención del cliente. En cambio, las invocaciones sincrónicas no cuentan con este mecanismo de reenvío, pero en algunos escenarios es preferible utilizarlas (e.g. cuando no se necesita realizar procesamiento en paralelo mientras se espera la respuesta de ZooKeeper).

En este proyecto se utilizaron operaciones tanto sincrónicas como asincrónicas. Por lo tanto se debió implementar un mecanismo de reenvío para las operaciones sincrónicas. Se reescribieron las operaciones de ZooKeeper de forma tal que en caso de producirse una excepción por pérdida momentánea de conexión, se reintenta una determinada cantidad de veces (configurable). De esta forma se evita que ante una falla en la comunicación del primer intento, el proceso no reciba un error que puede afectar el transcurso normal de la ejecución.

Los reintentos se realizan únicamente si se recibe una excepción por pérdida de conexión. En cualquier otro caso (respuesta exitosa u otro tipo de excepción) la respuesta de ZooKeeper debe ser procesada por el cliente. Por ejemplo, si en uno de los intentos para crear un znode se obtiene una excepción por pérdida de conexión, pero el znode alcanzó a ser creado, en el siguiente intento se obtendrá una excepción dado que el znode ya existe. En este caso no se debe reintentar, sino que debe ser el cliente quien procese la excepción.

4.2.9.3. Fallos en el Master-WMS

Un fallo en el Master-WMS dejaría al sistema imposibilitado de controlar y coordinar la ejecución de los pedidos. Por este motivo, el sistema cuenta con un conjunto de clientes ZooKeeper (denominados Secondary Master) utilizados como respaldo del proceso maestro, que pueden ser agregados dinámicamente durante la ejecución del sistema.

Los Secondary Master participan de un algoritmo de elección de líder (*leader election*). Este algoritmo (propuesto en [9]) asegura que en todo momento exista un sólo cliente ejecutando con el rol del Master-WMS y que ante su caída, otro proceso asuma este rol.

El algoritmo implementado parte del znode */masters*. Todos los procesos que intentan ejecutar como maestro compiten creando un znode secuencial y efímero bajo */masters*. El proceso que logra crear el znode con menor número de secuencia, asume el rol de Master-WMS del sistema,

mientras que el resto de los procesos asumen el rol de Secondary Master. Cada proceso que asume el rol de Secondary Master coloca un *watch* en el znode con el mayor número de secuencia anterior al suyo y procede a ejecutar en modo pasivo. Ante la caída del propietario del znode observado, el proceso es notificado y deberá determinar nuevamente qué rol asumir. En la figura 4.12 se muestra un ejemplo de la estructura de znodes definida.

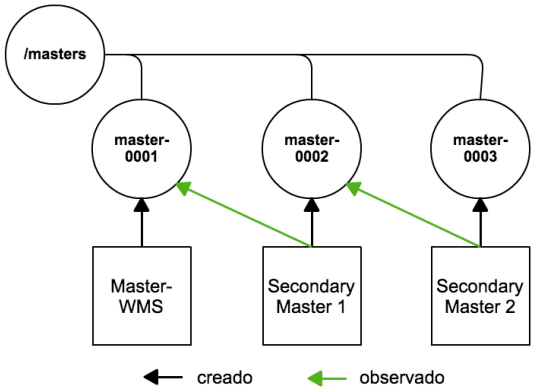


Figura 4.12: Estructura de znodes para el algoritmo de elección de líder.

Una variante simplificada del algoritmo de elección de líder consiste en que todos los procesos coloquen un *watch* en el znode con menor número de secuencia, y ante el borrado de este znode (debido a la caída del líder) realicen el chequeo correspondiente para determinar el nuevo líder. Sin embargo, esta variante implica que la notificación de la caída del líder sea recibida por todos los clientes. Si la cantidad de clientes es alta se generea una pérdida de rendimiento, ya que cada cliente debe obtener los hijos del znode */masters* invocando la operación *getChildren* (la operación más costosa de ZooKeeper).

4.2.9.4. Fallos en los Workers

Los Workers del sistema pueden sufrir fallos relacionadas a la ejecución de las simulaciones o a la comunicación con otros componentes (e.g. servidores ZooKeeper). Los fallos en las simulaciones son capturados y comunicados al usuario mediante un cambio de estado en la tarea, permitiendo que el Worker siga ejecutando. En cambio, los fallos de comunicación con otros componentes del sistema pueden implicar que el Worker termine su ejecución. El Master-WMS debe ser capaz de detectar la finalización abrupta de los Workers e intervenir para minimizar las consecuencias.

Las notificaciones a través de *watches* pueden ser utilizadas para detectar la caída de un cliente ZooKeeper. La creación de un znode efímero permite que los clientes que hayan colocado un *watch* sobre este znode sean notificados ante la caída del propietario del znode. Esta técnica se aplica para que el proceso Master-WMS sea notificado ante caídas de los procesos Worker del sistema.

El Master-WMS coloca un *watch* en el znode */workers*, y cada Worker crea un znode efímero bajo */workers* al iniciar. De esta forma, cuando un Worker falla, su znode efímero es destruido por ZooKeeper y el Master-WMS es notificado (ya que se produjo un cambio en el znode */workers*). Al recibir la notificación, el Master-WMS reasigna las tareas de los Workers que fallaron a los Workers activos.

Capítulo 5

Análisis experimental

Este capítulo describe el análisis experimental realizado sobre los diferentes componentes que integran el sistema construido. Los experimentos realizados abarcan el estudio y evaluación de los algoritmos desarrollados para la estimación de costos y asignación de simulaciones, así como los mecanismos de tolerancia a fallos implementados.

5.1. Cargas de trabajo

La carga de trabajo utilizada para el análisis experimental se basa en las simulaciones definidas en un trabajo previo de investigación de simulaciones de MCell y FERNET sobre OurGrid [7]. Estas simulaciones están inspiradas en el estudio de moléculas especiales llamadas *transcription factor* (TF) y su interacción con otro tipo de moléculas llamadas *binding site* (B).

Se definieron tres escenarios distintos, clasificados por su configuración en pequeño (p), mediano (m) y grande (g), haciendo referencia al número de moléculas liberadas al comienzo de cada simulación (ver tabla 5.1). Para cada uno de estos escenarios se definieron 8 instancias en las que se varían los parámetros de MCell que definen las dimensiones y características del volumen que contiene las moléculas. En total, se trabaja con 24 simulaciones distintas.

parámetro	**descripción**	**p**	**m**	**g**
$\#TF$	Número de moléculas TF iniciales	415	1660	4150
$\#B$	Número de moléculas B iniciales	1000		
$\#[TF$ - $B]$	Número de moléculas $[TF$ - $B]$ iniciales	85	340	850
D_{TF}	Coeficiente de difusión para TF	$55 \times 10^{-8}\ cm^2/s$		
D_B	Coeficiente de difusión para B	0		
$D_{[TF-B]}$	Coeficiente de difusión para $[TF - B]$	0		
k_{on}	Constante de control de unión de TF y B	$1{,}7 \times 10^6\ M^{-1}s^{-1}$		
k_{off}	Constante de control de desvinculación de TF y B	$5\ s^{-1}$		
$TIME\_STEP$	Paso de tiempo preferido para integración numérica	1×10^{-5}		

Tabla 5.1: Parámetros configurados para cada escenario definido.

En la tabla 5.2 se puede observar un resumen de la configuración de MCell utilizada en cada una de las 24 instancias definidas y su tiempo de ejecución secuencial en el cluster FING. El tiempo total de procesamiento para una ejecución secuencial de las 24 instancias es aproximadamente 10.81 horas, dedicando 2 núcleos de CPU para la ejecución de MCell+FERNET.

escenario	arista	nro. instancia	intervalo	tiempo de ejecución (s)
1	1.5	1	0.05	210
		2	0.10	177
		3	0.20	217
		4	0.50	1030
	3.0	5	0.05	318
		6	0.10	214
		7	0.20	198
		8	0.50	362
2	1.5	9	0.05	765
		10	0.10	672
		11	0.20	809
		12	0.50	4068
	3.0	13	0.05	1160
		14	0.10	780
		15	0.20	712
		16	0.50	1500
3	1.5	17	0.05	2044
		18	0.10	1715
		19	0.20	2002
		20	0.50	9796
	3.0	21	0.05	2271
		22	0.10	2002
		23	0.20	1596
		24	0.50	3798

Tabla 5.2: Detalle de las instancias del experimento y tiempo de ejecución individual [7].

Las instancias descritas en la tabla 5.2 son utilizadas en los experimentos realizados en este capítulo, trabajando sobre una configuración del sistema habilitada para utilizar hasta 8 núcleos del cluster FING, con un máximo de 2 Workers por máquina virtual, y fijando en 2 la cantidad de núcleos requeridos por cada Worker. Para realizar la ejecución de las 24 simulaciones partiendo de la configuración escogida, el sistema crea 2 máquinas virtuales con 2 Workers cada una, utilizando un total de 8 núcleos de CPU en el cluster FING (ya que cada Worker requiere 2 núcleos).

5.2. Experimentos realizados

Los experimentos que se describen en esta sección fueron realizados en el ambiente de ejecución construido sobre OpenNebula en el cluster FING. El objetivo de estos experimentos es validar el sistema construido mediante un análisis experimental sobre sus principales funcionalidades.

5.2.1. Análisis de eficiencia en el algoritmo de asignación

El objetivo del experimento que se presenta en esta sección es mostrar la mejora que se obtiene en el tiempo total de ejecución al utilizar el algoritmo de asignación implementado (descrito en la sección 4.2.4). Se compara el algoritmo implementado con un algoritmo de distribución estática uniforme, que le asigna a cada Worker la misma cantidad de tareas (sin tomar en cuenta el costo relativo de cada tarea ni las características de hardware de cada Worker).

Aplicar una distribución estática uniforme es un criterio razonable si no se conoce previamente el costo de las simulaciones. Esta distribución genera buenos resultados si las simulaciones tienen tiempos similares de procesamiento. De lo contrario, se podría sobrecargar a un subconjunto de Workers y dejar al resto ociosos.

Para realizar una evaluación y determinar cómo afecta el algoritmo de asignación de tareas aplicado por el Master-WMS al el tiempo total de ejecución, se ejecuta el sistema en dos oportunidades con los mismos recursos y la misma carga de trabajo, variando el algoritmo de asignación en cada ejecución. Para ambas ejecuciones se utilizaron 4 Workers y una carga de trabajo conformada por las 24 instancias descritas en 5.2.

La tabla 5.3 contiene las asignaciones realizadas por ambos algoritmos, y la figura 5.1 muestra una gráfica comparativa del tiempo de procesamiento empleado por cada Worker para la ejecución de las simulaciones asignadas por ambos métodos. Los Workers realizan el procesamiento en forma paralela, por lo tanto el tiempo total de la ejecución de las 24 instancias está dado por el Worker que demore más tiempo en ejecutar sus simulaciones.

algoritmo	worker	instancias asignadas	tiempo(h)	total(h)
Optimizado	1	20	2.8	3.63
	2	10,12,5,22,4,15,8,7	2.83	
	3	1,24,11,23,2,17,16,6	3.63	
	4	9,14,13,3,21,19,18	3.29	
Distribución uniforme	1	17,24,16,6,21,3	3.37	4.7
	2	10,9,18,5,4,15	1.68	
	3	19,14,23,13,20,7	4.7	
	4	1,12,11,22,8,2	2.3	

Tabla 5.3: Tiempo de procesamiento por Worker para cada algoritmo de asignación.

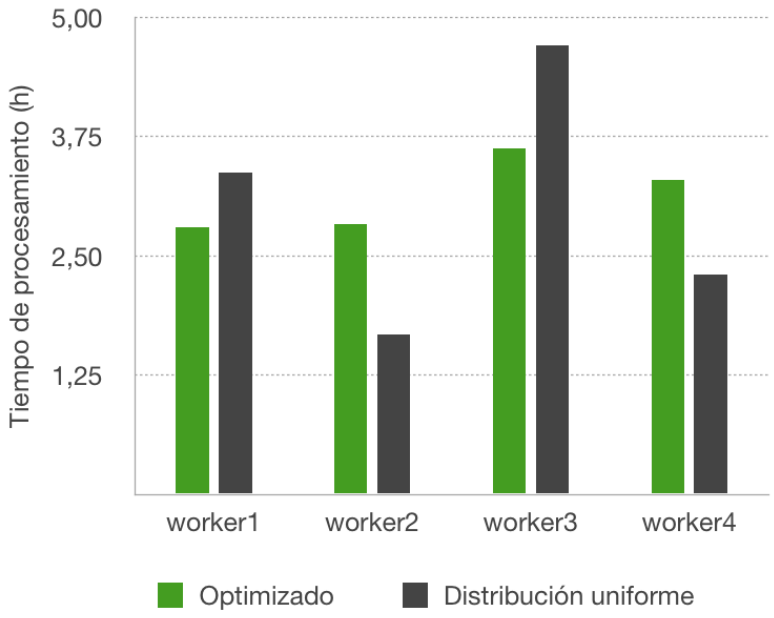


Figura 5.1: Tiempo de procesamiento de los algoritmos de asignación.

El Worker número 3 fue el más sobrecargado por el algoritmo de distribución estática uniforme y por el algoritmo implementado, consumiendo un total de 4.7 y 3.6 horas de procesamiento respectivamente. Se observa una mejora de 23.4 % en el tiempo de ejecución aplicando el algoritmo implementado sobre el algoritmo de distribución estática uniforme.

El algoritmo implementado requiere que el tiempo de ejecución de las tareas sea previamente estimado. Por lo tanto, para este experimento se debió ejecutar el Weight Estimator (componente encargado de realizar las estimaciones). El estimador se configuró con 5000 iteraciones por estimación (ver sección 4.2.6).

Las ejecuciones del Master-WMS y del estimador se realiza de forma solapada. Esto significa que mientras el Master-WMS coordina la ejecución de una carga de trabajo, el estimador procesa las nuevas simulaciones que arriben al sistema, conformando la siguiente carga de trabajo. De todas formas, para ser más estrictos en el experimento, se puede sumar el tiempo consumido por el estimador al tiempo total de ejecución de la carga de trabajo. Sumando el tiempo consumido por el estimador (46.5 minutos), se obtiene igualmente una mejora de 6.4 % en el tiempo de ejecución aplicando el algoritmo implementado sobre el algoritmo de distribución estática uniforme.

Otro de los resultados que se desprende del experimento es la mejor distribución de la carga que realiza el algoritmo implementado. La figura 5.1 muestra la diferencia entre el Worker que tarda más y el que tarda menos utilizando ambos algoritmos. Mientras que para el algoritmo implementado la diferencia es aproximadamente de 50 minutos, el algoritmo de distribución estática uniforme muestra una diferencia de 3 horas. Este balance de carga es la clave para minimizar el tiempo de la ejecución aprovechando mejor los recursos disponibles.

5.2.2. Estimación de simulaciones

El proceso de ejecución del sistema se puede resumir en tres etapas: estimar el costo de las simulaciones, asignar las simulaciones a los Workers y por último ejecutarlas. Cada una de estas actividades es realizada por un componente distinto de la solución. Un cambio en la estimación del tiempo de ejecución para las simulaciones afecta directamente a la asignación, debido a que el algoritmo implementado para asignar las simulaciones a los Workers utiliza la estimación como parámetro de entrada. A su vez, la asignación resultante incide en el tiempo total de ejecución del sistema.

El objetivo del experimento que se presenta en esta sección es analizar el comportamiento del sistema cuando se realizan cambios en la configuración del proceso de estimación y evaluar la influencia de estos cambios en el tiempo total de ejecución.

Para estimar una simulación, el estimador implementado ejecuta una cantidad reducida de iteraciones (configurable) y proyecta de manera lineal el tiempo de ejecución a la cantidad real de iteraciones de la simulación (ver sección 4.2.6). Se experimentó variando la cantidad de iteraciones que se ejecutan para realizar la estimación, utilizando los recursos y los escenarios de prueba definidos en la sección 5.1. En la tabla 5.4 se detallan los tiempos obtenidos al variar el número de iteraciones ejecutadas para las estimaciones. Se muestra el tiempo de ejecución del sistema en segundos y el tiempo requerido para realizar las estimaciones de las 24 instancias.

#iteraciones para estimar	t. de estimación (s)	t. de ejecución (s)	t. total (s)
2500	1452	12575	14027
5000	2788	13055	15843
7500	3636	12606	16242
10000	5042	12654	17696

Tabla 5.4: Tiempos de ejecución del sistema variando el número de iteraciones en la estimación

Las figuras 5.2 y 5.3 muestran en una gráfica de columnas la comparativa entre el tiempo de ejecución real y el estimado de cada una de las 24 instancias, utilizando 5000 y 10000 iteraciones para estimar. En las figuras 5.4 y 5.5 se grafica el porcentaje de error entre la estimación realizada y el tiempo real de ejecución para cada una de las instancias.

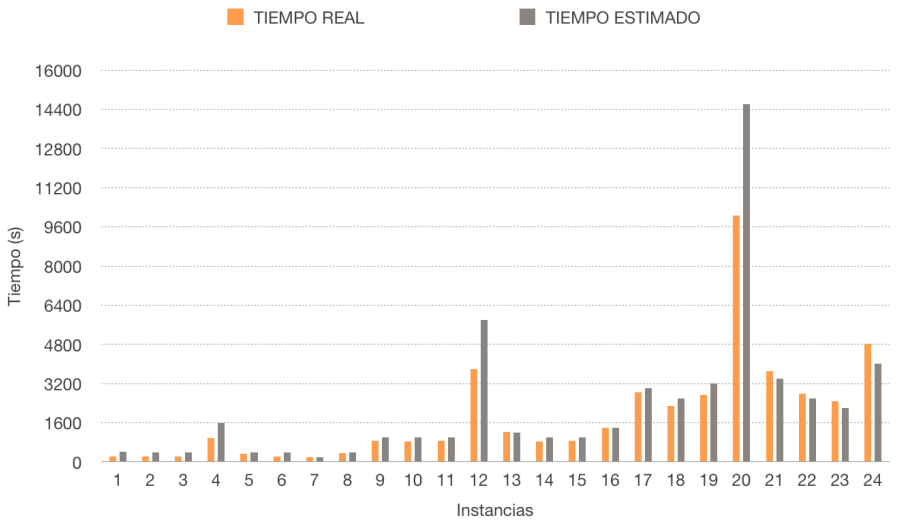


Figura 5.2: Comparativa tiempo de ejecución y estimación (5000 iteraciones).

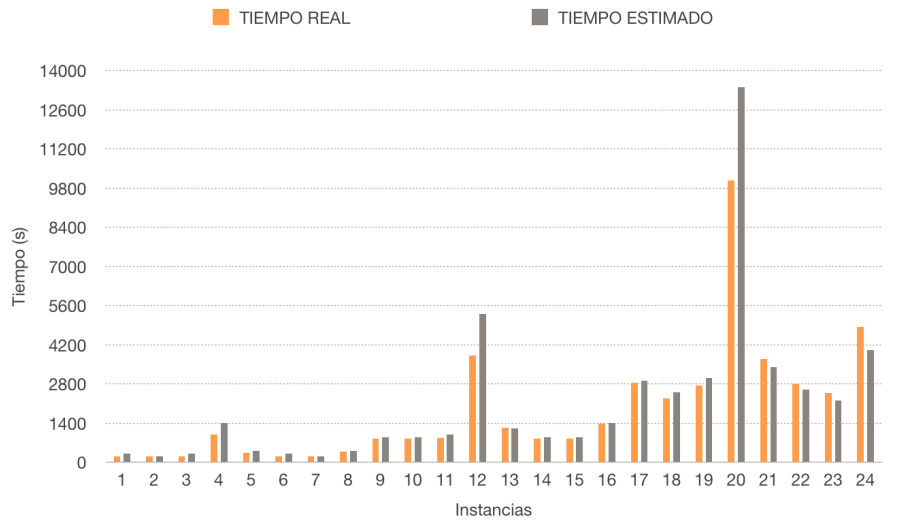


Figura 5.3: Comparativa tiempo de ejecución y estimación (10000 iteraciones).

Figura 5.4: Error absoluto entre el tiempo de ejecución y estimación (5000 iteraciones).

Figura 5.5: Error absoluto entre el tiempo de ejecución y estimación (10000 iteraciones).

Intuitivamente se puede suponer que aumentar las iteraciones utilizadas para estimar mejoraría la calidad de la estimación y en consecuencia el tiempo total de ejecución del sistema. Sin embargo, se verificó una mejora en la calidad de la estimación que no se trasladó al tiempo final. Por ejemplo, el tiempo de ejecución de las instancias estimadas con 2500 iteraciones fue de 12575 segundos, mientras que el tiempo de ejecución de las instancias estimadas con 10000 iteraciones fue de 12654 segundos (ver tabla 5.4). Por otro lado, al aumentar el número de iteraciones del estimador aumenta considerablemente el tiempo de estimación. Por ejemplo, el tiempo de estimación con 2500 iteraciones fue de 1452 segundos y el tiempo de estimación con 10000 iteraciones fue de 5042 segundos, empleando 3590 segundos adicionales. Para obtener mejoras en el tiempo total, el aumento del tiempo de estimación debería ser menor a la mejora lograda en el tiempo de ejecución.

Los escenarios definidos en la sección 5.1 varían en el número de moléculas liberadas. Dentro de cada escenario se definieron 8 instancias que varían en el tamaño de la estructura contenedora de las moléculas (definido a partir del parámetro *edge size*) y de la malla (definido a partir del parámetro *step size*). La estructura contenedora es dividida por MCell en subvolúmenes de cómputo (la cantidad de subvolúmenes se calcula como $(2 + edge\ size/step\ size)^3$).

Las simulaciones 4, 12 y 20 corresponden a la cuarta instancia de cada escenario, presentando la misma cantidad de subvolúmenes (125) y diferente cantidad de moléculas liberadas. Se observó que estas simulaciones son las que presentan menor cantidad de subvolúmenes (se deduce a partir de la tabla 5.2), mayor tiempo de ejecución (ver figuras 5.2 y 5.3) y mayor error de estimación (ver figuras 5.4 y 5.5) dentro de cada escenario [1]. Estas observaciones denotan la importancia que tienen los parámetros de MCell relacionados al volumen que contiene las moléculas. El tamaño de la estructura contenedora, la partición en subvolúmenes de cómputo y la densidad de moléculas

[1] La mayoría de las instancias del escenario 1 presentan un error de estimación alto (mayor al 50 %), similar al de la instancia 4. Esto puede deberse a que son simulaciones de muy bajo tiempo de ejecución (del orden de minutos).

dentro de cada subvolúmen, son parámetros que impactan directamente en el tiempo de ejecución de una simulación. Por lo tanto, la calidad de las estimaciones podría ser mejorada incorporando estos parámetros al proceso de estimación, realizando un estudio en profundidad de cómo afectan al tiempo de ejecución.

5.2.3. Recuperación ante fallos

Desde el momento en que el sistema desarrollado entra en funcionamiento, es necesario gestionar de forma adecuada los fallos que puedan ocurrir. La naturaleza distribuida y los diversos componentes que integran el sistema implican que son varios los puntos a considerar para evitar que surjan inconvenientes. En esta sección se describen y analizan algunos de los posibles escenarios de fallos que se pueden presentar en el sistema desarrollado, y se estudia cómo son manejados para asegurar la confiabilidad y disponibilidad del sistema.

5.2.3.1. Falla del Master-WMS

El Master-WMS es el componente que realiza la distribución de las cargas de trabajo y la gestión de los fallos de los Workers. Al culminar el proceso de asignación de tareas, el Master-WMS inicia un ciclo de control que es realizado periódicamente con el objetivo de detectar y corregir inconsistencias que puedan generarse como consecuencia del fallo de un Worker (e.g. una tarea que no fue reasignada).

El mecanismo utilizado para tolerar fallos en el Master-WMS consiste en mantener ejecutando paralelamente distintas instancias del proceso Secondary Master. Las instancias del proceso Secondary Master están alertas a posibles fallos que puedan ocurrir en el Master-WMS para tomar su lugar y asegurar la continuidad de la ejecución del sistema. En la sección 4.2.9.3 se describe detalladamente el mecanismo utilizado para tolerar los fallos del Master-WMS.

En el experimento presentado en esta sección se busca analizar el tiempo que el Master-WMS permanece fuera de servicio y el tiempo que le insume a un maestro secundario tomar el rol de primario. Los recursos y la configuración utilizada en el experimento es la descrita en la sección 5.1. Para analizar la tolerancia a fallos del sistema ante una caída del proceso Master-WMS, se ejecutó un maestro secundario en una máquina virtual con las mismas características de la máquina virtual donde se ejecuta el Master-WMS.

En la primer parte del experimento, se escogió un conjunto de simulaciones compuesto por las instancias 1, 2, 3, 5, 6, 7, 8, 9, 10 y 15 descritas en la tabla 5.2. Este conjunto de instancias se ejecutó en 5 ocasiones diferentes. En cada una de las ejecuciones se forzó la detención del Master-WMS en distintos momentos y se midió el tiempo que tardó el maestro secundario en ocupar el rol de primario. Luego, se realizaron otras 5 ejecuciones del mismo conjunto de instancias pero sin detener el Master-WMS para comparar los valores promedio del tiempo total de ejecución.

Los resultados obtenidos en la primer parte del experimento se pueden observar en la tabla 5.5. El tiempo promedio de las 5 ejecuciones con fallo del Master-WMS fue de 1272.8 segundos y el tiempo promedio de las 5 ejecuciones sin detener el Master-WMS fue de 1272 segundos. La diferencia entre los promedios es prácticamente nula, ya que una vez que el Master-WMS recupera las simulaciones pendientes de ejecución y las asigna a los Workers, se dedica a realizar tareas de verificación y control periódicamente. La carga de trabajo real de las ejecuciones se realiza sobre los procesos Workers y no es interrumpida ante un fallo del Master-WMS.

tiempo hasta falla (s)	tareas completadas hasta falla	tiempo de recuperación (s)	tiempo total de ejecución (s)
300	1 de 11	30.1	1299
500	2 de 11	30.0	1296
700	2 de 11	30.0	1286
900	3 de 11	30.2	1275
1200	9 de 11	30.1	1208

Tabla 5.5: Parte 1 del experimento para el análisis de fallos del Master-WMS. Tiempos de recuperación del Master-WMS en 5 ejecuciones distintas sobre las mismas simulaciones.

El tiempo transcurrido entre la caída del Master-WMS y el momento en que el maestro secundario ocupa su lugar fue similar en todas las ejecuciones. Se observa que la recuperación del Master-WMS es independiente del momento que ocurre la falla. Esto se debe a que los factores que más pesan en el tiempo de recuperación son el timeout de la sesión ZooKeeper y la latencia de la red. Cuando la falla se produce, los servidores ZooKeeper dejan de recibir los *hearbeats* del Master-WMS y la sesión expira cuando se alcanza el timeout indicado al iniciar la conexión. En el momento que ocurre el timeout, el maestro secundario es notificado por ZooKeeper y obtiene el rol de Master-WMS. Se verifica en los resultados obtenidos que los tiempos de recuperación medidos (tiempo sin Master-WMS) coinciden con los 30 segundos de timeout de sesión de ZooKeeper.

Para la segunda parte del experimento se utilizó un conjunto de simulaciones compuesto por las instancias 3, 6, 9, 11, 14 y 15 (definidas en la tabla 5.2), siguiendo el mismo procedimiento que en la primer parte del experimento. Los resultados obtenidos se pueden observar en la tabla 5.6. Estos resultados reafirman las conclusiones de la primera parte: los fallos en el Master-WMS no afectan el tiempo de ejecución final y el tiempo de recuperación coincide con el timeout de ZooKeeper configurado. A su vez, se comprobó que el tiempo de recuperación del Master-WMS es independiente de la carga de trabajo que se está ejecutando en el sistema.

tiempo hasta falla (s)	tareas completadas hasta falla	tiempo de recuperación (s)	tiempo total de ejecución (s)
200	0 de 6	30.0	1208
400	1 de 6	30.1	1222
700	1 de 6	30.0	1194
900	1 de 6	30.0	1201
1100	4 de 6	30.1	1205

Tabla 5.6: Parte 2 del experimento para el análisis de fallos del Master-WMS. Tiempos de recuperación del Master-WMS en 5 ejecuciones distintas sobre las mismas simulaciones.

5.2.3.2. Falla de procesos Worker

Para validar la tolerancia a fallos de los procesos Worker, se analizaron los procedimientos de detección de fallos y reasignación de simulaciones que implementa el Master-WMS ante el fallo de un Worker. Se utilizó la configuración del sistema y el conjunto de simulaciones descritos en la sección 5.1. Las instancias 12, 20 y 24 fueron excluidas por tener tiempos de ejecución muy altos (del orden de horas) en comparación al resto. Estas instancias no son relevantes para el experimento y dificultan la visualización de los resultados.

El experimento comenzó con la ejecución del Master-WMS. Las instancias fueron asignadas a los 4 Workers disponibles. Luego de $31'46''$ de iniciado el sistema se forzó el fallo del Worker 3. Como consecuencia, ZooKeeper eliminó el znode efímero del Worker y notificó al Master-WMS. La notificación fue recibida 30.2 segundos después del fallo del Worker (este tiempo se

corresponde con el timeout de sesión de ZooKeeper). Finalmente, el Master-WMS obtuvo las tareas no finalizadas que habían sido asignadas al Worker 3 y las asignó al resto de los Workers.

La gráfica de la figura 5.6 muestra el estado del sistema en el momento previo al fallo del Worker 3. En el eje horizontal se representan los Workers activos del sistema y en el vertical los costos de ejecución estimados de las simulaciones no finalizadas (separados por lineas negras).

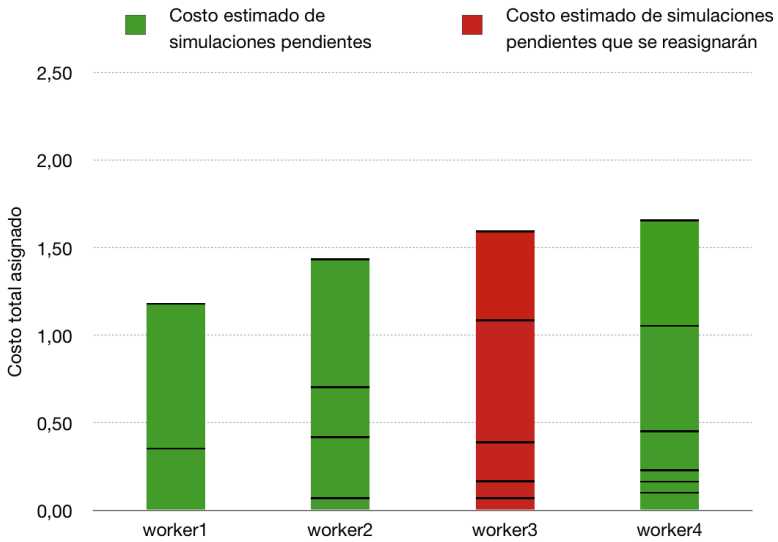


Figura 5.6: Costo de ejecución asignado previo al fallo del Worker 3.

La figura 5.7 muestra el estado del sistema en el momento posterior al proceso de reasignación. Se puede observar que las simulaciones no finalizadas del Worker 3 (en rojo) fueron asignadas al resto de los Workers.

Figura 5.7: Costo de ejecución asignado posterior a la reasignación de tareas.

Intuitivamente se puede concluir que el fallo de uno o varios de los Workers que cuentan con simulaciones asignadas, implica un aumento en el tiempo total de ejecución en la mayoría de los casos. Este aumento se produce por el traslado hacia los Workers activos de la carga de trabajo que dejan pendiente los Workers que fallaron. Como trabajo futuro se propone mejorar el método de reasignación de simulaciones, considerando el costo de las simulaciones que tienen asignadas los Workers que permanecen activos al momento del fallo. Esta consideración puede mejorar la distribución de las simulaciones reasignadas, minimizando el aumento del tiempo final de ejecución.

5.2.3.3. Falla de Servidor Zookeeper

Los servidores ZooKeeper gestionan la coordinación entre el Master-WMS y los Workers. Al mismo tiempo, dan soporte al sistema de detección de fallos. Debido a la relevancia ZooKeeper en el sistema, es fundamental asegurar la disponibilidad del servicio ante la falla de algunos de los servidores.

ZooKeeper puede ser configurado para funcionar incluso cuando alguno de los servidores del ensemble no esté disponible. Para un ensemble de n servidores, la cantidad de servidores que pueden fallar para que el servicio continúe operativo debe ser menor a $n/2$. Por ejemplo, para soportar la falla de un servidor son necesarios al menos 3 servidores en el ensemble y para soportar la falla de 2 servidores son necesarios al menos 5 servidores. Por este motivo, es recomendable contar con una cantidad impar de servidores. En la sección 4.2.9 se describe en detalle el mecanismo de tolerancia a fallos implementado por ZooKeeper.

Para el experimento presentado en esta sección se utilizó un ensemble con 3 servidores ZooKeeper (denominados zk1, zk2 y zk3). De esta manera, el sistema es capaz de seguir funcionando con 2 servidores sin detener el servicio.

La primer etapa del experimento consistió en ejecutar el sistema con las instancias 1, 2, 3, 5, 6, 7, 8, 9, 10 y 15 (descritas en la tabla 5.2), y detener uno de los servidores ZooKeeper para simular una falla. A los $17'30''$ de ejecución se detuvo manualmente el servicio ZooKeeper en el servidor zk3. La detención del servicio provocó que las sesiones de 2 de los Workers del sistema migraran a los servidores zk1 y zk2. La tabla 5.7 muestra las conexiones de los Workers con los servidores ZooKeeper antes y después de la falla.

	worker 1	**worker 2**	**worker 3**	**worker 4**
antes del fallo	zk2	zk3	zk3	zk1
después del fallo	zk2	zk2	zk2	zk1

Tabla 5.7: Conexiones de Workers a servidores ZooKeeper.

La segunda etapa del experimento consistió en ejecutar el sistema con el mismo conjunto de instancias utilizado en la etapa anterior y los tres servidores ZooKeeper operativos. No se encontró una diferencia significativa entre el tiempo de ejecución del sistema con los tres servidores activos (1272 segundos) y el tiempo de ejecución con la falla de uno de los servidores (1290 segundos). Esto se debe a que la conexión de los clientes (Workers y Master-WMS) con los servidores se mantiene en un hilo de ejecución paralelo. Cuando se detecta la falla (pérdida de la conexión) los procesos conectados al servidor que falló intentan conectarse nuevamente al servicio a través de alguno de los servidores que permanecen activos. La reconexión es lo suficientemente rápida para evitar que la sesión expire. Manejar la sesión de ZooKeeper en un hilo independiente permite a los Workers continuar con la ejecución de las simulaciones MCell+FERNET sin interrupciones.

Para culminar el experimento se realizó la detención del servicio ZooKeeper en dos de los servidores del ensemble. El resultado obtenido fue el esperado, el servicio ZooKeeper quedó inoperante y en consecuencia el Master-WMS y los Workers que estaban conectados finalizaron su ejecución.

5.3. Resumen del análisis experimental

Los experimentos realizados permitieron validar la tolerancia a fallos de los principales componentes que integran la solución y estudiar la efectividad de los algoritmos implementados.

Los resultados del análisis confirmaron que la implementación del algoritmo de asignación de simulaciones cumple satisfactoriamente su objetivo. Se obtuvo una distribución de la carga de trabajo que redujo el tiempo de ejecución total del sistema con respecto a una distribución estática uniforme y a ejecuciones secuenciales.

En los experimentos realizados sobre el estimador de tiempos de ejecución, se verificó que el incremento de la cantidad de iteraciones reduce el error de la estimación. Sin embargo, se observó que mejorar la calidad de la estimación (reducir el error) no siempre implica una reducción del tiempo de ejecución final. Por otro lado, se observó que el tiempo de ejecución de una simulación no sólo se ve afectado por el número de moléculas liberadas, también es afectado por la densidad de moléculas por subvolumen de cómputo.

En los experimentos donde se provocó el fallo del Master-WMS se comprobó que el tiempo de recuperación (i.e. el tiempo que tarda un maestro secundario en obtener el rol de primario) no depende del momento en el que ocurre la falla ni de la carga de trabajo que está ejecutando. Se observó que el tiempo de recuperación depende del timeout de sesión configurado entre clientes y servidores ZooKeeper.

Los experimentos realizados mostraron la correcta respuesta del sistema ante fallos de los procesos Workers. La carga de trabajo pendiente de los Workers que fallaron fue reasignada a los Workers activos. Por otro lado, se constató la continuidad del servicio ZooKeeper ante el fallo de uno de los servidores del ensemble.

Capítulo 6

Conclusiones y trabajo futuro

Este capítulo presenta las conclusiones finales del proyecto y las temáticas abiertas para realizar trabajos futuros que complementen la investigación.

6.1. Conclusiones

Este trabajo ha presentado la investigación y los detalles técnicos de la construcción de un sistema distribuido y tolerante a fallos, siguiendo el paradigma Cloud Computing, aplicado al estudio del desarrollo embrionario mediante la simulación de diferentes escenarios biológicos y el análisis de fluorescencia.

Cloud Computing es un paradigma de computación distribuida que permite integrar recursos computacionales para proveer una plataforma de computo potente, ideal para la ejecución de cargas de trabajo que suelen demandar alto poder de computo. En el marco de este proyecto, este paradigma sienta las bases para el diseño de una arquitectura que brinda soporte a la ejecución distribuida de simulaciones. OpenNebula permitió el desarrollo de una plataforma cloud mediante la virtualización de los recursos disponibles, permitiendo que el sistema resulte beneficiado de la alta flexibilidad y elasticidad que caracterizan a las plataformas cloud.

Como principal concepto de las tareas de análisis del software, se puede concluir que el proceso de instalación de OpenNebula en modalidad standalone presenta mayor complejidad en relación a una instalación estándar, en especial cuando no se cuenta con permisos de administrador. Una instalación en modalidad standalone requiere compilar el producto e instalar las dependencias necesarias manualmente. Esta modalidad no es la habitual ni la más recomendada, por este motivo no cuenta con documentación detallada. En este contexto, la instalación de OpenNebula requirió un esfuerzo adicional al esperado. Sin embargo, una vez instalado, OpenNebula simplificó la gestión de la infraestructura, facilitando la distribución de los componentes del sistema sobre los recursos virtualizados. La API XML-RPC de OpenNebula permitió desarrollar un mecanismo de integración con otras instancias para incrementar la capacidad de cómputo del sistema.

El servicio ZooKeeper, utilizado para la comunicación y coordinación de procesos distribuidos en el cloud, demostró ser una herramienta eficiente y de gran utilidad en el contexto de este proyecto. En especial, Zookeeper es muy útil para la implementación de los mecanismos de tolerancia a fallos de los procesos Master-WMS y Worker. Durante el análisis experimental se validó la tolerancia a fallos nativa de ZooKeeper, comprobando que el servicio permaneció disponible luego de eliminar uno de los tres servidores que conforman el ensemble, y que quedó inhabilitado cuando se eliminaron dos servidores.

El algoritmo de asignación de simulaciones implementado brinda una solución que, para los propósitos de esta investigación, es lo suficientemente precisa y se obtiene en un tiempo despreciable en comparación al tiempo de ejecución de una carga de trabajo. Se comprobó que este algoritmo genera una distribución más eficiente de la carga de trabajo en comparación a una distribución estática uniforme. En los experimentos realizados, el algoritmo implementado alcanzó una mejora de 23.4 % en el tiempo de ejecución de una carga de trabajo en comparación

al algoritmo de distribución estática uniforme, y una mejora de 66.4 % respecto a una ejecución secuencial.

El método de estimación de simulaciones implementado generó resultados aceptables que permitieron realizar una buena distribución de la carga de trabajo mediante el algoritmo de asignación. Por otro lado, a partir del análisis experimental se observó que el tiempo de ejecución de una simulación es afectado en gran medida por el número de moléculas liberadas, el número de iteraciones, el paso de tiempo de la simulación y la densidad de moléculas por subvolumen de cómputo. Estas observaciones pueden ser utilizadas para implementar métodos de estimación alternativos.

El sistema distribuido implementado respondió satisfactoriamente a los experimentos de tolerancia a fallos realizados. Se comprobó que la carga de trabajo y el momento en el que ocurre un fallo en el Master-WMS son factores que no inciden en el tiempo de recuperación. Además, se comprobó que el tiempo de recuperación del Master-WMS está determinado por el timeout de sesión configurado en ZooKeeper, siendo despreciable en comparación al tiempo total de ejecución de una carga de trabajo. Por otro lado, se verificó el correcto funcionamiento del sistema ante el fallo de un Worker. Las tareas asignadas al Worker que falló fueron distribuidas sobre el resto de los Workers activos. Al igual que cuando ocurre un fallo en el Master-WMS, la detección del fallo del Worker coincide con el timeout de sesión configurado en ZooKeeper.

La plataforma cloud construida, en conjunto con las técnicas y algoritmos propuestos, permitió la ejecución paralela y eficiente de simulaciones MCell+FERNET, realizando una contribución práctica al estudio de fenómenos biológicos complejos.

6.2. Trabajo futuro

A partir del trabajo realizado surgen dos lineas de investigación que resulta interesante plantear como trabajo futuro: la relacionada a mejorar la eficiencia en la ejecución paralela de simulaciones y la vinculada a los mecanismos para ampliar la capacidad de cómputo del sistema.

Para mejorar la eficiencia en la ejecución de simulaciones, sería interesante estudiar mecanismos alternativos de estimación y distribución de la carga de trabajo. Se propone realizar estudios analíticos sobre el impacto de los parámetros de MCell en el tiempo de ejecución de una simulación, para encontrar relaciones cuantificables que contribuyan a mejorar la calidad de las estimaciones. Con respecto a la distribución de la carga de trabajo, se plantea como trabajo futuro la implementación de un algoritmo que resuelva el problema ITMAP. Mejorar la estimación de las simulaciones e implementar un algoritmo que optimice la asignación de recursos en todos los escenarios, es clave para minimizar el tiempo total de ejecución de una carga de trabajo en el sistema.

Con el objetivo de favorecer la colaboración entre la Facultad de Ingeniería y la Universidad de Buenos Aires, sería interesante integrar las instancias de OpenNebula instaladas en los clusters FING y TUPAC a nivel de autenticación, mediante mecanismos de autorización genéricos como el Lenguaje de Marcado para Confirmaciones de Seguridad (*Security Assertion Markup Language, SAML*). Esta integración constituye un primer paso de cara a una integración que permita la unificación de los recursos computacionales en un único cloud privado, que posibilitaría el acceso a los recursos de ambos clusters mediante un único servicio, generando una visión global del sistema.

El nodo del cluster FING utilizado para el despliegue de la infraestructura debió ser trasladado a una red DMZ, quedando aislado del resto de los nodos del cluster. Por lo tanto, para ampliar la capacidad de cómputo del sistema con más recursos del cluster FING, es necesario introducir nuevamente el nodo a la red interna configurando el firewall de manera acorde. Esta tarea implica un análisis de impacto sobre el resto de los activos de software funcionando en el cluster y sobre la seguridad general de la infraestructura.

El servidor de base de datos y el servidor web suponen puntos únicos de fallo (Single Point of Failure, SPOF) para el funcionamiento del sistema. Para cubrir estos puntos de fallo, se propone implementar mecanismos de replicación en la base de datos y balanceo de carga en el servidor web.

Apéndice A

MCell+FERNET

A.1. Archivo de configuración de MCell

Las simulaciones ejecutadas por Mcell se especifican en formato MDL. Un archivo MDL es un archivo de texto que contiene comandos separados por espacios para ser interpretados y ejecutados por MCell.

A.1.1. Estructura de un archivo MDL

Los comandos de un archivo MDL pueden ser agrupados en cinco diferentes grupos, que usualmente deben seguir el orden que se presenta a continuación.

- Inicialización: este grupo de comandos fija parámetros globales de la simulación (e.g. salto de tiempo y duración).
- Definición de moléculas: estos comandos especifican el nombre y constante de difusión de las moléculas en la simulación.
- Definición de reacciones: este conjunto de comandos especifican la reacciones que pueden ocurrir entre las moléculas y su tasa de ocurrencia.
- Especificación de la geometría: estos comandos describen las membranas y otros límites donde la simulación ocurre.
- Especificación de la salida: estos comandos especifican qué datos se deben obtener como salida de la simulación.

A.1.2. Comandos

El listado A.1 muestra el grupo de comandos de inicialización del archivo de configuración MDL de MCell.

```
TIME_STEP = t     // Set the simulation time step to t seconds. 1e-6
                  is a common value.

ITERATIONS = N    // Run the simulation for N iterations
```

Listado A.1: Grupo de comandos de inicialización en un archivo de configuración de MCell.

El listado A.2 muestra el grupo de comandos de definición de moléculas del archivo de configuración MDL de MCell.

```
DEFINE_MOLECULE name     // Define a single molecule called name
{
  DIFFUSION_CONSTANT = D      // This molecule diffuses in space with
                              diffusion constant D. D can be zero,
                              in which case the molecule does not move

  DIFFUSION_CONSTANT_2D = D   // This molecule is constrained to a surface
                              and diffuses with diffusion constant D.
}
```

Listado A.2: Grupo de comandos de definición de moléculas en un archivo de configuración de MCell.

El listado A.3 muestra el grupo de comandos de definición de reacciones del archivo de configuración MDL de MCell.

```
DEFINE_REACTIONS
{
  reactants -> products [rate]      // Define a reaction that occurs
                                    between one, two or three reactants (names of
                                    molecules, separated by +) and produces an arbitrary
                                    number of products (also separated by +), with a
                                    specified rate

  reactants -> products [rate]:name // As above, and call the reaction name so it can
                                    be referred to by count statements
}
```

Listado A.3: Grupo de comandos de definición de reacciones en un archivo de configuración de MCell.

Las unidades del parámetro *rate* en la definición de reacciones unimoleculares y bimoleculares son las siguientes:

- $\left[s^{-1}\right]$ para reacciones unimoleculares.
- $\left[M^{-1}s^{-1}\right]$ para las reacciones bimoleculares entre moléculas volumétricas o una molécula volumétricas y una superficial, siendo M la molaridad de la solución.
- $\left[\mu m^2 N^{-1}s^{-1}\right]$ para reacciones bimoleculares entre dos superficies de molécula, siendo N la cantidad de reactivos.

Se presentan ejemplos de reacciones en la tabla A.1. El listado A.4 muestra el grupo de comandos de especificación de la geometría del archivo de configuración MDL de MCell. Cada tipo de superficie se define por su nombre (único en la simulación y no debe coincidir con ningún nombre de moléculas). Es posible definir objetos geométricos con los comandos del archivo MDL, por ejemplo el listado A.5 muestra la definición de una caja.

```
DEFINE_SURFACE_CLASS    // Define a single surface type called name
name
{
  surface property commands
}
```

Listado A.4: Grupo de comandos de especificación de la geometría en un archivo de configuración de MCell.

Ejemplo	Explicación
$A \rightarrow B$ [100]	La molécula A se transforma en la molécula B a una velocidad de $100s^{-1}$
$A \rightarrow A + B$ [100]	La molécula A emite moléculas de B a una velocidad de $100s^{-1}$
$A \rightarrow NULL$ [100]	La molécula A se destruye a una velocidad de $100s^{-1}$
$A + B \rightarrow A$ [1e6]	La molécula A destruye la molécula B a una velocidad de $10^6 M^{-1} s^{-1}$
$A + B \rightarrow A + C$ [1e6]	La molécula A catalíticamente convierte B en C a una velocidad de $10^6 M^{-1} s^{-1}$
$A + B \rightarrow A + B + C$ [1e6]	La colisión entre A y B catalíticamente genera C a una velocidad de $10^6 M^{-1} s^{-1}$

Tabla A.1: Ejemplo de definición de reacciones mediante comandos MCell.

```
name BOX       // This defines a box object called name
{
  CORNERS = [ x1 , y1 , z1 ],[ x2 , y2 , z2 ]   // The box object has
                                                 corners as specified
  ASPECT_RATIO = a       //Make sure that the ratio of the long to short
                        side of each triangle making up the box is no
                        more than a. The smallest allowed value is 2.
                        The default is to not care about triangle shape.
}
```

Listado A.5: Ejemplo de comandos de especificación de la geometría de una caja en un archivo de configuración de MCell.

Existen dos tipos diferentes de salida en MCell, la salida de visualización y la salida de conteo. La salida de visualización contiene las moléculas del modelo de un modo apropiado para la visualización o análisis que requiere el conocimiento preciso de la ubicación de las partículas. La salida de conteo contiene reportes estadísticos como el numero total de moléculas de cierto tipo, el numero de veces que ocurrió una reacción dentro de cierto objeto del modelo, etc. El listado A.6 muestra el grupo de comandos de especificación de la salida del archivo de configuración MDL de MCell.

```
VIZ_OUTPUT   // Define a new visualization output block
{
  viz output commands
}
```

Listado A.6: Grupo de comandos de especificación de la salida en un archivo de configuración de MCell.

En el caso de MCell modificado, es en el bloque de visualización donde se especifica el modo de ejecución JOINED o PIPED (por ejemplo el listado A.7).

```
VIZ_OUTPUT
{
  MODE = JOINED
  FILENAME = "./viz_data/seed_" & seed & "/Scene"
  MOLECULES
  {
     NAME_LIST {A B}
     ITERATION_NUMBERS {ALL_DATA @ ALL_ITERATIONS}
  }
}
```

Listado A.7: Ejemplo de comandos de especificación de la salida en un archivo de configuración de MCell.

A.2. Archivo de configuración de FERNET

El archivo de configuración de FERNET permite especificar los seis modos diferentes de ejecución, donde cada modo determina el tipo de escaneo de microscopio que se quiere simular. El archivo de configuración de FERNET se organiza en bloques de configuración. En un bloque se especifican los parámetros generales que aplican para todos los modos de ejecución, los otros seis bloques especifican los parámetros de configuración particulares de cada modo de ejecución.

El listado A.8 es un ejemplo de bloque del archivo de configuración de FERNET, con los parámetros comunes a todos los modos de ejecución. En este ejemplo, la molécula A emitirá en el canal 0 con un brillo de 100000 cpsm, la molécula B emitirá en el canal 0 con un brillo de 10000 cpsm y en el canal 1 con 100000 cpsm.

```
channel0: // Channel 0 configuration (mandatory)
{
  on = 1;               // channel state (1 = on, 0 = off)
  molec = ("A","B");    // list of molecule names from MCell that emit in channel 0
  bright = (100000,10000); // list with brightness in counts per second
                           per molecule for channel 0, length must be
                           equal to number of molecs in channel
};

channel1: // Channel 1 configuration (for dual color experiments, optional)
{
  on = 1;
  molec = ("B");
  bright = (100000);
};

kappa = 200e-9; // time of single fluorescence event in seconds
                (already optimized parameter, be careful if editing)
w_xy = 0.2;     // PSF waist in XY plane
w_z = 1.0;      // PSF waist in Z direction
```

Listado A.8: Ejemplo de bloque de parámetros comunes en un archivo de configuración de FERNET.

El listado A.9 es un ejemplo de bloque del archivo de configuración de FERNET, con los parámetros del modo de ejecución Point. En este ejemplo, el archivo de salida de FERNET que contiene la traza de fluorescencia se denomina *prefix_c0.txt* para el canal 0 y *prefix_c1.txt* para el canal 1.

```
shiftx = 0.0;           // X center of PSF
shifty = 0.0;           // Y center of PSF
shiftz = 0.0;           // Z center of PSF
prefix = "point";       // name for photon output file
```

Listado A.9: Ejemplo de bloque de parámetros para el modo de ejecución Point en un archivo de configuración de FERNET.

El listado A.10 es un ejemplo de bloque del archivo de configuración de FERNET, con los parámetros del modo de ejecución Multi. En este ejemplo, se crea una rejilla de $nPSFX \times nPSFY$ PSFs (función de dispersión de punto), donde cada centro se separa *dx* y *dy* micrómetros en cada eje. Los archivos de salida se denominan *prefix_XXX_cY.txt*, donde XXX es un número entre 0 y $(nPSFXnPSFY) - 1$ y Y es el canal correspondiente. Además un archivo auxiliar es creado con el número de PSF y su centro (X, Y).

```
prefix = "multi";   // prefix for output name
shiftz = 0.0;       // Z center of PSF
nPSFX = 5;          // number of PSF in X axis
dx = 0.2;           // separation between PSF in X axis
nPSFY = 5;          // number of PSF in Y axis
dy = 0.2;           // separation between PSF in Y axis
```

Listado A.10: Ejemplo de bloque de parámetros para el modo de ejecución Multi en un archivo de configuración de FERNET.

El listado A.11 es un ejemplo de bloque del archivo de configuración de FERNET, con los parámetros del modo de ejecución Line. En este ejemplo, se escanea una línea de ($n\_columns \times shift$) micrómetros centrados en *(centerx,centery,centerz)* a lo largo del eje x. El tiempo que el láser de escaneo toma para volver a la posición de escaneo inicial puede ser simulado con el parámetro *deadtime*.La imagen de salida que contiene el perfil XT de la fluorescencia es denominada *tiffname_cY.tif*, donde Y es el número de canal.

```
deadtime = 0.001;               // dead line time in seconds
centerx = 0.0;                  // X center of PSF (um)
centery = 0.0;                  // Y center of PSF (um)
centerz = 0.0;                  // Z center of PSF (um)
n_columns = 15;                 // number of columns
shift = 0.2;                    // shift between columns (um)
tiffname = "linescan";  // output TIFF name prefix
```

Listado A.11: Ejemplo de bloque de parámetros para el modo de ejecución Line en un archivo de configuración de FERNET.

El listado A.12 es un ejemplo de bloque del archivo de configuración de FERNET, con los parámetros del modo de ejecución Image. Este ejemplo es similar al modo Line, excepto que el área de escaneo es de $width \times height$ pixeles centrada en (0,0,*centerz*).

```
deadtime = 0.001;     // dead line time in seconds
centerz = 0.0;        // Z center of image
pixel = 0.05;         // pixel size in um
width = 64;           // image width in pixels
height = 64;          // image height in pixels
tiffname = "image";   // output TIFF name
```

Listado A.12: Ejemplo de bloque de parámetros para el modo de ejecución Image en un archivo de configuración de FERNET.

El listado A.13 es un ejemplo de bloque del archivo de configuración de FERNET, con los parámetros del modo de ejecución Stack. Este ejemplo es similar al modo Image, a excepción de que el escenario es escaneado entre las posiciones *top_z* y *bot_z* a intervalos *step* para obtener un stack 3D.

```
deadtime = 0.001;        // dead line time in seconds
pixel = 0.05;            // pixel size in um
width = 64;              // image width in pixels
height = 64;             // image height in pixels
tiffname = "image";      // output TIFF name
top_z = 1.0;             // top Z position of the stack in um
bot_z = -1.0;            // bottom Z position of the stack in um
step = 0.1;              // Z plane separation
```

Listado A.13: Ejemplo de bloque de parámetros para el modo de ejecución Stack en un archivo de configuración de FERNET.

El listado A.14 es un ejemplo de bloque del archivo de configuración de FERNET, con los parámetros del modo de ejecución SPIM. En este ejemplo, la imagen es generada por un sensor CCD. La fluorescencia de una área de $width \times height$ del escenario simulado centrado en (0,0,*centerz*) es agrupada en intervalos de *frame_t* segundos.

```
NA = 1.6;            // numerical aperture of the microscope
lambda = 500.0;      // emission wavelength in nm
waist = 0.5;         // axial waist of the excitation PSF, overrides
                     the value of w_z in common block
pixel = 0.1;         // pixel size in um
width = 64;          // image width in pixels
height = 64;         // image height in pixels
centerz = 0.00;      // Z center of the image in um
frame_t = 0.001;     // frame time in seconds
tiffname = "image"; // output TIFF name
```

Listado A.14: Ejemplo de bloque de parámetros para el modo de ejecución SPIM en un archivo de configuración de FERNET.

Bibliografía

[1] A.A.Puntambekar. *Design And Analysis Of Algorithms*, chapter 5. Technical Publications Pune, 1 edition, 2010.

[2] J. Angiolini and E. Mocskos. FERNET: Fluorescence Emission Recipes and Numerical routines Toolkit, 2014. URL `http://www.dc.uba.ar/inv/licar/FERNET`. Consultada Mayo 2014.

[3] J. Angiolini, N. Plachta, E. Mocskos, and V. Levi. Exploring the dynamics of cell processes through Monte Carlo simulations of advanced fluorescence microscopy experiments. *Biophysical Journal*, 108:2613–2618, 2015.

[4] S. Arora and M.C. Puri. A variant of time minimizing assignment problem. *European Journal of Operational Research*, 110:314–325, 1998.

[5] G.T. Balls, S.B. Baden, T. Kispersky, T.M. Bartol, and T.J. Sejnowski. A large scale monte carlo simulator for cellular microphysiology. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 42–, 2004.

[6] Cloud-Init. Cloud-init documentation, 2014. URL `http://cloudinit.readthedocs.org/en/latest/`. Consultada Julio 2014.

[7] M. Da Silva, S. Nesmachnow, M. Geier, E. Mocskos, J. Angiolini, V. Levi, and A. Cristobal. Efficient Fluorescence Microscopy Analysis over a Volunteer Grid/Cloud Infrastructure. *First HPCLATAM-CLCAR Joint Conference, Latin American High Performance Computing Conference*, pages 113–127, 2014.

[8] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET). Cluster TUPAC, 2015. URL `http://tupac.conicet.gov.ar/`. Consultada Agosto 2015.

[9] The Apache Software Foundation. ZooKeeper 3.4 Documentation, 2014. URL `https://zookeeper.apache.org/doc/r3.4.6/`. Consultada Julio 2014.

[10] Red Hat. KVM, 2008. URL `http://www.linux-kvm.org/page/Main_Page`. Consultada Julio 2014.

[11] Red Hat. WildFly, 2013. URL `http://wildfly.org/`. Consultada Agosto 2015.

[12] Red Hat. Managing services with Systemd, 2015. URL `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/System_Administrators_Guide/chap-Managing_Services_with_systemd.html`. Consultada Mayo 2015.

[13] F. Junqueira and B. Reed. *ZooKeeper: Distributed Process Coordination.* O'Reilly, 2013.

[14] R.A. Kerr, T.M. Bartol, B. Kaminsky, M. Dittrich, J.J. Chang, S.B. Baden, T.J. Sejnowski, and J.R. Stiles. Fast Monte Carlo Simulation Methods for Biological Reaction-Diffusion Systems in Solution and on Surfaces. *SIAM Journal on Scientific Computing*, 30:3126–3149, 2008.

[15] Libvirt. Libvirt virtualization API, 2014. URL `http://libvirt.org/`. Consultada Julio 2014.

[16] M. Mattess, C. Vecchiola, and R. Buyya. Managing Peak Loads by Leasing Cloud Infrastructure Services from a Spot Market. In *12th IEEE International Conference on High Performance Computing and Communications*, pages 180–188, 2010.

[17] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards & Technology, 2011.

[18] R. Moreno-Vozmediano, R.S. Montero, and I.M. Llorente. IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures. *Computer*, 45:65–72, 2012.

[19] S. Nesmachnow. Computación científica de alto desempeño en la Facultad de Ingeniería, Universidad de la República. *Revista de la Asociación de Ingenieros del Uruguay*, 61:12–15, 2010.

[20] Multiscale Modeling of Biological Systems (MMBioS). Monte Carlo Cell, 2013. URL `http://www.mcell.org`. Consultada Mayo 2014.

[21] The OpenNebula Project. About The OpenNebula Project, 2002. URL `http://opennebula.org/`. Consultada Mayo 2014.

[22] The OpenNebula Project. OpenNebula GitHub, 2008. URL `https://github.com/OpenNebula/one/`. Consultada Agosto 2015.

[23] TechNet. Designing the Demilitarized Zone, 2015. URL `https://technet.microsoft.com/en-us/library/cc961351.aspx`. Consultada Diciembre 2015.

[24] P. Turner, L. Petzold, A. Shiflet, I. Vakalis, K. Jordan, and S. John. Undergraduate Computational Science and Engineering Education. *SIAM Review*, 53:561–574, 2011.

[25] C. Vecchiola, S. Pandey, and R. Buyya. High-Performance Cloud Computing: A View of Scientific Applications. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, pages 4–16, 2009.

[26] U.R. Venkata. *A Performance Model and Load Balancer for a Parallel Monte-Carlo Cellular Microphysiology Simulator*. PhD thesis, University of California, San Diego, 2004.

[27] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010.