UNIVERSIDAD AUTÓNOMA METROPOLITANA

Construcción de un Sistema de Almacenamiento Distribuido Basado en Requerimientos

Idónea Comunicación de Resultados
que para obtener el grado de
Maestro en Ciencias

presenta el

Lic. Diego Rodrigo Guzmán Santamaría

Trabajo dirigido por:

Dra. Reyna C. Medina Ramírez

Dr. Ricardo Marcelín Jiménez

México D.F., a November 9, 2011

A mi hermosa Miriam, por todo el amor y apoyo que me ha brindado, por ser mi inspiración y compañera de vida.

A mis padres Antonio y Martha, y hermana Ángeles, porque gracias a ellos soy lo que soy.

A todos mis amigos, porque su apoyo ha sido muy importante durante todo este tiempo.

A mis asesores Carolina y Ricardo, por su valiosa guía y consejos para realizar este trabajo, y por todo lo que he aprendido de ellos.

A mis sinodales Víctor y Luis, por sus comentarios que enriquecieron en mucho este trabajo.

A mi universidad, la UAM, por la oportunidad de formarme como profesional.

Al CONACYT e INFOTEC por los recursos otorgados para la realización de mis estudios de maestría.

A todos ellos quiero decirles ¡Muchas gracias por su apoyo!

Contenido

Lista de figuras

Lista de Algoritmos

Lista de tablas

Parte I

Idónea Comunicación de Resultados

1 Introducción

La información es un bien colectivo e intangible del cual dependen los procesos de las organizaciones modernas. El volumen de la información que éstas manejan ha crecido y seguirá haciéndolo en los próximos años. Si se quiere enfrentar esta tendencia, no se puede seguir pensando en los mismos mecanismos de almacenamiento utilizados hasta hoy. Son varias las razones que obligan a buscar alternativas; entre ellas destacan, que su operación puede resultar muy costosa, tienen un límite en su capacidad de crecimiento, son más vulnerables ante ataques, desastres naturales o fallas y pueden quedar fácilmente rebasados cuando se les somete a cargas de trabajo masivas y concurrentes. Asimismo, se observan dos tendencias que en realidad son expresiones del mismo fenómeno, pero a diferente escala:

1. Existe una necesidad creciente de intercambiar información entre organizaciones, aún cuando sus formatos de registro y las relaciones entre sus datos sean diferentes. Se puede pensar como ejemplo en el caso de dos instituciones de salud que requieren intercambiar expedientes clínicos.

2. Se requiere que la información pueda moverse con agilidad en los equipos de trabajo dentro de una organización. Como es el caso de un proyecto de largo plazo en el que se debe garantizar la disponibilidad de los documentos generados, a través del tiempo. En este escenario sería de mucho valor que un usuario pudiera recuperar los contenidos producidos a partir de las nociones o conceptos comunes a los participantes y que articulan el quehacer del equipo. Por ejemplo, un profesional que llega a un grupo de investigación desearía recuperar aquellos contenidos relacionados con su trabajo a partir de conceptos propios de su especialidad.

La primera tendencia implica la necesidad de la interoperabilidad entre sistemas informáticos. Es claro que las organizaciones han invertido muchísimos recursos en la construcción de sus plataformas informáticas y que ello cancela la posibilidad de implantar un sólo estándar que unifique la representación de la información. El problema debe abordarse estableciendo mecanismos de conversión que traduzcan los datos desde una entidad emisora a una meta-representación que luego pueda "aterrizarse" sobre la plataforma de una entidad receptora.

La segunda tendencia pone en relieve la importancia de la recuperación de la información compartida por un colectivo, donde los usuarios puedan buscar los documentos relacionados con ciertos conceptos o temas, en vez de buscar archivos por nombre. Esto les da la capacidad de encontrar incluso aquellos documentos cuya existencia ignoran, pero que pueden ser útiles para su trabajo, sobretodo si se trata de un grupo que basa sus actividades en un ambiente colaborativo.

Sin embargo, la conclusión más importante que puede obtenerse de esta reflexión es que, en los hechos, la información ya no se encuentra almacenada en un sólo punto. En realidad, se encuentra distribuida y requiere una revisión de los mecanismos de intercambio para obtener el máximo beneficio y gestionar su crecimiento. Se trata de repensar la manera como la información se comparte en el interior y con el exterior de una organización.

En los últimos años, los sistemas de almacenamiento distribuido (SAD) han sido objeto de un gran interés por parte de las empresas y los grupos de investigación en el tema. Aún cuando la función básica de estos sistemas consiste en repartir archivos sobre un conjunto de discos conectados en red, se reconoce también que un diseño cuidadoso puede producir funcionalidades mejoradas y atributos de calidad superiores, particularmente aquellos relacionados con la integridad, confidencialidad y disponibilidad de la información. Un SAD se convierte en una opción muy atractiva para la gestión de un volumen de información que puede crecer con el tiempo y alcanzar escalas masivas[McC04, Gra03].

1.1 Definición del Problema

El diseño de un SAD requiere definir:

1. El conjunto de requisitos funcionales y no funcionales que se busca atender.
2. Los parámetros de desempeño que garanticen la calidad de servicio.
3. La entidad o espacio elemental de almacenamiento.

Los requerimientos funcionales se refieren a las necesidades del usuario o cliente final del sistema. En tanto, los requerimientos no funcionales se refieren a los principios de diseño del ingeniero o arquitecto que lo construirá.

Los parámetros de desempeño de un SAD incluyen aspectos como la cantidad de información redundante, el balance de carga en los dispositivos de almacenamiento y la disponibilidad de la información. Por cuanto se refiere a la entidad elemental de almacenamiento, de primera instancia podría pensarse en los discos asociados con las computadoras participantes en un SAD. Sin embargo, en vista de la escala a la que se puede crecer y la complejidad de las

operaciones que pueden tener lugar, se considera contraproducente asumir una dependencia tan fuerte sobre dispositivos individuales. Es deseable construir un dispositivo virtual que exhiba una vida útil más prolongada y un comportamiento más estable que lo que puede esperarse de los dispositivos físicos a partir de los cuales se construya.

1.2 Contribución

Atendiendo las consideraciones anteriores, en este documento se describe el diseño y construcción de un prototipo para un SAD, denominado celda de almacenamientoi. Una celda está integrada por una colección de dispositivos de almacenamiento, coordinados por un conjunto de servidores. Una solución con estas características puede ofrecer las mismas capacidades que un sistema de almacenamiento de alto desempeño, pero con costos de construcción y operación mas accesibles. Lo anterior hace de la celda un candidato ideal para el soporte de aplicaciones de almacenamiento con fuertes restricciones de operación.

1.3 Justificación

Toda organización maneja y genera información sensible que no puede permitirse almacenar utilizando servicios brindados por terceros y, a menudo, las soluciones de almacenamiento "en casa" que existen en el mercado, pueden resultar tan costosas que quedan fuera de su presupuesto.

Las instituciones de salud, por ejemplo, requieren almacenar expedientes electrónicos bajo fuertes exigencias tales como la disponibilidad y la confidencialidad. Una clínica familiar no puede costear una solución de almacenamiento de alto desempeño, a menos que disponga de alternativas como las que se proponen en este proyecto.

1.4 Objetivos

- **Objetivo General**
  Desarrollar una arquitectura genérica para la construcción de sistemas de almacenamiento distribuido basados en requerimientos, en la cual se especifican los componentes básicos que deben integrar un sistema de este tipo, con la finalidad de que en trabajos futuros puedan desarrollarse en su totalidad cada uno de ellos, respetando la interfaz definida por la arquitectura definida en este trabajo.

- **Objetivos Particulares**
  - Definir los requerimientos funcionales y no funcionales (restricciones y atributos de calidad) de un sistema de almacenamiento distribuido.
  - Diseñar una arquitectura que pueda adaptarse a nuevas tecnologías y reutilizar recursos de bajo costo.

1.5 Metodología

De la literatura del tema, que es muy extensa y heterogénea, pudimos reconocer un conjunto de procedimientos que son comunes a todos los proyectos con alcances y objetivos semejantes al nuestro. Para tal efecto, seguimos la siguiente metodología:

1. Identificar las características y componentes que deben tomarse en consideración para diseñar un sistema de almacenamiento distribuido.
2. Identificar los parámetros utilizados para medir el desempeño de un sistema de éste tipo.
3. Proponer una arquitectura flexible, que cumpla con todos los aspectos anteriormente reconocidos.
4. Emplear la arquitectura propuesta, para construir un prototipo.

1.6 Cómo leer este documento

En el capítulo 2, profundizamos en el trabajo relacionado citando algunos de los resultados más relevantes que encontramos en la literatura del área. Reconocemos las características fundamentales de un sistema de almacenamiento distribuido, sus requerimientos (tanto funcionales como no funcionales), principios que hay que considerar al momento de diseñar un sistema de éste tipo y bajo qué parámetros podemos medir su desempeño.

En el capítulo 3 proponemos una arquitectura enfocada en la construcción de un sistema de almacenamiento distribuido, basándonos en los principios de diseño identificados.

Una vez definida nuestra propuesta, explicamos en el capítulo 4, la forma en que implementamos un prototipo de sistema de almacenamiento que cumple con la arquitectura que proponemos. Dado que este prototipo cuenta con una funcionalidad muy limitada, sería muy aventurado calificar su desempeño; el propósito de este prototipo es poner en práctica nuestra porpuesta, sin embargo, el diseño de la arquitectura no escapó de tener en cuenta

los parámetros de desempeño que identificados durante la etapa de investigación. Lo que se refleja los mecanismos empleados por el prototipo para controlar estos parámetros de desempeño, que pueden consultarse en la sección 3.4. Todo esto servirá para mejorar el rendimiento de las futuras versiones del prototipo o de otras implementaciones que sigan la arquitectura propuesta.

En el capítulo 5 recapitulamos el contenido de este documento, damos una panorámica de las posibilidades de crecimiento que tiene el sistema y el trabajo que queda por hacer para lograrlo.

Al final del documento, en el apéndice A mostramos con cierto nivel de detalle los casos de uso identificados durante la etapa de diseño de nuestra propuesta. En el apéndice B proporcionamos un manual para instalar el prototipo de la celda de almacenamiento, indicando el equipo de cómputo y dependencias necesarias para su correcto funcionamiento. Finalmente en el apéndice C incluimos la documentación completa del prototipo, incluyendo la definición de las clases y sus métodos.

2 Antecedentes

En este capítulo se presenta una visión panorámica sobre los sistemas de almacenamiento distribuido. Se busca entender que son el resultado de una evolución en los sistemas de cómputo que fue propiciada por las tecnologías para la comunicación. Con estos cambios aparecen también nuevas formas de evaluar las prestaciones soportadas. Enseguida se describen diferentes criterios para entender el vasto conjunto de sistemas que caben en la definición de almacenamiento distribuido. Luego, se presentan algunos ejemplos de esta tecnología, que describen hitos o etapas importantes. Por último se revisan algunos conceptos sobre redes P2P y principios de diseño que se utilizan en la construcción de sistemas de almacenamiento distribuido.

2.1 La evolución del almacenamiento

Los sistemas de almacenamiento han experimentado cambios en la medida en que las capacidades de comunicación, provistas por las redes de datos, han pasado a formar parte del hardware de la propia computadora. Podemos mencionar algunos de las etapas que dan testimonio de este cambio, observadas a lo largo de los últimos veinte años [R.J05].

DAS (Direct Attached Storage) Los recursos de almacenamiento están conectados directamente a la computadora y reservados, normalmente, para su uso exclusivo. Los datos se entregan mediante una interfaz de entrada y salida por bloques, p.ej. tipo SCSI. La computadora tiene acceso a los datos mediante el sistema de archivos que tiene implementado.

SAN (Storage Area Network) Es una arquitectura que integra recursos de almacenamiento remotos, de forma que aparecen ante el cliente como si fueran locales. El usuario puede realizar operaciones de lectura y escritura a nivel de bloques. Para garantizar un desempeño satisfactorio, se utiliza una red local de alta velocidad, p.ej. basada en fibra óptica.

NAS (Network Attached Storage) En este caso, los recursos de almacenamiento se ofre-

cen a los clientes por medio de una red de comunicaciones y a través de un sistema de archivos, por ejemplo, NFS, CIFS. Esto les permite leer y escribir como si se tratara de operaciones locales.

iSCSI (Internet SCSI) Esta arquitectura permite que los clientes accedan a un sistema de almacenamiento remoto sobre Internet, a nivel de entrada/salida por bloques. Puede entenderse como una arquitectura SAN sobre TCP/IP.

Puede considerarse que estos sistemas representan los antecedentes inmediatos de la tecnología de almacenamiento distribuido. No obstante, cuando se requiere aumentar el volumen de la información administrada, se observa que los costos crecen a una tasa superior a la que crece la capacidad de los mismos. Estas soluciones resultan incosteables a partir de un punto, luego del cual la única posibilidad es distribuir las responsabilidades del almacenamiento sobre un conjunto de nodos conectados en red.

Por su parte, [Yia01] menciona que a la par de esta evolución, cambiaron también los criterios para evaluar el desempeño de los sistemas de almacenamiento. Mientras que el almacenamiento centralizado se evalúa en términos de costo, persistencia, ancho de banda y latencia; estas figuras de mérito ceden su lugar o se complementan con nuevas medidas tales como la coherencia, la disponibilidad, la seguridad, la interoperabilidad y la tolerancia a fallas.

En [Pla06] se observa un esfuerzo para clasificar los sistemas de almacenamiento distribuido a partir de nueve criterios diferentes, que representan nueve formas de abordar su estudio:

Propósito del sistema. Se refiere a la función primordial del sistema, la cual puede ir más allá de las operaciones básicas de almacenamiento y recuperación de información (almacenamiento simple, sistemas de archivos, publicación de contenidos, sistemas de alto desempeño, *middleware* para soporte de aplicaciones distribuidas, sistemas por encargo).

Arquitectura de almacenamiento. Se refiere a la forma en que se distribuyen las responsabilidades entre los miembros del sistema y la manera en que se concretan las interacciones entre estos (cliente-servidor, P2P).

El ambiente de operación. Se refiere a las consideraciones que influyen en el diseño y la arquitectura, relacionadas con las fronteras del sistema y las garantías de uso que pueden asumirse por parte de los usuarios (confiable, parcialmente confiable, no confiable).

Los patrones de utilización. Se refiere a la discusión sobre cómo se clasifican las cargas de trabajo que puede soportar un sistema de esta naturaleza (confiable, parcialmente confiable, no confiable).

La consistencia de la información. Se refiere a la importancia que se le da al manejo sincronizado de las réplicas que pueden coexistir en el sistema (fuerte, optimista).

La seguridad. Se refiere al cuidado que se le otorga a la información sensible (acceso controlado, confidencialidad, anonimato, protección contra la censura, etc.).

La administración de entidades autónomas. Se refiere a que, en la medida que aumenta el tamaño de la red, las partes implicadas pueden estar administradas desde entidades diferentes, que sin embargo deben coordinarse para proporcionar un servicio unificado (auto-configuración, auto-optimización, auto-reparación, auto-protección).

La federación de componentes. Es un criterio que complementa al anterior y se refiere a la necesidad de administrar la heterogeneidad de recursos (descubrimiento de recursos, intercambio de servicios, etc.).

Encaminamiento. Se refiere a la necesidad de construir los caminos que llevan hasta el punto donde la información se almacena (estáticos/dinámicos, distribuidos/centralizados).

2.2 Ejemplos Sobresalientes

Algunos casos paradigmáticos de esta nueva generación de sistemas son: Napster, Gnutella, Farsite, OceanStore. Más recientemente pueden citarse los casos de BigTable [Cha06], Dynamo [DeC07], Ceph [Wei07] y Keyspace [Tre01].

Napster [BR01] Fue un sistema P2P para compartir archivos MP3. Cada usuario contribuía con su espacio de almacenamiento y sus propios recursos de red. La búsqueda de los archivos se realizaba mediante claves que empataban con los metadatos asociados a cada archivo. Al igual que la búsqueda, la operación de indexado se realizaba en forma centralizada desde un sitio principal. Por otro lado, una vez establecidas las conexiones entre los clientes, la transferencia final ocurría directamente entre estos.

Gnutella [Rip02] Utiliza un protocolo descentralizado construido sobre una estructura auto-configurable. Una consulta difunde una cadena de datos que cada receptor es libre de interpretar como le parezca. Por ejemplo, algún nodo puede tratar de empatar la consulta con el nombre del archivo, otro con su contenido y un tercero, puede procesar la consulta de cualquier otra forma. La consulta y las operaciones de solicitud se propagan usando un protocolo de inundación. La transferencia tiene lugar directamente entre las partes interesadas.

Farsite [AA02] Es un sistema de archivos distribuido y jerárquico, diseñado pensando en ofrecer confiabilidad, disponibilidad y confidencialidad. El acceso a los archivos se controla mediante firmas electrónicas. Todos los archivos se encriptan antes de almacenarse. Existe un directorio distribuido que registra los sitios donde se guardan los originales y sus réplicas, así como las versiones de cada archivo.

OceanStore [**Rhe01**] Es un sistema de almacenamiento de escala global que busca ofrecer propiedades tales como la persistencia, alta disponibilidad, y desempeño, aún bajo condiciones dinámicas de trabajo, i.e. fallas o ataques y cambios en las condiciones de la red de comunicaciones.

BigTable [**Cha06**] Es un sistema construido por encargo, para almacenar datos estructurados. Sus diseñadores crearon una API y un conjunto de funciones con las que un cliente puede desarrollar nuevas aplicaciones.

Dynamo [**DeC07**] Fue pensado como un almacen de datos sensibles, eventualmente consistentes. Los arquitectos de Dynamo, diseñaron una plataforma en la que los clientes pueden programar el nivel de calidad de servicio que esperan recibir en sus aplicaciones. También es un sistema diseñado por encargo.

CEPH [**Wei07**] Es un sistema de archivos distribuido, de código libre y abierto, que busca ofrecer capacidades de almacenamiento masivo, en el orden de los petabytes, para aplicaciones científicas de alto desempeño (ver figura 2.1).

Keyspace[**Tre01**] Es un SAD para datos críticos, diseñado para satisfacer tres requerimientos: consistencia fuerte, tolerancia a fallas y alta disponibilidad. El código fuente es libre y abierto, al igual que en el caso de CEPH [Wei07].

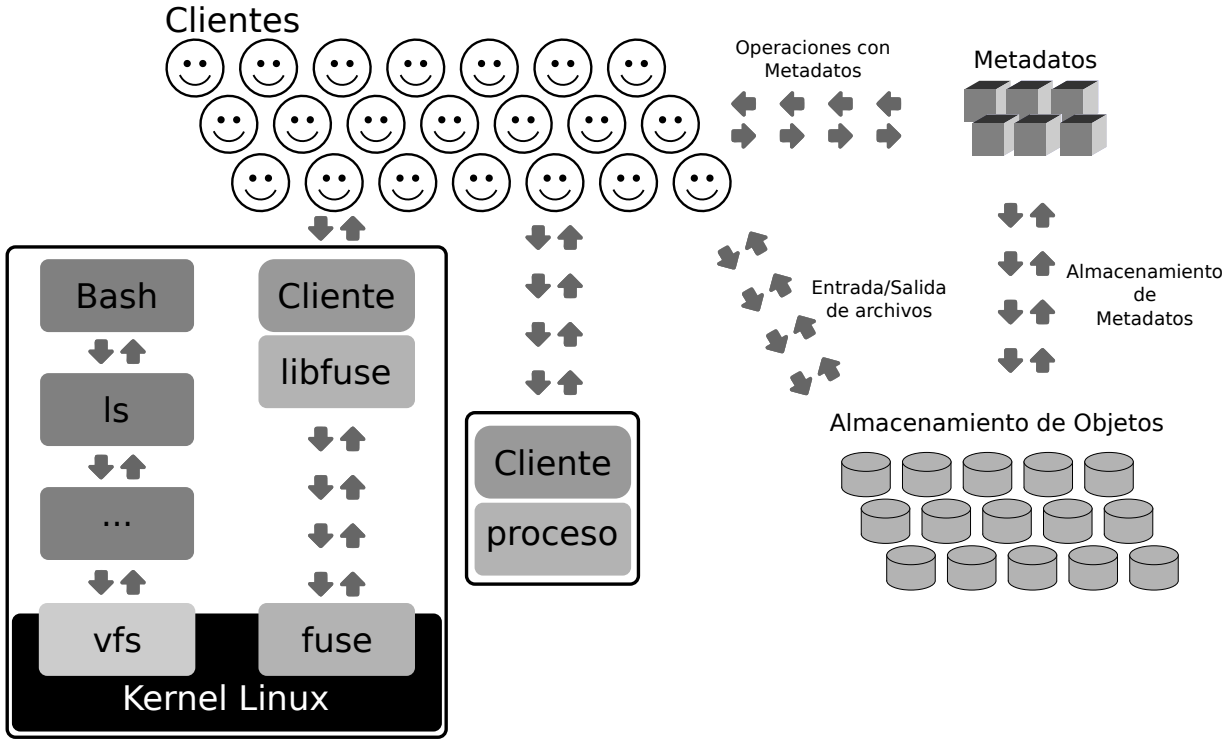


Figura 2.1: Vista panorámica del sistema CEPH [Wei07].

2.3 Almacenamiento basado en sistemas P2P

Recientemente se pueden identificar dos momentos importantes en la evolución de los sistemas de almacenamiento: el primero, cuando se migró la funcionalidad de un sistema centralizado hacia los sistemas distribuidos estáticos; el segundo, cuando la movilidad de los nodos introdujo incertidumbre en la estructura del sistema obligando a los diseñadores a abandonar el esquema cliente/servidor para optar por un esquema *Peer to Peer* (P2P). Tanto la distribución como la movilidad trajeron consigo retos particulares que han sido objeto de mucho del trabajo en el área.

Los sistemas P2P son sistemas distribuidos que consisten de nodos interconectados capaces de auto-organizarse en topologías de red, con el propósito de compartir recursos tales como contenido, ciclos de CPU, almacenamiento y ancho de banda. También son capaces de adaptarse a las fallas y acomodar poblaciones temporales de nodos mientras mantienen niveles de desempeño aceptables, sin requerir de intermediación o apoyo de una autoridad o servidor centralizado [SD04]. Puede entendérseles como una manera de estructurar aplicaciones distribuidas de modo que los nodos que componen al sistema asuman roles simétricos [Gro07]. En suma, se les puede caracterizar por tres propiedades: son auto-organizados, simétricos y distribuidos [Rou03].

En un sistema P2P, un nodo que se integra a la red agrega recursos y no representa, como en los sistemas cliente/servidor clásicos, carga adicional que pone en riesgo la escalabilidad del sistema. Sin embargo, un nodo que se ausenta, pone en riesgo la persistencia y la coherencia de la información y, eventualmente, la seguridad. Las preocupaciones clave de los diseñadores de sistemas P2P son [Has05]: la garantía de simetría en los roles de los nodos, la descentralización de las funcionalidades, la operación con participantes voluntarios sin compromisos, la localización rápida de recursos, el balance de carga, la protección contra las oscilaciones abruptas de la infraestructura, el anonimato, la escalabilidad, la persistencia de la información y la seguridad.

Las redes P2P se pueden clasificar a partir de diferentes criterios. Según su topología, se les divide en redes estructuradas y no estructuradas. En las primeras, la gráfica que se forma con los enlaces entre los peers se construye de acuerdo con un principio o procedimiento que produce una estructura regular. En tanto, en las redes no estructuradas los nodos que se unen al sistema obtienen diferentes grados de conexidad. En las redes estructuradas, suele utilizarse un método denominado tabla de hash distribuido (DHT) que genera una llave para la localización de los recursos y dirigir la búsqueda de los mismos. Ejemplos de estos sistemas son CAN, Chord, Pastry y Tapestry [BKK$^+$03]. Por otro lado, en las redes no estructuradas se produce una inundación con el nombre del recurso que se busca, como es el caso de Gnutella y Kazaa [RT04] ó bien, se centralizan las consultas en un solo punto, como en Napster [BR01].

Por su parte, Hasan et al. [Has05] listan lo que a su juicio son los beneficios que se pueden conseguir en el diseño de un sistema de archivos distribuido (DFS) basado en el paradigma P2P: simetría, balance de carga, localización de recursos, protección contra picos de demanda, escalabilidad, persistencia de la información y seguridad. Lo que, a su vez, se traduce en soluciones de diseño, tales como: mecanismos de emplazamiento, de localización, de codificación, y de acceso concurrente.

Risson y Moors [RT04] enfatizan la importancia de las operaciones de búsqueda en las redes P2P, partiendo de la noción de índice. El índice es un número que ubica a un documento en un espacio de almacenamiento. Las redes puede clasificarse por la existencia o ausencia de un índice (estructuradas ó no estructuradas), por la ubicación del índice (local, centralizado ó distribuido) o, por el significado que se le otorga a un índice (libre de semántica ó sensible a la semántica).

En una red basada en una tabla de hash distribuido o DHT (libre de semántica), el nombre del documento se proporciona a una función de dispersión que devuelve un índice, el cual codifica la posición del espacio donde debe localizarse dicho documento. Este mecanismo pasa por alto el contenido del documento que se maneja. En contraste, en un mecanismo de emplazamiento sensible a la semántica, se espera que aquellos documentos que se encuentren relacionados semánticamente deberán emplazarse conservando un sentido de cercanía en el espacio donde se almacenan.

Algunas propuestas sugieren que es posible compatibilizar los dos enfoques de búsqueda. Se espera que la búsqueda semántica pudieran asignarse a una capa que funcionaría encima de un servicio DHT. Sin embargo, aún queda por verse si ésta sería una solución eficiente [BKK+03].

2.4 Principios de diseño

A partir de la revisión de varias fuentes especializadas, fue posible reconocer los aspectos más relevantes por considerar, al momento de proponer la arquitectura para un nuevo sistema de almacenamiento distribuido.

Harren y su equipo [Har02] propusieron una arquitectura de tres capas: almacenamiento, DHT y consulta. Por su parte, Sung et al. [Sun05] consideran que una arquitectura de almacenamiento distribuido debe ubicar o reconocer las entidades que dan solución a cuatro problemas:

1. ¿Cómo se identifica el lugar donde se encuentra la información?
2. ¿Cómo se solicita esta información?
3. ¿Cómo se intercambia la información entre repositorios?

4. ¿Cómo se preserva la consistencia de la información?

En CEPH [Wei07] se resaltan tres principios de diseño:

1. Desacoplamiento entre datos y metadatos (no existe una tabla de asignación de archivos, sino una función que calcula la asignación de manera seudoaleatoria).

2. Manejo dinámico y distribuido de los metadatos (los servidores de metadatos pueden variar el espacio de almacenamiento que coordinan, dependiendo de la carga de trabajo).

3. Espacios autónomos y confiables de almacenamiento de objetos. Un dispositivo de almacenamiento de objetos u OSD[1] se compone de un CPU, una interfaz de red, una memoria cache local y un disco adjunto, se supone que los OSD se agrupan en clusters que se automonitorean, integran nuevos equipos, migran contenidos y reparan, en caso necesario.

El sistema de archivos de Google (GFS[2] [San03]) ofrece también un conjunto interesante de principios que merecen atención. Por un lado, se trata de un cluster o cúmulo de dispositivos de almacenamiento de bajo costo (*commodities*), administrados por una colección redundante de controladores en los que residen los metadatos, y que realizan también funciones de monitoreo y supervisión. Los clientes se conectan con alguno de los controladores y estos los reexpiden hacia los dispositivos de almacenamiento (*chunk servers*) que se encargan de completar el almacenamiento y la recuperación de la información. Los archivos se guardan en bloques de información que se replican en varios dispositivos de almacenamiento. Si un archivo excede el tamaño de un "chunk" entonces se corta en tantos pedazos como sea necesario. Un cluster puede incluir hasta algunos miles de dispositivos de almacenamiento administrados por unos cuantos controladores redundantes (en espejo). El sistema de archivos de Google, es el bloque de construcción elemental a partir del cual se desarrollan aplicaciones de almacenamiento de gran escala, como el sistema BigTable [Cha06].

[1] *Object Storage Device* por sus siglas en inglés.
[2] *Google File System*

3 Nuestra Propuesta

En este capítulo se revisan las consideraciones de diseño que guiaron la solución que proponemos.

Muchas organizaciones disponen de una infraestructura moderna de telecomunicaciones y cómputo que, sin embargo, se encuentra subutilizada. De poder coordinarse con eficiencia, esta misma infraestructura les permitiría disponer de una capacidad de almacenamiento en el orden de los terabytes o incluso petabytes, sin caer en costos agregados, al mismo tiempo que podrán mejorar varios de sus procesos. Para aprovechar estos recursos y gestionar su crecimiento hay que comenzar reconociendo la heterogeneidad de los componentes que participarían en un proyecto de tales características: diferentes marcas, modelos, sistemas operativos, capacidades, patrones de utilización, contenidos y, en muchos casos, hay que considerar que no existe y no se juzga pertinente introducir la figura de una entidad central coordinadora. Se busca entonces, construir un sistema distribuido, auto-organizado, en donde los participantes puedan asumir roles cooperativos y simétricos, es decir, que al mismo tiempo puedan ofrecer y recibir recursos hacia y desde el colectivo en el que se integran, minimizando en lo posible las dependencias sobre puntos que concentren algún tipo de actividad.

3.1 Consideraciones de diseño

Luego de revisar los trabajos destacados en el área, pudimos reconocer los aspectos más importantes que deben tomarse en cuenta al momento de proponer la arquitectura para un nuevo sistema de almacenamiento distribuido. La tabla 3.1 condensa los elementos de diseño que se repiten en las referencias consultadas.

Como hemos adelantado, el diseño de un Sistema de Almacenamiento Distribuido requiere:

1. El conjunto de requisitos funcionales y no funcionales que se busca atender.
2. Los parámetros de desempeño que garanticen la calidad de servicio.
3. La entidad o espacio elemental de almacenamiento.

Aspecto	Harren	Sung	Hasan	CEPH	GFS
Administración dinámica y distribuida de metadatos				✓	
Balance de carga			✓		✓
Consultas via DHT	✓				
Control de acceso		✓			
Escalabilidad			✓		✓
Espacios autónomos y confiables de almacenamiento				✓	✓
Identificar información		✓			
Localización de recursos			✓		
Persistencia			✓		
Preservar la consistencia		✓			
Seguridad			✓		
Separación datos - metadatos.				✓	✓
Simetría			✓		

Tabla 3.1: Aspectos a Considerar en el Diseño de un SAD

Los requerimientos funcionales se refieren a las necesidades del usuario o cliente final del sistema. En tanto, los requerimientos no funcionales se refieren a los atributos de calidad que tienen que satisfacerse para que el sistema funcione adecuadamente. Los parámetros de desempeño de un SAD evalúan la eficiencia en el uso de los recursos implicados en la solución de almacenamiento, así como la calidad de servicio que se ofrece. Por cuanto se refiere a la entidad elemental de almacenamiento, de primera instancia podría pensarse en los discos asociados con las computadoras participantes en un SAD. Sin embargo, en vista de la escala a la que se puede crecer y la complejidad de las operaciones que pueden tener lugar, se considera contraproducente asumir una dependencia tan fuerte sobre dispositivos individuales. Es deseable construir un dispositivo virtual que exhiba una vida útil más prolongada y un comportamiento más estable que lo que puede esperarse de los dispositivos físicos a partir de los cuales se construya.

3.2 Requerimientos funcionales

El diseño de un SAD incluye la formalización de los procesos de tratamiento de la información y los aspectos arquitectónicos que gobiernan su construcción. A partir de estos se identifican los requerimientos funcionales y no funcionales del sistema, respectivamente.

Los requerimientos funcionales definen el comportamiento interno del software: cálculos, detalles técnicos, manipulación de datos y otras funcionalidades específicas que muestran cómo los casos de uso serán llevados a la práctica. Son complementados por los requerimientos no funcionales, que se enfocan en el diseño o la implementación [Som06]. En este sentido, los primeros responden a la pregunta ¿qué se necesita?, mientras que los segundos guian el diseño de la arquitectura al describir cualidades que debe tener el producto final.

Monitoreo y control Se requiere una entidad del sistema capaz de llevar un control de los

componentes en operación, verificar los parámetros de desempeño del sistema (balance de carga, redundancia, disponibilidad) e invocar a los mecanismos de regulación o reparación [Des91], por ejemplo, cuando haya que migrar contenidos de un disco viejo a un disco nuevo, o cuando el sistema se desvíe de sus especificaciones.

Con el fin de atender este requerimiento, el prototipo es capaz de registrar todos los eventos que suceden en él mediante el uso de bitácoras, empleando estas bitácoras el sistema será capaz de detectar fallas, y en consecuencia lanzar el procedimiento correspondiente para recuperarse de la misma.

Metadatos Los metadatos son la información a partir de la cual se gestiona la aplicación. Son los datos relacionados con los usuarios, sus cuotas, los atributos de acceso a los recursos almacenados, la localización física de los archivos, tamaño, relaciones, etcétera. En breve, se trata de una base de datos. Sin embargo, su diseño incluye decisiones vinculadas no solo con su contenido y operaciones, sino con su localización. Una base de datos centralizada ofrece una administración muy sencilla, pero puede convertirse en un cuello de botella que limita la calidad de servicio. En contraste, una base de datos distribuida puede ofrecer un alto desempeño pero, a cambio, se requieren mecanismos de acceso sincronizado y control de información replicada.

Los metadatos del prototipo son almacenados en una base de datos relacional. El manejador de base de datos elegido es PostgreSQL. La estructura de estos metadatos es definida en capítulo 4.

Consistencia y sincronización En un SAD se necesita definir una política para sincronizar el manejo concurrente de contenidos, esto quiere decir que se debe definir de qué forma se soportan las solicitudes simultáneas de acceso a un mismo recurso. Se pueden soportar, por ejemplo, varias lecturas simultaneas y una sola escritura.

Existen archivos, tanto de los usuarios como del propio sistema, que pueden encontrarse replicados en varios sitios. En estas circunstancias, una operación de escritura que modifique una réplica deberá repercutirse con el fin de garantizar la consistencia del conjunto de copias. Luego de este aseguramiento, cualquier entidad tendría acceso a la misma información, independientemente de la copia que recibiera. Sin embargo, existe una ventana de tiempo durante la cual el sistema cuenta con al menos dos versiones del mismo archivo. En este lapso, la falla de algún componente involucrado pondría en riesgo el éxito de la operación. Las especificaciones del sistema deben considerar, por tanto, la clase de fallas que pueden tolerarse y los protocolos con los que se les haría frente, llegado el caso.

En el prototipo se optó por un mecanismo muy sencillo para preservar la consistencia, el cual consiste en establecer que todos los objetos almacenados son inmutables, con lo que evitamos la necesidad de introducir un mecanismo de sincronización al momento

de realizar operaciones de lectura; en cuanto a la operación de escritura, el mecanismo a seguir es el siguiente: Cada vez que un usuario quiere "editar" un objeto previamente almacenado en el sistema, lo que realmente sucede es que el objeto modificado se almacena como un archivo nuevo, que corresponde a la siguiente versión del archivo original, a partir de ese momento, todas las consultas de ese objeto obtendrán como resultado la nueva versión del archivo.

Integridad y confidencialidad La información se expone a un medio con limitaciones físicas por el que debe viajar o almacenarse. Este medio representa una fuente de errores que pueden modificar los dígitos binarios que componen un archivo. Preservar la integridad es garantizar que el contenido que el usuario recupera del sistema de almacenamiento corresponde al contenido que almacenó en él originalmente, aún en presencia de factores que pueden degradar el servicio de almacenamiento. Por otro lado, la confidencialidad se refiere a la garantía para acceder a un archivo y su contenido, solo en los casos que el usuario que lo solicita tiene derechos sobre el mismo.

Una característica común de los SAD es el uso de la redundancia de información para proveer tolerancia a fallas y, en particular, integridad. La estrategia más sencilla consiste en la replicación de archivos. Esta técnica se utiliza por ejemplo en PAST [AP01] y Farsite [AA02]. En estos sistemas, los componentes de almacenamiento participan con sus capacidades individuales para guardar réplicas de cada archivo en custodia. En contraste, existen sistemas como OceanStore [Kub00] o Intermemory [Che99], en los que la redundancia se implementa mediante técnicas de detección y correción de errores, por ejemplo usando códigos de la familia Reed-Solomon, códigos de red, fuentes digitales, entre otros. En estos casos, cada archivo se transforma en un conjunto de bloques codificados que luego son alojados en los componentes de almacenamiento disponibles.

La selección de una estrategia en particular tiene un fuerte impacto en aspectos operacionales tales como el costo y la administración. Estudios recientes [RL05] sugieren que cuando los dispositivos exhiben un comportamiento estable de largo plazo, el sistema puede sacar provecho de una estrategia conservadora basada en códigos. En tanto, si los dispositivos tienen un comportamiento intermitente, el sistema debe basarse en una estrategia más agresiva, usando replicación. También se sabe de estrategias mixtas que combinan lo mejor de cada opción.

Por cuanto se refiere a la confidencialidad, se trata de dar solución a un requerimiento de diseño teniendo en cuenta qué tanto los usuarios, como el propio sistema pueden almacenar información sensible y deben disponer de esta garantía. La información puede quedar expuesta al uso no autorizado ya sea durante su transmisión, o bien, durante su almacenamiento. En ambos casos se deben tomar las previsiones que limiten el riesgo. Sin embargo, la confidencialidad de la información no solo depende de su aseguramiento criptográfico, se deben considerar los mecanismos que limitan el acceso a

la información dependiendo del rol que cada usuario tiene asignado en una organización.

La satisfacción de la integridad se logra en el prototipo, aplicando el algoritmo MD5 a los contenidos que son almacenados, con la finalidad de verificar que los datos no se hayan corrompido, y en caso de que esto ocurra, detectar en qué momento del proceso de almacenamiento o recuperación, ocurrió la pérdida de datos, en tanto que la confidencialidad se atiende por medio del manejo de cuentas y grupos de usuarios que le permiten al sistema otorgar o denegar el acceso a los objetos almacenados, en función de los privilegios asociados con su cuenta.

Indexación y búsqueda Cada uno de los dispositivos de almacenamiento que participan debe contar con un identificador único que lo distinga del resto. Esta decisión dota al conjunto de una estructura a través de la cual es posible definir la posición donde se guarda un archivo, pero también define la manera de acceder al dispositivo que corresponde. Los mecanismos tradicionales encontrados en los sistemas de archivos no parecen adaptarse bien al manejo de volúmenes masivos. En estos casos se requieren mecanismos ágiles y distribuidos para emplazar y localizar contenidos. En los sistemas P2P, por ejemplo, se observa que los algoritmos basados en tablas de dispersión distribuida (DHT[1]) recuperan información basándose en un identificador único [BKK+03]. Se proporciona el nombre del archivo que se busca y la función de dispersión lo relaciona con la posición o identificador del dispositivo en el que se guarda. Sin embargo, para efectuar las consultas más elaboradas no basta proporcionar el nombre de un archivo, en muchos casos se requieren consultas semánticas, que se refieren a conceptos o palabras clave que describen el contenido de un grupo de documentos.

La indexación y búsqueda de información, queda "resuelta" a nivel del prototipo haciendo uso de los metadatos; sin embargo no se contempló en esta versión del sistema el uso de un índice distribuido (aunque está considerado en la arquitectura) debido a que no se necesita para esta etapa del proyecto.

Interfaz de aplicación Los usuarios de un SAD podrán dar de alta y baja sus equipos, conectarse, desconectarse, almacenar archivos, borrarlos y recuperar contenidos usando diferentes criterios de búsqueda. Por otro lado, cada máquina deberá interactuar con un conjunto heterogéneo de componentes incluyendo programas escritos en diferentes lenguajes, diferentes sistemas operativos, diferentes capacidades de procesamiento y almacenamiento, entre varias posibilidades. Por lo anterior, es muy importante garantizar que la interfaz de aplicación facilite la interoperabilidad. Igualmente es importante que la instalación de las diferentes versiones del SAD tengan un impacto mínimo en las máquinas de los usuarios finales. Desde esta perspectiva, los usuarios deberían considerarse como clientes de un servicio. Como en el caso de los abonados del sistema telefónico, a quienes se les instala un equipo cuando se suscriben y para ellos resultan

[1] *Distributed Hash Table* por sus siglas en inglés

transparentes las actualizaciones posteriores o cambios en el sistema que les da soporte.

Este requerimiento puede satisfacerse dotando al prototipo de una interfaz que le sirva para conectarse con otros sistemas y así poder brindar servicios de almacenamiento separando lo más posible la interfaz de aplicación del núcleo del sistema, y que la conexión entre ellos se realice a través de un API .

3.3 Requerimientos no funcionales

Los requerimientos no funcionales se refieren a los aspectos de diseño que gobiernan la arquitectura de las soluciones. A partir de ellos se establecen criterios y prioridades que se busca atender en el proceso de desarrollo del software.

Flexibilidad Aún faltan estudios concluyentes que definan la mejor manera de resolver o proveer cada una de las funciones que se requieren en un SAD. Por otro lado, parece razonable pensar que un sistema de este tipo pueda evolucionar y encontrar nuevas maneras de proveer las funciones que ya tiene implantadas. Por lo anterior, un diseño estratégico pasaría por definir estas funciones como módulos o cajas negras con entradas y salidas bien establecidas, dejando la posibilidad de variar su construcción, en tanto las interfaces permanezcan sin cambios.

Por un lado, siguiendo una de las ideas propuestas en [Sun05] los componentes del sistema, pueden estar presentes o no (en el caso del prototipo, pueden estar activos o no), dependiendo de las tareas que van a desempeñar en el sistema, lo que significa que los módulos de la arquitectura están debilmente acoplados entre sí. Por otro lado, es muy sencillo agregar funcionalidad al prototipo debido a su diseño, dado que los componentes del sistema pueden verse como cajas negras, con una interfaz bien definida que brinda la posibilidad de socializar el desarrollo del proyecto repartiendo la responsabilidad de implementar los componentes por separado y finalmente integrarlos para construir el producto final.

Interoperabilidad La posibilidad de intercambiar información proveniente de distintas fuentes, sugiere que ésta puede registrarse de acuerdo con formatos diferentes. Ya se ha mencionado que se trata de una tendencia que cobrará mayor relevancia con el tiempo. Dependiendo del nivel operativo en el que se aborde el problema, habrá aspectos particulares implicados en la solución. En todo caso, la interoperabilidad será posible siempre que se definan mecanismos estandarizados para invocar o proveer servicios entre las entidades involucradas. Se trata de un principio de diseño que complementa a la modularidad y que cobra mayor relevancia en la misma medida en que crece la heterogeneidad de los componentes que buscan coordinarse.

Dado el diseño de la interfaz de usuario y comunicaciones , el prototipo puede adaptarse facilmente para interactuar con otros sistemas, este aspecto se explicará más a detalle en la sección A.11.

Escalabilidad Se refiere a la manera en que se planea el crecimiento del sistema. Se entiende que al aumentar el número de componentes de almacenamiento crece la capacidad del sistema, pero igualmente pueden crecer los conflictos por el uso compartido de recursos, comenzando por la designación de los sitios de almacenamiento, los mecanismos para el emplazamiento y localización de la información, las políticas de acceso, entre muchos aspectos que deben considerarse. Visto de otra forma, el problema puede plantearse como una pregunta, ¿cómo pasar de un sistema basado en una red local, a un sistema del tamaño de la Internet, de modo que no se degraden sus prestaciones mientras crece? Desde esta perspectiva, la escalabilidad se puede entender como un principio de diseño que permite que los sistemas evolucionen de lo sencillo a lo complejo.

La interacción con el sistema se hará a través de una interfaz que le permitirá agregar más nodos para aprovechar sus recursos, estos pueden ingresar al sistema de diferentes maneras, las cuales se mencionan en el capítulo 5.

Confiabilidad y Tolerancia a Fallas La confiabilidad es una garantía de continuidad de los servicios que ofrece un sistema. Se entiende que un sistema es más proclive a fallas, a medida que crece su complejidad. Para poder soportar las diferentes garantías de servicio que ofrece, un SAD deberá contar por un lado, con una entidad que despliegue un monitoreo continuo de las capacidades del sistema y, por otro lado, de un conjunto de recursos redundantes a los que pueda recurrirse, cada vez que se asuma que un componente ha caido en falla y deba intervenirse. Sin embargo, las fallas que pueden presentarse en un sistema no se reducen a la caída de un componente que queda fuera de servicio. En algunos casos se presentan comportamientos intermitentes, erráticos e, incluso, algunos tan sutiles como los que caracterizan a las denominadas fallas bizantinas. La especificaciones del diseño deberán considerar las clase de fallas que pueden tolerarse y, en consecuencia, los mecanismos con los que se deberá hacerles frente.

El prototipo es en sí mismo, un espacio de almacenamiento confiable y autónomo, debido a la manera en que almacena los datos; dicho mecanismo se explica a detalle en el capítulo 4.

3.4 Parámetros de Desempeño

El beneficio inmediato del almacenamiento distribuido es que se logra la independencia entre la información y su almacén. Visto de otra forma, los archivos que se guardan en un repositorio colectivo no dependen de un solo dispositivo para su recuperación. Si un archivo

estuviera guardado en una sola máquina, entonces la falla de ésta cancelaría su recuperación. Naturalmente que, si se guardan varias copias del mismo en otras tantas máquinas, entonces se consigue esta independencia del almacén, pero a un costo que puede ser excesivo. Los costos de participar en un SAD se determinan a través de sus parámetros de desempeño. Como se ha visto, las referencias consultadas no coinciden totalmente en las medidas que sirven para evaluar el trabajo de estos sistemas. Hemos elegido una lista breve de parámetros que miden la eficiencia en el uso de los recursos y la calidad de servicio [Laz09]:

Redundancia El propósito de un sistema tolerante a fallas es detectar y, si es posible, reparar los errores antes de que se manifiesten en la interfaz del usuario. La clave del diseño consiste en el uso de la redundancia. Este término se refiere a los recursos adicionales que se incluyen en el sistema, que no se usarían en una situación ideal. La redundancia se requiere para enmascarar la falla, luego que se detecta el error. Se pueden distinguir tres tipos de redundancia: de recursos físicos, de tiempo y de información. La primera se refiere a la replicación de los componentes del sistema. La segunda, a la repetición de una acción de cómputo o de comunicaciones, sobre el dominio del tiempo. Por último, la redundancia de la información se refiere a una técnica de codificación que introduce un exceso de dígitos binarios para representar la información. En un sistema bien diseñado se espera conseguir la máxima protección contra fallas, usando la menor cantidad de recursos redundantes. Entonces, la redundancia se evalúa contabilizando el conjunto de recursos en exceso y comparándolo con el número de fallas que con ello se consigue tolerar.

El prototipo es capaz de ofrecer dos tipos de "niveles de servicio", las técnicas de generación de redundancia empleadas para cada uno de estos niveles de servicio son la replicación simple y la codificación de los datos utilizando el ADI (ver sección A.11, ambas técnicas ofrecen diferentes grados de compromiso entre la redundancia generada y la disponibilidad que ayudan a alcanzar.

Balance de carga Se refiere a la cantidad de información que un dispositivo recibe en resguardo, en relación con el resto de los componentes de almacenamiento . Se necesita establecer un criterio, ya sea de equidad ó de justicia en función de las características y capacidades de los nodos que participan en el sistema; de tal forma que puedan aprovecharse los recursos disponibles de la mejor manera posible. Porsteriormente hay que medir el cumplimiento de esta característica.

La equidad se refiere a asignar, idealmente, la misma cantidad de información a todos los dispositivos del conjunto; este criterio resulta adecuado en un ambiente homogéneo. La justicia, por su parte, se refiere al hecho de asignar a cada nodo, una cantidad de trabajo adecuada según sus capacidades; en este caso, componentes con mejores recursos reciben mas carga de trabajo. En cualquiera de las circunstancias, el balance de carga se evalúa mediante una estadística que caracteriza la cantidad de información

asignada a los dispositivos. Una medida de dispersión, como la desviación estándar, describe el grado de cumplimiento del criterio de asignación ideal.

Partiendo del supuesto de que todos los nodos internos tienen las mismas capacidades tanto de procesamiento y almacenamiento de datos, y que no se sabe el tamaño de los archivos que se van a guardar en la celda, definimos en la sección A.11, un criterio que garantiza que ninguna unidad de datos excederá cierta cota, además se procura seleccionar el lugar en el que se almacenará o procesará cada unidad de datos en base a un criterio que mantenga balanceada la carga que reciben los nodos participantes; en el caso de este prototipo, el criterio de selección consiste en acomodar a los espacios en que se almacena la información en un círculo lógigo y empleando el mecanismo de Round Robin para seleccionar al responsable para almacenar o procesar los datos. Será tema de otro trabajo evaluar mecanismos alternativos de selección con el fin de mejorar el balance de carga.

Disponibilidad Este parámetro tiene diferentes medidas a través de las cuales se evalúa. Puede referirse a la probabilidad de encontrar un recurso cuando se le busca, pero también implica el tiempo de respuesta medido desde el momento en que se solicita el documento, hasta el momento en que se entrega al solicitante (latencia). Puede entendérsele como un indicador de la satisfacción de los usuarios cuando invocan al sistema buscando recuperar un contenido. Por otro lado, la calidad de servicio implica un costo asociado con los recursos donde se almacena la información y con el ancho de banda que se requiere para su recuperación. No es lo mismo guardar un archivo en un solo punto del sistema y transmitirlo hasta el lugar donde se requiera, que codificarlo y guardarlo en varios puntos. La segunda opción implica que el documento está más disponible, pero también implica que se han utilizado un mayor número de almacenes para soportar el mismo servicio. Un SAD debería permitir a cada usuario elegir el grado de servicio con el que espera recuperar su información.

El parámetro de disponibilidad es atendido por nosotros empleando técnicas de redundancia que ayuden a mejorar la probabilidad de recuperar los contenidos almacenados en la celda aún en presencia de fallas.

En la tabla 3.2 se observa cómo los aspectos de diseño identificados en la sección 3.1 tienen impacto sobre los parámetros de desempeño que acabamos de mencionar.

Principio	Redundancia	Balance de Carga	Disponibilidad
Administración dinámica y distribuida de metadatos	✓	✓	✓
Consultas via DHT			✓
Control de acceso		✓	✓
Escalabilidad		✓	✓
Espacios autónomos y confiables de almacenamiento	✓		✓
Identificar información			✓
Localización de recursos		✓	✓
Persistencia	✓		✓
Preservar la consistencia	✓		✓
Seguridad	✓		✓
Separación Datos - Metadatos.	✓	✓	✓
Simetría		✓	

Tabla 3.2: Aspectos a Considerar en el Diseño de un SAD vs. Parámetros de desempeño

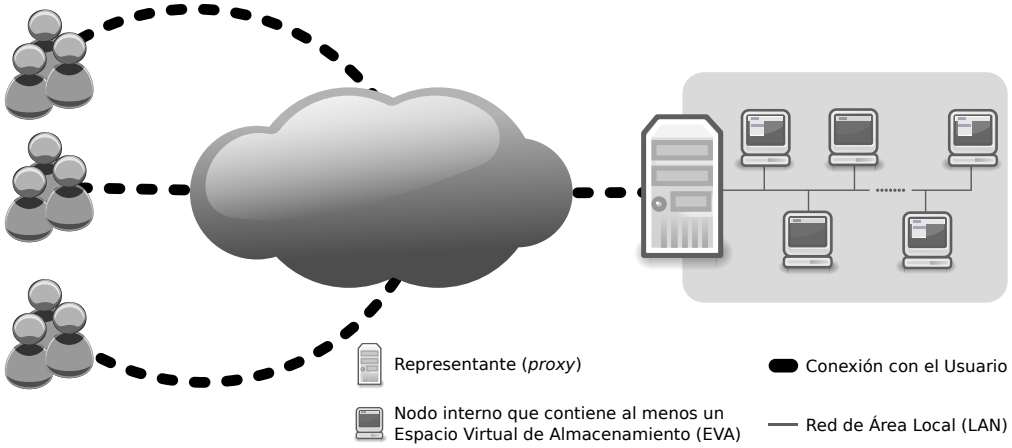


Figura 3.1: Celda de almacenamiento integrada por un conjunto de dispositivos bajo la coordinación de un representante.

3.5 Dos casos de uso: almacenamiento y recuperación de archivos

Se propone la construcción de un dispositivo virtual llamado **celda de almacenamiento**. Cada celda estará integrada por un conjunto de dispositivos de almacenamiento denominados nodos internos, los cuales son coordinados por un nodo representante. Puede pensarse en cada celda como una red P2P no estructurada, donde el dispositivo de coordinación hace el papel de supernodo. Se sabe que un conjunto con estas características puede presentar una interfaz única a través de la que ofrece capacidad extendida y parámetros de confiabilidad mejorados [Que10].

Se trata de un caso similar al del protocolo TCP, que ofrece un servicio con garantía de entrega y que, sin embargo, se construye partiendo del protocolo IP, que no ofrece garantías (lo que

se conoce como *"best effort"*). La clave de este comportamiento aparentemente paradójico se encuentra en la aplicación de recursos redundantes. La confiabilidad se construye sobre la base de la redundancia de componentes de almacenamiento y redundancia de información. En la figura 3.1 se ofrece una vista panorámica de la celda de almacenamiento.

Por sí misma cada celda es un sistema de almacenamiento distribuido que cubre todos los requerimientos considerados en este capítulo. Buscamos un diseño flexible, basado en componentes de software que puedan implementarse en cada dispositivo de almacenamiento, o bien, puedan repartirse sobre nodos especializados.

3.5.1 Almacenamiento de archivos

Para hacer mas ágil la lectura del resto de este documento, hay algunos términos que conviene definir:

Unidad de datos Cualquier secuencia de bits encapsulada en un archivo, es decir cualquier archivo, leido en binario.

UMA Unidad Máxima de Almacenamiento. Es la longitud máxima permitida para almacenar una unidad de datos. Cualquier archivo que exceda esta medida tiene que dividirse en fragmentos que cumplan con esta convención. La UMA es un análogo de la unidad máxima de transferencia (MTU) que se usa en las redes de conmutación de paquetes. Se sabe que las solicitudes de atención con longitud arbitraria pueden degradar sensiblemente el desempeño de un sistema, dando lugar a colas de espera que crecen muy rápidamente. Se trata del mismo principio de la caja rápida que se usa en los supermercados. Es decir, un cliente puede formarse en una caja rápida, siempre que la cantidad de productos que va a comprar no exceda un límite. La definición de la UMA tiene impacto sobre el balance de carga, porque una unidad pequeña permite conseguir una distribución más homogenea de los datos. Sin embargo, también puede producir un exceso de carga para los procesadores durante el almacenamiento y la recuperación que impacte en los tiempos de servicio.

Fragmento Cada una de las unidades de datos con longitud menor o igual a la UMA, y que aún no han sido codificadas.

Bloque Cada una de las unidades de datos a que da lugar un fragmento, luego de aplicar sobre éste replicación simple o una técnica de codificación. En el caso de la replicación simple, un fragmento da lugar a dos bloques idénticos. En el caso de ADI, cada fragmento da lugar a n bloques, también denominados dispersos.

ADI Algoritmo de dispersión de información [Rab89]. Convierte una unidad de datos en n dispersos tales que bastan cualesquiera m de ellos para reconstruir la unidad original,

con $n > m > 1$. La relación entre estos parámetros juega un papel muy importante en la definición de la cantidad de información redundante y la tolerancia a fallas. Cuando m es cercana a n, entonces el algoritmo tolera pocas fallas, pero igualmente requiere poca información redundante. Cuando m es cercana a 1, el algoritmo soporta un mayor número de fallas, pero produce una cantidad muy grande de información redundante. También se tiene que n debe ser mayor o igual a 3.

Checksum Es un mecanismo para verificar la integridad de una unidad de datos. Se parte de un polinomio especial $g(x)$ con coeficientes binarios, denominado polinomio generador. Enseguida se genera una representación polinomial de la unidad de datos $m(x)$, que se multiplica por x^r, donde r es el grado de $g(x)$. El resultado se divide por $g(x)$ y se toma el residuo de la división $d(x)$. Se hace la suma $m(x) * x^r + d(x)$ y se pasa a formato binario otra vez. Esta es la nueva unidad de datos. Puede entenderse como una secuencia compuesta de dos partes. La primera es idéntica a los datos originales, la segunda, que consta de r bits, se denomina secuencia de verificación de integridad. Luego de que se ha almacenado o transmitido, se puede utilizar un proceso semejante y comparar las secuencia de verificación resultante con la que acompaña a los datos. Si éstas no coinciden se dice que los datos se han corrompido. En cuyo caso la unidad de datos debe desecharse.

EVA Espacio Virtual de Almacenamiento. Se trata de una "caja" ficticia, con capacidad acotada, a la que se asocian los bloques de datos que van a almacenarse. Cada disco puede contener uno o varios EVA, este caso puede presentarse cuando se tienen discos de diferentes capacidades.

Round Robin Es una técnica que consiste en acomodar lógicamente un conjunto de recursos en una cola circular para seleccionar lo más equitativamente posible cada uno de ellos y mejorar el balance de carga.

Cuando un usuario desea almacenar un archivo, se conecta a la celda a través de su representante, el cual, luego de registrar la operación de almacenamiento, delegará la responsabilidad de almacenar el archivo en uno de los nodos internos de la celda. Cuando el nodo seleccionado reciba la orden de almacenar un archivo, si su tamaño excede la UMA definida en el sistema, lo dividirá en tantos fragmentos como sea necesario de tal forma que ninguno exceda ésta medida; acto seguido, el nodo enviará cada uno de los fragmentos resultantes a sus vecinos (y conservará algunos en caso de ser necesario) para que sean procesados, según el nivel de servicio que se le esté otorgando al usuario (serán replicados o codificados usando el algoritmo de dispersión de información), para generar los bloques que finalmente se almacenarán en los espacios virtuales de almacenamiento designados. Tal intercambio de mensajes, se ilustra en la figura 3.2.

En el algoritmo 1 se define la acción que realiza el representante de la celda cuando recibe la orden de almacenar un archivo, esta acción se ilustra en la figura 3.2a.

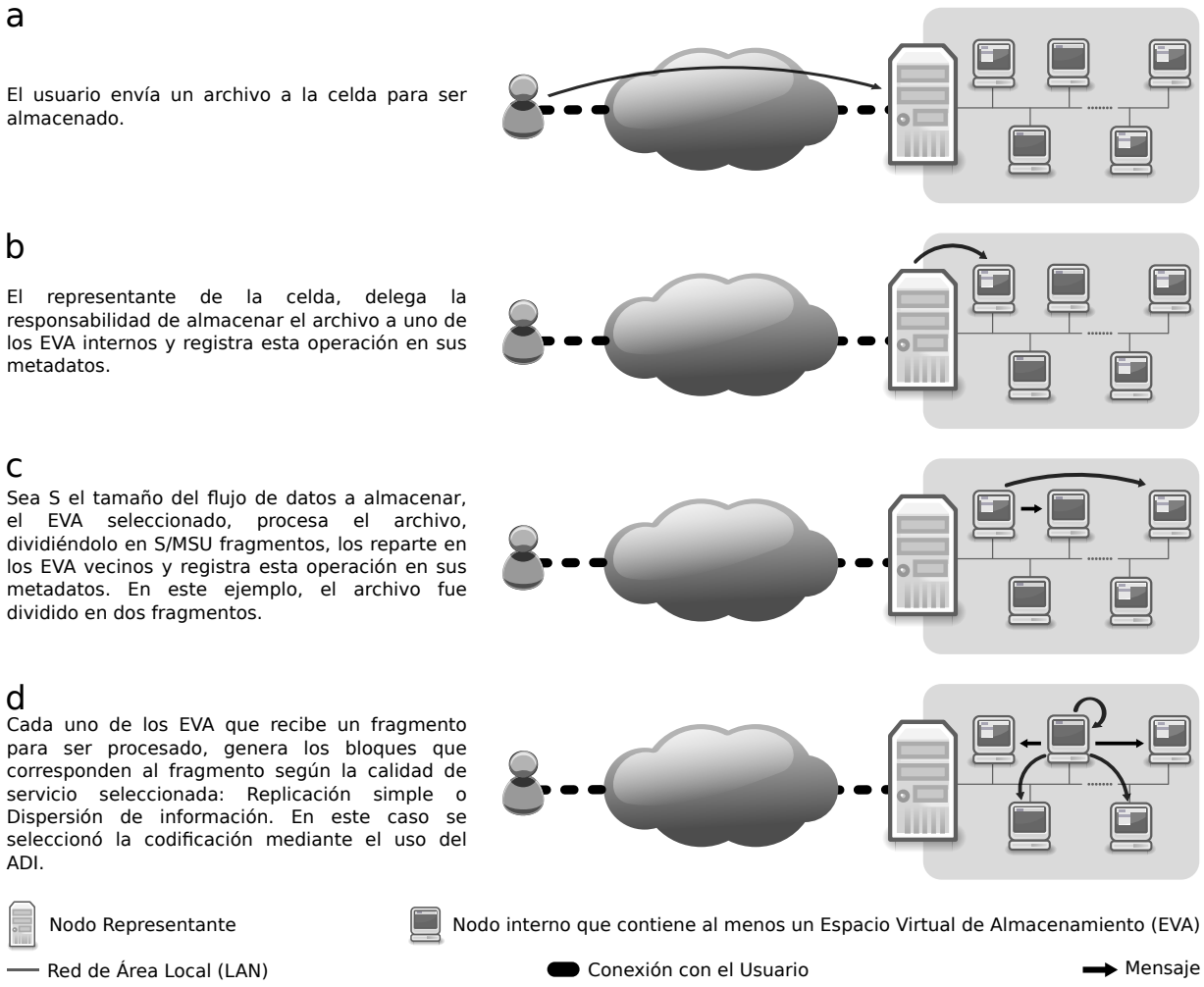


Figura 3.2: Secuencia de mensajes durante el proceso de almacenamiento

Algoritmo 1 El representante de la celda recibe la orden de almacenar un archivo

Entrada: Una tupla de la forma: (DB, F)

Salida: Una tupla de la forma $(estado, version)$

1: $destino \leftarrow siguiente(V)$
2: $(id, version) \leftarrow registraArchivo(DB, F)$
3: $estado \leftarrow enviaArchivo(destino, F)$
4: **devolver** $(estado, version)$

En el algoritmo 2 se ilustra el algoritmo que sigue un EVA para almacenar un archivo. El algoritmo recibe como entrada una tupla de la forma (V, S, C, A, DB, MSU) donde:

V Es un conjunto que contiene todos los EVA que integran la celda de almacenamiento.

S Un flujo de datos que contiene el archivo que se va a almacenar.

C Una cadena de caracteres que contiene el *checksum* del archivo que se quiere almacenar, con la finalidad de verificar que el archivo fue recibido correctamente.

A Una referencia para identificar el archivo dentro del sistema.

DB Un objeto que representa el controlador de la base de datos en donde se almacenan los metadatos generados durante la operación.

MSU Un número entero positivo que representa el tamaño de la unidad máxima de almacenamiento.

Algoritmo 2 Procesar archivo

Entrada: Una tupla de la forma: (V, S, C, A, DB, MSU)
Salida: Un valor que indica si la operación se realizó exitosamente o hubo algún error.

```
1: exito ← Verdadero
2: sí C = getCheckSum(S) entonces
3:    i ← 0
4:    F ← rebana(S, MSU)
5:    para f ∈ F hacer
6:       destino ← siguiente(V)
7:       checksum ← getCheckSum(f)
8:       id ← registraFragmento(DB, f, A, i, checksum, talla(f), destino)
9:       sí destino = YO() entonces
10:         procesaFragmentoLocal(f, checksum, id)
11:      otro
12:         conexion ← conectar(destino)
13:         procesaFragmento(f, checksum, id)
14:      fin sí
15:      i ← i + 1
16:   fin para
17: otro
18:   exito ← Falso
19: fin sí
20: devolver exito
```

Para entender mejor el algoritmo 2, hay algunas líneas del mismo que conviene explicar:

En la línea 4 se llama a un procedimiento que simplemente divide el flujo de datos S en fragmentos de tamaño menor o igual a la unidad máxima de almacenamiento dada por el parámetro MSU.

En la línea 6 se selecciona el siguiente EVA al que se le va a asignar que procese un fragmento de acuerdo con cierto criterio, para el caso de este prototipo, los EVA que componen la celda, son ordenados lógicamente en una cola circular, y el EVA seleccionado es el siguiente de la lista.

En la línea 7 se calcula el *checksum* de cada uno de los fragmentos.

Algoritmo 3 Procesar Fragmento

Entrada: Una tupla de la forma: (C, F)
Salida: Un valor que indica si la operación se realizó exitosamente o hubo algún error.

1: $exito \leftarrow$ **Verdadero**
2: **sí not** $C = getCheckSum(F)$ **entonces**
3: $exito \leftarrow$ **Falso**
4: **otro**
5: $procesaFragmentoLocal(F, checksum)$
6: **fin sí**
7: **devolver** $exito$

3.5.2 Recuperación de archivos

La operación inversa al proceso de almacenamiento es la recuperación, en ella, el usuario le solicita a la celda, a través de su representante, la entrega de un archivo. Cuando el representante recibe la solicitud, consulta sus metadatos para saber a qué EVA le delegó la responsabilidad de almacenar el archivo, y le envía un mensaje con la orden de recuperarlo.

Cuando el EVA correspondiente recibe dicha solicitud por parte del representante, busca en sus metadatos, a quienes de sus vecinos les delegó el procesamiento de los fragmentos que generó a partir del archivo solicitado, y en consecuencia les solicita que le devuelvan el fragmento que procesaron.

Cuando cada uno de los EVA's que procesaron un fragmento recibe la orden de devolverlo, revisa en sus metadatos, que tipo de bloques generó para el fragmento que procesó, en caso de haberlo replicado, simplemente devuelve la réplica que almacenó, ya sea en su propio disco, o en el EVA vecino. En el caso de haber brindado el nivel de servicio ADI, el nodo consulta en sus metadatos a quienes de sus vecinos les dio a guardar los bloques del fragmento que es su responsabilidad de volver, recupera algunos de ellos para regenerar el fragmento, y lo devuelve.

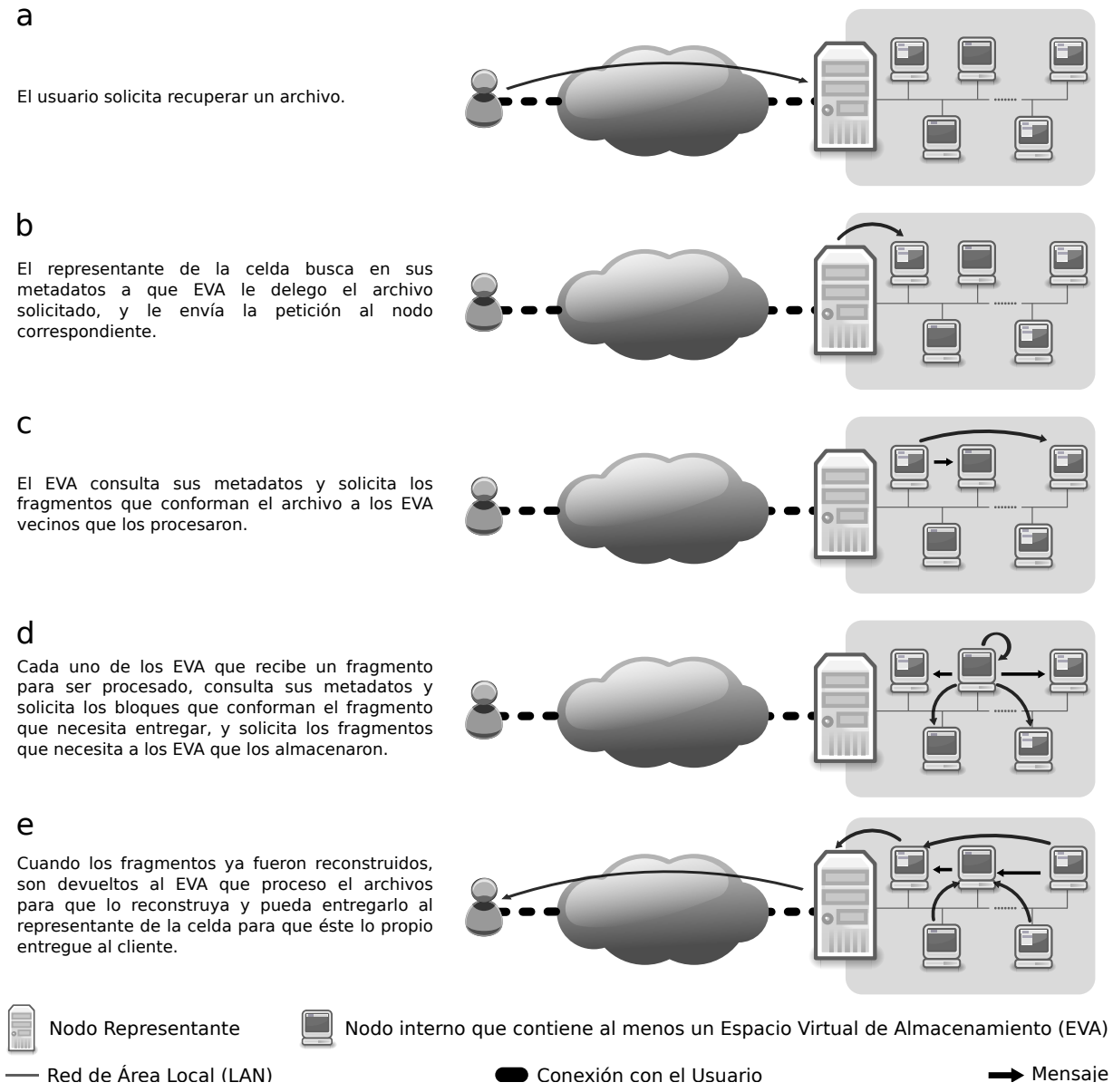


Figura 3.3: Secuencia de mensajes durante el proceso de recuperación

Cuando todos los fragmentos han sido recuperados, el EVA responsable de entregar el archivo une los fragmentos y le entrega el archivo reconstruido al representante de la celda, quien hace lo propio al devolver el archivo al usuario.

3.6 Definiendo el espacio de almacenamiento

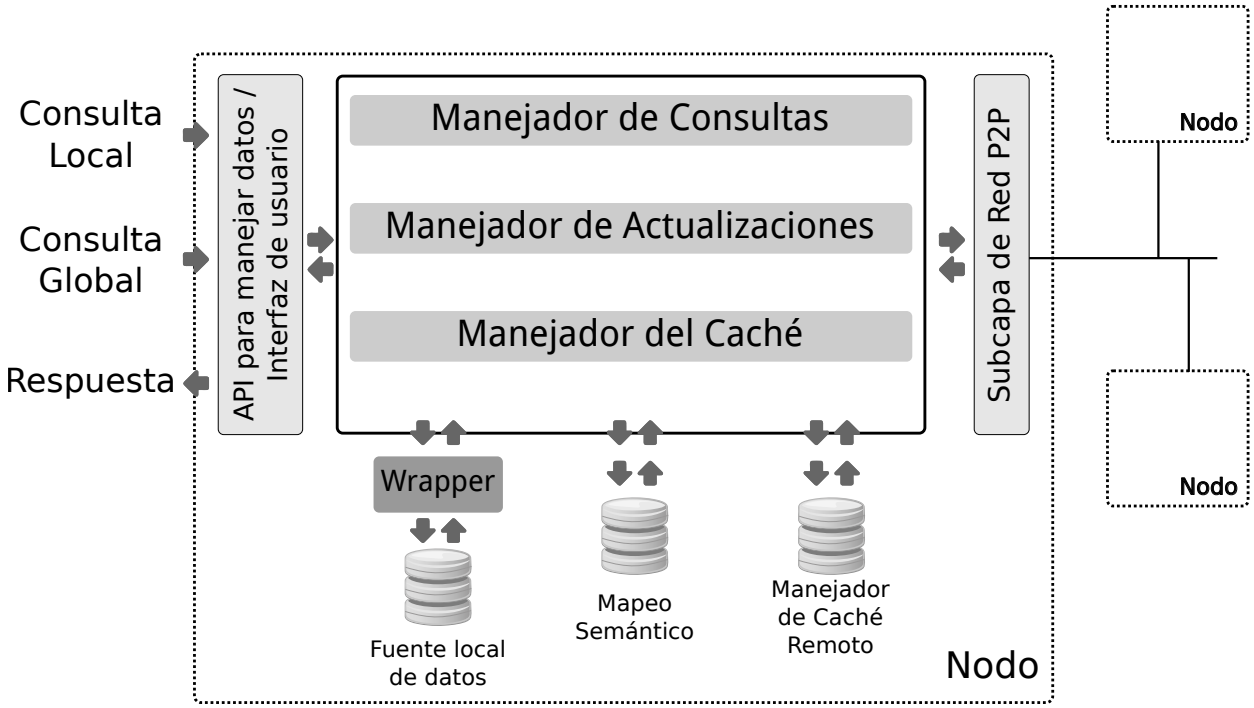


Figura 3.4: Arquitectura de referencia tomada del trabajo *A survey of data management in peer-to-peer systems* [Sun05]

Partiendo de las consideraciones presentadas observamos la conveniencia de adoptar la arquitectura de referencia propuesta en [Sun05]. En dicho trabajo, los autores proponen que un sistema para manejar datos basado en P2P, necesita separar las interfaces, tanto de comunicaciones, de usuario como los datos, del núcleo del sistema, por otro lado en dicho núcleo, se necesitan componentes responsables de manejar las consultas recibidas del exterior, y controlar tanto el caché, como las actualizaciones de la información que almacenan, tal como se muestra en la figura 3.4. Los autores también proponen que, dependiendo de la funcionalidad específica de un nodo, algunos de los componentes pueden aparecer o no, o pueden combinarse para ajustarse a las necesidades del sistema.

A pesar de la flexibilidad que nos ofrece el diseño de [Sun05], la arquitectura de referencia tiene deficiencias considerables con respecto a los aspectos de diseño que identificamos, por ejemplo,

no considera un componente que se encargue de monitorear y controlar el comportamiento del sistema, ni considera la posibilidad de introducir redundancia en los datos almacenados. Es por esto que a partir de la arquitectura de referencia, nos dimos a la tarea de modificarla para adaptarla a los requerimientos que identificamos.

Luego de un proceso de análisis la arquitectura evolucionó hacia una segunda etapa, en la cual ya se contemplan componentes específicos para atender aspectos tales como la redundancia, el manejo de fallas, índices, caché y metadatos, así como las interfaces de comunicación hacia la red, el usuario y aplicaciones externas (ver figura 3.5).

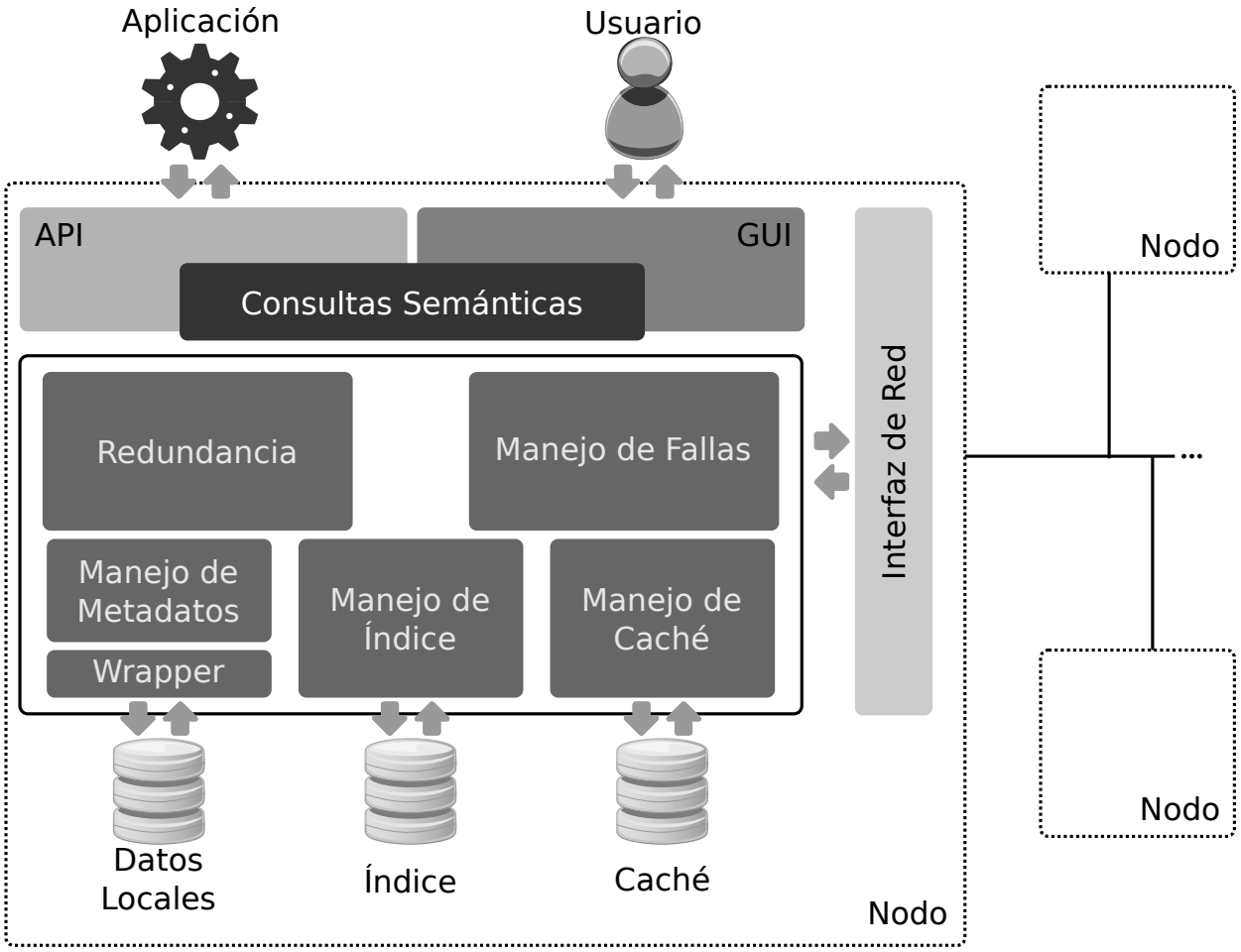


Figura 3.5: Arquitectura preliminar

Aún así, la arquitectura preliminar (mostrada en la figura 3.5), tiene algunas deficiencias, por ejemplo, la interfaz de usuario no está totalmente desacoplada del núcleo del sistema. No se considera tampoco, cómo conectar el EVA con mecanismos externos (por ejemplo el ADI), para agregar redundacia, o un mecanismo para conectarlo con un DHT, con el

fin de escalar el sistema hacia una red P2P estructurada. Por todo esto, en un segundo refinamiento que condujo a la propuesta final (ilustrada en la figura 3.6). En ella aparecen nuevos elementos entre los que destacan las interfaces para delegar responsabilidades sobre componentes independientes e incluso externos al sistema, como es el caso del manejo de metadatos, o de índices basados en DHT; además se hace explícito el desacoplamiento entre la interfaz de usuario y el núcleo del sistema, al igual que el acceso a consultas. Con esta última decisión se espera poder atender consultas sensibles a la semántica, o libres de semántica, a través de una interfaz de programación de aplicaciones.

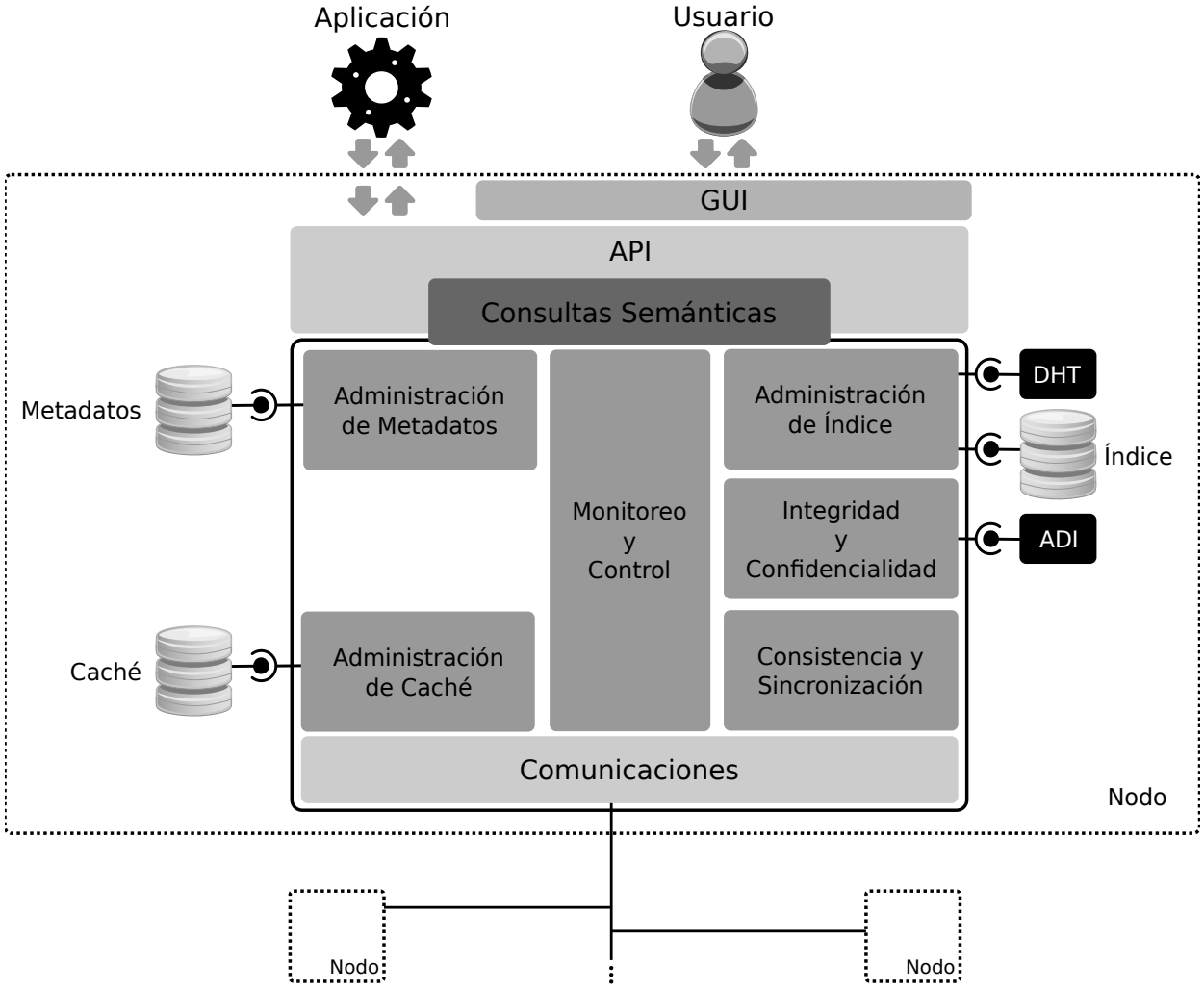


Figura 3.6: Arquitectura propuesta

4 Construcción del prototipo

En este capítulo exponemos la implementación de un prototipo de celda de almacenamiento que cumple con nuestra propuesta. Para ello, la infraestructura disponible constó de cinco computadoras conectadas en red, cuyas características básicas son:

- Procesador Intel Pentium Dual-Core a 2.7 GHz.
- 2 GB the memoria RAM.
- Disco duro con capacidad 500 GB.
- Un switch *Cisco Small Business* SF 1000-08.

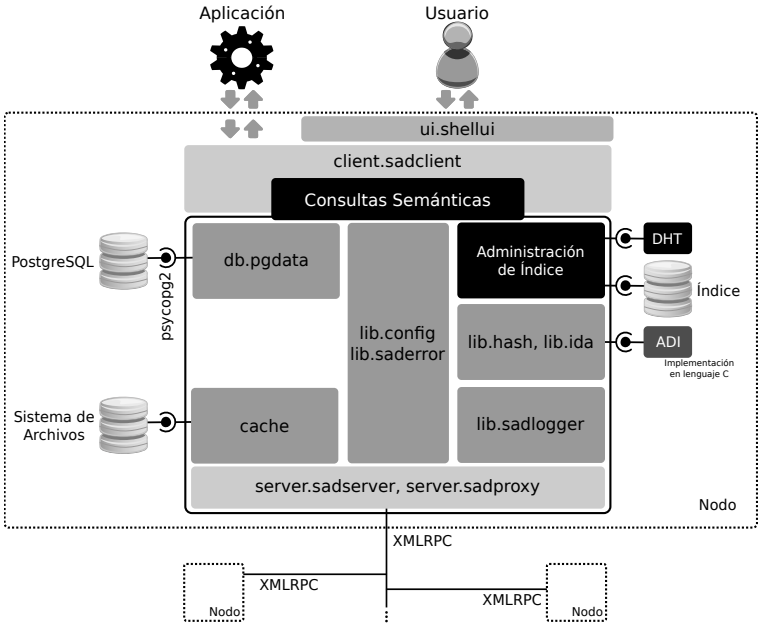


Figura 4.1: Correspondencia entre la implementación del prototipo y la arquitectura propuesta

4.1 Funcionalidad del Sistema

Con el objetivo de probar el modelo arquitectónico propuesto, implementamos una celda de almacenamiento, utilizando la infraestrucura mencionada anteriormente, sobre las computadoras disponibles, instalamos cinco espacios virtuales de almacenamiento, y un representante, el cual es el responsable de interactuar con los usuarios y delegar las tareas de almacenamiento y recuperación de archivos.

La idea de la celda de almacenamiento es obtener un dispositivo virtual, que sea capaz de garantizar la integridad y confidencialidad de los datos almacenados en él, es decir, es un espacio de almacenamiento autónomo y confiable, esto es un aspecto de diseño considerado por [Wei07]. Para efectos de este prototipo, la celda que implementamos tiene la capacidad de otorgar, en un principio, dos diferentes "niveles de servicio" a sus usuarios: almacenar sus datos replicados, o codificados utilizando el algoritmo de dispersión de información [Rab89], ambas opciones ayudan a mejorar la integridad y disponibilidad de la información. La replicación simple es una estrategia más económica en términos de procesamiento, pero es más costosa en cuanto al consumo de espacio de almacenamiento se refiere, por otro lado, la codificación por medio del algoritmo de dispersión de información, aunque más costosa en procesamiento, ofrece las ventajas de ser más económica en costo de almacenamiento y abre la posibilidad de soportar la falla de hasta dos espacios virtuales de almacenamiento, según la implentación actual del algoritmo.

En la figura 4.1 ilustramos la correspondencia entre la arquitectura que proponemos y el prototipo que desarrollamos. En este diagrama expresamos cuales de los módulos corresponden con los componentes de la arquitectura propuesta.

4.2 Modelo de Comunicación

Como modelo de comunicación elegimos XMLRPC [Mrr06], porque permite conectar clientes y servidores, sin importar el lenguaje de programación en que estén escritos; de cualquier manera, el diseño de la arquitectura no impone restricciones sobre el modelo de comunicación que pueda elegirse en otras implementaciones (Ver figura 4.2).

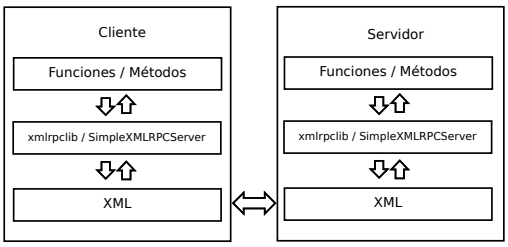


Figura 4.2: Modelo XMLRPC [Mrr06]

El proceso general de comunicación, se ilustra en la figura 4.3. Cuando un **Cliente** desea enviar un mensaje x para realizar una petición i, lo pasa como parámetro a su método

correspondiente f_i, desde donde se invoca un método privado especial, ilustrado en la figura como *enviar*(), que recibe dos parámetros: uno de ellos es una clave k_i correspondiente a la petición realizada y el otro es el mensaje x. Con estos parámetros, el **Cliente** invoca el método remoto ilustrado como *atender*, quien es el encargado de recibir el mensaje x, toma la clave k_i, y con ella, el **Servidor** selecciona el servicio correspondiente del **Manejador** para atender la petición del **Cliente**. Una vez que la operación fue realizada, el valor de retorno del método *atender*() es devuelto al **Cliente**, para reportar el estado en que terminó la invocación remota.

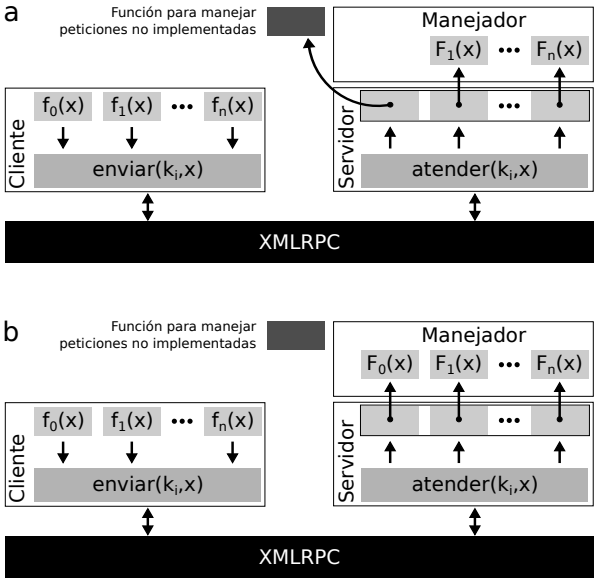


Figura 4.3: Esquema general de comunicación entre nodos

Por otro lado, el diseño que se muestra en la figura 4.3a permite que si la funcionalidad para algún tipo de petición no está implementada, se puede dirigir el procesamiento a un método "por defecto", de esta manera, si se quiere agregar una nueva funcionalidad en el **Servidor**, basta con implementar un método en el **Manejador**, y conectarlo con la "casilla" correspondiente del **Servidor**, tal como se muestra en la figura 4.3b.

La estructura básica de los mensajes consiste en envíar tuplas de la forma $(codigo, datos)$, donde *codigo* es un número entero que sirve para identificar el tipo de mensaje, en caso de una petición, indica de qué petición se trata, y en caso de las respuestas del servidor, indica si la operación terminó exitosamente o no; por otro lado, la entrada *datos* consta de una estructura de datos que contiene el mensaje en sí, que puede ser, por ejemplo los datos necesarios para reconstruir un objeto.

Con estos conceptos, definimos clases "abstractas" tanto para el cliente, el servidor y el manejador de peticiones que se conecta él. En el caso de los clientes, la clase base es RawClient , a partir de ella especializamos dos tipos de cliente, el primero, la clase Client, se encarga de implementar el API que conecta el núcleo del sistema con su usuario final, ya sea una persona u otra aplicación; el segundo, la clase NodeClient, se encarga de habilitar a los EVA para intercambiar mensajes con sus vecinos y el nodo representante, esta relación entre las clases cliente, se ilustra en la figura 4.4.

En el caso de los servidores, se define una clase base llamada Node (definida en el módulo server.sadnode) de la cual se especializan dos clases: Proxy y Server. Las instancias de estas

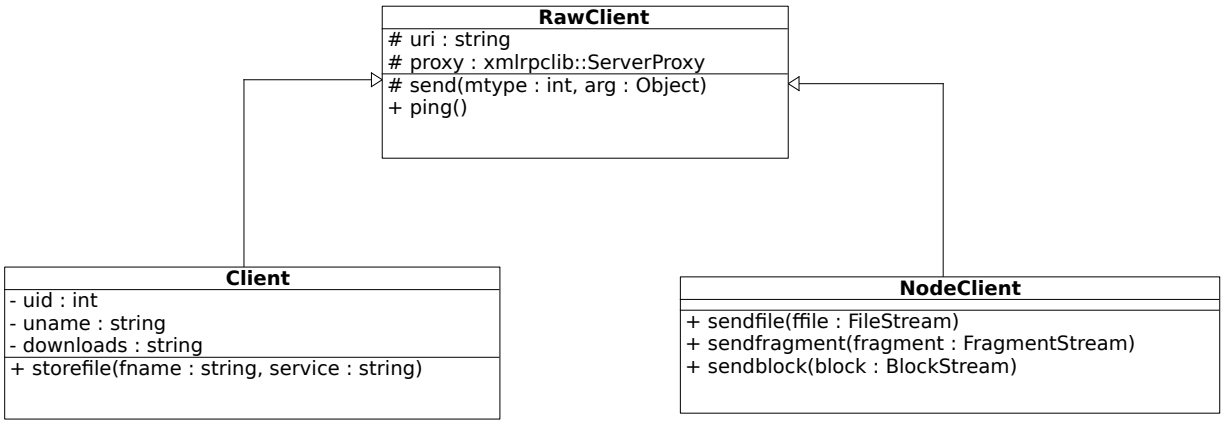


Figura 4.4: Definición de las clases cliente

clases se encargan de las labores del representante y los EVA respectivamente.

Tanto Proxy como Server, utilizan un manejador de peticiones para atender a sus clientes, estos son instancias de las clases ProxyHandler y ServerHandler, ambos, subclases de Handler, clase que también está definida en el módulo server.sadnode.

Finalmente, dotamos a los servidores de la capacidad de atender varias peticiones al mismo tiempo, agregando un "listener" a la clase Node, llamado NodeListener (también definido en el módulo server.sadnode), que es una clase vacía que simplemente hereda de las clases ThreadingMixIn (definida en el módulo SocketServer) y SimpleXMLRPCServer (definida en el módulo SimpleXMLRPCServer), todas estas relaciones entre las clases mencionadas, se ilustran en la figura 4.5.

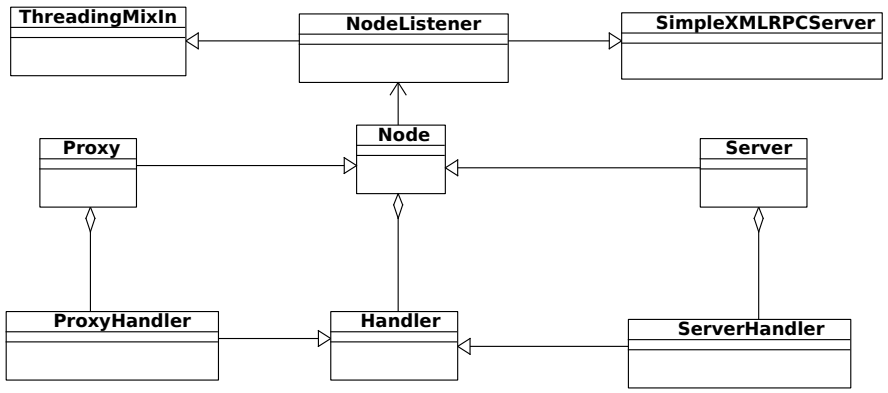


Figura 4.5: Definición de las clases que componen los Espacios Virtuales de Almacenamiento (EVA)

4.3 Modelo de Datos

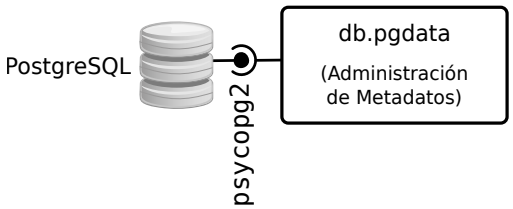


Figura 4.6: Conexión con la Base de Datos

En la arquitectura propuesta, el almacenamiento de los metadatos es delegado a un sistema manejador de base de datos, en este prototipo, el manejador elegido es PostgreSQL, el cual se conecta con la aplicación a través del módulo psycopg2, tal como se ilustra en la figura 4.6, con el módulo pgdata, del paquete db, en éste módulo se implementan las clases necesarias para interactuar con un sistema manejador de base de datos dado. Si se quiere extender la interoperabilidad del prototipo para que pueda utilizar un manejador diferente, bastará con implementar una clase especial para cada uno, respetando la interfaz definida por la clase PGDataBase definida en el módulo db.pgdata; la idea es que toda interacción del sistema con la base de datos sea a través de una clase de este tipo.

Por otro lado, se requieren tipos de datos que representen los objetos que se van a procesar dentro de la celda, a saber, usuarios, archivos, fragmentos, y bloques (tanto réplicas como fragmentos codificados), qué además sean apropiados para enviarse empleando el modelo de comunicación descrito en la sección 4.2. Para lograr esto, es necesario transformar los objetos en una estructura que sea adecuada para formar parte de un mensaje, lo cual se logra definiendo una clase base para todos estos tipos de datos, de la cual se derivan todos los demás, tal como se ilustra en la figura 4.7

4.4 Metadatos

Los metadatos son uno de los aspectos más importantes para el diseño del prototipo; estos son utilizados para registrar los eventos que ocurren en el sistema durante los procesos de almacenamiento y recupreración de información. En los metadatos se registran los archivos que entran a la celda, qué EVA está encargado de procesarlo para dividirlo en fragmentos, qué EVA se encargó de crear los bloques de cada fragmento y en qué EVA se guardan los bloques producidos. En resumen, los metadatos hacen posible que el sistema funcione y pueda recuperarse de las fallas que sufra.

En el prototipo, los metadatos son almacenados en una base de datos relacional, la cual tiene la misma estructura para cada EVA, sin embargo se encuentra desacoplada de su implementación, como se ilustra en la figura 4.6. El diseño de los metadatos se ilustra en la figura 4.8, y lo explicamos a continuación:

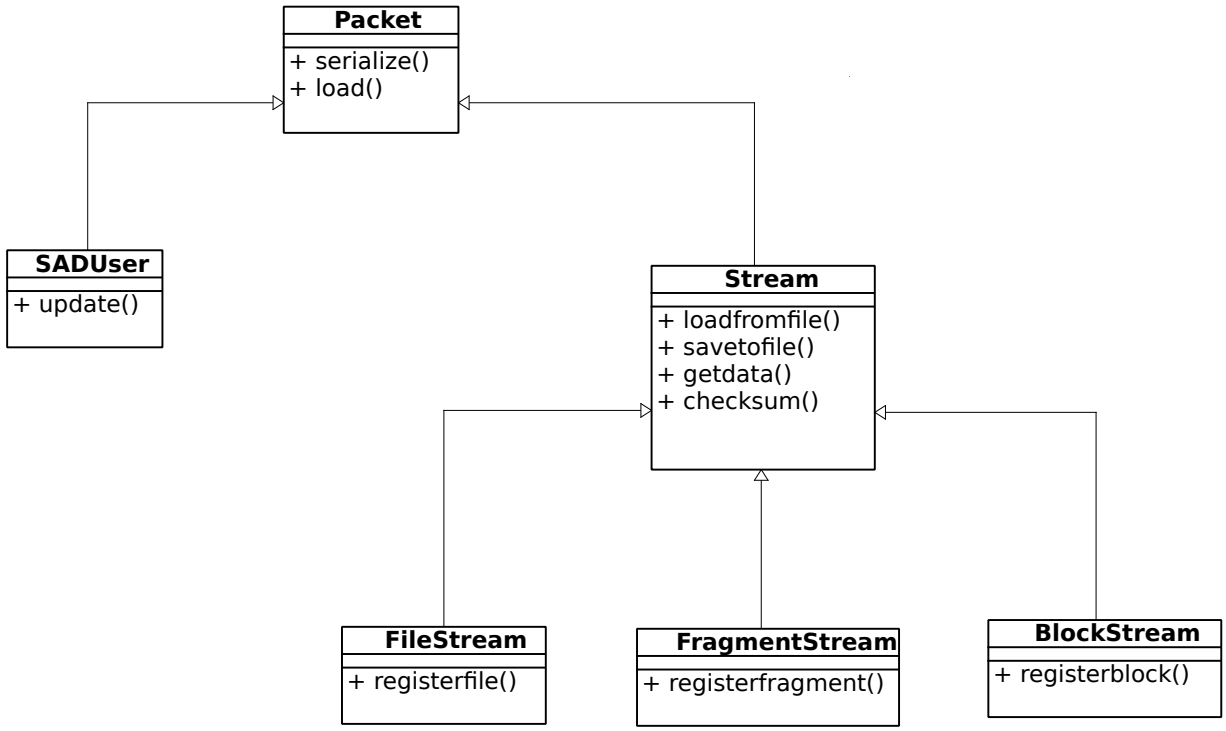


Figura 4.7: Modelo de Datos

users Es un registro de los usuarios del sistema, utilizado por el representante de la celda para autenticar y autorizar el acceso de un usuario al sistema.

groups Es un catálogo de los grupos de usuario existentes.

membership Es una interrelación de cardinalidad "muchos a muchos" que vincula las tablas users y groups, es decir que un usuario puede compartir sus contenidos con otros, creando grupos y asociando otros usuarios y sus archivos a los grupos que desee.

services Es un catálogo de los servicios o niveles de servicio disponibles en la celda de almacenamiento, es decir, que además de poder almacenar archivos utilizando replicación simple y la codificación mediante ADI, se pueden agregar otros servicios que proporcionen una mayor variedad en los niveles de servicio que puede ofrecer la celda.

usr2srv Es una interrelación de cardinalidad "muchos a muchos" que vincula las tablas users y services, la cual permite brindar distintos niveles de servicio a los usuarios.

files Es un registro empleado para tener un control sobre los archivos que se han almacenado en la celda, hacia qué EVA fueron delegados, a qué usuario pertenecen, y en qué versión se encuentran.

fragments Cuando un EVA procesa un archivo y lo divide en fragmentos, utiliza esta tabla

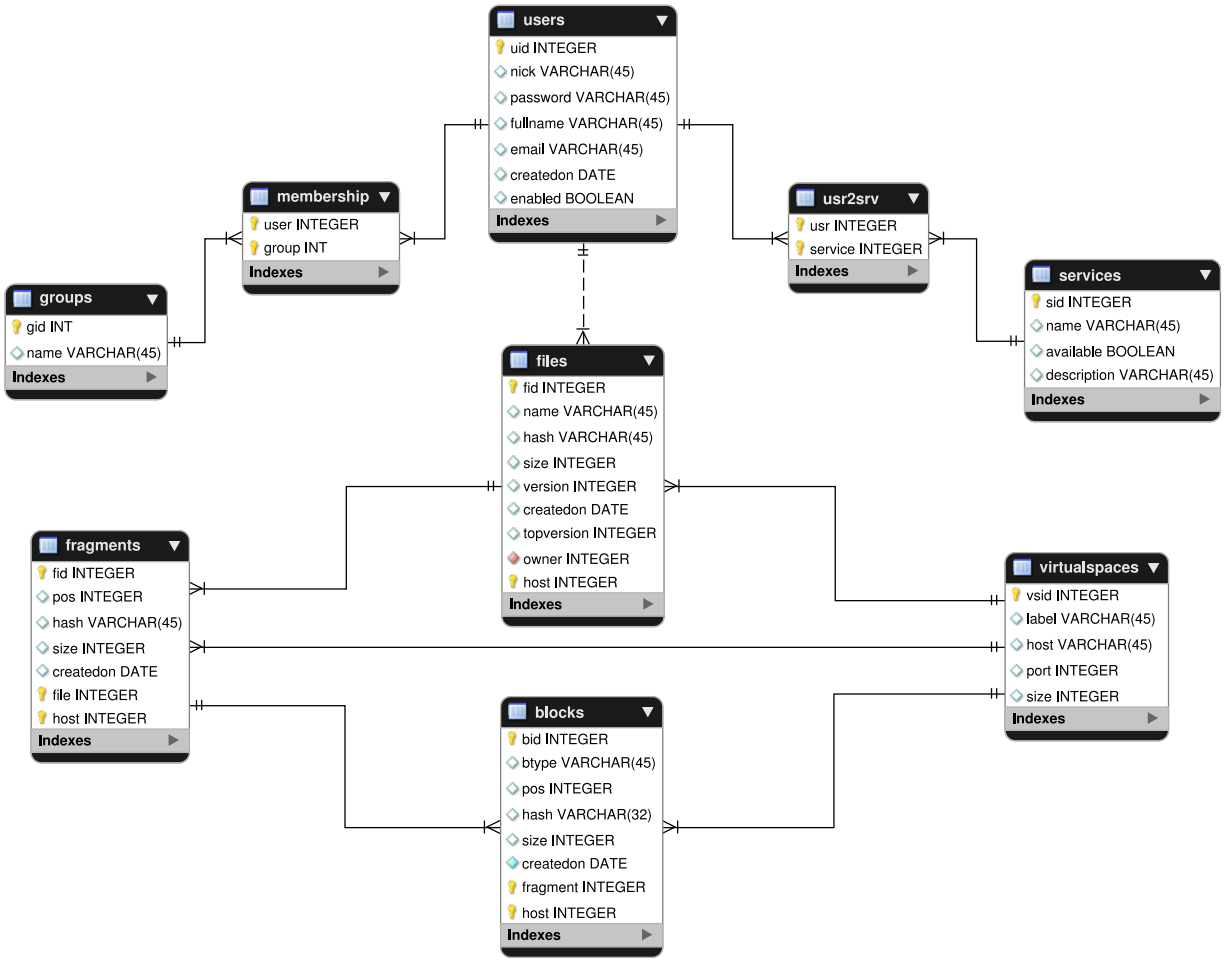


Figura 4.8: Estructura de los metadatos utilizados por el sistema.

para llevar un registro de en cuántos fragmentos fue dividido, a qué archivo pertenece cada fragmento, y a cuál de sus EVA vecinos delegó el procesamiento de cada uno de los fragmentos generados.

blocks De manera similar, la tabla blocks es utilizada para llevar el registro de todos los boques generados, a qué fragmento pertenecen y de qué tipo de fragmento se trata (dependiendo del nivel de servicio con que se almacenó el archivo).

virtualspaces Es un catálogo de los EVA disponibles dentro del sistema, el cual provee una posibilidad de almacenamiento al no limitar el número de EVA que pueden componer la celda de almacenamiento.

4.5 Pruebas

Dado que el prototipo aún se encuentra en una etapa temprana de su desarrollo, las pruebas que se le realizaron se enfocaron a verificar su funcionamiento del mismo. Durante el proceso de desarrollo el prototipo fue puesto en funcionamiento sobre tres diferentes ambientes de pruebas; esto con la finalidad de verificar el comportamiento que tienen los espacios virtuales de almacenamiento cuando comparten el mismo nodo interno, o cuando interactuan con EVAs que residen en diferentes sistemas operativos.

4.5.1 Ambiente homogéneo

En este ambiente se utilizaron cinco computadoras con las mismas caractarísticas de hardware y el mismo software instalado. Se emplearon cinco computadoras de escritorio para implementar la celda de almacenamiento con el siguiente software:

- Sistema Operativo Ubuntu Linux versión 10.10
- Intérprete de Python versión 2.6
- Manejador de base de datos PostgreSQL versión 8.4

Además el cliente de la celda de almacenamiento fue instalado en una computadora portátil con sistema operativo Ubuntu Linux versión 10.04, intérprete de Python versión 2.6 y manejador de base de datos PostgreSQL versión 8.4.

Utilizando estos recursos, cada uno de los cinco EVA que componen la celda se alojó en cada una de las computadoras de escritorio, sin embargo una de las computadoras hospedó también al proceso representante de la celda.

4.5.2 Ambiente heterogéneo

El segundo ambiente de pruebas, un poco más variado, constaba de seis computadoras de escritorio, el siguiente software disponible:

- Tres computadoras con sistema operativo Linux Fedora versión 15
- Dos computadoras con sistema operativo Slackware Linux versión 13.1
- Una computadora con sistema operativo Ubuntu Linux versión 10.04
- Todas las computadoras contaban con la versión 2.7 del intérprete de Python
- Las computadoras con sistema operativo Linux Fedora tienen instalado el manejador de base de datos PostgreSQL versión 9
- Tanto las dos máquinas con sistema operativo Slackware Linux como la computadora con sistema operativo Ubuntu Linux, tienen instalado el manejador de base de datos PostgreSQL versión 8.4

4.5.3 Ambiente reducido

Este ambiente de pruebas consta de una sola computadora que hospeda a todos los EVA, al representante de la celda y la aplicación cliente. Ciertamente este escenario no es muy útil para poner en operación el sistema, pero fue el más utilizado en la etapa de desarrollo del proyecto. Para este ambiente se empleará una computadora con sistema operativo OpenSuSE versión 11.4, intérprete de Python versión 2.7 y manejador de base de datos PostgreSQL versión 8.4.

5 Conclusiones y perspectivas

Durante este trabajo identificamos los requerimientos que un sistema de almacenamiento distribuido debe satisfacer, poniendo especial énfasis en la (re)utilización de la infraestructura de comunicaciones y cómputo de bajo costo.

Atendiendo a la necesidad que tienen las organizaciones para almacenar y compartir la información que generan, nos dimos a la tarea de proponer una arquitectura que puede entenderse como un conjunto de bloques de construcción. Una vez articulados, estos bloques forman un dispositivo virtual denominado **celda de almacenamiento**.

El diseño de la celda reconoce los siguientes requerimientos funcionales:

1. El monitoreo y control de las operaciones.
2. La gestión de metadatos.
3. La consistencia y sincronización de la información.
4. La integridad y la confidencialidad de los datos.
5. El soporte del indexado y búsqueda de la información.

Requerimiento	**Propuesta**
Monitoreo y Control	Todos los eventos ocurridos en el sistema se registran en bitácoras. (*logs*).
Metadatos	Los metadatos se almacenan en una base de datos relacional (sección 4.4).
Consistencia y Sincronización	Los objetos almacenados en la celda son inmutables, cada vez que un usuario pretende almacenar un archivo que ha cambiado, se crea un nuevo archivo en la celda que corresponde a una nueva versión del archivo original.
Integridad	Algoritmo de Dispersión de Información, Replicación Simple y Algoritmo de *Checksum*.
Confidencialidad	Manejo de usuarios, grupos y privilegios de acceso sobre los objetos almacenados.
Indexación y Búsqueda	Pendiente, dado que sólo tenemos una celda de almacenamiento.
Interfaz de Aplicación	Se emplea un **API** através de la cual es posible conectar una interfaz de usuario, en este caso, basada en línea de comandos, para interactuar con la celda de almacenamiento.

Tabla 5.1: Requerimientos funcionales cubiertos por el prototipo

6. El soporte de la interfáz de usuario.

La solución que se propone en el prototipo para cada uno de estos requerimientos se ilustra en la tabla 5.1, asimismo, el diseño se realizó bajo los siguientes requerimientos no funcionales:

1. Flexibilidad.
2. Interoperabilidad.
3. Escalabilidad.
4. Confiabilidad.

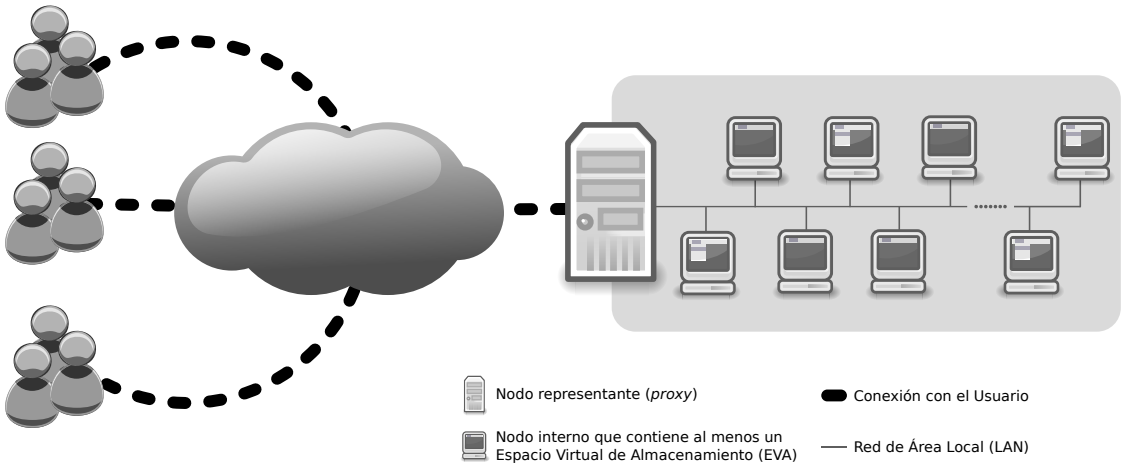


Figura 5.1: Extendiendo la celda agregando EVA's.

Reconocemos que los requerimientos que orientaron nuestras decisiones tienen impacto sobre un conjunto de parámetros de desempeño a los que dimos preferencia; estos son: La cantidad de información redundante, el balance de carga en los dispositivos de almacenamiento y la disponibilidad de la información.

También es cierto que en los sistemas complejos, como es el caso, la arquitectura juega un papel crítico en la definición sus atributos de calidad, en este caso: la flexibilidad, tolerancia a fallas, escalabilidad e incluso, la rapidez de propagación de la información [Bar03, Wat99]. Por lo tanto, para atender las necesidades futuras de crecimiento, la arquitectura que diseñamos le ofrece a la celda cuatro posibilidades:

- Por un lado, se pueden incorporar al interior de la celda más dispositivos de almacenamiento (de igual forma, que contengan uno o mas EVA) gestionados por el mismo representante como se muestra en la figura 5.1.
- Gracias a su interfaz, la celda permite agregar "nodos volátiles" que, en coordinación con el representante de la celda, pueden ofrecer sus capacidades de almacenamiento en

determinados momentos, con esto se puede brindar a los usuarios un nivel de servicio intermedio, en el que la disponibilidad de los archivos que almacenan en la celda está sujeta al tiempo que los nodos volátiles están conectados al sistema. Esta mejora es transparente para el usuario, debido a que siempre utilizará la cela a través de su representante (ver figura 5.2).

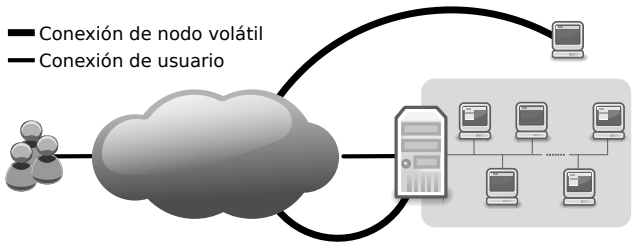


Figura 5.2: Agregando nodos volátiles a la celda de almacenamiento

- Considerando que el representante de la celda es un potencial cuello de botella, es posible mitigar ese riesgo utilizando un *arreglo* de nodos representantes, cuya carga sea balanceada por un componente *broker*[Bass03] que se asegure que la carga de trabajo en cada uno de los nodos representantes sea razonable, tal como se ilustra en la figura 5.3.

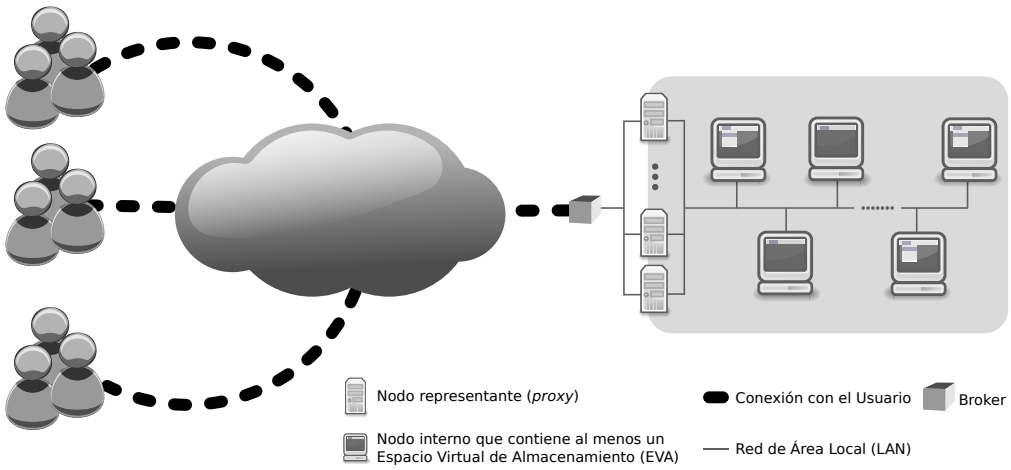


Figura 5.3: Arreglo de nodos representantes para evitar cuellos de botella.

- Dado que la celda puede entenderse como un dispositivo virtual cuyo representante cuenta con una interfaz única se ofrece la posibilidad de que participe en una red P2P estructurada (ver figura 5.4) con mecanismos de tipo DHT para búsqueda e indexado (figura 5.5).

Este enfoque dual, de red P2P no estructurada en un contexto local, y de red P2P estructurada en un contexto global, abre la posibilidad de ofrecer varios tipos de servicios de

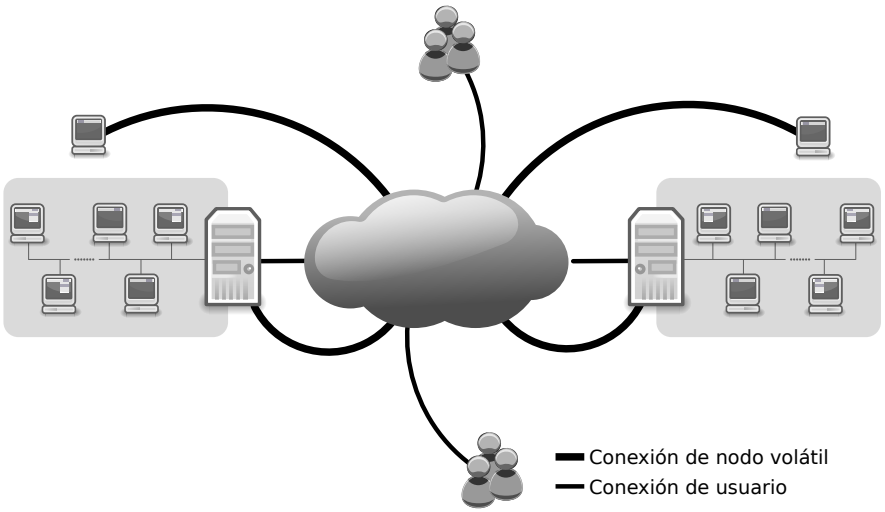


Figura 5.4: Comunicando celdas de almacenamiento

almacenamiento que puede clasificarse en dos grandes categorías: locales y remotos. Evidentemente, cada servicio trae implicados diferentes niveles de redundancia.

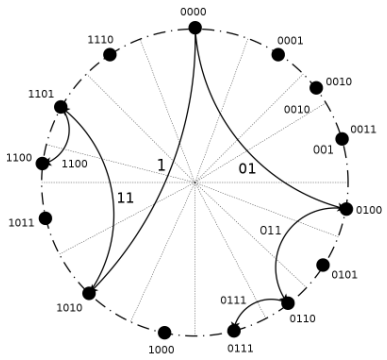


Figura 5.5: Red P2P estructurada (Chord)

En un SAD se requiere encontrar un punto de equilibrio entre el costo de las comunicaciones y la disponibilidad de los datos. Para sistemas pequeños o medianos, el dispositivo de almacenamiento se encuentra en la vecindad de la fuente donde se origina la información y donde se consulta. Sin embargo, para un sistema de mayor escala, el almacenamiento puede resultar muy caro cuando el EVA correspondiente se encuentra lejos del sitio donde se requiere la información. En este sentido el almacenamiento local puede proveer, entre otras funcionalidades, de una "memoria" caché. En tanto, el almacenamiento remoto sirve para proveer un servicio fuera de sitio. El primero garantiza la disponibilidad de la información de corto plazo. En tanto, el segundo soporta una disponibilidad de largo plazo. La combinación de ambos sería el principio de un mecanismo de aseguramiento de la continuidad de los negocios (business continuity plan). Por su parte, cada celda quedaría a cargo de vigilar las condiciones locales del balance de carga.

Parte II

Apéndices

A Casos de uso

En este apéndice se encuentra una descripción de los casos de uso identificados del sistema los cuales se describen desde la perspectiva del representante de la celda. A grandes rasgos, la funcionalidad del prototipo se ilustra en el diagrama de la figura A.1.

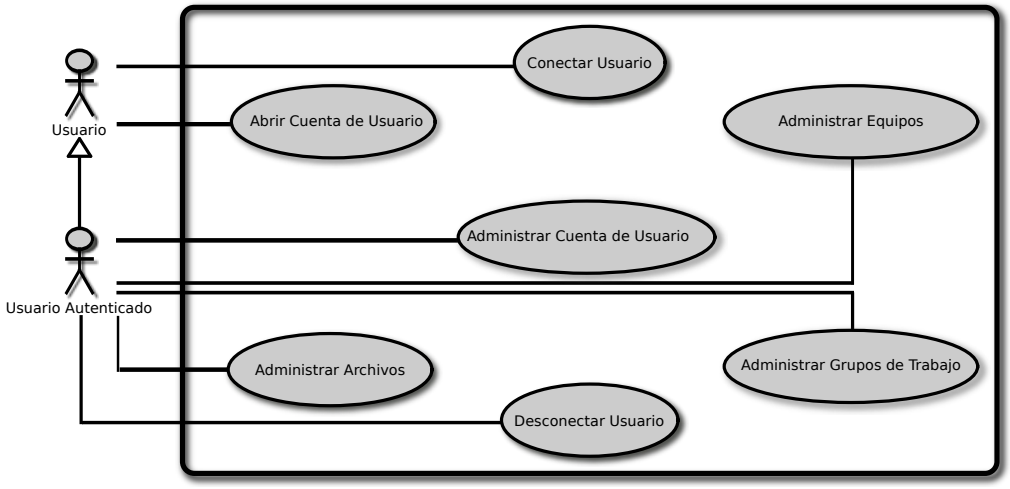


Figura A.1: Vista panorámica de la funcionalidad de la celda.

A.1 Abrir cuenta de usuario

Descripción Dado que el servicio de almacenamiento requiere garantizar la confidencialidad de los datos, es necesario conocer la identidad de cada uno.

Precondición El usuario no está registrado.

Poscondición El usuario obtiene una cuenta para hacer uso del servicio de almacenamiento.

Disparador Un usuario no autenticado solicita registrarse en el sistema.

Excepciones

- La cuenta de usuario solicitada ya fue asignada anteriormente.
- Ocurrió un error en la conexión con la celda durante el proceso de registro.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.1

	Acción
1	El usuario solicita una cuenta.
2	El sistema le solicita al usuario sus datos personales.
3	El usuario proporciona los datos solicitados.
4	El sistema registra los datos proporcionados.
5	El sistema crea la cuenta del usuario.
6	El sistema notifica al usuario que la cuenta fue creada.

Tabla A.1: Secuencia normal del caso de uso “Abrir cuenta de usuario”

A.2 Administrar cuenta de usuario

Descripción Actualiza los datos de la cuenta de usuario.

Precondición La cuenta que quiere actualizarse existe dentro del sistema.

Poscondición Los datos del usuario están actualizados.

Disparador Un usuario autenticado solicita al sistema modificar sus datos.

Excepciones

- Los datos proporcionados por el usuario no son correctos.
- Ocurrió un error al momento de almacenar los datos nuevos del usuario
- Ocurrió un error en la conexión con la celda durante el proceso de actualización.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.2

	Acción
1	El usuario solicita que se modifiquen los datos de su cuenta.
2	El sistema le solicita al usuario sus datos personales actualizados.
3	El usuario proporciona los datos solicitados.
4	El sistema actualiza los datos del usuario.
5	El sistema le notifica al usuario que sus datos fueron actualizados.

Tabla A.2: Secuencia normal del caso de uso "Administrar cuenta de usuario"

A.3 Eliminar cuenta de usuario

Descripción Permite al administrador de la celda eliminar cuentas de usuario.

Precondición El usuario ingresó al sistema con el rol de administrador.

Poscondición La cuenta de usuario seleccionada fue eliminada

Disparador El administrador del sistema solicita la eliminación de una cuenta de usuario.

Excepciones

- La cuenta de usuario que quiere ser eliminada no existe dentro del sistema.
- Ocurrió un error al momento de eliminar los archivos asociados a la cuenta.
- Ocurrió un error al momento de eliminar los metadatos asociados a la cuenta.
- Ocurrió un error en la conexión con la celda durante el proceso de eliminación de la cuenta.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.3

	Acción
1	El administrador le indica al sistema que quiere eliminar una cuenta de usuario.
2	El sistema pide que el administrador confirme que desea eliminar la cuenta.
3	El administrador confirma la petición para eliminar la cuenta.
4	El sistema elimina todos los archivos que pertenecen a la cuenta seleccionada.
5	El sistema elimina todos los metadatos relacionados con la cuenta seleccionada.
6	El sistema notifica al administrador que la cuenta fue eliminada.

Tabla A.3: Secuencia normal del caso de uso "Eliminar cuenta de usuario"

A.4 Un usuario se contecta a la celda

Descripción Le permite a un usuario conectarse a la celda para hacer uso del servicio de almacenamiento.

Precondición El usuario aún no se ha conectado al sistema.

Poscondición El usuario está conectado al sistema.

Disparador Un usuario no autenticado solicita iniciar sesión.

Excepciones

- La cuenta de usuario no existe.
- La cuenta de usuario existe, pero la contraseña porporcionada no es la correcta.
- Ocurrió un error en la conexión con la celda durante el proceso de autenticación.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.4

	Acción
1	El usuario solicita conectarse a la celda proporcionando un nombre de usuario.
2	El sistema verifica que exista una cuenta con el nombre del usuario que quiere conectarse.
3	El sistema le solicita al usuario que porporcione su contraseña.
4	El usuario porporciona su contraseña.
5	El sistema verifica que la contraseña corresponda al usuario que quiere conectarse.
6	El sistema le permite el acceso al usuario.

Tabla A.4: Secuencia normal del caso de uso "Un usuario se conecta a la celda"

A.5 Un usuario se desconecta de la celda

Descripción Le permite a un usuario autenticado cerrar su sesión

Precondición El usuario está conectado al sistema.

Poscondición La sesión del usuario ha terminado.

Disparador Un usuario autenticado solicita terminar su sesión.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.5

	Acción
1	El usuario le indica al sistema que desea desconectarse de la celda.
2	El sistema notifica al usuario que su sesión ha finalizado.
3	El sistema cierra la conexión del usuario.

Tabla A.5: Secuencia normal del caso de uso "Un usuario se desconecta de la celda"

A.6 Crear grupo de trabajo

Descripción Le permite al usuario crear un grupo de trabajo para compartir sus archivos con los miembros de éste.

Precondición El usuario está conectado a la celda.

Poscondición El grupo indicado ha sido creado.

Disparador Un usuario autenticado solicita la creación de un grupo de trabajo.

Excepciones

- El grupo que se desea crear ya existe.
- Ocurrió un error al momento de registrar el nuevo grupo.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.6

	Acción
1	El usuario le indica al sistema que quiere crear un nuevo grupo.
2	El sistema le solicita al usuario el nombre del nuevo grupo.
3	El usuario le proporciona al sistema el nombre del nuevo grupo.
4	El sistema crea el grupo de trabajo con el nombre especificado por el usuario.

Tabla A.6: Secuencia normal del caso de uso "Crear grupo de trabajo"

A.7 Invitar usuarios a grupos de trabajo

Descripción Le permite a un usuario compartir archivos con otros usuarios a través de grupos de trabajo.

Precondición El usuario está conectado a la celda y ha creado un grupo de trabajo.

Poscondición Los usuarios seleccionados han sido invitados a unirse a un grupo de trabajo.

Disparador Un usuario autenticado crea un grupo de trabajo.

Excepciones

- El grupo de trabajo al que se quiere invitar a los usuarios, no existe.
- Alguno de los usuarios que fueron invitados no existe dentro del sistema.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.7

	Acción
1	El usuario le indica al sistema que quiere invitar usuarios a un determinado grupo de trabajo.
2	El sistema verifica que el grupo de trabajo seleccionado exista.
3	El sistema le solicita al usuario la lista de invitados a formar parte del grupo de trabajo.
4	El usuario proporciona la lista de los invitados a formar parte del grupo de trabajo.
5	El sistema envía las invitaciones a los usuarios que forman parte de la lista de invitados.
6	El sistema le indica al usuario que sus invitaciones fueron enviadas.

Tabla A.7: Secuencia normal del caso de uso “Invitar usuarios a grupos de trabajo”

A.8 Aceptar invitación a grupo de trabajo

Descripción Le permite a un usuario aceptar una invitación para unirse a un grupo de trabajo.

Precondición El usuario fue inivtado a unirse a un grupo de trabajo.

Poscondición El usuario se unió al grupo de trabajo al que fue invitado.

Disparador Un usuario inicia su sesión.

Excepciones

- El usuario rechazó la invitación para unirse al grupo de trabajo.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.8

	Acción
1	El sistema le notifica al usuario que fue invitado para unirse a un grupo de trabajo.
2	El usuario acepta la invitación para unirse al grupo de trabajo.
3	El sistema agrega al usuario al grupo de trabajo.
4	El sistema notifica a quien envió la invitación, que el usuario se ha unido al grupo de trabajo.

Tabla A.8: Secuencia normal del caso de uso “Aceptar invitación a grupo de trabajo”

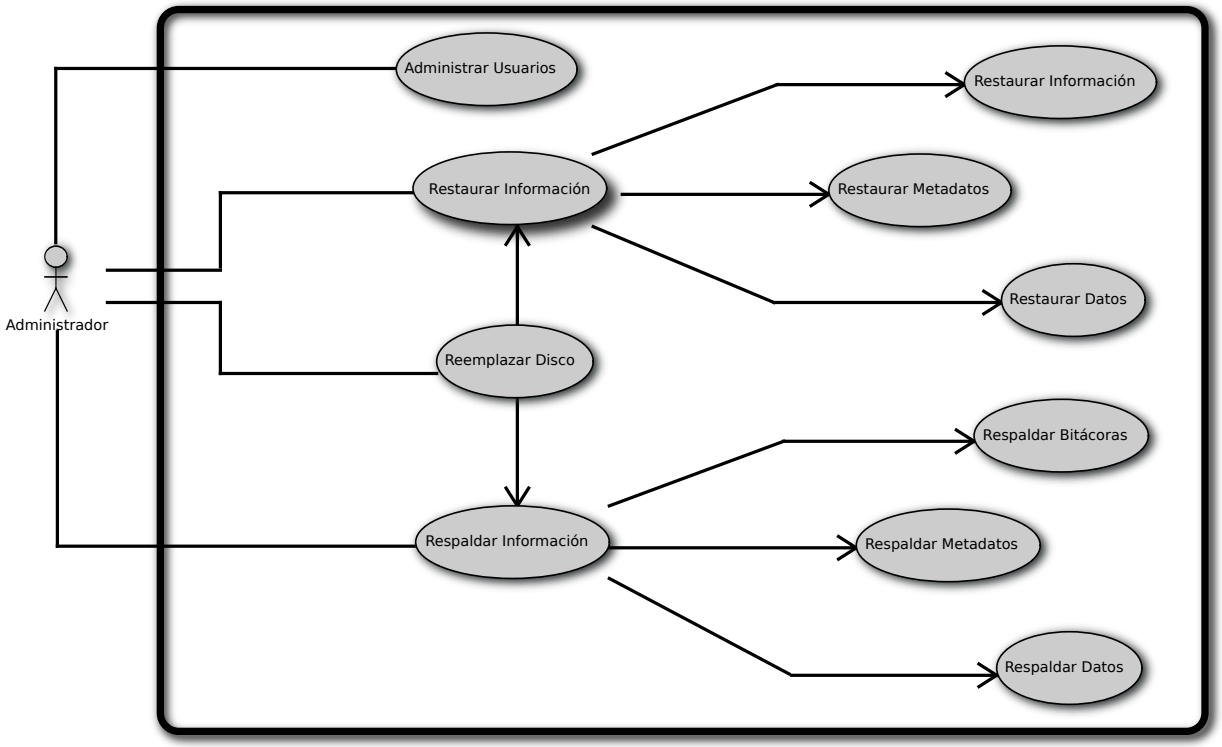


Figura A.2: Administración de la celda de almacenamiento.

A.9 Quitar un usuario de un grupo de trabajo

Descripción Le permite a un usuario desligar a otro de un grupo de trabajo.

Precondición El usuario es dueño de un grupo de trabajo que contiene al menos a otro integrante.

Poscondición El integrante seleccionado ha sido retirado del grupo de trabajo.

Disparador El usuario solicita eliminar un integrante de un grupo de trabajo.

Excepciones

- El integrante seleccionado no pertenece al grupo de trabajo.
- El grupo de trabajo no pertenece al usuario.
- El grupo de trabajo no existe.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.9

	Acción
1	El usuario solicita retirar un integrante de un grupo de trabajo.
2	El sistema le solicita al usuario que indique el nombre del integrante.
3	El usuario porporciona el nombre del integrante.
4	El sistema verifica que el integrante forme parte del grupo de trabajo.
5	El sistema desliga al integrante del grupo de trabajo.

Tabla A.9: Secuencia normal del caso de uso "Quitar un usuario de un grupo de trabajo"

A.10 Eliminar grupo de trabajo

Descripción Le permite a un usuario eliminar un grupo de trabajo.

Precondición El grupo que quiere ser eliminado existe.

Poscondición El grupo de trabajo fue eliminado.

Disparador El usuario solicita que se elimine un grupo de trabajo.

Excepciones

- El grupo de trabajo que se quiere eliminar no existe.
- El usuario no es dueño del grupo de trabajo que quiere eliminar.
- El grupo de trabajo no está vacío.
- La conexión con la base de datos falla al momento de realizar la operación que elimina el grupo de trabajo.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.10

	Acción
1	El usuario solicita la eliminación de un grupo de trabajo.
2	El sistema verifica que el grupo de trabajo exista.
3	El sistema verifica que el grupo de trabajo pertenezca al usuario.
4	El sistema verifica que el grupo de trabajo esté vacío.
5	El sistema elimina el grupo de trabajo.

Tabla A.10: Secuencia normal del caso de uso "Eliminar grupo de trabajo"

A.11 Almacenar un archivo

Descripción Le permite a un usuario almacenar un archivo en la celda de almacenamiento.

Precondición El usuario está conectado a la celda.

Poscondición El archivo ha sido almacenado en la celda.

Disparador El usuario solicita que se almacene un archivo en la celda.

Excepciones

- El archivo no pudo almacenarse debido a que hubo un error al transmitirse hacia la celda.
- El usuario cancela la operación mientras está en progreso.
- La conexión con la base de datos falla al momento de realizar la operación de almacenamiento.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.11

	Acción
1	El usuario envía un archivo a la celda para que sea almacenado.
2	El sistema recibe el archivo
3	El sistema selecciona un EVA para que realice la operación de almacenamiento.
4	El sistema envía el archivo al EVA seleccionado.
5	El sistema genera y registra los metadatos que corresponden a la operación de almacenamiento en su base de datos
6	El sistema le indica al usuario que el archivo fue almacenado correctamente.

Tabla A.11: Secuencia normal del caso de uso “Almacenar un archivo”

A.12 Listar archivos almacenados

Descripción Le permite al usuario consultar qué archivos tiene almacenados en la celda.

Precondición El usuario se ha conectado a la celda.

Poscondición El usuario recibe la lista de los archivos que tiene en la celda

Disparador El usuario solicita la lista de sus archivos

Excepciones

- La conexión con la base de datos falla al momento de realizar la consulta.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.12

	Acción
1	El usuario solicita al sistema la lista de los archivos que ha almacenado.
2	El sistema consulta en sus metadatos la información solicitada.
3	El sistema envía al usuario la información solicitada.
4	El usuario recibe la lista de archivos que tiene almacenados en la celda.

Tabla A.12: Secuencia normal del caso de uso "Listar archivos almacenados"

A.13 Compartir archivo

Descripción Le permite a un usuario compartir un archivo con los miembros de un grupo de trabajo.

Precondición El archivo seleccionado no está compartido con el grupo de trabajo.

Poscondición Los miembros del grupo de trabajo pueden tener acceso al archivo compartido.

Disparador Un usuario le indica al sistema que quiere compartir un archivos.

Excepciones

- El usuario no pertenece al grupo de trabajo al que quiere compartir el archivo.
- La conexión con la base de datos falla al momento de realizar la operación.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.13

	Acción
1	El usuario le indica al sistema que quiere compartir un archivo, indicando el nombre del archivo y del grupo de trabajo.
2	El sistema asocia el archivo con el grupo de trabajo indicado.
3	El sistema le notifica a los integrantes del grupo que el usuario ha compartido un archivo con ellos.

Tabla A.13: Secuencia normal del caso de uso “Compartir archivo”

A.14 Alta de equipo

Descripción Le permite a un usuario agregar un equipo de cómputo para que le preste sus capacidades de almacenamiento a la celda, desempeñando el rol de nodo volatil, como se muestra en la figura 5.2 del capítulo 5.

Precondición El usuario está conectado a la celda.

Poscondición El equipo está asociado a la celda de almacenamiento.

Disparador El usuario solicita que se asocie un equipo a la celda.

Excepciones

- La conexión con la base de datos falla al momento de realizar la operación.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.14

	Acción
1	El usuario solicita al sistema que registre un equipo.
2	El sistema solicita al usuario que le proporcione los datos del equipo.
3	El sistema registra la información del equipo.

Tabla A.14: Secuencia normal del caso de uso “Alta de equipo”

A.15 Baja de equipo

Descripción Le permite a un usuario retirar uno de sus equipos de la celda de almacenamiento.

Precondición El equipo seleccionado está ligado a la celda.

Poscondición El equipo ya no está ligado a la celda.

Disparador El usuario le indica al sistema que desligue un equipo de la celda.

Excepciones

- El equipo especificado no está asociado a la celda.
- El usuario no es dueño del equipo que trata de dar de baja.
- Ocurrió un error en la conexión con la celda mientras la operación está en progreso.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.15

	Acción
1	El usuario le indica al sistema que desea dar de baja un equipo.
2	El sistema le solicita al usuario que ingrese su contraseña.
3	El usuario le proporciona su contraseña al sistema.
4	El sistema verifica que el equipo pertenezca al usuario.
5	El sistema da de baja el equipo seleccionado.

Tabla A.15: Secuencia normal del caso de uso “Baja de equipo”

A.16 Conectar equipo

Descripción Le permite a un equipo registrado, conectarse a la celda.

Precondición El equipo que quiere conectarse está registrado en la celda.

Poscondición El equipo está conectado y listo para prestar sus recursos a la celda.

Disparador El equipo iniciá el servicio de almacenamiento e intenta conectarse a la celda.

Excepciones

- El equipo no está asociado a la celda.
- Ocurrió un error en la conexión con la celda mientras la operación está en progreso.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.16

	Acción
1	El equipo le envía al sistema una petición para conectarse a la celda.
2	El sistema verifica la identidad del equipo.
3	El sistema verifica que el equipo esté registrado y autorizado para conectarse a la celda.
4	El sistema establece un enlace con el equipo para que la celda pueda utilizar sus recursos.

Tabla A.16: Secuencia normal del caso de uso “Conectar equipo”

A.17 Desconectar equipo

Descripción Le permite a un equipo desconectarse de la celda de almacenamiento.

Precondición El equipo está conectado a la celda.

Poscondición El equipo se ha desconectado de la celda.

Disparador El equipo envía una petición al sistema para desconectarse de la celda.

Excepciones

- Ocurrió un error en la conexión con la celda mientras la operación está en progreso.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.17

	Acción
1	El equipo envía una petición al sistema para desconectarse de la celda.
2	El sistema verifica la identidad del equipo.
3	El sistema envía una notificación al equipo informándole que la conexión ha sido terminada.
4	El sistema cierra la conexión con el equipo.

Tabla A.17: Secuencia normal del caso de uso “Desconectar equipo”

A.18 Respaldar metadatos y bitácoras

Descripción Le permite al sistema respaldar sus metadatos y bitácoras.

Precondición El sistema está en funcionamiento.

Poscondición Los metadatos del sistema.

Disparador El caso de uso se dispara de manera periódica, la frecuencia con que se dispara, es un parámetro configurable.

Excepciones

- Ocurrió un error en la conexión con a la base de datos la operación está en progreso.

El flujo de eventos normal para este caso de uso se muestra en la tabla A.18

	Acción
1	El sistema respalda sus metadatos en un archivo.
2	El sistema respalda sus bitácoras en un archivo comprimido.
3	El sistema almacena los archivos de respaldo en la celda.

Tabla A.18: Secuencia normal del caso de uso “Desconectar equipo”

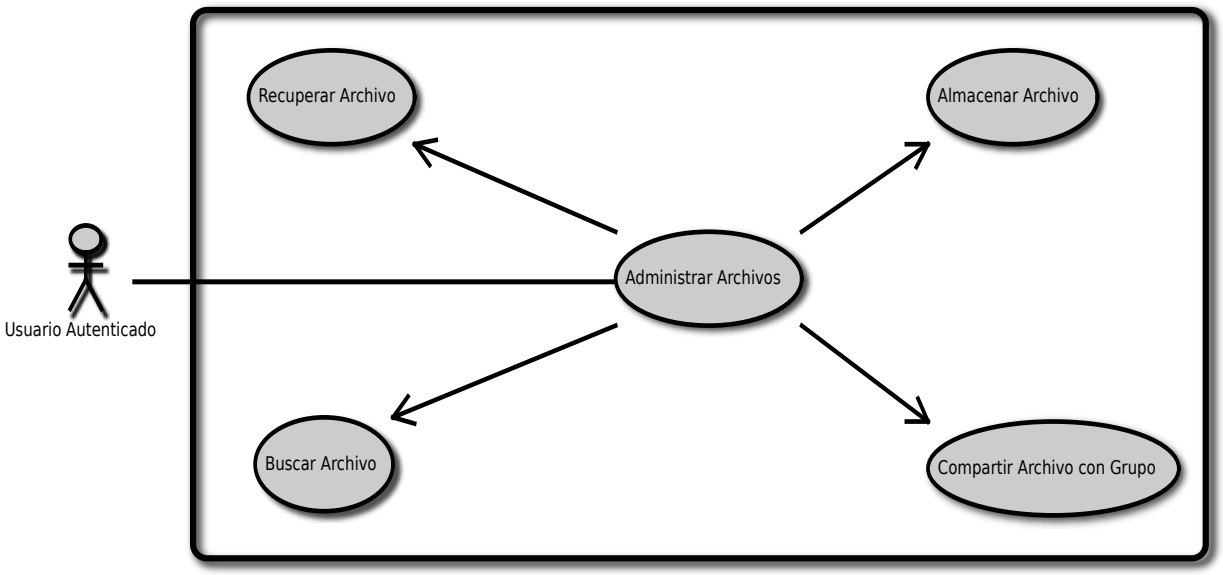


Figura A.3: Manejo de Archivos

B Manual

Como ya se mencionó desde el capítulo 4, con la finalidad de poner a prueba el diseño arquitectónico propuesto, construimos un prototipo de la celda de almacenamiento.

En este apéndice se muestran las ideas e instrucciones necesarias para poner en funcionamiento una celda de almacenamiento distribuida sobre una red de área local.

B.1 Infraestructura necesaria

Dada la implementación disponible del ADI [Rab89], la celda necesita al menos de seis procesos, uno de ellos se desempeñará como el representante de la celda, y los otros cinco, serán los EVA . Como se definió en la sección 3.5.1, un nodo interno, puede albergar uno o mas espacios virtuales de almacenamiento (EVA), cada uno de ellos es un proceso independiente que cuenta con su propio archivo de configuración, su propio conjunto de metadatos y un espacio exclusivo para que almacene los bloques que reciba, tal como se muestra en la figura B.1, por lo que la celda puede crecer en cuanto al número de nodos internos disponibles, y los EVA que hospedan. Hay que mencionar que aunque el nodo representante realizará tareas diferentes, el código que tiene disponible es el mismo que el de los nodos internos, previendo que en un futuro, el diseño de la celda pueda evolucionar para soportar la sustitución del nodo representante a partir de uno de los nodos internos (ver figura B.2).

Por lo anterior, la infraestructura necesaria para lograr los niveles de calidad en el servicio del prototipo consta de:

- Al menos cinco computadoras con las mismas capacidades, para desempeñarse como **nodos internos**.
- Una computadora, equipada con dos interfaces de red, que le permitan comunicarse con los nodos internos y con una red externa, este equipo es el que se desempeñará como el nodo representante.
- Un *switch* para montar la red local en la que se conectarán los nodos que componen la

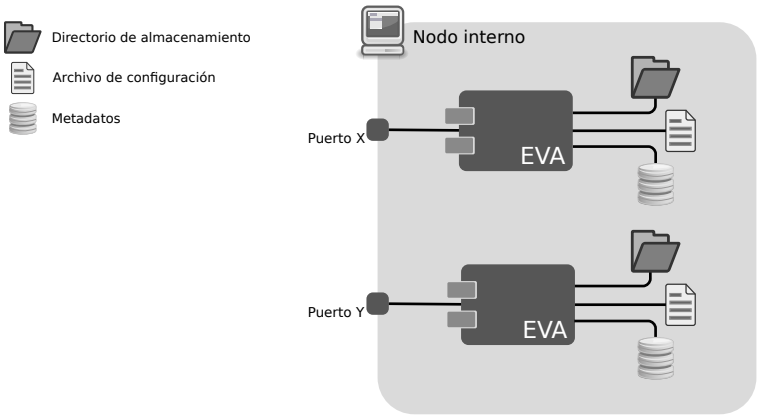


Figura B.1: Varios EVA residentes en un nodo interno

celda. La capacidad de éste también tendrá un impacto significativo en el rendimiento de la celda.

B.2 Dependencias de software

Para funcionar correctamente, el prototipo depende de dos componentes externos. Para ejecutar el código del EVA se necesita instalar en cada uno de los nodos participantes un intérprete de Python y para gestionar los metadatos se requiere PostgreSQL, dado que cada EVA necesita tener su propia base de datos.

El procedimiento para instalar tanto Python como PostgreSQL varía dependiendo del sistema operativo, por lo que detallarlo sale del alcance de este trabajo.

Una vez instalado PostgreSQL, en el sistema se crea un usuario llamado postgres, primero hay que asignarle una contraseña a este usuario como se muestra en el listado B.1.

Listado B.1: Cambiando la contraseña del usuario postgres

```
$ passwd postgres
Changing password for postgres.
New password: # ingresar nueva clave de acceso
Reenter new password: # repetir nueva clave de acceso
$
```

Una vez hecho esto, hay que crear una base de datos para cada uno de los EVA, en este caso, dado que instalamos un manejador de base de datos en cada uno de los nodos, hay que crear tantas bases de datos, como EVA's vayan a residir en cada nodo, por ejemplo, si la base de datos se llamará *celda*, entonces hay que ejecturar los comandos mostrados en el listado B.2.

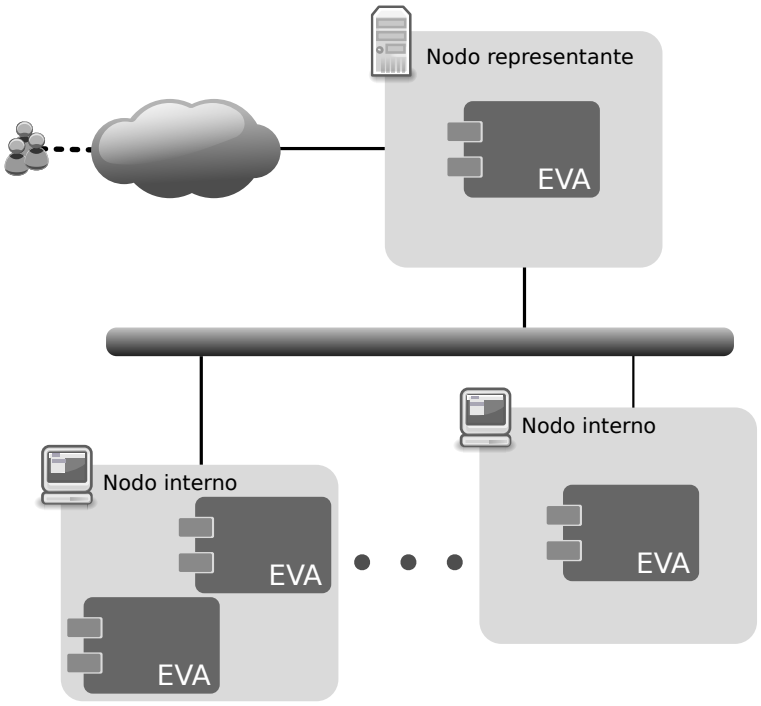


Figura B.2: Posible configuración para la celda de almacenamiento

Listado B.2: Creando una base de datos

```
$ su - posgres
Password: # ingresar clave de acceso
$ createdb celda
```

Ya con la base de datos creada, es recomendable crear un usuario dentro de PostgreSQL que sea exclusivo para la celda de almacenamiento, un nombre apropiado puede ser *admincelda* (ver listado B.3).

Listado B.3: Creando un usuario en PostgreSQL

```
$ createuser -DRSP admincelda
Type password for new rol:
Retype it again:
$
```

Posteriormente hay que configurar PostgreSQL para que solicite su contraseña al usuario al momento de intentar conectarse a la base de datos, esto se logra editando el archivo *data/pg_hba.conf* dentro del directorio inicial del usuario postgres (ver listado B.4).

Listado B.4: Editando el archivo de pg_hba.conf

```
$ su - postgres
```

```
Password: # ingresar clave de acceso
$ vim data/pg_hba.conf
```

En este archivo identificamos ejemplos de cómo editar el archivo para permitir el acceso a los usuarios, especificando las bases de datos a las que pueden conectarse y el método de autenticación que emplearán para ello (listado B.5).

Listado B.5: Ejemplos de configuración de acceso en el archivo pg_hba.conf

```
# local      DATABASE  USER  METHOD  [OPTIONS]
# host       DATABASE  USER  CIDR-ADDRESS  METHOD  [OPTIONS]
# hostssl    DATABASE  USER  CIDR-ADDRESS  METHOD  [OPTIONS]
# hostnossl  DATABASE  USER  CIDR-ADDRESS  METHOD  [OPTIONS]
```

Suponiendo que la base de datos se llama celda y el usuario se llama admincelda, hay que instertar una línea en el archivo como la que se muestra en el listado B.6.

Listado B.6: Permitiendo el acceso al usuario admincelda para la base de datos celda

```
local celda admincelda md5
```

Lo cual le indica a PostgreSQL que el usuario *admincelda* pude conectarse desde el equipo *local* (localhost) a la base de datos *celda* utilizando su contraseña, la cual será protegida por el algoritmo MD5.

Luego de esto es necesario reiniciar el servicio de PostgreSQL, nuevamente, esto es diferente en función del sistema operativo que se esté utilizando.

B.3 Instalación de los nodos

Para tener instalado el prototipo, hay que copiar el código fuente en cada uno de los nodos internos que van a formar parte de la celda. Por cada EVA en un nodo interno, se necesita:

1. Un archivo de configuración.
2. Una base de datos.
3. Un directorio en el que almacenará los bloques que reciba en resguardo.

El archivo de configuración es contiene listas de pares **clave = valor** separadas en secciones, que especifican valores como el puerto por el que el EVA escuchará las peticiones de sus vecinos, el directorio en el que almacenará los bloques etc. (ver figura B.2). En el listado B.7 se muestra un ejemplo del archivo de configuración.

Listado B.7: Plantilla de un archivo de configuración para lun EVA o el nodo representante

```
1
2  [General]
3  address  = 192.168.0.1  # Direccion donde se hospeda el EVA
4  port     = 4242         # Puerto asignado al EVA
5  storage  = /storage/eva # Directorio para almacenar los bloques
6  msu      = 10M          # UMA
7
8  [DataBase]
9
10 dbname   = celda        # Nombre de la base de datos
11 user     = admincelda   # Usuario que se conecta a la base de datos
12 password = algunaclave  # Clave del usuario
13
14 [Log]
15
16 logfile  = /var/log/celda/celda.log # archivo de bitacoras
17
18 [IDA] # Rutas hacia los algoritmos de dispersion y recuperacion
19
20 disperse = /ruta/hacia/ida/dis
21 recover  = /ruta/hacia/ida/rec
```

Por cada uno de los EVA residentes en un nodo es necesario que exista un archivo de configuración como este, los datos que aparecen en él indican entre otras cosas el tamaño de la UMA, el puerto por el que el proceso escuchará las peticiones, el directorio en el que se almacenarán los bloques , la ruta hacia los programas encargados de codificar los bloques usando el ADI y la ruta hacia el archivo en el que se almacenarán las bitácoras.

Una vez creado el archivo de configuración el siguiente paso es crear los directorios necesarios, el directorio en el que se van a almacenar los bloques, y el directorio del archivo de bitácoras.

Por defecto, el sistema buscará el archivo de configuración en el directorio */etc/celda/celdad.conf*, asumiendo que el sistema se está ejecutando en un ambiente Linux, si no es el caso, la ruta hacia el archivo de configuración puede especificarse mediante la opción correspondiente (ver sección B.4).

Para copiar el código a todos los nodos internos, puede usarse cualquier medio, desde simplemente copiarlo en un dispositivo de almacenamiento, como una memoria USB, subirlo a un servidor y descargarlo en cada uno de los nodos.

Una vez que el código se ha colocado en su lugar, hay que agregarlo a las variables de entorno PATH y PYTHONPATH ; la primera es necesaria para poder invocar el servicio de la celda solamente por su nombre (celdad), y la segunda es necesaria para que el intérprete de Python pueda encontrar los paquetes que componen el prototipo. De nuevo, la forma de hacer esto varía según el sistema operativo que se esté utilizando.

Cuando ya se tiene todo acomodado en su lugar, es necesario construir las bases de datos para cada uno de los EVA y el representante de la celda; como ya hemos dicho con anterioridad, el código tanto de los EVA como del nodo representante es idéntico y en consecuencia, la

estructura de la base de datos para cada uno, también lo es. Junto con el código del prototipo se proporciona un script llamado *install*, cuya finalidad es crear las tablas necesarias para el funcionamiento de los nodos. Es recomendable consultar las opciones disponibles en este comando mediante la opción *–help*, como se muestra en el listado B.8.

Listado B.8: Opciones del comando para construir la base de datos.

```
$ ./install --help
Usage: install [options]

Options:
  -h, --help            show this help message and exit
  -m MACHINES, --machines=MACHINES
                        Sets machines file
  -u USERS, --users=USERS
                        Sets users file
  -c CONF, --conf=CONF
```

Los valores que deben ser proporcionados en las opciones del listado B.8 son las siguientes:

-m, –machines Recibe la ruta hacia un archivo que contiene una lista de los EVA que formarán parte de la celda de almacenamiento, en el listado B.9 se muestra un ejemplo de como luce este archivo.

-u, –users Recibe la ruta hacia un archivo que contiene una lista de usuarios, esta opción se utiliza para dar de alta usuarios de prueba, en el listado B.10, se muestra un ejemplo de como luce este archivo.

-c, –config Reciba la ruta hacia un archivo de configuración con el formato mostrado en el listado B.7.

Listado B.9: Ejemplo de un archivo que contiene la lista de EVAs para dar de alta en la base de datos

```
1  # Add to this file your SAD virtual spaces, in this format
2  # Agrega a este archivo los datos de tus espacios virtuales
3  # de almacenamiento, de acuerdo con el siguiente formato:
4  #
5  # [nombre] protocolo://direccion [puerto] [capacidad]
6  #
7  # Donde:
8  #
9  #   * [nombre] ............ Es el nombre del EVA.
10 #   * protocol://hostname . Es la URI donde se hospeda el EVA.
11 #   * [puerto] ............ Puerto asignado al EVA.
12 #   * [capacity] .......... Indica la capacidad (en Bytes) del EVA.
13
14 arte01 http://192.168.3.142 4242 209715200
15 arte02 http://192.168.3.143 4242 209715200
16 arte03 http://192.168.3.144 4242 209715200
17 arte04 http://192.168.3.145 4242 209715200
18 arte05 http://192.168.3.146 4242 209715200
```

Listado B.10: Ejemplo de un archivo que contiene la lista de usuarios para dar de alta en la base de datos

```
1
2
3  # Tabla de usuarios de prueba.
4  # Usuario  Clave          Nombre real        Correo         Servicio
5  root       passroot       Celda              root@arte         1
6  diego      passdiego      Diego Guzman       diego@arte        1
```

Suponiendo que los archivos de los listados B.7, B.9 y B.10 se llaman *celdad.conf*, *maquinas.conf* y *usuarios.conf* respectivamente, y que todos se encuentran en el directorio */etc/celda/*, el script *install* se puede invocar como se muestra en el listado B.11

Listado B.11: Creando la base de datos de un EVA o el nodo representante.

```
$ cd /ruta/hacia/el/script
$ ./install -c /etc/celda/celdad.conf -m /etc/celda/maqunas.conf \
-u /etc/celda/usuarios.conf
```

B.4 Iniciando el servicio de almacenamiento

Una vez instalados correctamente todos los componentes, es posible iniciar los servicios en los nodos participantes, tanto en los internos, como en el representante, tomando en cuenta que el nodo representante debe ser aquel que es visible desde el exterior de la red en que se encuentran los otros nodos (ver figura B.2).

Listado B.12: Uso del prototipo

```
$ celdad -h
Usage: celdad [options]

Options:
  -h, --help            show this help message and exit
    -c CONFIG, --config=CONFIG
                        Config file
    -m MODE, --mode=MODE  proxy | node
```

Los valores que deben ser proporcionados en las opciones del listado B.12 son las siguientes:

-m, –mode Tiene dos posibles valores: *proxy* y *node*, los cuales le indican al programa, respectivamente, si el nodo va a desempeñar el rol de nodo representante, o espacio virtual de almacenamiento.

-c, –config Reciba la ruta hacia un archivo de configuración con el formato mostrado en el listado B.7.

Hay que destacar que estos dos valores son opcionales, si ninguno de ellos es especificado, el sistema se iniciará por defecto en modo *node*, el cual corresponde a que el nodo desempeñe el rol de espacio virtual de almacenamiento (EVA), y el archivo de configuración*configuración, archivo de* que intentará leer es */etc/celda/celdad.conf* y en caso de que éste último no exista, el sistema lanzará una excepción, por lo tanto, si se desea iniciar que el nodo será un EVA, y el archivo de configuración es el predeterminado, el sistema puede inicializarse simplemente llamando al script *celdad* sin ningún parámetro (ver el listado B.13), y en caso de que se desee iniciar el sistema como nodo representante, hay que especificar el parámetro *–m proxy* o *--mode=proxy* según se prefiera (ver el listado B.14).

Listado B.13: Iniciar un EVA sin especificar ningún parámetro al inicio

```
$ celdad
        Listening Port: 4242
        Storage Path:   /storage/celda
        MSU:            10485760 bytes
```

Listado B.14: Iniciar al representante de la celda especificando el parámetro *--mode*

```
$celdad --mode=proxy
        Listening Port: 4242
        Storage Path:   /storage/celda
        MSU:            10485760 bytes
```

En ambos casos, el comando reporta en su salida el puerto por el que escucha las peticiones, el directorio donde almacenará los bloques que reciba y el tamaño en bytes de la MSU. En el momento en que todos los nodos participantes estén activos, la celda estará lista para ser utilizada a través del cliente.

B.5 Uso del cliente

La aplicación cliente también cuenta con su archivo de configuración, que aunque más sencillo, también contiene datos importantes. Un ejemplo de este archivo de configuración es el que se muestra en el listado B.15.

Listado B.15: Ayuda para el cliente de la celda

```
[General]

# URL del representante de la celda.
proxy     = http://192.168.86.99:4242

# Directorio local en el que se guardaran los datos recuperados.
downloads = /home/diacus/Descargas
```

```
# Cantidad maxima de intentos fallidos para conectarse a la celda.
maxtries  = 3

[Log]

# Archivo de bitacoras para el cliente.
logfile   = /home/diacus/.celda/celda.log
```

El cliente de la celda de almacenamiento es una aplicación con una interfaz de línea de comandos. Una pequeña ayuda de cómo invocar el cliente para conectarse a la celda se muestra en el listado B.16.

Listado B.16: Ayuda para el cliente de la celda

```
$ celda -h
Usage: celda [options]

Options:
  -h, --help            show this help message and exit
  -c CONFIG, --config=CONFIG
                        Config file
  -u USER, --user=USER  User name
  -p PROXY, --proxy=PROXY
                        Proxy's URL
```

Típicamente es necesario especificar la url del representante y el nombre del usuario para conectarse a la celda; acto seguido, se le solicitará la contraseña del usuario especificado (ver el listado B.17).

Listado B.17: Conectando el cliente a la celda de almacenamiento

```
$ celda -u admin -p http://192.168.3.142:4242
Password: *******
celda:> _
```

Una vez conectado al sistema, el usuario puede interactuar con la celda mediante el uso de los comandos disponibles, los cuales se muestran en la tabla B.1.

Los parámetro que necesita cada uno de estos comandos puede consultarse directamente en el cliente, ejecutando el comando *help* como se indica en el listado B.18.

Listado B.18: Obteniendo ayuda desde el cliente.

```
celda:> help [comando]
```

Donde *[comando]* es cualquiera de los comandos mostrados en la tabla B.1.

Comando	**Propósito**
post	Envía un archivo a la celda para ser almacenado.
get	Recupera un archivo de la celda especificando su nombre.
ls	Consulta la lista de archivos que están almacenados en la celda.
lls	Consulta el contenido del un directorio local.
cd	Cambia el directorio de trabajo local.
pwd	Consulta el directorio de trabajo actual.
adduser	Agrega un usuario.
addgroup	Agrega un grupo.
whoami	Le pregunta al sistema ¿cuál es el usuario con el que se está conectado a la celda.
exit	Finaliza la conexión con la celda de almacenamiento.

Tabla B.1: Comandos disponibles para utilizar la celda de almacenamiento.

C Documentación del prototipo

Una de las partes mas importantes del diseño, es la definición de una interfaz que permita construir sobre la celda aplicaciones que proporcionen otros servicios (ver el capítulo C.1). Además de la interfaz de programación de aplicaciones, es muy útil conocer la implementación del prototipo (ilustrada en la figura C.1), para abrir la posibilidad de realizar otras implementaciones de la celda, por ejemplo utlizando otras plataformas de desarrollo, modelos de comunicación, etc.

En este apéndice se presenta una descripción detallada de la implementación del prototipo, y en particular de la interfaz de programación de aplicaciones, mediante la cual se puede utilizar la celda como un servicio de almacenamiento.

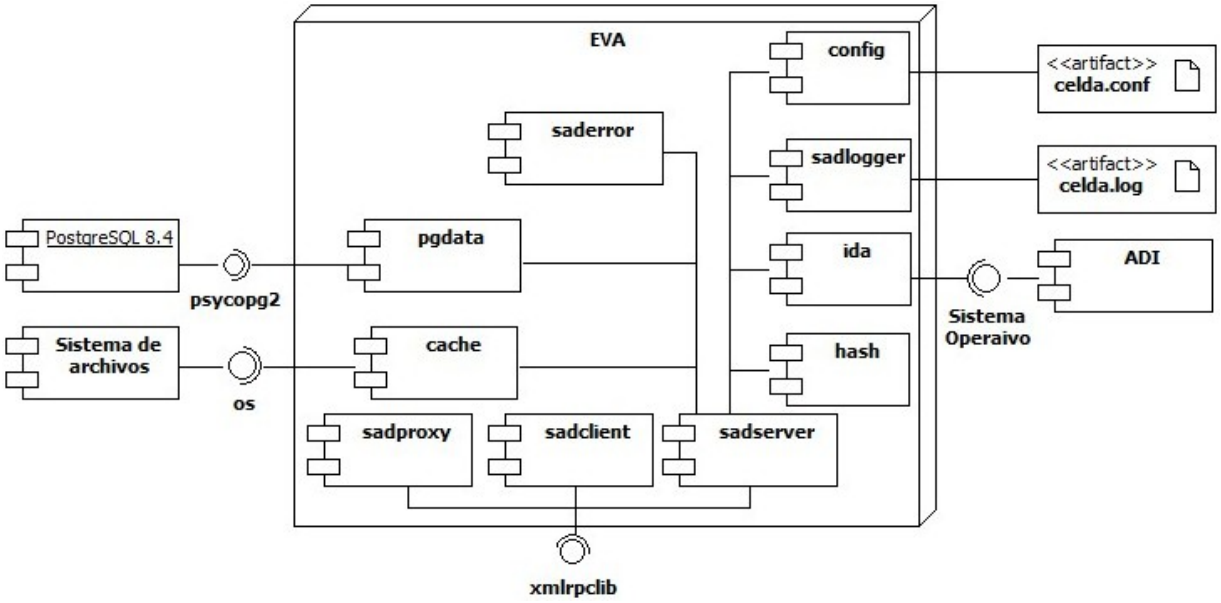


Figura C.1: Componentes que integran el prototipo

C.1 Paquete client

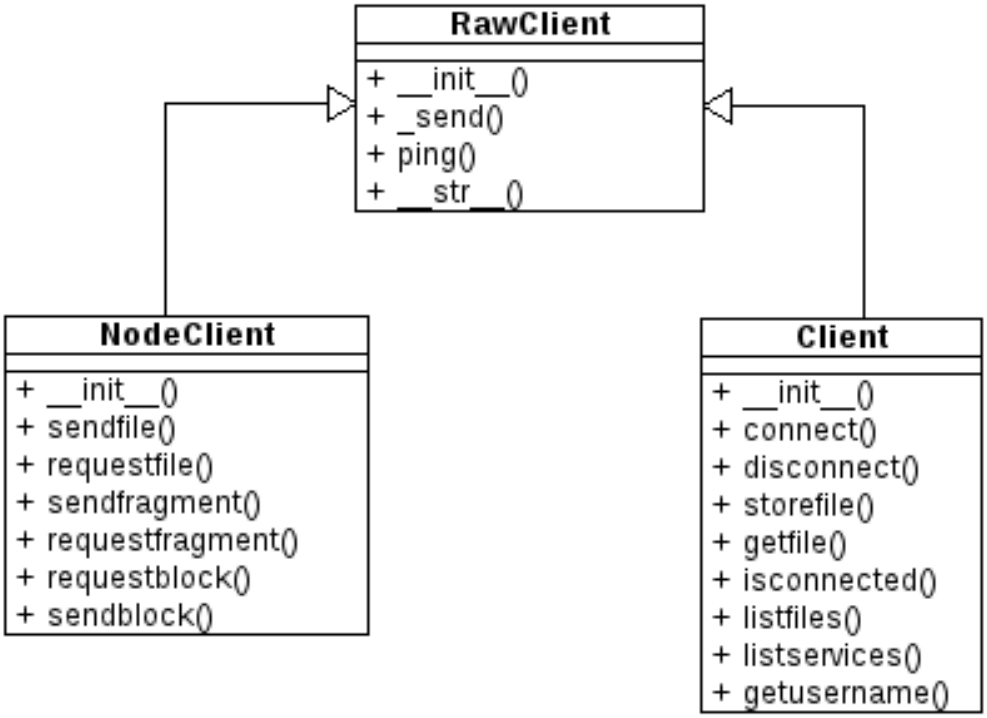


Figura C.2: Clases cliente

C.1.1 Módulos

- **sadclient**: En este módulo se implementan las clases desde las cuales se crean los clientes del sistema, tanto la interfaz con el usuario final, como la interfaz que utilizan los EVA, el representante de la celda o el usuario final para enviar solicitudes a sus vecinos (ver figura C.2).
(Sección C.2, p. 81)

C.1.2 Variables

Nombre	Descripción
__package__	**Valor:** `None`

C.2 Módulo client.sadclient

En este módulo se implementan las clases desde las cuales se crean los clientes del sistema, tanto la interfaz con el usuario final, como la interfaz que utilizan los EVA, el representante de la celda o el usuario final para enviar solicitudes a sus vecinos.

C.2.1 Variables

Nombre	Descripción
__package__	**Valor:** `'client'`

C.2.2 Clase RawClient

Subclases conocidas:
client.sadclient.Client, client.sadclient.NodeClient

Clase base para los clientes del sistema.

En ésta clase se implementan los métodos básicos _send, ping, y __str__

Métodos

__init__(*self*, *uri*)

Constructor de la clase RawClient

Parámetros

`uri`: Contiene la URI del representante de a celda.

(type=Cadena de caracteres)

ping(*self*)

Envía un mensaje de prueba para verificar que el servidor esté funcionando.

__str__(*self*)

Devuelve una cadena de caracteres que representa al cliente.

C.2.3 Clase Client

client.sadclient.RawClient
└
client.sadclient.Client

Clase que define los clientes usados para conectar a los usuarios con el representante de la celda.

Las instancias de esta clase pueden entenderse como el API a través de la cual, tando personas como otras aplicaciones pueden gozar de los servicios que ofrece la celda de almacenamiento.

Métodos

__init__(*self*, *uri*, *downloads*)

Constructor de la clase Client

Parámetros

`uri:` Representa la URI del representante de a celda.

(type=Cadena de caracteres.)

`downloads:` Indica el directorio en el que se van a almacenar los archivos recuperados desde la celda de almacenamiento.

(type=Cadena de caracteres.)

Redefine: client.sadclient.RawClient.__init__

connect(*self*, *user*, *password*)

Establece una conexión con el representante de la celda. Devuelve un indicador del estado de la conexión.

Parámetros

`user:` Representa el nombre del usuario.

(type=Cadena de caracteres.)

`password:` Conraseña asociada al usuario.

(type=Cadena de caracteres.)

disconnect(*self*)

Cierra la conexión con el servidor

storefile(*self*, *fname*, *service*)

Envía un archivo a la celda de almacenamiento. Devuelve una tupla que contiene el estado de la petición y en caso de éxito, el número de versión actual del archivo enviado.

Parámetros

`fname:` El nombre del archivo.

(type=Cadena de caracteres.)

`service:` Indica el nivel de servicio con el que se almacenará el archivo.

(type=Cadena de caracteres.)

getfile(*self*, *fname*)

Recupera un archivo de la celda de almacenamiento. Devuelve una instancia de la clase FileStream, que contiene el archivo solicitado.

Parámetros

`fname:` El nombre del archivo que se desea recuperar.

(type=Cadena de caracteres.)

isconnected(*self*)

Devuelve Verdadero si la conexión se ha establecido exitosamente, Falso en otro caso.

listfiles(*self*)

Devuelve una lista con los nombres de archivo asociados al usuario que está actualmente conectado.

listservices(*self*)

Devuelve una lista que contiene los servicios que están disponibles para el usuario que se encuentra actualmente conextado.

getusername(*self*)

Devuelve el nombre de usuario actual.

Heredado de client.sadclient.RawClient(Sección C.2.2)

__str__(), ping()

C.2.4 Clase NodeClient

client.sadclient.RawClient
└
client.sadclient.NodeClient

Cliente utilizado por los EVA y el representante para enviar solicitudes a sus vecinos.

Métodos

__init__(*self*, *uri*)

Constructor de la clase NodeClient

Parámetros

`uri`: Contiene la dirección del servidor, el formato de la URI es el siguiente: protocolo://servidor:puerto

(type=Cadena de caracteres.)

Redefine: client.sadclient.RawClient.__init__

sendfile(*self*, *ffile*)

Envía una instancia de la clase FileStream hacia el nodo con el que el cliente está conectado.

Parámetros

`ffile`: El archivo a transmitir.

(type=Instancia de la clase FileStream)

requestfile(*self*, *fname*, *uid*)

Solicita el archivo llamado "fname" perteneciente al usuario con identificador "uid" al nodo con el que el cliente está conectado. Devuelve una instancia de la case FileStream que contiene el archivo solicitado.

Parámetros

`fname`: El nombre del archivo solicitado.

(type=Cadena de caracteres.)

`uid`: Identificador de usuario del propietario del archivos.

(type=Número entero.)

sendfragment(*self*, *fragment*)

Envía un fragmento hacia el nodo con el que el cliente está conectado.

Parámetros

`fragment`: Fragmento a transmitir.

(type=Instancia de la clase FragmentStream.)

requestfragment(*self*, *fragment*)

Solicita un fragmento al EVA que fue encargado de procesarlo.

Parámetros

`fragment`: Un descriptor del fragmento, que contiene todos los metadatos necesarios para identificar al EVA que es responsable de su procesamiento

requestblock(*self*, *block*)

Solicita un bloque al EVA responsable de almacenarlo.

Parámetros

`block`: Un descriptor del bloque que quiere recuperarse, que contiene todos los metadatos necesarios para identificar al EVA que es responsable de almacenarlo.

sendblock(*self*, *block*)

Envía un bloque hacia el nodo con el que el cliente está conectado.

Parámetros

`block`: bloque a transmitir.

(type=Instancia de la clase BlockStream)

Heredado de client.sadclient.RawClient(Sección C.2.2)

__str__(), ping()

C.3 Paquete data

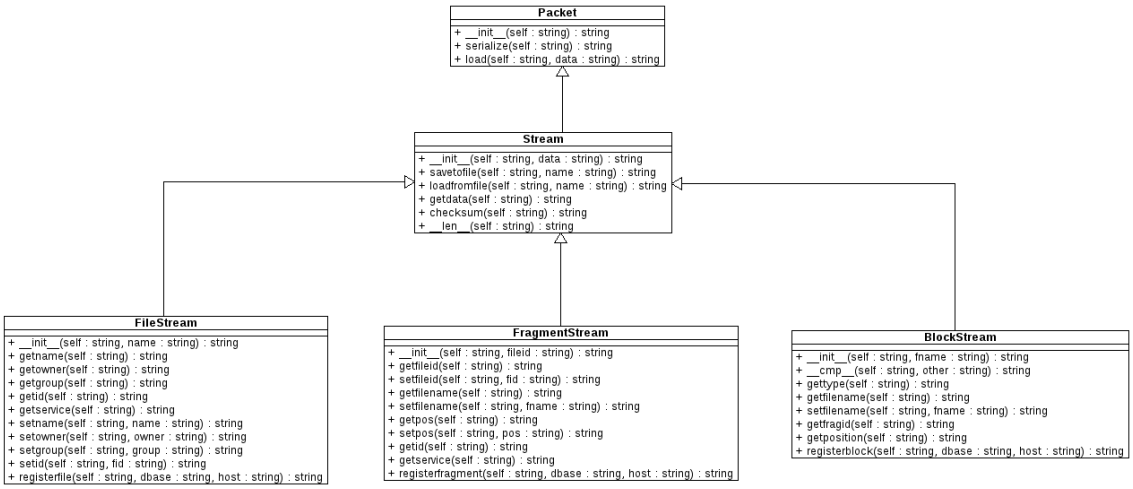


Figura C.3: Clases del paquete data

C.3.1 Módulos

- **message**: En este módulo se definen las clases abstractas Message y Packet
  (Sección C.4, p. 88)
- **stream**: En este módulo se definen las clases Stream, FileStream, FragmentStream y Blockstream, utilizadas para transportar flujos de datos.
  (Sección C.5, p. 91)
- **users**: En este módulo se define la clase SADUser
  (Sección C.6, p. 103)

C.3.2 Variables

Nombre	Descripción
__package__	**Valor:** `None`

C.4 Módulo data.message

En este módulo se definen las clases abstractas Message y Packet

Las cuales sirven para construir los mensajes que se intercambian entre los EVA, el nodo representante y el usuario final.

C.4.1 Variables

Nombre	Descripción
__package__	**Valor:** `None`

C.4.2 Clase Message

Un mensaje es básicamente una tupla de dos objetos, donde el primero de ellos es un número entero que sirve de etiqueta para que el receptor sepa de qué tipo de mensaje se trata, y el segundo miembro es el mensaje en sí.

Esta clase contiene los códigos de mensaje de los posibles tipos que pueden ser enviados entre los EVA y el representante.

Variables de clase

Nombre	Descripción
SAVESTREAM	**Valor:** `1000`
SAVEFRAGMENT	**Valor:** `1001`
SAVEBLOCK	**Valor:** `1002`
SAVECOPY	**Valor:** `1003`
LOADSTREAM	**Valor:** `1004`
LOADFRAGMENT	**Valor:** `1005`
LOADBLOCK	**Valor:** `1006`
ADDUSER	**Valor:** `1010`

continúa en la siguiente página

Nombre	Descripción
DISUSER	**Valor:** `1011`
ADDGROUP	**Valor:** `1012`
DELGROUP	**Valor:** `1013`
ADDNODE	**Valor:** `1014`
LISTFILES	**Valor:** `1015`
LISTSERVS	**Valor:** `1016`
CONNECT	**Valor:** `1100`
DISCONNECT	**Valor:** `1100`
SUCCESS	**Valor:** `2000`
FAILURE	**Valor:** `2001`
ACCESS_DENIED	**Valor:** `2002`

C.4.3 Clase Packet

Subclases conocidas:
data.users.SADFile, data.users.SADUser, data.stream.Stream

Cuando se quiere enviar un objeto a través de la red, lo que en realidad se hace es copiar el estado del objeto en una estructura de datos y enviarla empaquetada en un mensaje, cuando el servidor recibe ese mensaje, crea una instancia nueva de la clase correspondiente y la inicializa con el estado que recibió en el mensaje, de este modo se crea el efecto de que el objeto fue transmitido.

Dado que los objetos que necesitan ser enviados pertenecen a clases diferentes, se tomó la desición de delcarar esta clase abstracta que le permita a los objetos de sus clases derivadas copiar su estado en un diccionario, el cual es la estructura de datos que será transmitida en cada mensaje.

Métodos

__**init**__(*self*)

serialize(*self*)

Devuelve un diccionario que contiene una copia del estado del objeto.

load(*self*, *data*)

Le asigna al objeto el estado contenido en el diccionario "data".

Parámetros

`data`: Contiene el estado que se desea asignar al objeto.

(type=Instancia de la clase dict.)

C.5 Módulo data.stream

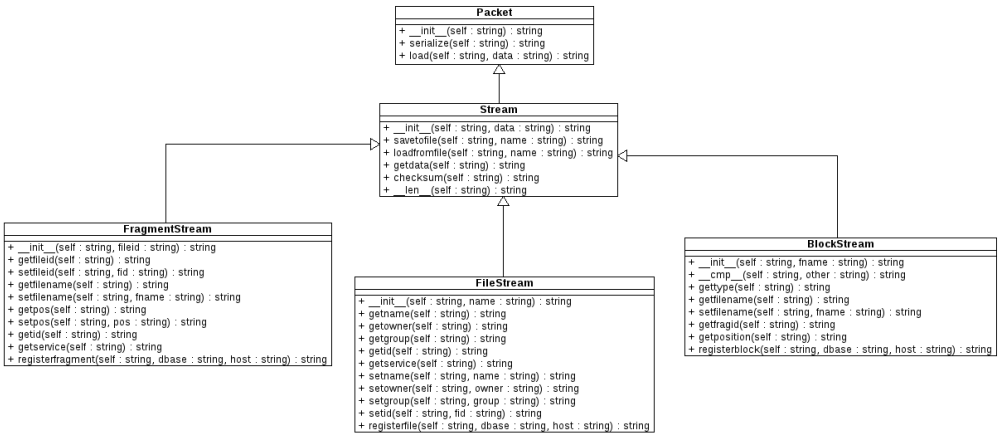


Figura C.4: Clases que de representan los flujos de datos

En este módulo se definen las clases Stream, FileStream, FragmentStream y Blockstream, utilizadas para transportar flujos de datos (ver figura C.4).

C.5.1 Variables

Nombre	Descripción
__package__	**Valor:** `'data'`

C.5.2 Clase Stream

data.message.Packet

data.stream.Stream

Subclases conocidas:
data.stream.BlockStream, data.stream.FileStream, data.stream.FragmentStream

Clase abstracta que representa un flujo de datos genérico

Métodos

__init__(*self*, *data=''*)

Constructor de la clase Stream.

Parámetros

`data`: Contenido del flujo de datos.

Redefine: data.message.Packet.__init__

savetofile(*self*, *name*)

Almacena el contenido del flujo en un archivo.

Parámetros

`name`: Nombre del archivo en el que se quiere almacenar el contenido del flujo.

(type=Cadena de caracteras.)

loadfromfile(*self*, *name*)

Recupera el contenido de un archivo y lo carga en el objeto.

Parámetros

`name`: El nombre del archivo que se desea cargar.

(type=Cadena de caracteres.)

getdata(*self*)

Devuelve una copia del flujo de datos.

checksum(*self*)

Devuelve una cadena que sirve para verificar la integridad de los datos contenidos en el flujo.

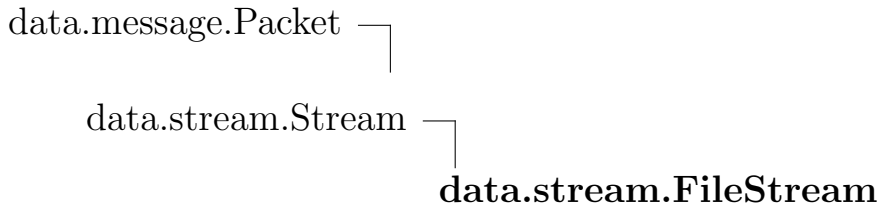
__**len**__(*self*)

Devuelve el tamaño del flujo de datos.

Heredado de data.message.Packet(Sección C.4.3)

load(), serialize()

C.5.3 Clase FileStream



Clase que representa el flujo de datos de un archivo.

Métodos

__init__(*self*, *name*=`''`, *owner*=`None`, *group*=`None`, *serv*=`''`, *data*=`''`)

Constructor de la clase FileStream.

Parámetros

`name:` Nombre del archivo.

(type=Cadena de caracteres)

`owner:` Identificador del dueño del archivo

(type=Número entero)

`group:` Identificador del grupo al que pertenece el dueño del archivo.

(type=Número entero)

`data:` Contenido del archivo

(type=Cadena de caracteres)

Redefine: data.message.Packet.__init__

getname(*self*)

Devuelve el nombre del archivo.

getowner(*self*)

Devuelve el identificador del dueño del archivo.

getgroup(*self*)

Devuelve el identificador del grupo al que pertenece el dueño del archivo.

getid(*self*)

Devuelve el identificador del archivo.

getservice(*self*)

Devuelve el nivel de servicio con el que se almacenó o almacenará el archivo.

setname(*self*, *name*)

Actualiza el nombre del archivo.

Parámetros

`name`: El nuevo nombre del archivo.

(type=Cadena de caracteres.)

setowner(*self*, *owner*)

Actualiza el identificador del propietario del archivo.

Parámetros

`owner`: Identificador del propietario del archivo.

(type=Número entero.)

setgroup(*self*, *group*)

Actualiza el identificador del grupo al que pertenece el dueño del archivo.

Parámetros

`group`: El identificador del grupo.

(type=Número entero.)

setid(*self*, *fid*)

Actualiza el identificador del archivo.

Parámetros

`fid`: Identificador del archivo.

(type=Número entero.)

registerfile(*self*, *dbase*, *host*)

Registra los metadatos que describen el archivo en la base de datos.

Parámetros

`dbase`: Controlador de la base de datos.

(type=En esta implementación es una instancia de la clase PGDataBase.)

`host`: Identificador del EVA en donde el archivo será procesado.

(type=Número entero.)

Heredado de data.stream.Stream(Sección C.5.2)

__len__(), checksum(), getdata(), loadfromfile(), savetofile()

Heredado de data.message.Packet(Sección C.4.3)

load(), serialize()

C.5.4 Clase FragmentStream

data.message.Packet
└ data.stream.Stream
└ **data.stream.FragmentStream**

Clase que representa un fragmento de archivo.

Métodos

__init__(*self*, *fileid*=0, *fname*='', *pos*=0, *service*='', *data*='')

Constructor de la clase FileStream.

Parámetros

`fileid`: Identificador del archivo al que pertenece el fragmento.

(type=Número entero.)

`fname`: Nombre del archivo al que pertenece el fragmento.

(type=Número entero.)

`pos`: Posición del fragmento en el archivo al que pertenece.

(type=Número entero)

`data`: Contenido del fragmento

(type=Cadena de caracteres.)

Redefine: data.message.Packet.__init__

getfileid(*self*)

Devuelve el identificador del archivo al que pertenece el fragmento.

setfileid(*self*, *fid*)

Actualiza el identificador del archivo al que pertenece el fragmento.

getfilename(*self*)

Devuelve el nombre del archivo al que pertenece el fragmento.

setfilename(*self*, *fname*)

Actualiza el nombre del archivo al que pertenece el fragmento.

getpos(*self*)

Devuelve la posición que ocupa el fragmento dentro del archivo al que pertenece.

setpos(*self*, *pos*)

Actualiza la posición que ocupa el fragmento dentro del archivo al que pertenece.

getid(*self*)

Devuelve el identificador del fragmento.

getservice(*self*)

Devuelve el nivel de servicio con que se almacenará el archivo al que pertenece el fragmento.

registerfragment(*self*, *dbase*, *host*)

Registra los metadatos que describen al fragmento en la base de datos.

Parámetros

`dbase:` Controlador de la base de datos.

(type=En esta implementación es una instancia de la clase PGDataBase.)

`host:` Identificador del EVA en donde el archivo será procesado.

(type=Número entero.)

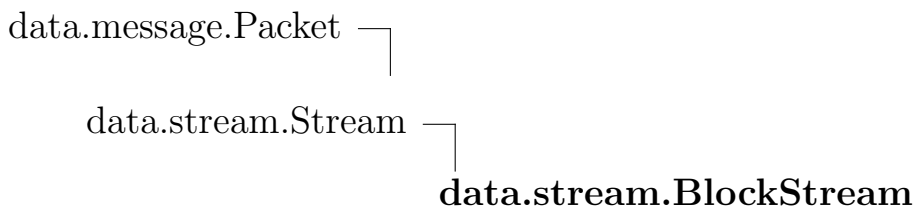
Heredado de data.stream.Stream(Sección C.5.2)

__len__(), checksum(), getdata(), loadfromfile(), savetofile()

Heredado de data.message.Packet(Sección C.4.3)

load(), serialize()

C.5.5 Clase BlockStream



Clase que representa un bloque, ya sea este una réplica simple o un disperso codificado usando el Algoritmo de Rabin.

Métodos

__init__(*self*, *fname*='', *btype*='copy', *fragid*=0, *pos*=0, *data*='')

Constructor de la clase BlockStream

Parámetros

`fname:` Archivo al que pertenece el bloque.
(type=Cadena de caracteres)

`btype:` El tipo del bloque (copia, disperso, etc.)
(type=Cadena de caracteres.)

`fragid:` Identificador del fragmento al que pertenece el bloque.
(type=Número entero.)

`pos:` Posición al que pertenece el bloque.
(type=Número entero.)

`data:` Contenido del bloque.
(type=Cadena de caracteres.)

Redefine: data.message.Packet.__init__

__cmp__(*self*, *other*)

Compara dos instancias de la clase BlockStream

gettype(*self*)

Devuelve el tipo del bloque (copia, disperso, etc.).

getfilename(*self*)

Devuelve el nombre del archivo

setfilename(*self*, *fname*)

Actualiza el nombre del archivo

Parámetros

`fname`: El nombre del archivo

(type=Cadena de caracteres)

getfragid(*self*)

Devuelve el identificador del fragmento al que pertenece el bloque.

getposition(*self*)

Devuelve la posición del bloque dentro del fragmento al que pertenece.

registerblock(*self*, *dbase*, *host*)

Registra los metadatos que describen al bloque en la base de datos.

Parámetros

`dbase:` Controlador de la base de datos.

(type=En esta implementación es una instancia de la clase PGDataBase.)

`host:` Identificador del EVA en donde el archivo será procesado.

(type=Número entero.)

Heredado de data.stream.Stream(Sección C.5.2)

__len__(), checksum(), getdata(), loadfromfile(), savetofile()

Heredado de data.message.Packet(Sección C.4.3)

load(), serialize()

C.6 Módulo data.users

En este módulo se define la clase SADUser

C.6.1 Variables

Nombre	Descripción
__package__	**Valor:** `'data'`

C.6.2 Clase SADUser

data.message.Packet
└
data.users.SADUser

Subclase de Packet, representa a un usuario dentro del sistema

Métodos

__**init**__(*self*, *uid*, *nick*, *passwd*, *fullname*, *email*, *createdon*)

SADUser clase constructor Constructor de la clase

Parámetros

`uid:`	Representa el identificador del usuario dentro del sistema. *(type=Número entero.)*
`nick:`	El nombre del usuario dentro del sistema. *(type=Cadena de caracteres.)*
`passwd:`	La contraseña del usuario. *(type=Cadena de caracteres.)*
`fullname:`	Nombre real del usuario. *(type=Cadena de caracteres.)*
`email:`	Dirección de correo electrónico del usuario. *(type=Cadena de caracteres)*
`createdon:`	Fecha de creación de la cuenta de usuario.

Redefine: data.message.Packet.__init__

setnick(*self*, *nick*)

Actualiza el nombre del usuario dentro del sistema.

Parámetros

`nick`: El nombre del usuario dentro del sistema.

(type=Cadena de caracteres.)

setpasswd(*self*, *passwd*)

Actualiza la contraseña del usuario dentro del sistema.

Parámetros

`passwd`: La contraseña del usuario.

(type=Cadena de caracteres.)

setfullname(*self*, *fullname*)

Actualiza el nombre real del usuario.

Parámetros

`fullname`: Nombre real del usuario.

(type=Cadena de caracteres.)

setemail(*self*, *email*)

Actualiza la dirección de correo electrónico del usuario.

Parámetros

`email`: Dirección de correo electrónico del usuario.

(type=Cadena de caracteres)

getid(*self*)

Devuelve el identificador del usuario.

getnick(*self*)

Devuelve el nombre del usuario dentro del sistema.

getfullname(*self*)

Devuelve el nombre real del usuario.

getemail(*self*)

Devuelve la dirección de correo electrónico del usuario.

getservices(*self*)

Devuelve una lista con los niveles de servicio disponibles para el usuario.

getpasswd(*self*)

Devuelve la contraseña del usuario.

update(*self*, *dbase*)

Actualiza la información del usuario en la base de datos.

Parámetros

`dbase`: Controlador de la base de datos

(type=En esta implementación es una instancia de la clase PGDataBase.)

Heredado de data.message.Packet(Sección C.4.3)

load(), serialize()

C.6.3 Clase SADFile

data.message.Packet
└
data.users.SADFile

Clase que describe un archivo:

Métodos

__init__(*self*, *fid*, *name*, *size*, *version*, *since*)

Constructor de la clase SADFile

Parámetros

`fid:` Identificador del archivo.
(type=Número entero.)

`name:` Nombre del archivo.
(type=Cadena de caracteres.)

`size:` Tamaño del archivo.
(type=Número entero.)

`version:` Versión del archivo.
(type=Número entero.)

`since:` Fecha de creación del archivo.

Redefine: data.message.Packet.__init__

__str__(*self*)

Devuelve una representación como cadena de caracteres, del descriptor de archivo.

Heredado de data.message.Packet(Sección C.4.3)

load(), serialize()

C.7 Paquete db

C.7.1 Módulos

- **pgdata**: En este módulo se define la interfaz entre el sistema y la base de datos, esta interfaz proporciona al sistema una capa de abstracción que le oculta los detalles de la estructura de los metadatos.
(Sección C.8, p. 110)

C.7.2 Variables

Nombre	Descripción
__package__	**Valor:** `None`

C.8 Módulo db.pgdata

En este módulo se define la interfaz entre el sistema y la base de datos, esta interfaz proporciona al sistema una capa de abstracción que le oculta los detalles de la estructura de los metadatos.

Si se quiere conectar el sistema con un manejador de base de datos diferente, simplemente hay que implementar el controlador correspondiente respetando la interfaz definida aquí.

C.8.1 Funciones

dbsetup(*database*)

Crea las tablas de la base de datos.

Parámetros

`database`: Controlador de la base de datos.

(type=Instancia de la clase PGDataBase.)

C.8.2 Variables

Nombre	Descripción
__package__	**Valor:** `'db'`

C.8.3 Clase PGDataBase

Controlador de la base de datos, actúa como una capa de abstracción entre el prototipo y una manejador de base de datos relacional PostgreSQL.

Métodos

__init__(*self*, *user*, *passwd*, *dbname*, *server*='')

Constructor de la clase

Parámetros

`user`: Nombre de usuario en el sistema manejador de base de datos.

(type=Cadena de caracteres.)

`passwd`: Contraseña del usuario en el sistema manejador de base de datos.

(type=Cadena de caracteres.)

`dbname`: Nombre de la base de datos

(type=Cadena de caracteres.)

`server`: Servidor en el que reside la base de datos.

(type=Cadena de caracteres.)

select(*self*, *table*, *fields*=())

Realiza una consulta para seleccionar registros de una base de datos.

Parámetros

`table`: El nombre de la tabla.

(type=Cadena de caracteres.)

`fields`: Los nombres de los campos que se quieren consultar

(type=Tupla de cadenas de caracteres.)

selectservices(*self*)

Devuelve los nombres de todos los servicios disponibles.

selectuserservices(*self*, *uid*)

Devuelve una lista que contiene los nombres de los servicios disponibles para un usuario dado.

Parámetros

`uid`: Identificador del usuario.

(type=Número entero.)

adduser(*self*, *nick*, *password*, *fullname*, *email*, *service*=`1`)

Registra un nuevo usuario en la base de datos.

Parámetros

`nick`: Nombre del usuario dentro del sistema.

(type=Cadena de caracteres.)

`password`: Contraseña del usuario.

(type=Cadena de caracteres.)

`fullname`: Nombre real del usuario.

(type=Cadena de caracteres.)

`email`: Dirección de correo electrónico del usuario.

(type=Cadena de caracteres.)

`service`: Indicador del nivel de servicio por defecto que se le brindará al usuario.

(type=Número entero.)

disableuser(*self*, *uid*)

Inhabilita un usuario

Parámetros

`uid`: Idenificador del usuario.

(type=Número entero.)

enableuser(*self*, *uid*)

Habilita una cuenta de usuario.

Parámetros

`uid`: Idenificador del usuario.

(type=Número entero.)

updateuser(*self*, *uid*, *nick*, *passwd*, *fullname*, *email*)

Actualiza la información de un usuario.

Parámetros

`nick`: Nombre del usuario dentro del sistema.

(type=Cadena de caracteres.)

`passwd`: Contraseña del usuario.

(type=Cadena de caracteres.)

`fullname`: Nombre real del usuario.

(type=Cadena de caracteres.)

`email`: Dirección de correo electrónico del usuario.

(type=Cadena de caracteres.)

selectusers(*self*)

Devuelve una lista que contiene todos los usuarios cuyas cuentas están habilitadas.

selectuser(*self*, *uid*=`None`, *nick*=`None`, *email*=`None`)

Devuelve la información de un usuario buscándolo por identificador, nombre de usuario o correo electrónico.

Parámetros

`uid`: Idenificador del usuario.

(type=Número entero.)

`nick`: Nombre del usuario dentro del sistema.

(type=Cadena de caracteres.)

`email`: Dirección de correo electrónico del usuario.

(type=Cadena de caracteres.)

addvirtualspace(*self*, *label*, *host*, *port*, *size*)

Registra un espacio virtual de almacenamiento (EVA)

Parámetros

`label`: Nombre del espacio virtual.

(type=Cadena de caracteres.)

`host`: Dirección IP o URI del nodo que contiene al EVA.

(type=Cadena de caracteres.)

`port`: Puerto por el que el EVA espera las peticiones.

(type=Número entero.)

`size`: Capacidad (en bytes) del EVA.

(type=Número entero.)

replacevirtualspace(*self*, *vid*, *host*)

Reemplaza la dirección IP o la URI en donde se hospeda el EVA.

Parámetros

`vid`: Identificador del EVA

(type=Número entero.)

`host`: Dirección IP o URI del nodo que contiene al EVA.

(type=Cadena de caracteres)

selectvirtualspaces(*self*)

Devuelve una secuencia que contiene los datos de todos los EVA registrados.

updatevirtualspace(*self*, *vid*, *name*, *host*, *port*, *size*)

Actualiza los datos de un EVA.

Parámetros

`vid`: Identificador del EVA

(type=Número entero.)

`name`: Nombre del EVA.

(type=Cadena de caracteres.)

`host`: Dirección IP o URI del EVA.

(type=Cadena de caracteres.)

`port`: Puerto por el que el EVA espera las peticiones.

(type=Número entero.)

`size`: Capacidad del EVA.

(type=Número entero.)

addgroup(*self*, *group*, *uid*)

Agrega un grupo de usuarios.

Parámetros

`group:` Nombre del grupo nuevo.

(type=Cadena de caracteres.)

`uid:` Identificador del usuario que crea el grupo.

(type=Número entero.)

delgroup(*self*, *gid*)

Elimina un grupo.

Parámetros

`gid:` El identificador del grupo que será destruido.

(type=Número entero.)

adduser2group(*self*, *uid*, *gid*)

Agrega un usuario a un determinado grupo de trabajo.

Parámetros

`uid:` Identificador del usuario.

(type=Número entero.)

`gid:` Identificador del grupo.

(type=Número entero.)

addservice(*self*, *name*, *description=''*)

Agrega un nuevo servicio al sistema.

Parámetros

`name:` Nombre del servicio.

(type=Cadena de caracteres.)

`description:` La descripción del servicio.

(type=Cadena de caracteres.)

bringservice2user(*self*, *usr*, *sid*)

Le otorga un servicio a un usuario dado.

Parámetros

`usr:` Identificador del usuario.

(type=Número entero.)

`sid:` Identificador del servicio.

(type=Número entero.)

disableservice(*self*, *sid*)

Suspende un servicio.

Parámetros

`sid:` Identificador del servicio.

(type=Número entero.)

storefile(*self*, *name*, *fhash*, *service*, *owner*, *gid*, *size*, *host*)

Registra un archivo en la base de datos. Devuelve una tupla que contiene la el número de versión actual del archivo y el identificador de registro.

Parámetros

`name:` Nombre del archivo

(type=Cadena de caracteres)

`fhash:` Cadena de verificación de integridad, en este caso, un MD5 del contenido del archivo.

(type=Cadena de caracteres)

`service:` Nombre del servicio de almacenamiento.

(type=Cadena de caracteres)

`owner:` Identificador de usuario del propietario del archivo.

(type=Número entero.)

`gid:` Identificador del grupo al que pertenece el dueño del archivo.

(type=Número entero.)

`size:` Tamaño del archivo.

(type=Número entero.)

`host:` EVA en elque esrá procesado el archivo.

(type=Número entero.)

selectfile(*self*, *fid*, *uid*)

Selecciona un archivo.

Parámetros

`fid`: Identificador del archivo.

(type=Número entero.)

`uid`: Identificador del usuario.

(type=Número entero.)

selectuserfiles(*self*, *uid*)

Busca los archivos que son propiedad de un usuario determinado, devuelve una lista que contiene descriptores de los archivos encontrados.

Parámetros

`uid`: Identificador del usuario.

(type=Identificador del usuario.)

storefragment(*self*, *fname*, *ffile*, *pos*, *fhash*, *size*, *host*)

Registra un fragmento en la base de datos.

Parámetros

`fname:` Nombre del archivo al que pertenece el fragmento.

(type=Cadena de caracteres)

`ffile:` Identificador del archivo al que pertenece el fragmento.

(type=Número entero.)

`pos:` Posición del fragmento dentro del archivo al que pertenece.

(type=Número entero.)

`fhash:` Cadena de verificación del fragmento.

(type=Cadena de caracteres)

`size:` Tamaño del fragmento.

(type=Número entero.)

`host:` Identificador del EVA en donde será procesado el fragmento.

(type=Núemro entero.)

storeblock(*self*, *fname*, *fragment*, *btype*, *pos*, *fhash*, *host*, *size*)

Registra un bloque en la base de datos.

Parámetros

`fname:` Nombre del archivo al que pertenece el bloque.

(type=Cadena de caracteres)

`fragment:` Identificador del bloque.

(type=Número entero.)

`btype:` El tipo de bloque, para esta implementación, los tipos disponibles son: copia e ida, para la replicación simple y el algoritmo de Rabin respectivamente.

(type=Cadena de caracteres)

`pos:` Posición del bloque dentro del fragmento al que pertenece.

(type=Número entero.)

`fhash:` Cadena de verificación del bloque.

(type=Cadena de caracteres)

`host:` Identificador del EVA en el que se va a almacenar el bloque.

(type=Número entero.)

`size:` Tamaño del bloque.

(type=Número entero.)

droptable(*self*, *table*)

Elimina una tabla.

Parámetros

`table`: Nombre de la tabla que se desea eliminar.

(type=Cadena de caracteres.)

query(*self*, *qry*)

Método utilizado para ejecutar directamente consultas SQL sobre la base de datos.

Parámetros

`qry`: Consulta SQL

(type=Cadena de caracteres.)

C.9 Paquete lib

C.9.1 Módulos

- **common**: En este módulo se definen funciones útiles para varios de los procesos ocurridos al interior de la celda.
  (Sección C.10, p. 124)
- **config**: Las clases de este módulo están encargadas de leer los archivos de configuración del prototipo para recurar los parámetros que rigen su comportamiento.
  (Sección C.11, p. 131)
- **hash**: La finalidad de este módulo es servir de interfaz entre el sistema y los diferentes algoritmos que pueden utilizarse para verificar la integridad de los datos.
  (Sección C.12, p. 136)
- **ida**: La finalidad de este módulo es servir de interfaz entre la celda de almacenamiento y los programas que implementan el algoritmo de dispersión de Rabin, tanto para dispersar fragmentos como para recuperarlos.
  (Sección C.13, p. 137)
- **saderror**: En este módulo se definen Excepciones especiales que sirven para describir los fallos que pueden presentarse en la celda de almacenamiento.
  (Sección C.14, p. 139)

C.9.2 Variables

Nombre	Descripción
__package__	**Valor:** `None`

C.10 Módulo lib.common

En este módulo se definen funciones útiles para varios de los procesos ocurridos al interior de la celda.

C.10.1 Funciones

readmachinesfile(*fname*)

Lee un archvo de texto en busca de datos para dar de alta espacios virtuales de almacenamiento. Devuelve una lista con los datos leidos.

Parámetros

`fname`: Nombre del archivo.

(type=Cadena de caracteres.)

readusersfile(*fname*)

Lee un archvo de texto en busca de datos para dar de alta usuarios. Devuelve una lista con los datos leidos.

Parámetros

`fname`: Nombre del archivo.

(type=Cadena de caracteres.)

readmsu(*msu*=`'1m'`)

Lee una cadena de caracteres que expresa el valor de la UMA, ya se a en Megabytes, Kilobytes, etc. y devuelve un valor numérico equivalente en bytes.

Parámetros

`msu`: Unidad máxima de almacenamiento.

(type=Cadena de caracteres.)

splitstream(*stream*=’’, *msu*=1048576)

Divide una cadena de caracteres en trozos de un tamaño dado o más pequeños, para luego devolverlos en una lista.

Parámetros

`stream`: La información que será dividida.

(type=Cadena de caracteres.)

`msu`: El valor máximo que puede medir un pedazo de información.

(type=Número entero.)

notimplementedfunction(*mtype*, *message*)

Función vacía, que sirve para indicar que hay funcionalidades no implementadas Empty function for not implemented services

Parámetros

`mtype`: Message’s type

`message`: Message’s content

C.10.2 Variables

Nombre	Descripción
__package__	**Valor:** `’lib’`

C.10.3 Clase MessagesENG

Lista de mensajes para la salida estandar y archivos de bitácora.

Métodos

__**init**__(*self*)
Constructr de la clase (vacío) vacío.

Variables de clase

Nombre	Descripción
CantConnectDB	**Valor:** `'Data Base connection refused, check if your service is o...`
Connecting	**Valor:** `'Connecting to the proxy: %s ...'`
ConnectionRefused	**Valor:** `'Connection refused.'`
CreateTable	**Valor:** `'Creating Table: %s.'`
CreateUser	**Valor:** `'Creating user: %s.'`
DataRecived	**Valor:** `'Data recived:\n%s'`
Dispersed	**Valor:** `'%s fragment dispersed'`
Dispersing	**Valor:** `'Dispersing %s fragment'`
FaultRaise	**Valor:** `'Fault %d ocurred: %s.'`
FileNotSaved	**Valor:** `'The file \'%s\' has not been saved.'`
FileSaved	**Valor:** `'The file \'%s\' has been saved. Current version: %d.'`
FileSplitUp	**Valor:** `'The file %s was split up in %d fragments.'`
InvalidOption	**Valor:** `'Wrong option.'`
ListingFiles	**Valor:** `'Listing %s\'s files.'`
ListingServs	**Valor:** `'Listing %s\'s available services.'`
MissingServer	**Valor:** `'You must type an URL.'`
MissingUser	**Valor:** `'You must type an user name.'`

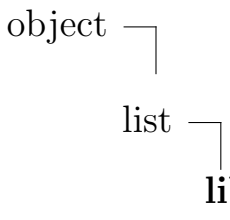
continúa en la siguiente página

Nombre	Descripción
MuchMistakes	**Valor:** 'You\'ve made so much mistakes. Good bye.'
NoIDALinux64	**Valor:** 'There\'s no IDA módulo for Linux x86_64 architecture.'
NoIDAWin32	**Valor:** 'There\'s no IDA module for Windows 32 bits.'
NoServsAvailable	**Valor:** 'There\'s no storage services available. Sorry.'
Notimplemented	**Valor:** 'This function has not been implemented yet.'
ProcessingBlock	**Valor:** 'Processing block %s.'
ProcessingFrag	**Valor:** 'Processing fragment %s.'
RequestProcessed	**Valor:** 'Request processed: %s.'
RequestRecived	**Valor:** 'Request recived: %s.'
SelectedService	**Valor:** 'Selected service: %s.'
SelUsrID	**Valor:** 'Selecting user by id: %d'
SelUsrMail	**Valor:** 'Selecting user by e-mail address: %s'
SelUsrNick	**Valor:** 'Selecting user by name: %s'
SendingFileTo	**Valor:** 'Sending file \'%s\' to %s virtual space (%s). current ve...
SendingFragTo	**Valor:** 'Sending %s file\'s fragment number %d to %s virtual spac...
ServerError	**Valor:** 'Server error [%s]: %s'
ServiceHalt	**Valor:** 'Service Halt.'
SmallFile	**Valor:** 'The file %s is smaller than MSU.'
StartingConnect	**Valor:** 'Starting connection protocol.'
TryAgain	**Valor:** 'Sorry, try again please.'

continúa en la siguiente página

Nombre	Descripción
TryingConnection	**Valor:** `'Trying to connecto to the server: %s'`
TryingLogin	**Valor:** `'%s user is trying to login.'`
UnknownError	**Valor:** `'Unknown Error.'`
UnsupportedReq	**Valor:** `'Unsupported Request %d.'`
UserConnected	**Valor:** `'The user %s has connected to the system.'`
UserRefused	**Valor:** `'Connection refused for %s user.'`

C.10.4 Clase CycleQueue



Una cola circular simple. En esta implementación del prototipo se utiliza esta cola como mecanismo de selección del EVA al que se le va a delegar alguna tarea; en futuras implementaciones de la celda de almacenamiento, puede cambiarse este mecanismo con el fin de mejorar el balance de carga.

Métodos

__init__(*self*, *content*=`None`)

Constructor de la clase.

Parámetros

`content`: El contenido de la cola circular.

(type=Una secuencia.)

Return Valor

new empty list

Redefine: object.__init__

choice(*self*)

Establece de una manera pseudoaleatoria el elemento que será elegido la próxima vez que se invoque el método nextval.

nextval(*self*)

Devuelve el siguiente valor de la cola.

currentpos(*self*)

Devuelve el índice del siguiente elemento de la cola.

current(*self*)

Devuelve el último elemento de la cola en ser seleccionado.

Heredado de list

__add__(), __contains__(), __delitem__(), __delslice__(), __eq__(), __ge__(), __getattribute__(), __getitem__(), __getslice__(), __gt__(), __iadd__(), __imul__(), __iter__(), __le__(), __len__(), __lt__(), __mul__(), __ne__(), __new__(), __repr__(), __reversed__(), __rmul__(), __setitem__(), __setslice__(), __sizeof__(), append(), count(), extend(), index(), insert(), pop(), remove(), reverse(), sort()

Heredado de object

__delattr__(), __format__(), __reduce__(), __reduce_ex__(), __setattr__(), __str__(), __subclasehook__()

Propiedades

Nombre	Descripción
Heredado de object	
__clase__	

Variables de clase

Nombre	Descripción
Heredado de list	
__hash__	

C.11 Módulo lib.config

Las clases de este módulo están encargadas de leer los archivos de configuración del prototipo para recurar los parámetros que rigen su comportamiento.

C.11.1 Variables

Nombre	Descripción
__package__	**Valor:** `'lib'`

C.11.2 Clase SADConfig

object
└ **lib.config.SADConfig**

Clase para generar el objetos de configuración para el nodo representante y cada uno de los EVA.

Métodos

__init__(*self*)

Constructor de la clase.

Redefine: object.__init__

setconfigfile(*self*, *path*)

Asigna la ruta hacia el archivo de configuración.

Parámetros

`path`: Ruta hacia el archivo de configuración.

(type=Cadena de caracteres.)

addoption(*self*, *section*, *option*, *value*)

Agrega un par "clave = valor" en una determinada sección, al objeto de configuración.

Parámetros

`section:` Nombre de la sección.

(type=Cadena de caracteres.)

`option:` Nombre de la opción.

(type=Cadena de caracteres.)

`value:` Valor.

(type=Cadena de caracteres.)

saveconfig(*self*)

Guarda los valores de configuración actuales.

loadconf(*self*)

Recupera los valores de configuración a partir del archivo especificado.

getdb(*self*)

Devuelve una referencia al controlador de base de datos.

getport(*self*)

Devuelve el número de pueto.

getstoragepath(*self*)

Devuelve la ruta en donde se están almacenando los bloques.

getmsu(*self*)

Devuelve el valor de la UMA.

getaddress(*self*)

Devuelve el nombre o la dirección IP del Nodo huesped.

getlogfile(*self*)

Devuelve la ruta hacia el archivo de configuración.

getdispath(*self*)

Devuelve la ruta hacia el programa encargado de dispersar los fragmentos.

getrecpath(*self*)

Devuelve la rutha hacia el programa encargado de recuperar los fragmentos a partir de sus bloques.

Heredado de object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

Propiedades

Nombre	Descripción
Heredado de object	
__class__	

C.11.3 Clase SADClientConfig

Clase para generar el objeto de configuración para el cliente.

Métodos

__init__(*self*)

Constructor de la clase.

setconfigfile(*self*, *path*)

Asigna la ruta hacia el archivo de configuración.

Parámetros

`path`: Ruta hacia el archivo de configuración.

(type=Cadena de caracteres.)

addoption(*self*, *section*, *option*, *value*)

Agrega un par "clave = valor" en una determinada sección, al objeto de configuración.

Parámetros

`section`: Nombre de la sección.

(type=Cadena de caracteres.)

`option`: Nombre de la opción.

(type=Cadena de caracteres.)

`value`: Valor.

(type=Cadena de caracteres.)

saveconfig(*self*)

Guarda los valores de configuración actuales.

loadconf(*self*)

Recupera los valores de configuración a partir del archivo especificado.

getdownloadspath(*self*)

Devuelve la ruta hacia el directorio en donde el cliente guarda los contenidos recuperados.

getproxy(*self*)

Devuelve la URI hacia el representante de la celda.

getmaxtries(*self*)

Pregunta a la celda de almacenamiento, cuantos intentos de acceso fallido tolera antes de negar el acceso.

getlogfile(*self*)

Devuelve la ruta hacia el archivo de configuración.

C.12 Módulo lib.hash

La finalidad de este módulo es servir de interfaz entre el sistema y los diferentes algoritmos que pueden utilizarse para verificar la integridad de los datos.

C.12.1 Funciones

byMD5(*stream*)

Calcula un hash md5 de una cadena de caracteres dada

Parámetros

`stream`: Datos a los que se les aplicará el algoritmo MD5

(type=Cadena de caracteres.)

getCheckSum(*stream*)

Calcula y devuelve una cadena que sirve para verificar la integridad de los datos. Contiene un diccionario cuyas claves son nombres de algoritmos hash, por ejemplo MD5 y SHA1, y sus valores son funciones que implementan dichos algoritmos. Dependiendo de la configuración del sistema, esta función puede utilizar cualquiera de los algoritmos disponibles.

Parámetros

`stream`: Los datos cuya integridad se quiere corroborar.

(type=Cadena de caracteres.)

C.12.2 Variables

Nombre	Descripción
__package__	**Valor:** `'lib'`

C.13 Módulo lib.ida

La finalidad de este módulo es servir de interfaz entre la celda de almacenamiento y los programas que implementan el algoritmo de dispersión de Rabin, tanto para dispersar fragmentos como para recuperarlos.

C.13.1 Funciones

disperse(*fragment*)

Dispersa un fragmento. una vez codificado el fragmento, se generan 5 archivos que contienen sus bloques dispersos, y la función devuelve una lista con las rutas a esos 5 archivos.

Parámetros

`fragment:` fragmento que va a ser codificado.

(type=Instancia de la clase FragmentStream.)

recover(*blockA*, *blockB*, *blockC*)

Recupera un fragmento a partir de cuales quiera tres de sus bloques codificados. Una vez terminado el proceso, la función devuelve una instancia de la clase FragmentStream.

Parámetros

`blockA:` Bloque asociado al fragmento que se desea recuperar.

(type=Instancia de la clase BlockStream.)

`blockB:` Bloque asociado al fragmento que se desea recuperar.

(type=Instancia de la clase BlockStream.)

`blockC:` Bloque asociado al fragmento que se desea recuperar.

(type=Instancia de la clase BlockStream.)

C.13.2 Variables

Nombre	Descripción
__package__	**Valor:** `'lib'`

C.14 Módulo lib.saderror

En este módulo se definen Excepciones especiales que sirven para describir los fallos que pueden presentarse en la celda de almacenamiento.

C.14.1 Variables

Nombre	Descripción
__package__	**Valor:** `'lib'`

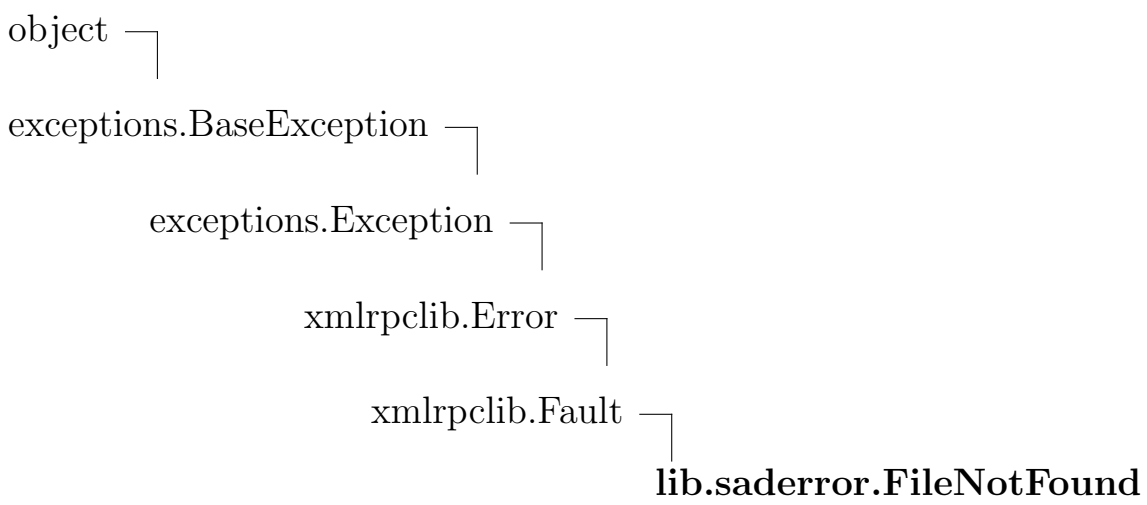
C.14.2 Clase SADError

Esta clase contiene definiciones de códigos de error.

Variables de clase

Nombre	Descripción
FILENOTFOUND	**Valor:** `100`
EMPTYSTREAM	**Valor:** `101`
CLIENTDISCONNECTED	**Valor:** `102`
NOTIMPLEMENTED	**Valor:** `103`
CORRUPTEDFILE	**Valor:** `104`
CONNECTIONERROR	**Valor:** `105`

C.14.3 Clase FileNotFound



Excepción lanzada cuando no se encuentra un archivo.

Métodos

__**init**__(*self*, *message*)
x.__init__(...) initializes x; see x.__class__.__doc__ for signature
Redefine: object.__init__ extit(inherited documentation)

Heredado de xmlrpclib.Fault

__repr__()

Heredado de xmlrpclib.Error

__str__()

Heredado de exceptions.Exception

__new__()

Heredado de exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(),

__setattr__(), __setstate__(), __unicode__()

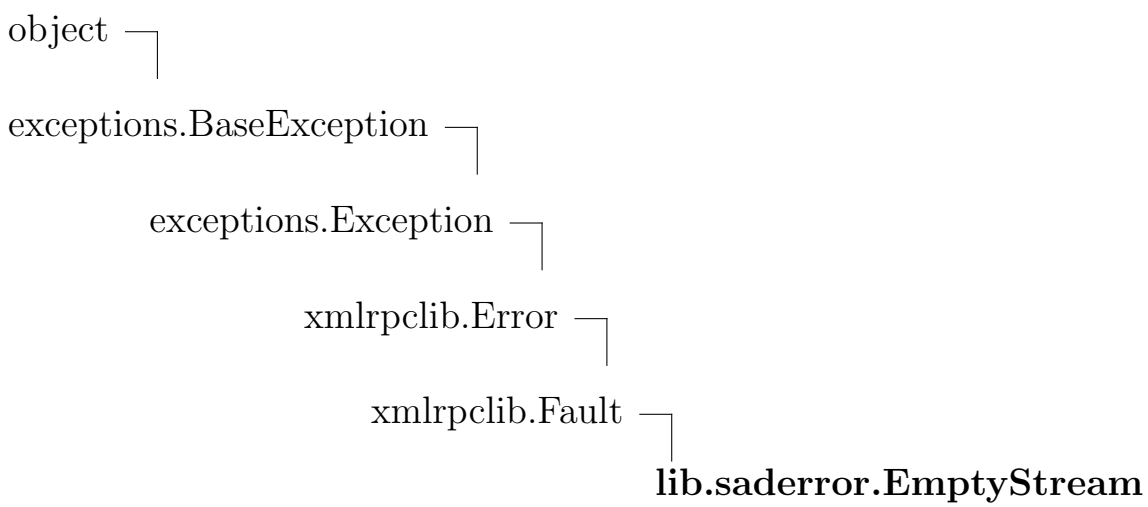
Heredado de object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Propiedades

Nombre	Descripción
Heredado de exceptions.BaseException	
args, message	
Heredado de object	
__class__	

C.14.4 Clase EmptyStream



Excepción lanzada cuando un flujo de datos está vacío.

Métodos

__**init**__(*self*, *message*)
x.__init__(...) initializes x; see x.__class__.__doc__ for signature
Redefine: object.__init__ extit(inherited documentation)

Heredado de xmlrpclib.Fault

__repr__()

Heredado de xmlrpclib.Error

__str__()

Heredado de exceptions.Exception

__new__()

Heredado de exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(),

__setattr__(), __setstate__(), __unicode__()

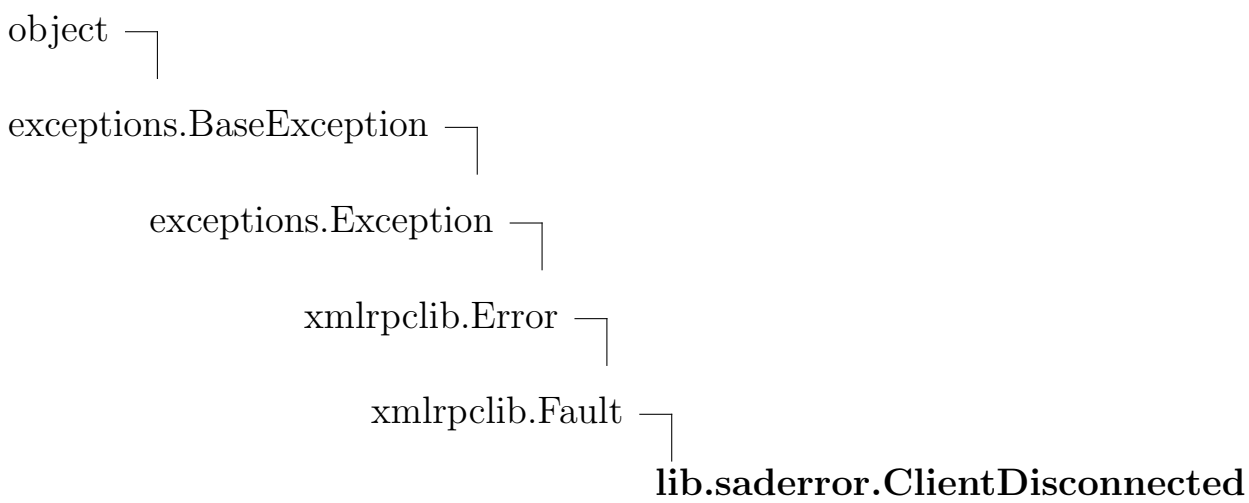
Heredado de object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Propiedades

Nombre	Descripción
Heredado de exceptions.BaseException	
args, message	
Heredado de object	
__class__	

C.14.5 Clase ClientDisconnected



Excepción lanzada cuando la conexión con la celda ha fallado.

Métodos

__**init**__(*self*, *message*)
x.__init__(...) initializes x; see x.__class__.__doc__ for signature
Redefine: object.__init__ extit(inherited documentation)

Heredado de xmlrpclib.Fault

__repr__()

Heredado de xmlrpclib.Error

__str__()

Heredado de exceptions.Exception

__new__()

Heredado de exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(),

__setattr__(), __setstate__(), __unicode__()

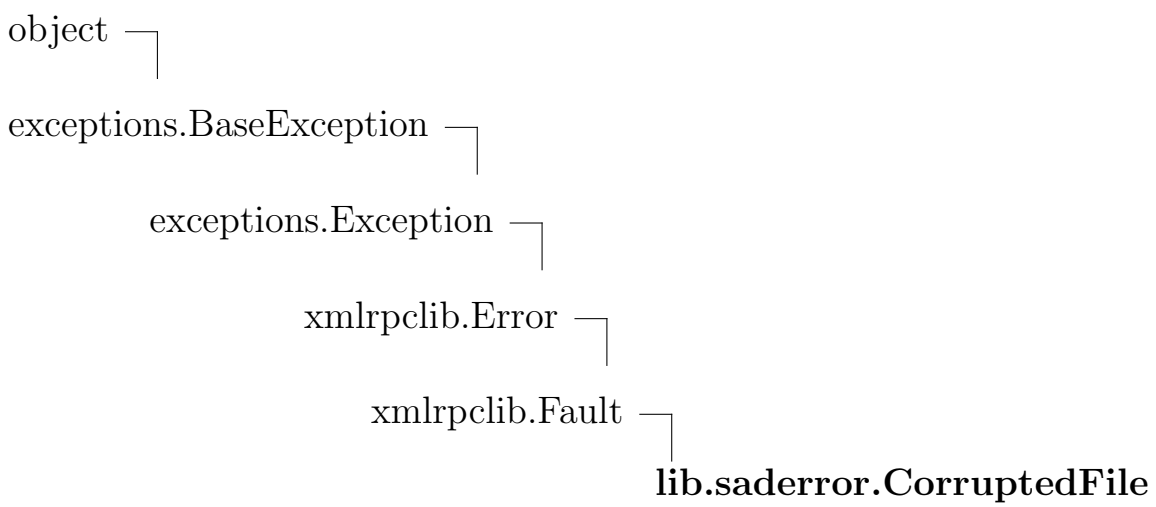
Heredado de object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Propiedades

Nombre	Descripción
Heredado de exceptions.BaseException	
args, message	
Heredado de object	
__class__	

C.14.6 Clase CorruptedFile



Excepción lanzada cuando un archivo, fragmento o bloque está corrupto.

Métodos

__**init**__(*self*, *message*)
x.__init__(...) initializes x; see x.__class__.__doc__ for signature
Redefine: object.__init__ extit(inherited documentation)

Heredado de xmlrpclib.Fault

__repr__()

Heredado de xmlrpclib.Error

__str__()

Heredado de exceptions.Exception

__new__()

Heredado de exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(),

__setattr__(), __setstate__(), __unicode__()

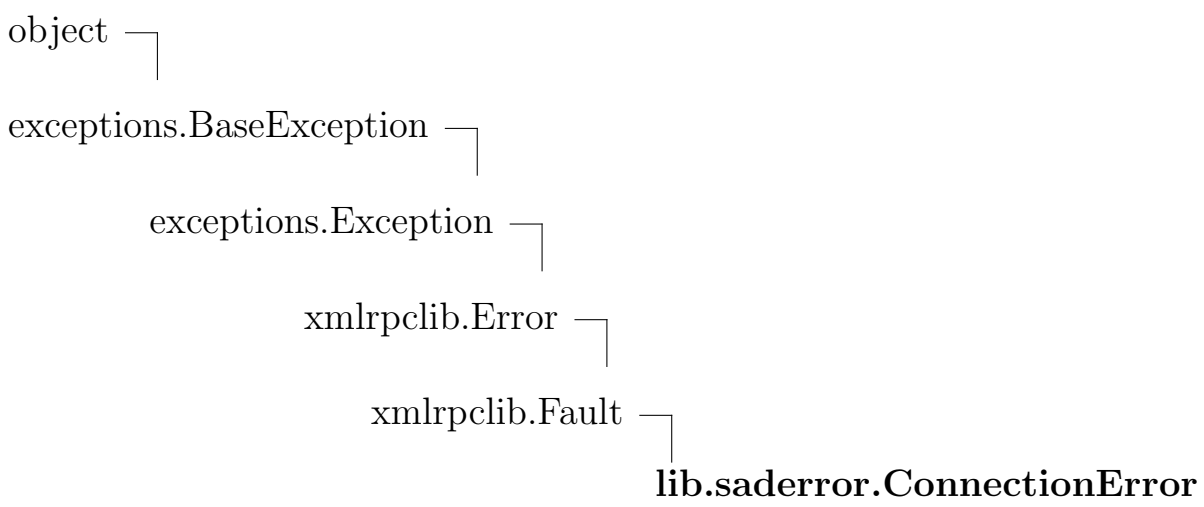
Heredado de object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Propiedades

Nombre	Descripción
Heredado de exceptions.BaseException	
args, message	
Heredado de object	
__class__	

C.14.7 Clase ConnectionError



Excepción lanzada cuando ocurre algún error con la conexión entre los EVA, el representante o el cliente

Métodos

__**init**__(*self*, *message*)
x.__init__(...) initializes x; see x.__class__.__doc__ for signature
Redefine: object.__init__ extit(inherited documentation)

Heredado de xmlrpclib.Fault

__repr__()

Heredado de xmlrpclib.Error

__str__()

Heredado de exceptions.Exception

__new__()

Heredado de exceptions.BaseException

__delattr__(), __getattribute__(), __getitem__(), __getslice__(), __reduce__(), __setattr__(), __setstate__(), __unicode__()

Heredado de object

__format__(), __hash__(), __reduce_ex__(), __sizeof__(), __subclasshook__()

Propiedades

Nombre	Descripción
Heredado de exceptions.BaseException	
args, message	
Heredado de object	
__class__	

C.15 Paquete server

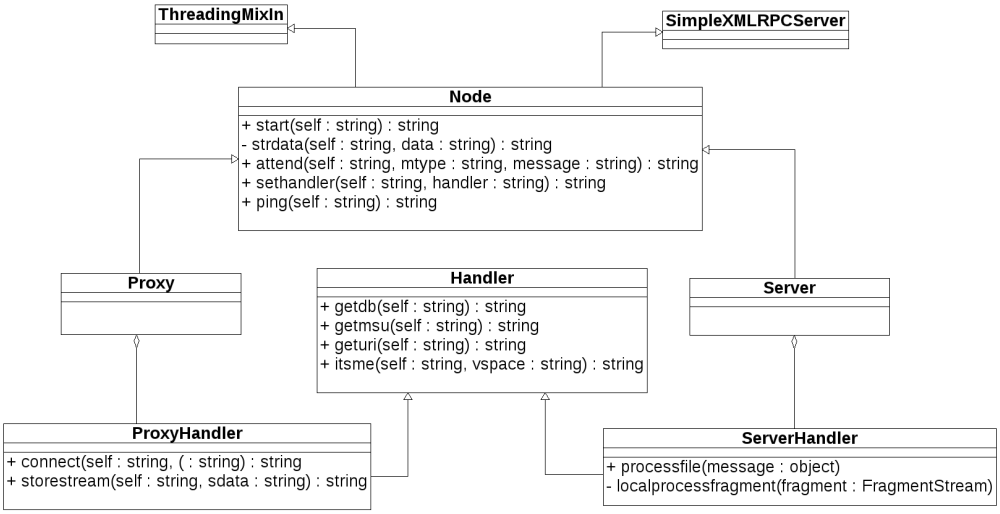


Figura C.5: Clases servidores

C.15.1 Módulos

- **common**: En éste módulo se definen la clase VirtualSpace utilizada por todos los EVA para representar a sus vecinos (ver la figura C.5).
  (Sección C.16, p. 152)
- **sadnode**: En éste módulo se definen las clases NodeListener, Handler y Node
  (Sección C.17, p. 156)
- **sadproxy**: En este módulo se definen las clases ProxyHandler y Proxy, derivadas de las clases sadnode.Handler y sadnode.Node respectivamente.
  (Sección C.18, p. 162)
- **sadserver**: En este módulo se definen las clases ServerHandler y Server, derivadas de las clases sadnode.Handler y sadnode.Node respectivamente.
  (Sección C.19, p. 165)

C.15.2 Variables

Nombre	Descripción
__package__	**Valor:** `None`

C.16 Módulo server.common

En éste módulo se definen la clase VirtualSpace utilizada por todos los EVA para representar a sus vecinos.

C.16.1 Variables

Nombre	Descripción
__package__	**Valor:** `'server'`

C.16.2 Clase VirtualSpace

Clase encargada de representar a un EVA, y permitirle a sus vecinos intractuar con él

Métodos

__init__(*self*, *vid*, *name*, *host*, *size*, *port*=4242)

Constructor de la clase.

Parámetros

`vid`: Identificador del espacio virtual de almacenamiento (EVA)

(type=Número entero.)

`name`: El nombre del EVA.

(type=Cadena de caracteres.)

`host`: Dirección IP o nombre del nodo que hospeda al EVA.

(type=Cadena de caracteres.)

`size`: Capacidad de almacenamiento del EVA.

(type=Número entero.)

`port`: Número de puerto por el que el EVA atenderá peticiones.

(type=Número entero.)

setname(*self*, *name*)

Actualiza el nombre del EVA.

Parámetros

`name`: The new virtual space name

(type=Cadena de caracteres.)

sethost(*self*, *host*)

Actualiza el nombre o dirección IP del nodo que hospeda al EVA.

Parámetros

`host`: El nuevo nombre o dirección IP.

(type=Cadena de caracteres.)

setsize(*self*, *size*)

Actualiza la capacidad de almacenamiento actual del EVA.

Parámetros

`size`: La nueva capacidad de almacenamiento del EVA.

(type=Número entero.)

setport(*self*, *port*)

Actualiza el número de puerto del EVA.

Parámetros

`port`: El nuevo número de puerto.

(type=the Número entero.)

getid(*self*)

Devuelve el identificador del EVA

getname(*self*)

Devuelve el nombre del EVA.

gethost(*self*)

Devuelve el nombre o dirección IP del nodo que hospeda al EVA.

getsize(*self*)

Devuelve la capacidad de almacenamiento del EVA.

geturi(*self*)

Devuelve la URI en que se encuentra hospedado el EVA.

update(*self*, *dbase*)

Actualiza el registro del EVA en la base de datos

Parámetros

`dbase`: Controlador de la base de datos.

(type=Para esta implementación es una instancia de la clse PGDataBase.)

__**str**__(*self*)

Devuelve la URI en que se encuentra hospedado el EVA.

__**del**__(*self*)

Destructor de la clase.

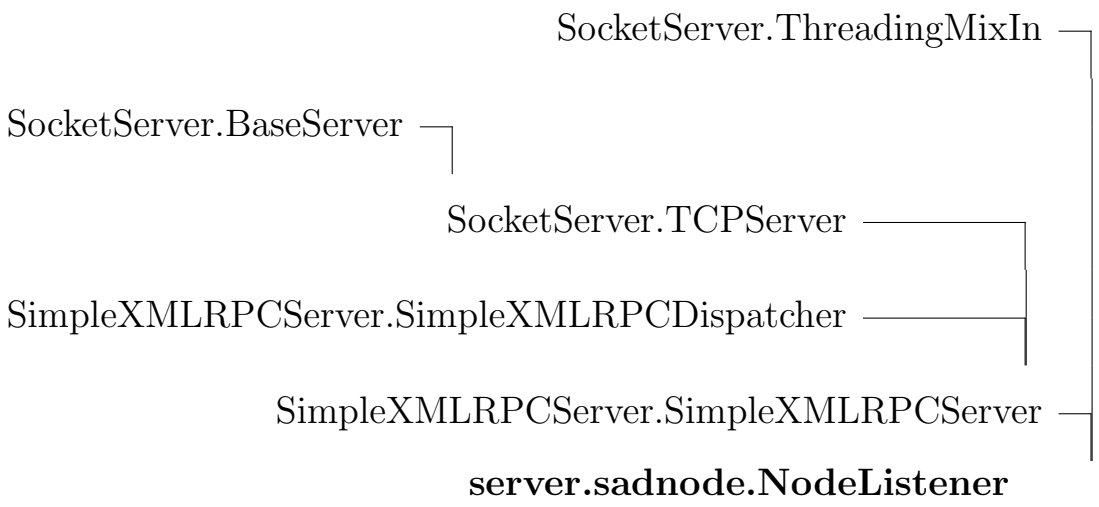
C.17 Módulo server.sadnode

En éste módulo se definen las clases NodeListener, Handler y Node

C.17.1 Variables

Nombre	Descripción
__package__	**Valor:** `'server'`

C.17.2 Clase NodeListener



Clase para manejar múltiples peticiones de manera concurrente.

Métodos

Heredado de SocketServer.ThreadingMixIn

process_request(), process_request_thread()

Heredado de SimpleXMLRPCServer.SimpleXMLRPCServer

__init__()

Heredado de SocketServer.TCPServer

close_request(), fileno(), get_request(), server_activate(), server_bind(), server_close(), shutdown_request()

Heredado de SocketServer.BaseServer

finish_request(), handle_error(), handle_request(), handle_timeout(), serve_forever(), shutdown(), verify_request()

Heredado de SimpleXMLRPCServer.SimpleXMLRPCDispatcher

register_function(), register_instance(), register_introspection_functions(),

register_multicall_functions(), system_listMethods(), system_methodHelp(), system_methodSignature(), system_multicall()

Variables de clase

Nombre	Descripción
Heredado de SocketServer.ThreadingMixIn	
daemon_threads	
Heredado de SimpleXMLRPCServer.SimpleXMLRPCServer	
allow_reuse_address	
Heredado de SocketServer.TCPServer	
address_family, request_queue_size, socket_type	
Heredado de SocketServer.BaseServer	
timeout	

C.17.3 Clase Handler

Subclases conocidas:
server.sadserver.ServerHandler, server.sadproxy.ProxyHandler

Manejador de peticiones genérico. Se encarga de leer los parámetros de configuración necesarios para el funcionamiento del nodo.

Métodos

__init__(*self*)

Constructor de la clase.

getdb(*self*)

Devuelve la una referencia al controlador de la base de datos.

getmsu(*self*)

Devuelve el tamaño de la unidad máxima de almacenamiento (UMA).

geturi(*self*)

Devuelve la URI del nodo.

itsme(*self*, *vspace*)

Verifica si el espacio virtual dado corresponde con el nodo en que reside la instancia del manejador de peticiones.

__str__(*self*)

Devuelve la URI del nodo.

C.17.4 Clase Node

Subclases conocidas:
server.sadserver.Server, server.sadproxy.Proxy

Clase generica que controla las acciones de un nodo.

Métodos

__init__(*self*)

Constructor de la clase.

start(*self*)

Inicia el servicio.

attend(*self*, *mtype*, *message*)

Método utilizado para invocar al manejador cuando se recibe una petición.

Parámetros

`mtype:` Código de petición

(type=Número entero.)

`message:` Datos de la petición

(type=Cualquier tipo de objeto)

sethandler(*self*, *handler*)

Asigna un manejador de peticiones al nodo.

Parámetros

`handler:` Manejador de peticiones.

(type=Instancia de la clase Handler o alguna de sus clases derivadas.)

ping(*self*)

Prueba si el servidor continúa funcionando.

C.18 Módulo server.sadproxy

En este módulo se definen las clases ProxyHandler y Proxy, derivadas de las clases sadnode.Handler y sadnode.Node respectivamente.

El propósito de los objetos de estas clases es dotar de funcionalidad específica al representante de la celda, para que atienda las peticiones del usuario final.

C.18.1 Variables

Nombre	Descripción
__package__	**Valor:** `'server'`

C.18.2 Clase ProxyHandler

server.sadnode.Handler
└ **server.sadproxy.ProxyHandler**

Manejador de peticiones para el representante de la celda.

Métodos

__init__(*self*)

Constructor de la clase.

Redefine: server.sadnode.Handler.__init__

connect(*self*, *args*)

Abre una conexión hacia el servidor

Parámetros

`args`: Una tupla de dos elementos que contiene el nombre del usuario y su contraseña.

(type=Tupla.)

storestream(*self*, *sdata*)

Registra un archivo en los metadatos del representante de la celda y lo envía hacia el EVA donde será procesado.

Parámetros

`sdata`: Los datos del archivo que se procesará.

(type=Diccionario.)

loadstream(*self*, *fname*)

Recupera un archivo

Parámetros

`fname`: Descriptor del archivo que se va a recuperar.

(type=Instancia de la clase SADFile.)

listfiles(*self*, *uid*)

Devuelve una lista de los archivos de un usuario dado.

Parámetros

`uid`: Identificador del usuario.

(type=Número entero.)

listservices(*self*, *uid*)

Devuelve una lista con los servicios disponibles para un usuario dado.

Parámetros

`uid`: Identificador del usuario.

(type=Número entero.)

Heredado de server.sadnode.Handler(Sección C.17.3)

__str__(), getdb(), getmsu(), geturi(), itsme()

C.18.3 Clase Proxy

server.sadnode.Node
└
server.sadproxy.Proxy

Las instancias de esta clase rigen el comportamiento del nodo representante.

Métodos

__init__(*self*)

Constructor de la clase, conecta los tipos de petición con el método del manejador de peticiones que les corresponde.

Redefine: server.sadnode.Node.__init__

Heredado de server.sadnode.Node(Sección C.17.4)

attend(), ping(), sethandler(), start()

C.19 Módulo server.sadserver

En este módulo se definen las clases ServerHandler y Server, derivadas de las clases sadnode.Handler y sadnode.Node respectivamente.

El propósito de los objetos de estas clases es dotar de funcionalidad específica a los espacios virtuales de almacenamiento, para que atienda las peticiones de sus vecinos y el representante de la celda.

C.19.1 Variables

Nombre	Descripción
__package__	**Valor:** `'server'`

C.19.2 Clase ServerHandler

server.sadnode.Handler

server.sadserver.ServerHandler

Las instancias de esta clase se encargan de manejar las peticiones que llegan al EVA.

Métodos

__init__(*self*)

Constructor de la clase

Redefine: server.sadnode.Handler.__init__

processfile(*self*, *args*)

Procesa el flujo de un archivo en fragmentos que no exceden el tamaño defindo de la UMA, y los reparte a otros EVA para que sean codificados.

Parámetros

`args`: Una tupla de dos elementos que contiene una cadena de caracteres para verificar que el archivo no se haya corrompido y la información del archivo.

(type=Tupla)

Raises

`CorruptedFile` Cuando se detecta que el archivo está corrupto.

processfragment(*self*, *args*)

Codifica un fragmento.

Parámetros

`args`: Una tupla de dos elementos que contiene una cadena de caracteres para verificar que los datos no se haya corrompido y la información del fragmento.

(type=Tupla)

Raises

`CorruptedFile` En caso de que detecte que el fragmento está corrupto.

processblock(*self*, *args*)

Procesa un bloque.

Parámetros

`args`: Una tupla de dos elementos que contiene una cadena de caracteres para verificar que los datos no se haya corrompido y la información del bloque.

(type=Tupla)

Raises

`CorruptedFile` En caso de que detecte que el bloque está corrupto.

Heredado de server.sadnode.Handler(Sección C.17.3)

__str__(), getdb(), getmsu(), geturi(), itsme()

C.19.3 Clase Server

server.sadnode.Node
└ **server.sadserver.Server**

Clase que controla el comportamiento de un EVA.

Métodos

__init__(*self*)

Constructor de la clase, conecta los tipos de petición con el método del manejador de peticiones que les corresponde.

Redefine: server.sadnode.Node.__init__

Heredado de server.sadnode.Node(Sección C.17.4)

attend(), ping(), sethandler(), start()

C.20 Paquete ui

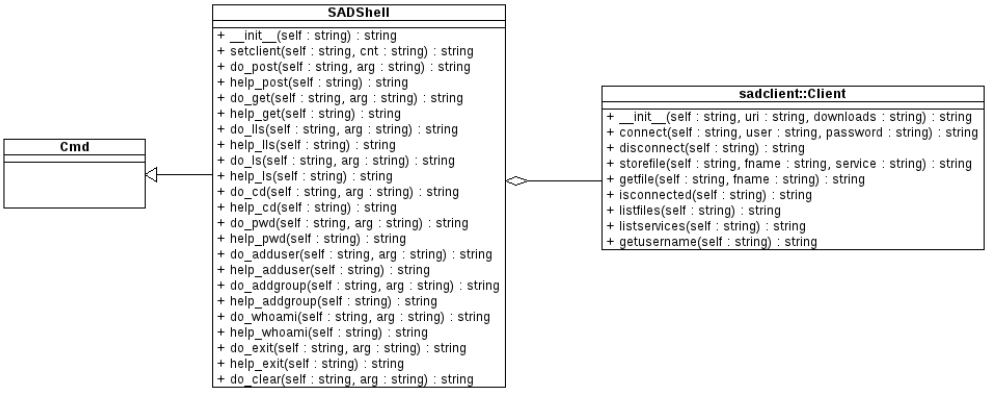


Figura C.6: Interfaz de usuario

C.20.1 Módulos

- **shellui**: En éste módulo se define la interfaz con el usuario final, en este caso, está implementada a travéz de un cliente de basado en línea de comandos.
  (Sección C.21, p. 169)

C.20.2 Variables

Nombre	Descripción
__package__	**Valor:** `None`

C.21 Módulo ui.shellui

En éste módulo se define la interfaz con el usuario final, en este caso, está implementada a travéz de un cliente de basado en línea de comandos.

C.21.1 Variables

Nombre	Descripción
__package__	**Valor:** `'ui'`

C.21.2 Clase SADShell

cmd.Cmd
└ **ui.shellui.SADShell**

Intérprete de comandos para conectarse a la celda de almacenamiento distribuido.

Métodos

__init__(*self*)

Constructor de la clase.

Redefine: cmd.Cmd.__init__

setclient(*self*, *cnt*)

Conecta un cliente a la interfaz de línea de comandos.

Parámetros

`cnt`: Cliente

(type=Instancia de la clase client.sadclient.Client)

do_post(*self*, *arg*)

Definición del comando post

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

help_post(*self*)

Ayuda en línea para el comando post.

do_get(*self*, *arg*)

Definición del comando get

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

help_get(*self*)

Ayuda en línea para el comando get.

do_lls(*self*, *arg*)

Definición del comando lls

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

help_lls(*self*)

Ayuda en línea para el comando lls.

do_ls(*self*, *arg*)

Definición del comando ls

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

help_ls(*self*)

Ayuda en línea para el comando ls.

do_cd(*self*, *arg*)

Definición del comando cd

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

help_cd(*self*)

Ayuda en línea para el comando cd.

do_pwd(*self*, *arg*)

Definición del comando pwd

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

help_pwd(*self*)

Ayuda en línea para el comando pwd.

do_adduser(*self*, *arg*)

Definición del comando adduser

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

help_adduser(*self*)

Ayuda en línea para el comando adduseer.

do_addgroup(*self*, *arg*)

Definición del comando addgroup

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

help_addgroup(*self*)

Ayuda en línea para el comando addgroup.

do_whoami(*self*, *arg*)

Definición del comando whoami

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

help_whoami(*self*)

Ayuda en línea para el comando whoami.

do_exit(*self*, *arg*)

Definición del comando exit

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

help_exit(*self*)

Ayuda en línea para el comando exit.

do_EOF(*self*, *arg*)

Definición del comando exit

Parámetros

`arg`: Argumentos del comando

(type=Cadena de caracteres.)

do_clear(*self*, *arg*)

Limpia la pantalla

Heredado de cmd.Cmd

cmdloop(), columnize(), complete(), complete_help(), completedefault(), completenames(), default(), do_help(), emptyline(), get_names(), onecmd(), parseline(), postcmd(), postloop(), precmd(), preloop(), print_topics()

Variables de clase

Nombre	Descripción
prompt	**Valor:** `'celda:> '`
Heredado de cmd.Cmd	

continúa en la siguiente página

Nombre	Descripción
doc_header, doc_leader, identchars, intro, lastcmd, misc_header, nohelp, ruler, undoc_header, use_rawinput	

Índice alfabético

Referencias

[AA02] Adya A., et. al. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Symp. On Operating Systems Design and Implementation (OSDI)*, pages 1–14, Boston, USA, December 2002. revisar los datos.

[AP01] Rowstron A. and Druschel P. Storage management and caching in past, a large-scale, persistent, peer-to-peer storage utility. *Operating Systems Review*, 35(5):188–201, 2001.

[Bar03] A. Barabási. *Linked: How Everything is Connected to Everything Else and What it Means for Business, Science and Everyday Life*. Plume, 2003.

[BKK+03] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Commun. ACM*, 46:43–48, February 2003.

[BR01] Carlsson B. and Gustavsson R. The rise and fall of napster – an evolutionary approach. In *Proceeding of the 6th International Conference on Active Media Technology*, pages 347, 354. LNCS- 2252, 2001.

[Cha06] Chang, F., et. al. Bigtable: A distributed storage system for structured data. In *Symp. On Operating Systems Design and Implementation (OSDI)*, 2006.

[Che99] Chen, Y., et. al. A prototype implementation of archival intermemory. In *4th ACM Conference on Digital Libraries*, pages 28–37, California, USA, 1999.

[DeC07] DeCandia, G., et. al. Dynamo: Amazon's highly available key-value storage. In *SOSP'07*, pages 205–220, Washington, USA, October 2007.

[Des91] Y. Deswarte. Tolérance aux fautes, sécurité et protection. In et. al. Balter, R., editor, *Construction des Systèmes d'exploitation Répartis*. INRIA, Paris, France, 1991.

[Gra03] J. Gray. What's next? a dozen information-technology research goals. *JACM*, 40:41–57, 2003.

[Gro07] P2P Research Group. Peer-to-peer research group: Charter. Technical report, July 2007. http://www.irtf.org.

[Har02] Harren, J., et. al. Complex queries in dht-based peer-to-peer networks. In *First International Workshop on Peer-to-Peer Systems*, Cambridge, USA, 2002 2002. submitted to Web Data Management, Winter 2005.

[Has05] Hasan, R., et. al. A survey of peer-to-peer storage techniques for distributed file systems. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, volume 2, pages 205–213, 2005.

[Kub00] Kubiatowicz, J., et. al. Oceanstore: An architecture for global-scale persistent storage. In *Architectural Support for Programming Languages and Operating Systems*, pages 190–201, New York, USA, 2000.

[Laz09] C.A.G. Lazalde. Evaluación de estrategias de redundancia de información en sistemas de almacenamiento p2p. Master's thesis, Depto. De Ing. Eléctrica, UAM-Iztapalapa, 2009.

[McC04] McClinton, D., et. al. 10 emerging technologies that will change your world. *Technology Review*, pages 31–50, 2004.

[Mrr06] Merrick, et. al. XML Remote Procedure Call. United States Patent No.: US 7,028,312 B1. April 11th. 2006

[Pla06] Placek, M., et. al. A taxonomy of distributed storage systems. Technical report, University of Melbourne, Grid Computing and Distributed Systems Lab., July 2006.

[Que10] Quezada, N.M., et. al. Fault-tolerance and load-balance tradeoff in a distributed storage system. *Computación y Sistemas*, 2010. accepted.

[Rab89] Michel O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal ofthe Association for Computing Machinery*, 36(2):335–348, April 1989.

[Rhe01] Rhea, S., et. al. Maintenance-free global data storage. *IEEE Internet Computing*, pages 40–49, September-October 2001.

[Rip02] Ripeanu, M., et. al. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing*, 6(1):50–57, February 2002.

[R.J05] Chevance R.J. *Server Architectures: Multiprocessors, Clusters, Parallel Systems, and Storage Solutions*. Digital Press, 2005.

[RL05] Rodrigues R. and LiskovB. High availability in dht's: Erasure coding vs replication. In *International Workshop on Peer-to-Peer Systems*, LNCS, pages 226–239, 2005. faltan datos.

[Rou03] Roussopoulos, M., et. al. P2P or Not 2 P2P? Technical report, July 2003. Arxiv preprint cs.NI/0311017.

[RT04] J. Risson and Moors T. Survey of research towards robust peer-to-peer networks: Search methods. Technical report, University of South Wales, Sidney, Australia, September 2004.

[San03] Sanjay, Ghemawat., et. al. The google file system. *SOSP'03 ACM*, October 2003.

[SD04] Androutsellis-Theotokis S. and Spinellis D. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.

[Som06] I. Sommerville. *Software Engineering*. 8th. edition, 2006.

[Sun05] Sung, A., et. al. A survey of data management in peer-to-peer systems. Waterloo, Canada, April 2005. submitted to Web Data Management, Winter 2005.

[Tre01] Trencseni, Marton., et. al. Keyspace: A consistently replicated highly-available key-value store. URL: *http://final-project-hum.googlecode.com/svn-history/r24/trunk/refer/references/Keyspace.pdf* Última revisión: 23 de agosto de 2011.

[Wat99] D.J. Watts. *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton University Press, 1999.

[Wei07] S. Weil. *CEPH: Reliable, Scalable, and High-Performance Distributed Storage*. PhD thesis, UC Santa Cruz, 2007.

[Yia01] Peter N. Yianilos. The evolving field of distributed storage. *IEEE Internet Computing*, pages 35–39, 2001.

[Bass03] Bass Len, et. al. *Software Architecture in Practice* SEI Series in Software Engineering 2nd. Edition Addison Wessly, Pearson Education

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

Universidad Autónoma Metropolitana – Iztapalapa
División de Ciencias Básicas e Ingeniería

CONSTRUCCIÓN DE UN SISTEMA DE ALMACENAMIENTO DISTRIBUIDO BASADO EN REQUERIMIENTOS

Idónea Comunicación de Resultados
que presenta el
Lic. Diego Rodrigo Guzmán Santamaría
para obtener el grado de
Maestro en Ciencias

Mis mejores deseos

Felicidades

11/11/2011

Asesores: Dra. Reyna Carolina Medina Ramírez — Dr Ricardo Marcelín Jiménez

Jurado Calificador:

Felicidades

Presidente:	Dr. Víctor Jesús Sosa Sosa	CINVESTAV - Tamaulipas
Secretario:	Ing. Luis Fernando Castro Careaga	UAM - Iztapalapa
Vocal:	Dra. Reyna Carolina Medina Ramírez	UAM - Iztapalapa

México D.F., noviembre de 2011