IDNet

Framework de conexión P2P de bases de datos distribuidas e independientes

Lorenzo José de la Paz Suárez
Juan Mas Aguilar

Director: José Luis Vázquez Poletti

Trabajo Fin de Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Curso 2017/2018

Autorización de difusión

Los abajo firmantes, matriculados en el Grado de Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores el presente Trabajo Fin de Grado: "IDNet: Framework de conexión P2P de bases de datos distribuidas e independientes", realizado durante el curso académico 2017-2018 bajo la dirección de José Luis Vázquez-Poletti en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Juan Mas Aguilar
Lorenzo José de la Paz Suárez

Madrid, 28 de Mayo de 2018

A José Luis Vázquez Poletti por su incansable dedicación y esfuerzo. Sin sus respuestas a nuestros correos a las dos de la madrugada y la cantidad de ideas y motivación que nos ha aportado no hubiésemos sacado adelante el proyecto.

Dedicatoria

A nuestras familias, por estar ahí en todo momento y animarnos incluso cuando estábamos frustrados y no entendían por qué ni cómo resolverlo.

A nuestros compañeros, porque esos momentos en la cafetería le levantan el ánimo a cualquiera. Gente que comparte lo mismo que tú y siempre te apoya.

Índice general

Índice de figuras

Resumen

"IDNet" es un framework consistente en la creación de una red de conexión Peer-to-Peer (P2P) para conectar bases de datos independientes y distribuidas, es decir, sin necesidad de discriminar la estructura interna de la misma y sin estar centralizadas las bases de datos.

El prototipo está diseñado con el objetivo de soportar la creación de Organizaciones Virtuales, en las cuáles los usuarios pueden obtener información de las bases de datos de sus vecinos, pudiendo realizar consultas a la base de datos vecina. La información sensible transmitida entre los vecinos se encuentra securizada mediante una capa de seguridad inherente al framework.

La red P2P está implementada mediante contenedores Docker en Cloud mediante el proveedor Amazon Web Services, aportándonos elasticidad y capacidad de respuesta a la misma.

Figura 1: *Logo de la aplicación[7]*

Palabras clave

Bases de datos, P2P, Organizaciones Virtuales, Cloud, Docker

Abstract

"IDNet" is a net-based framework consisting on the creation of Peer-to-Peer (P2P) networks to connect independent and distributed databases. In other words, without the need of differentiate the internal structure and without being logically or geographically centralized.

The prototype is designed to create Virtual Organizations, in which users would be able to obtain data from their neighbours' databases. By this means, users would be able to make queries between the databases previously referenced. Sensitive data will pass through a variety of security protocols built in the core of the application to avoid its content to be revealed.

The network is built onto a Cloud structure. Although we chose Amazon Web Services as our Cloud Service Provider, the vendor lock-in is minimal to let the user choose his provider freely. Cloud computing provides us with an outstanding flexibility. We use Docker containers to isolate the node and make its portability easier for everyone.

Figura 2: *Application Logo[7]*

Keywords

Databases, Peer-to-Peer, Virtual Organizations, Cloud, Docker

Capítulo 1

Introducción

Con el avance de las comunicaciones nos presentamos ante una nueva sociedad *"hiperconectada"*. Las empresas y administraciones públicas hacen un uso masivo de Internet y las tendencias de mercado implican un volcado casi absoluto sobre el mismo. Servicios online, alojamiento, ventas... Todo se desarrolla ahora de cara a la red y explota sus capacidades y bondades.

No obstante, existe una excepción a esta regla. Las infraestructuras críticas no son capaces de aprovechar esta inmensidad de recursos disponibles. Esto es debido a la sensibilidad de su información o al temor de que alguien se infiltre en sus sistemas de vital importancia. Muchas de ellas se aislan completamente de Internet con el objetivo de securizarse aun más. Pero esto es, en nuestra opinión, un desperdicio. Los datos que se generan podrían ser computados a mayores velocidades y con mayor eficiencia utilizando muchos de los componentes de la red. Incluso el mero hecho de que dos centros puedan intercambiar datos ya sería una gran avance.

Con esa idea hemos creado IDNet. Un Framework para que dos bases de datos puedan conectarse de forma autónoma e intercambiar información de forma independiente, distribuida y, sobretodo, segura. Lo consideramos como un primer paso para tirar esos muros sin comprometer la seguridad de los datos.

Capítulo 2

Motivación

Actualmente nos encontramos limitados. El "aislamiento" de las bases de datos es un hecho por no hablar de las incompatibilidades entre bases de datos con misma estructura pero distinto desarrollador (ej. Oracle, MySQL, MariaDB, etc). Nuestro trabajo pretende marcar un precedente y abrir un nuevo canal de comunicación y transmisión de la información.

No es raro encontrarse con que existen varias bases de datos individuales en distintos sectores de una entidad (empresa, administración, etc) pero llegado un momento, ante la necesidad de trabajar con todas ellas juntas, se necesita crear de nuevo una macrobase de datos que aúne las anteriores. La solución más sencilla para este tipo de casos, sería mantener el mismo tipo de base de datos (tanto estructura interna como desarrollador), de forma que las tablas y las conexiones puedan manejarse con facilidad. No obstante, esta aproximación al problema no permite a una entidad adecuar sus sistemas de almacenamiento a las características de sus distintos perfiles (volumen de datos, características de los mismos, nivel de seguridad, temporalidad).

Si hablamos de estructuras críticas, el tema se vuelve más espinoso. La seguridad de la que deben disponer es su factor más limitante. El sistema más seguro es aquel que, de forma total y absoluta no tenga acceso a la Red. Esto está basado en un hecho reconocido abiertamente: Internet no fue hecho para ser seguro.

Ha sido con el paso del tiempo que se ha demostrado la necesidad de securizar los protocolos ya establecidos. Esto, que puede servir para mejorar la seguridad de los datos de un navegante común, es del todo insuficiente para entidades que traten datos sensibles de cualquier índole. Es por ello que no pueden arriesgarse a usar plataformas que ampliarían enormemente sus capacidades y acelerarían sus procesos. Hablamos, por ejemplo, de un hospital que pudiera acceder a las investigaciones y expedientes de los centros de investigación de referencia con una sola petición. Empresas que puedan usar indistintamente bases de datos SQL y NoSQL en función de los datos que necesitan usar en sus distintos ámbitos sin miedo a aislarlos. Comisarías de Policía que pudieran usar bases de datos independientes y consultarlas como si fueran una. Y todo ello de forma distribuida geográficamente y segura.

Con este proyecto queremos iniciar un primer acercamiento para poder abordar este problema tan complicado. Todo ello aprovechando el impulso de las nuevas tendencias de computación a gran escala como el Cloud. Este último, escogido por su elasticidad y robustez.

Capítulo 3

Estado del arte

A través de este capítulo se va a presentar el estudio inicial del mercado realizado para encontrar qué aplicaciones presentan similitud con nuestro proyecto. Todo ello con el fin de encontrar las ventajas y desventajas que subyacen en ellas, y para poder obtener aquellas características que nos puedan ayudar en la consecución de nuestro proyecto.

IDNet busca sacar provecho de tales proyectos en los ámbitos en los que se encuentra: bases de datos SQL y NoSQL, arquitecturas Peer-to-Peer, Organizaciones Virtuales, Cloud y Docker. IDNet busca gestionar bases de datos, tanto SQL como NoSQL, con el fin de compartir información sobre ellas a través de la red. También, busca formar una red anónima en la que los nodos pertenecientes a ella solo conozcan los nodos adyacentes a sí mismo. Con esta asociación de vecinos, se busca que se creen Organizaciones Virtuales en las que los usuarios tengan una característica en común, como puede ser el ámbito estatal en el que se encuentren, por ejemplo una red de hospitales. Y, por último, se busca el uso de las tecnologías puntas existentes en el mundo actual, más concretamente, el uso de contenedores Docker y de su despliegue en Cloud.

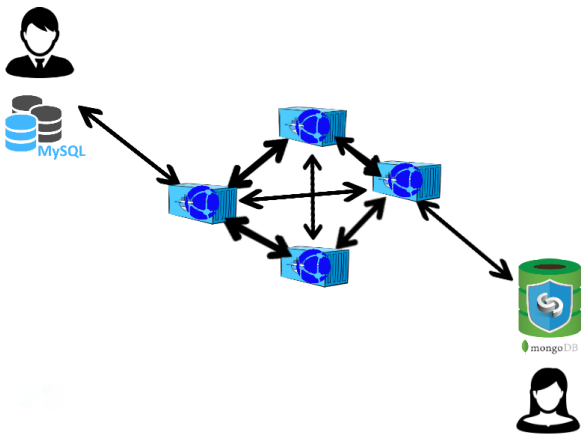


Figura 3.1: *Diagrama de la aplicación*

Tras analizar varias aplicaciones llegamos a la conclusión de que no había ninguna aplicación que englobara las funcionalidades previstas. Únicamente pudimos obtener ideas relacionadas de cada uno de los ámbitos mencionados.

3.1. ToroDB

ToroDB es una tecnología diseñada para la relación de bases de datos SQL con bases de datos NoSQL orientadas a documentos, como por ejemplo MongoDB. Internamente, replica documentos en MongoDB en PostgreSQL, una base de datos relacional. Esta tecnología se asemeja mucho a nuestra idea de gestionar tanto bases de datos SQL como NoSQL, pero nos limita la capacidad de usar otras bases de datos SQL como MySQL.

Más información en la siguiente URL: https://www.torodb.com/

3.2. FreeNet

FreeNet es una plataforma Peer-to-Peer diseñada para una comunicación resistente a la censura, con el fin de que los usuarios puedan compartir información de forma anónima. En esta aplicación, la información se encuentra descentralizada a través de los nodos que la componen. Sin embargo, FreeNet está centrada en la compartición de información de cualquier índole, mientras que nosotros buscamos compartir información relacionada con datos sensibles pertenecientes a bases de datos. Más información en la siguiente URL: https://freenetproject.org/

3.3. Alliance P2P

Alliance P2P es un software que permite crear redes P2P privadas y con acceso controlado (necesitas permiso para entrar a la red). Mayoritariamente pensada y usada para la compartición de archivos en Internet. Este software interpreta muy bien nuestra idea de crear Organizaciones Virtuales (redes P2P privadas) pero no se centran en la compartición segura de información sensible ni tienen compatibilidad con bases de datos. El software se puede descargar de sourceforge en la siguiente URL: http://alliancep2p.sourceforge.net/

Capítulo 4

Descripción general

IDNet es, a priori, un modelo de red que permite conectar bases de datos e intercambiar querys de forma anónima y segura. Sus principales bazas son el uso de criptografía simétrica y asimétrica así como la implementación de protocolos propios de conexión y enrutado. La red está organizada con una topología P2P que permite el anonimato y aumenta la seguridad de las comunicaciones. Del mismo modo, está implementada para favorecer el uso de *"Organizaciones Virtuales"*, un concepto que permitirá a IDNet establecer ámbitos privados. Todo ello soportado sobre una estructura en cloud.

4.1. Ventajas

IDNet conlleva una serie de ventajas:

- **Distribución Geográfica**. IDNet permite que dos bases de datos se puedan comunicar independientemente de dónde se sitúen físicamente. El uso de AWS favorece está "independencia" geográfica.
- **Arquitectura de la Base de Datos**. A través de IDNet, dos bases de datos pueden comunicarse independientemente de la arquitectura interna que usen o de la representación de los propios datos.
- **Seguridad de las comunicaciones**. Los datos intercambiados pueden ser de un nivel de sensibilidad extremadamente alto. Por ello, IDNet se vale de técnicas de cifrado simétrico y asimétrico para asegurar la privacidad de los datos y la legitimidad de las comunicaciones.
- **Anonimato de los nodos**. Los nodos Cliente no son conscientes nunca de las direcciones IP del resto de nodos, esto se asegura con un sistema eficaz de alias.

4.2. Características técnicas

IDNet está compuesta por distintos tipos de nodos. Cada uno de ellos realiza una serie de funciones que permiten la correcta comunicación entre ellos. Cabe destacar las siguientes funcionalidades de cada nodo:

4.2.1. Nodo Cliente

- Procesamiento de las querys.
- Conexión a las distintas bases de datos.
- Conexión al GateKeeper y obtención de los nodos vecinos con los que se puede comunicar.
- Autenticación del usuario que usa el software.
- Interfaz gráfica para facilitar el uso.
- Creación del demonio e implementación de los protocolos diseñados.

4.2.2. Nodo GateKeeper

- Propagación de los mensajes del cliente.
- Anonimización de los mensajes del cliente.
- Enrutado de los mensajes que viajan por la red.
- Identificación de las conexiones nuevas.
- Proveer al nodo Cliente de los vecinos disponibles.

Capítulo 5

Arquitectura

Dentro de la estructura interna de IDNet podemos distinguir varios ámbitos sobre los que se van a realizar actuaciones. A gran escala podemos definir un ámbito en cliente y otro en Cloud. Pasamos a explicarlos más en detalle.

5.1. Arquitectura interna del Cliente

IDNet está planteado como una red anónima P2P, por lo que cada nodo instalado en cliente cumple ambos roles, tanto el de servidor como el de cliente. Por ello, y dada la necesidad de que el servidor esté escuchando "continuamente" las peticiones del resto de nodos, se ha dividido el nodo en dos componentes. El cliente, que corre como una aplicación y a través del cual se puede interactuar mediante una interfaz de usuario; y el servidor, el cuál no puede ser accedido por el usuario y se ejecuta como un demonio en *background* para responder las peticiones de los demás.
La nomenclatura usada para nombrar a estos dos componentes es la siguiente:

`IDNetSoftware: cliente` `IDNetDaemon: servidor`

En la figura 5.1 se puede ver el esquema general de la interacción entre dos nodos clientes mostrando explícitamente la función de cada componente mencionado.

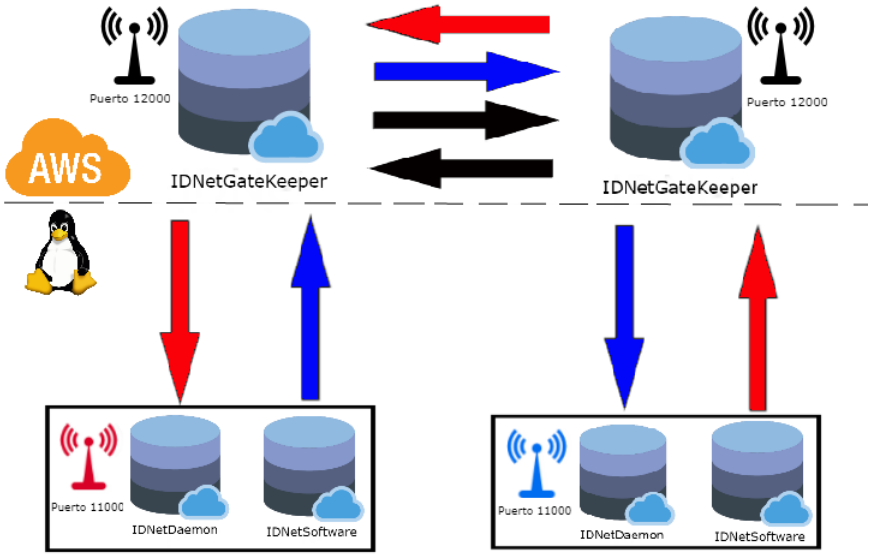


Figura 5.1: *Esquema general de la interacción entre dos nodos Cliente*

Las flechas de color azul y rojo corresponden con dos interacciones distintas entre dos nodos Cliente. Por otro lado, las flechas negras corresponden con la información transmitida entre los nodos GateKeeper.
Se puede comprobar que ambos nodos Cliente tienen a su demonio corriendo en el puerto 11000. Los GateKeeper escuchan las peticiones de los nodos Cliente por el puerto 11000 y las peticiones de otros nodos GateKeeper, por el puerto 12000.

Gestión de las bases de datos

Antes de comenzar a explicar en detalle cada componente en la arquitectura interna del cliente, cabe mencionar que ambos componentes manejan información concerniente a las bases de datos añadidas en la aplicación. La información gestionada es la siguiente:

- **Nombre de la base de datos**.
- **Tipo de la base de datos**: MySQL o MongoDB. La elección de estos dos tipos de bases de datos se debe a que cada uno de ellos representa al grupo de bases de datos al que pertenece. MySQL para las bases de datos relacionales y MongoDB para las bases de datos no relacionales.[6]
- **Usuario**: este campo es opcional, para aquellos casos en los que no se requiera un usuario y contraseña para acceder a la base de datos. Por defecto, en una instalación de MongoDB nos encontramos con este caso.
- **Contraseña**: de forma similar al campo anterior es opcional.

Desde IDNetSoftware se gestiona la información de acceso a las bases de datos, mientras que desde IDNetDaemon nos conectamos a tales bases de datos con el fin de obtener información de ellas.

5.1.1. IDNetSoftware

IDNetSoftware es la aplicación que se encargar de comunicarse con los vecinos para solicitar datos.

Siguiendo el esquema general de interacción entre nodos Cliente de la figura 5.1, es el encargado de comenzar la comunicación con otro nodo Cliente. Y, como se comentará en la sección 5.1.2, se comunica con el IDNetDaemon del nodo receptor Cliente.

Las funciones que realiza IDNetSoftware son las siguientes:

- **Inicio de sesión en la aplicación**. Mediante la comunicación con la base de datos IDNet (sección 5.2.3) inicia sesión el usuario. Tras iniciar sesión, se obtendrán los archivos de configuración *info.conf* y *neighbours.conf*.
- **Establecimiento de la conexión con el vecino**. Interactúa con el IDNetDaemon del nodo receptor. En la sección 5.3.1 se explican las estructuras de los mensajes enviados para el establecimiento de la conexión.
- **Solicitud del esquema de una base de datos de un vecino**. En la sección 5.3.2 se explica la estructura del mensaje enviado.
- **Realización de una consulta a una base de datos de un vecino**. En la sección 5.3.3 se explica la estructura del mensaje enviado.
- **Autoriza a los vecinos la consulta de una base de datos propia**. Para ello, añade una base de datos existente en la aplicación. Tal base de datos debe de existir en el sistema y debe de estar activado el servidor propio de la base de datos. Para mayor información véase el caso de uso 7.1, en el cuál se ven todas las posibilidades existentes para que se dé tal autorización.

Para la realización de las funciones mencionadas, se ha realizado una estructura del código fuente que se asemeja al siguiente diagrama.

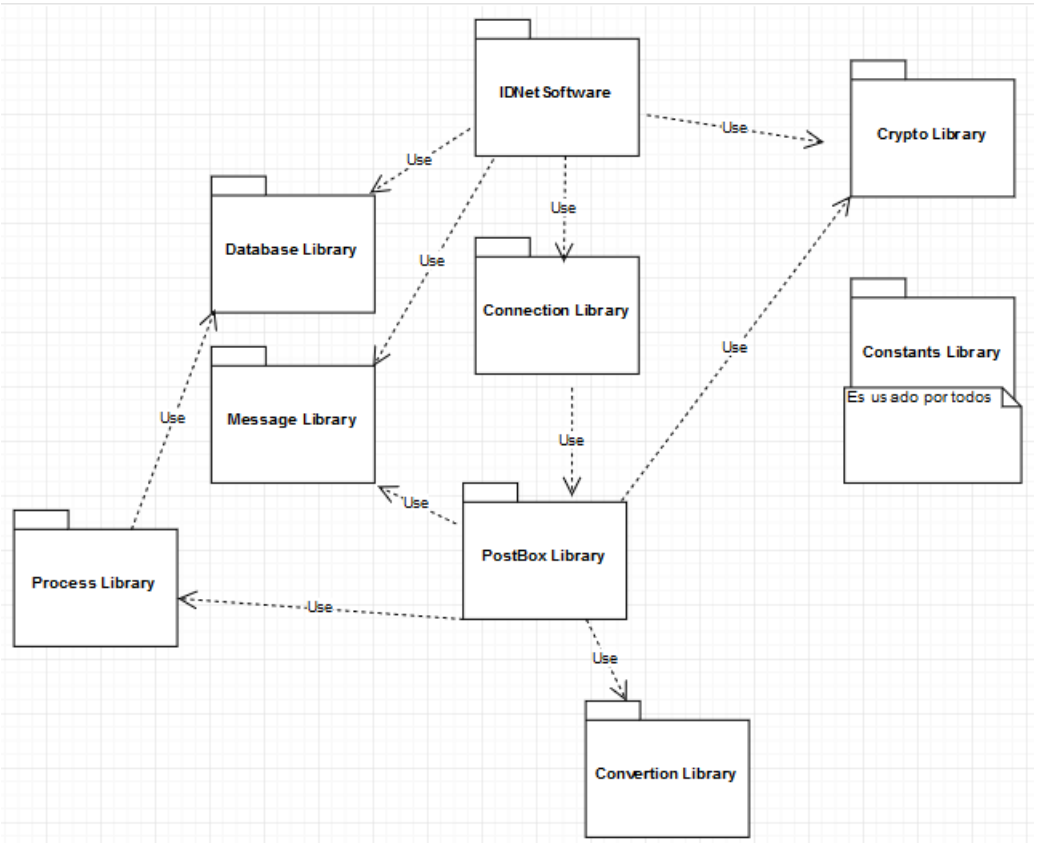


Figura 5.2: *Diagrama de módulos IDNetSoftware*

Como se puede ver en la figura 5.2, se ha modularizado IDNetSoftware de la siguiente forma:

- **IDNetSoftware**: contiene la clase principal de la aplicación. Además, contiene todas aquellas clases que son ventanas y diálogos de la aplicación. Un ejemplo de estas clases mencionadas son: MainWindow, AddDatabaseDialog, UsuariosOVDialog...
- **Connection Library**: contiene la clase *Client*, encargada de realizar las conexiones de cliente hacia los servidores de los vecinos.
- **Message Library**: módulo responsable de la creación de mensajes y del parseo de los mensajes recibidos. La clase exclusiva del módulo es *Message*.
- **Crypto Library**: la clase principal del módulo, *Crypto*, es la encargada de la criptografía de la aplicación.
- **PostBox Library**: es la encargada de actuar como si de un buzón se tratara. Se encarga de tratar y procesar los mensajes, por ello utiliza los módulos Message Library y Crypto Library.

- **Database Library**: módulo encargado de todo lo concerniente a las bases de datos. En primer lugar, gestiona la información relativa a las bases de datos propias del nodo Cliente. Por otra parte, maneja la información relativa a las bases de datos de los vecinos. Por último, se encarga de la comunicación con la base de datos remota de IDNet.
- **Process Library**: módulo que actúa de intermediario entre el módulo PostBox Library y Database Library.
- **Convertion Library**: módulo encargado de la conversión de tipos en la aplicación. Simplifica el procesamiento y libera carga de trabajo en los demás módulos.
- **Constants Library**: su clase principal, *Constants*, contiene todas las variables y métodos constantes relativos a la aplicación.

Ficheros de soporte

IDNetSoftware se vale de cuatro ficheros fundamentales de configuración para su desempeño.

- ***info.conf***: fichero encargado de obtener la información sobre el usuario. Se obtiene tras iniciar sesión en la aplicación. Estructura del archivo:

```
nombre=<nombre de usuario>|code:<código>;
```

Figura 5.3: *Estructura archivo info.conf*

- ***databases.conf***: fichero encargado de almacenar la información concerniente a las bases de datos propias autorizadas desde la aplicación. Estructura del archivo:

```
<tipo BBDD>*<nombre BBDD>[ |<usuario BBDD>*<contraseña encriptada> ];
<tipo BBDD>*<nombre BBDD>[ |<usuario BBDD>*<contraseña encriptada> ];
<tipo BBDD>*<nombre BBDD>[ |<usuario BBDD>*<contraseña encriptada> ];
...
```

Figura 5.4: *Estructura archivo databases.conf*

- ***neighbours.conf***: fichero que almacena la información sobre los vecinos de la aplicación. Estructura del archivo:

```
<vecino1>=<tipo BBDD>,<nombre BBDD>;
<vecino2>=<tipo BBDD>,<nombre BBDD>;
<vecino3>=<tipo BBDD>,<nombre BBDD>;
...
```

Figura 5.5: *Estructura archivo neighbours.conf*

- ***neighboursDatabases.conf***: fichero que almacena la información sobre los vecinos y sus bases de datos. El fichero se va creando según se van realizando comunicaciones con distintos vecinos. Estructura del archivo:

```
<vecino1>;
<vecino2>;
<vecino3>;
...
```

Figura 5.6: *Estructura archivo neighboursDatabases.conf*

Puedes encontrar ejemplos de estos archivos de configuración mencionados en el apéndice B.

5.1.2. IDNetDaemon

Como bien se ha comentado, IDNetDaemon es un servidor ejecutado en *background* encargado de llevar a cabo la resolución de todas las peticiones con destino el nodo Cliente. Dado que no necesita que la aplicación del Cliente esté activa, es capaz de responder a todas las peticiones que le lleguen. Pensamos que independizar los dos procesos favorecería a la red.

Siguiendo el esquema general de la red de la figura 5.1, cuando un nodo Cliente se comunica con otro nodo Cliente, el IDNetDaemon del nodo receptor Cliente se encarga de responder a todas las peticiones del IDNetSoftware del nodo emisor Cliente. Resumiendo su funcionalidad, actúa como el "portavoz" del nodo Cliente.

Internamente, las funciones que realiza IDNetDaemon son las siguientes:

- **Registro del nodo Cliente en la aplicación**. Para ello, envía la solicitud de registro al nodo GateKeeper que tiene asignado. En la sección 5.3.7 se explica la estructura del mensaje enviado.
- **Establecimiento de la conexión con el vecino**. Interactúa con el IDNetSoftware del nodo Cliente emisor.
- **Gestión de la conexión con las bases de datos propias**. Sólo se encargará de aquellas bases de datos que hayan sido asignadas por el IDNetSoftware.

Para la realización de tales funciones, se ha diseñado el siguiente diagrama.

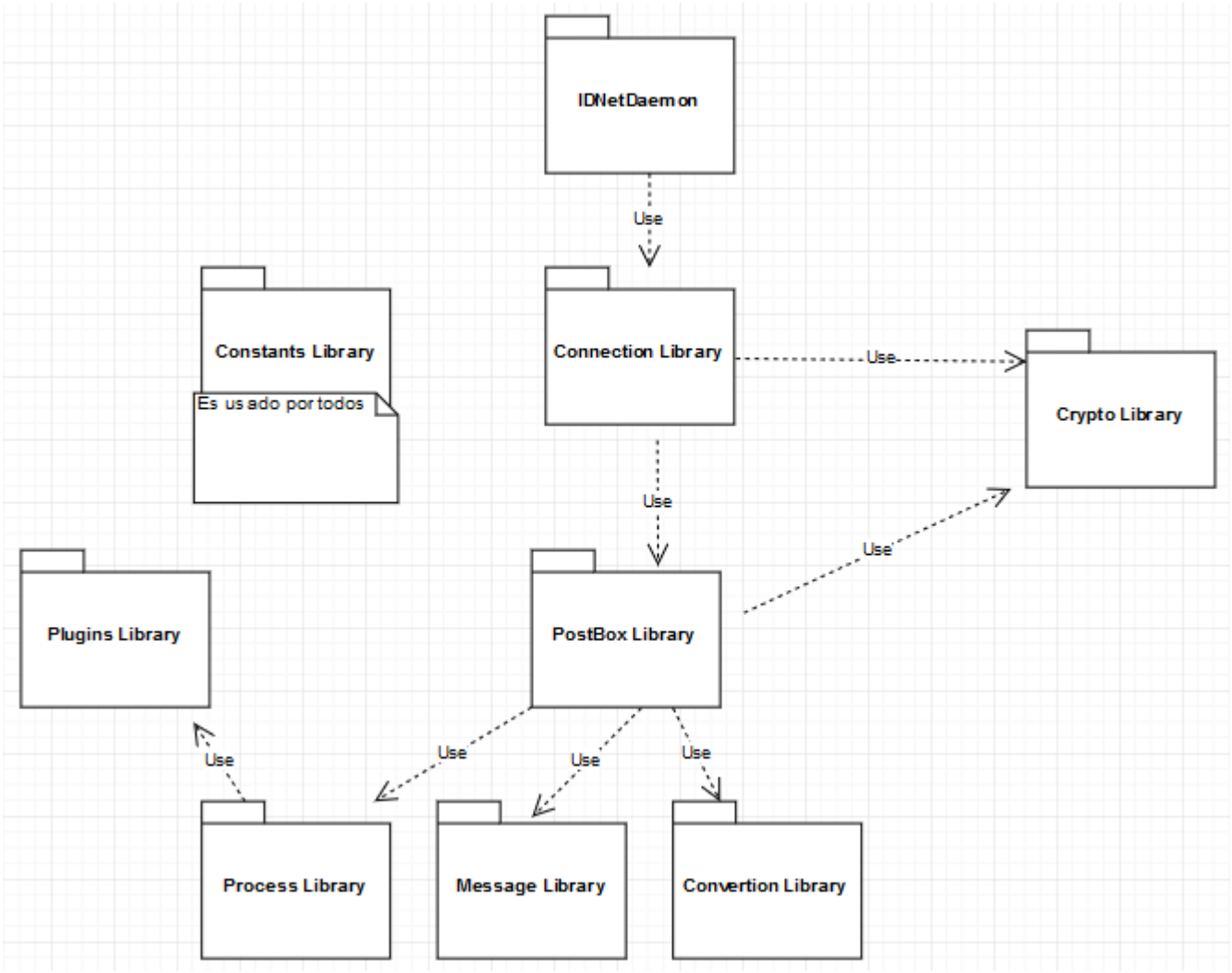


Figura 5.7: *Diagrama de módulos IDNetDaemon*

El diagrama anterior presenta los siguientes módulos:

- **IDNetDaemon**: contiene la clase principal. La ejecución de la clase principal sólo se puede realizar mediante *mono-service*. Para mayor información acuda a la sección 6.1.2.
- **Connection Library**: contiene dos clases principales: *Server* y *RegisterClient*. La primera es el servidor encargado de escuchar a los clientes vecinos, mientras que la segunda clase es la encargada de registrar al nodo Cliente en la aplicación.
- **Message Library**: módulo similar al de la sección 5.1.1.
- **Crypto Library**: módulo similar al de la sección 5.1.1.

- **PostBox Library**: módulo similar al de la sección 5.1.1.
- **Plugins Library**: presenta similitud con respecto al módulo DatabaseLibrary de IDNetSoftware. La diferencia está en que, en lugar de manejar la información relativa a las bases de datos, es la encargada de conectarse a ellas y de obtener la información pertinente. Presenta dos *Plugins* principales: PluginMongo y PluginMySQL.
- **Process Library**: módulo similar al de la sección 5.1.1
- **Convertion Library**: módulo similar al de la sección 5.1.1
- **Constants Library**: módulo similar al de la sección 5.1.1

Ficheros de soporte

IDNetDaemon se vale de dos ficheros fundamentales de configuración para su desempeño.

- ***info.conf***: similar a los usados por IDNetSoftware.
- ***databases.conf***: similar a los usados por IDNetSoftware.

5.2. Arquitectura interna en la Nube

Una de las partes más críticas de la aplicación es la parte correspondiente al Cloud. No solo dependemos de un proveedor externo (aunque intentamos minimizar al máximo el *vendor locking*) sino que además, la seguridad es uno de los aspectos fundamentales a tener en cuenta. Esto choca directamente con la definición que tenemos del Cloud actualmente. Por lo tanto, hemos desarrollado una arquitectura de nodos en Cloud con la seguridad como uno de los principales objetivos.

5.2.1. Nodo GateKeeper

Los nodos que se encargan de transmitir el mensaje a través de la red y alojados en Cloud son los denominados GateKeeper (GK en adelante). Esta nomenclatura no es azarosa sino que ejemplifica perfectamente el trabajo que desempeña.

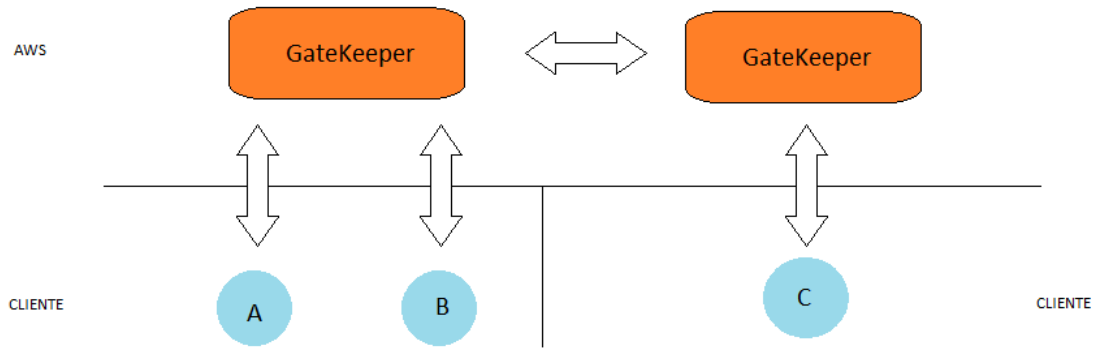


Figura 5.8: *Esquema General de la distribución de los GateKeeper en la red*

Las funciones actuales de los GK son las siguientes:

- Actúan como entrada y salida de los mensajes a la red Cloud. Cada uno de los nodos Cliente están asignados a un GK y se comunican con él y sólo con él.
- Crea una tabla de rutas con los demás GK para establecer el encaminamiento de los mensajes. Se utiliza un método de encaminamiento por vectores de distancias parecido a RIP.
- Comprueba la legitimidad de los Clientes conectados a través de él a la red.
- Anonimiza los mensajes que pasan a través de él.

Para conseguir este comportamiento, el GK tiene una estructura interna de servidor/cliente con hilos paralelos siguiendo un modelo como el ejemplifica el diagrama de clases del Apéndice C (C.3).

El GK se compone de un demonio ejecutándose en *background* que lanza tres grandes hilos de ejecución. Uno de ellos se corresponde con el servidor que escucha las peticiones que le llegan desde otro GK vecino (puerto 12000). El segundo ejecuta las peticiones provenientes de un nodo Cliente (puerto 11000); mientras que el tercero se encarga de anunciar periódicamente (actualmente cada tres minutos) la tabla de rutas a sus vecinos de la red. Este modelo *multithread* permite la ejecución paralela de varios trabajos sin bloquear el resto de operaciones.

A continuación pasaremos a detallar el comportamiento del GK en función de las peticiones que le llegan.

Comportamiento de un GateKeeper con respecto a un Nodo Cliente

Como hemos mencionado anteriormente. El GK sirve como puerta de entrada del nodo Cliente a la red. Para hacer efectiva esta funcionalidad se propusieron los siguientes objetivos:

- **Debía haber un protocolo de conexión del nodo Cliente al GK**. Es decir, cada vez que el nodo Cliente se activara (la parte cliente), el GK debía de verificar que ese nodo es realmente un nodo de la red.
- **A la hora de transmitir un mensaje en la red**, esta transmisión debía ser segura. Es decir, para el GK, la dirección IP de entrada debía ser irrelevante. Esto choca directamente con el objetivo anterior.
- **El GK debería informar** al nodo cliente de los nodos con los que se puede comunicar.

Estos objetivos contrapuestos suponían un reto de diseño para la arquitectura interna del nodo.

Dado que la seguridad era uno de los principales objetivos de este proyecto, primero se abordó el segundo objetivo. La respuesta fue más o menos sencilla: un cola anonimizadora. El GK recibe los mensajes provenientes del puerto 11000 y los encola para un tratamiento posterior. Esto puede suponer un cuello de botella al rendimiento del nodo por lo que se se puede recurrir a balanceadores de carga para minimizar el impacto. El GK no considera la ip de origen para reenviar el paquete, solo la de destino. No obstante, ¿cómo se asegura de que ese paquete procede de un cliente legítimo? En este momento es dónde entra en juego el primer objetivo descrito anteriormente.

Mediante el protocolo de conexión, el GK se asegura de la existencia de un cliente concreto. Cuando un usuario inicia la aplicación de IDNet en su ordenador, debe realizar un proceso de login. Dicho login coteja los datos con una base de datos de la aplicación alojada en la nube y genera un código aleatorio. Ese código será el que se use en los mensajes de conexión para verificar la identidad del nodo. A partir de ese momento, el GK se asegura que ese nodo existe y lo registra en su tabla de rutas.

No obstante, el problema sigue persistiendo, el GK no debería tratar con la IP de origen directamente. Para ello usamos *alias* para los nodos cliente. Con cada mensaje los parámetro de origen y destino no se tratan con direcciones IP, sino con alias. Solo los GK conocen la relación entre un alias y su IP y solo lo consultan una vez. De esta forma podemos evitar ataques de tipo MITM al mismo tiempo que el GK verifica la identidad sin comprometer al cliente.

Por último, pero no menos importante y como medida extra de seguridad, el nodo Cliente no dispone de direcciones IP de ningún otro nodo. Simplemente sus alias. De esta forma, durante el protocolo de conexión entre un cliente y un GK, una vez verificada la identidad del cliente, el GK informa al susodicho de los nodos con los que puede comunicarse. No obstante, solo le aporta los alias. Como hemos mencionado anteriormente, solo el GK debería conocer la relación entre una IP y su alias.

Comportamiento del GateKeeper con respecto a mensajes de otros GateKeeper

En este sentido se nos presentan dos casos de uso:

- Anunciamiento de tablas de rutas.
- Retransmisión de un mensaje.

El nodo escucha las peticiones de otros GK por el puerto 12000. De la misma forma que con el cliente, el nodo receptor no considera la dirección de origen, sino la de destino. En caso de que la dirección de destino sea él mismo, significa que es un mensaje de propagación de tablas de rutas y fusiona la tabla recibida con la que suya propia. En caso de que la dirección de destino sea otra distinta, comprueba el siguiente paso en la tabla de rutas. Si es un nodo conectado a él mismo, lo entrega directamente al cliente; si no, lo reenvía a otro GK.

Ficheros de soporte

El GK se vale de dos ficheros fundamentales para su desempeño.

- ***neighbours.xml***
- ***routes.xml***

Ambos fichero xml son vitales para las comprobaciones, el encaminamiento y el protocolo de intercambio y cálculo de rutas.

El funcionamiento del protocolo de encaminamiento es simple. Un nodo recibe un mensaje de todos los nodos vecinos en los que el nodo receptor figura en su tabla de vecinos (neighbours.xml). En dicho mensaje figura la tabla de rutas del nodo emisor. Dicha tabla está diseñada de forma que no se envíen los nodos cuya dirección de *"hop"* sea igual a la del nodo receptor. De esta forma se evitan bucles en el encaminamiento. Igualmente, todas las distancias que se envían están incrementadas en una unidad (porque sería un salto más). El nodo receptor procesa la tabla y en base a las distancias decide si un camino es más conveniente que otro y reescribe su tabla de rutas.

Protocolo de encaminamiento por vectores de distancias

Los nodos disponen de un fichero *routes.xml*. Dicho fichero almacena la información de cada uno de los nodos Cliente existentes en la red. Para cada nodo Cliente el GateKeeper almacena su dirección IP (la cual solo usa para resolver el alias), el propio alias, la dirección del próximo GK a través del cual se va a comunicar con el nodo Cliente y la distancia (nodos) entre el GK y el Cliente destino. La estructura del fichero sería la mostrada en la figura. Véase Figura 5.9

```
<?xml version="1.0" encoding="UTF-8"?>
<routes>
  <route>
    <d_node></d_node>
    <d_hop></d_hop>
    <name></name>
    <distance></distance>
  </route>
</routes>
```

Figura 5.9: *Fichero routes.xml del GateKeeper*

Cuando un nodo GK recibe un mensaje de otro nodo GK que va dirigido a él mismo, lo identifica como un mensaje de propagación de rutas. A partir de este momento, el nodo receptor compara cada una de las rutas de la nueva tabla con la que ya tenía almacenada previamente. Si recibe la ruta de un nodo Cliente que no conoce, lo incorpora a su fichero de rutas. Si ya existe ese cliente, compara las distancias, elige la menor y actualiza tanto la distancia como la dirección de salto.

Igualmente, el propio nodo retransmite su propia tabla de rutas. Para ello genera una tabla para cada uno de los nodos existentes en *neighbours.xml*. Posteriormente, elimina las entradas cuya dirección de salto es el nodo al que va a enviar la tabla. Finalmente las envía a sus destinos. Este proceso se repite periódicamente.

La estructura del fichero *neighbours.xml* es la mostrada en la figura. Véase Figura 5.10

```
<?xml version="1.0" encoding="utf-8" ?>
<neighbours>
    <node></node>
</neighbours>
```

Figura 5.10: *Fichero neighbours.xml del GateKeeper*

5.2.2. Arquitectura en AWS

Hemos elegido Amazon Web Services como proveedor Cloud para desplegar los GateKeeper en la red. No obstante, para hacer el despliegue más sencillo usaremos además Docker. Esta tecnología nos permite encapsular el nodo en un entorno de ejecución propio que luego podremos clonar y desplegar sobre las máquinas de Amazon de forma rápida y sencilla.

Durante el desarrollo del proyecto hemos reparado en que, posiblemente, los GateKeeper puedan ser objeto de ataques de denegación de servicio (DoS o DDoS). Además, aunque no hubiera ataques, presentan un claro cuello de botella en cuanto al rendimiento de la red. La elección de AWS nos permite también resolver este cuello de botella ya que nos da la posibilidad de montar balanceadores de carga sobre los nodos en la red y poder gestionar de forma elástica las peticiones.

5.2.3. Base de datos IDNet

Para el almacenamiento de los usuarios de la aplicación disponemos de una base de datos MySQL alojada en el proveedor Amazon Web Services gracias al servicio Amazon RDS (Servicio de bases de datos relacionales), el cuál nos permite una alta disponibilidad, una seguridad inherente y una mayor accesibilidad a la base de datos.

La base de datos está compuesta por una tabla llamada "usuariosIDNet" que contiene los siguientes campos:

- ***code***: código aleatorio generado en el IDNetDaemon y almacenado desde la creación del usuario.
- ***user***: nombre de usuario.
- ***password***: contraseña.

Como bien se comentará en la sección 5.4.2, las contraseñas no se encuentran almacenadas en texto plano.

El acceso a la base de datos IDNet lo realizan los dos nodos de la aplicación, es decir, tanto el nodo Cliente –solo IDNetSoftware– como el nodo GateKeeper.
Dentro del nodo Cliente, IDNetSoftware es el encargado de insertar en la base de datos los usuarios registrados, además de comprobar los usuarios existentes en el inicio de sesión.
Por otro lado, el nodo GateKeeper comprueba que los mensajes recibidos por el puerto del cliente se hagan por un cliente legítimo, es decir, por un usuario que se encuentre en la base de datos IDNet.

5.3. Protocolo de comunicación de mensajes

Hay tres tipos de comunicaciones entre los nodos:

- **Comunicación nodo Cliente - nodo Cliente**
- **Comunicación nodo Cliente - nodo Gatekeeper**
- **Comunicación nodo Gatekeeper - nodo Gatekeeper**

Para estas comunicaciones hemos implementado un protocolo de comunicación de mensajes estructurado en formato XML.

A través de las siguiente secciones vamos a detallar los tipos de mensajes existentes en la comunicación. La nomenclatura en el tipo de mensaje se basa en la siguiente especificación. Véase figura 5.11.

```
mensaje 0{Relación con nodo Cliente}{Función}
```

Figura 5.11: *Nomenclatura de los mensajes*

Si en la comunicación ambos nodos son clientes se especifica con **0**, mientras que si en la comunicación un nodo cliente se comunica con un nodo Gatekeeper, es un **1**. Y, por otro lado, la función, que ya es especifica para cada relación existente.

5.3.1. Mensaje 001

Este tipo de mensaje se ha dividido en dos debido a que la conexión entre los nodos cliente es necesaria realizarla en dos tiempos consecutivos: 001a y 001b.

Mensaje 001a

El mensaje 001a es enviado por el cliente emisor IDNetSoftware que quiere realizar la conexión hacia el cliente receptor IDNetDaemon. Se caracteriza por el envío de su clave pública al nodo cliente receptor, el cual guardará dicha clave para encriptar asimétricamente sus futuros mensajes. Ver Figura 5.3.1.

```
<root>
    <message_type>001a</message_type>
    <source>Cliente emisor</source>
    <destination>Cliente destino</destination>
    <key>-----BEGIN PUBLIC KEY-----
        Clave pública del cliente emisor
        -----END PUBLIC KEY-----
    </key>
</root>
```

Figura 5.12: *Mensaje 001a*

Como se puede comprobar en la imagen, tenemos los siguientes campos:

- Message type: tipo de mensaje.
- Source: nodo cliente emisor.
- Destination: nodo cliente receptor.
- Key: clave pública del nodo cliente emisor.

En el apéndice A puedes encontrar ejemplos concretos de mensajes 001a.

Mensaje 001b

El mensaje 001b es enviado por el cliente emisor IDNetSoftware que quiere realizar la conexión hacia el cliente receptor IDNetDaemon. Se caracteriza por el envio de la información necesaria para llevar a cabo la encriptación simétrica en futuros mensajes. Debido a la securización de la comunicación que se comentará en la sección 5.4.1, este mensaje está encriptado con la clave pública del nodo cliente receptor, por lo que el mensaje se envía con los elementos XML encriptados. Ver Figura 5.13.

```
<root>
  <message_type>001b</message_type>
  <source>Cliente emisor</source>
  <destination>Cliente destino</destination>
  <encripted>
    <key>Clave</key>
    <IV>Vector de inicialización</IV>
  </encripted>
</root>
```

Figura 5.13: *Mensaje 001b*

Los campos, como se puede ver en la imagen, son los siguientes:

- Message type: tipo de mensaje.
- Source: nodo cliente emisor.
- Destination: nodo cliente receptor.
- Key: clave para la encriptación simétrica AES.
- IV: vector de inicialización aleatorio para la encriptación simétrica AES.

En el apéndice A puedes encontrar ejemplos concretos de mensajes 001b.

5.3.2. Mensaje 002

El mensaje 002 es enviado por el cliente emisor IDNetSoftware que quiere realizar la conexión hacia el cliente receptor IDNetDaemon. En este mensaje se solicita la obtención del esquema de la base de datos que se quiere consultar. Ver Figura 5.14.

```
<root>
    <message_type>002</message_type>
    <source>Cliente emisor</source>
    <destination>Cliente destino</destination>
    <encripted>
        <db_name>Nombre de la base de datos</db_name>
        <db_type>Tipo de base de datos</db_type>
    </encripted>
</root>
```

Figura 5.14: *Mensaje 002*

Los campos son los siguientes:

- Message type: tipo de mensaje.
- Source: nodo cliente emisor.
- Destination: nodo cliente receptor.
- Encripted: este elemento contiene todos aquellos campos que son necesarios encriptarlos por motivos de seguridad. Dichos campos son los siguientes:
  - Db name: nombre de la base de datos.
  - Db type: tipo de base de datos.

En el apéndice A puedes encontrar ejemplos concretos de mensajes 002.

5.3.3. Mensaje 003

El mensaje 003 es enviado por el cliente emisor IDNetSoftware que quiere realizar la conexión hacia el cliente receptor IDNetDaemon. Es el mensaje que lleva a cabo la realización de una consulta a una base de datos del nodo cliente receptor. La estructura del mensaje depende del tipo de base de datos que usemos. Ver Figuras 5.15 y 5.16.

```
<root>
    <message_type>003</message_type>
    <source>Cliente emisor</source>
    <destination>Cliente destino</destination>
    <encripted>
        <db_name>Nombre de la base de datos</db_name>
        <db_type>mysql</db_type>
        <body>
            <query>
                <select>Campo de selección</select>
                <from>Tabla de la base de datos</from>
                <where>Filtro de la consulta</where>
                <orderby>Ordenación de los resultados</orderby>
            </query>
        </body>
  </encripted>
</root>
```

Figura 5.15: *Mensaje 003 para MySQL*

```
<root>
  <message_type>003</message_type>
  <source>Cliente emisor</source>
  <destination>Cliente destino</destination>
  <encripted>
    <db_name>Nombre de la base de datos</db_name>
    <db_type>mongodb</db_type>
    <body>
      <query>
        <collection>Colección de la base de datos</collection>
        <filter>Filtrado de la consulta</filter>
        <projection campo1="Condición" campo2="Condición" campo3="Condición" ... />
        <sort>Ordenación de los resultados</sort>
        <limit>Límite del número de resultados</limit>
      </query>
    </body>
  </encripted>
</root>

Condición: aparición en los resultados. 1:Sí  0:No
```

Figura 5.16: *Mensaje 003 para MongoDB*

Los campos de los mensajes son:

- Message type: tipo de mensaje.
- Source: nodo cliente receptor.
- Destination: nodo cliente emisor.
- Encripted: de forma similar al mensaje 003, contiene todos aquellos campos que son necesarios encriptarlos. Dichos campos son los siguientes:
  - Db name: nombre de la base de datos.
  - Db type: tipo de base de datos.
  - Body: el cuerpo del mensaje contiene la consulta.
    Para las bases de datos MySQL contiene los campos: select, from, where y orderby.
    Por otro lado, para las bases de datos MongoDB contiene los campos: colleciton, filter, projection, sort y limit.

En el apéndice A puedes encontrar ejemplos concretos de mensajes 003.

5.3.4. Mensaje 004

El mensaje 004, de forma análoga al mensaje 001, se realiza en dos tiempos consecutivos: 004a y 004b.

Mensaje 004a

El mensaje 004a es enviado por el cliente receptor IDNetDaemon como respuesta al mensaje 001a enviado por el cliente emisor IDNetSoftware. Presenta la misma función que el mensaje 001a de la sección 5.3.1.
Ver Figura 5.17.

```
<root>
  <message_type>004a</message_type>
  <source>Cliente emisor</source>
  <destination>Cliente destino</destination>
  <key>-----BEGIN PUBLIC KEY-----
       Clave pública del cliente emisor
      -----END PUBLIC KEY-----
  </key>
</root>
```

Figura 5.17: *Mensaje 004a*

Como se puede ver en el mensaje 001b, los campos de la imagen son similares.
En el apéndice A puedes encontrar ejemplos concretos de mensajes 004a.

Mensaje 004b

El mensaje 001b es enviado por el cliente receptor IDNetDaemon como respuesta al mensaje 001b enviado por el cliente emisor IDNetSoftware. Tiene la función principal de informar de las bases de datos del receptor que están disponibles para el cliente emisor. Debido a la securización de la infraestructura que se comentará en la sección 5.4, este mensaje está encriptado con criptografía simétrica AES gracias al mensaje 001b, el cual nos da la información necesaria para llevar a cabo dicha encriptación. Ver Figura 5.18.

```
<root>
    <message_type>001b</message_type>
    <source>Cliente emisor</source>
    <destination>Cliente destino</destination>
    <encripted>
        <databases>
            <database db_type="tipoDeBaseDeDatos1">Nombre de la base de datos1</database>
            <database db_type="tipoDeBaseDeDatos2">Nombre de la base de datos2</database>
        </databases>
    </encripted>
</root>
```

Figura 5.18: *Mensaje 004b*

Los campos, como se puede ver en la imagen, son los siguientes:

- Message type: tipo de mensaje.
- Source: nodo cliente receptor.
- Destination: nodo cliente emisor.
- Encripted: elemento que contiene el campo *databases*, y éste contiene a su vez el campo:
    - Database: dentro del elemento XML tenemos el nombre de la base de datos. Como atributo del elemento nos encontramos con el tipo de base de datos.

En el apéndice A puedes encontrar ejemplos concretos de mensajes 004b.

5.3.5. Mensaje 005

Este mensaje es enviado por el cliente receptor IDNetDaemon como respuesta al mensaje 002 enviado por el cliente emisor IDNetSoftware. Tiene la función principal de responder con el esquema de la base de datos requerida en el mensaje 002.
De forma similar a los mensajes enviados por el cliente emisor a partir del mensaje 001, es decir, los mensajes 002, 003 y 004, encriptamos este mensaje con criptografía simétrica AES. Ver Figuras 5.19 y 5.20.

```
<root>
  <message_type>005</message_type>
  <source>Cliente emisor</source>
  <destination>Cliente destino</destination>
  <encripted>
    <db_name>Nombre de la base de datos</db_name>
    <db_type>mysql</db_type>
    <body>
        <database name="nombre de la base de datos">
            <table name="nombre de la tabla1">
                <col name="nombre columna11" type="tipo columna11" />
                <col name="nombre columna12" type="tipo columna12" />
                ...
            </table>
            <table name="nombre de la tabla2">
                <col name="nombre columna21" type="tipo columna21" />
                <col name="nombre columna21" type="tipo columna21" />
                ...
            </table>
            ...
      </database>
    </body>
  </encripted>
</root>
```

Figura 5.19: *Mensaje 005 para MySQL*

```
<root>
  <message_type>005</message_type>
  <source>Cliente emisor</source>
  <destination>Cliente destino</destination>
  <encripted>
    <db_name>Nombre de la base de datos</db_name>
    <db_type>mongodb</db_type>
    <body>
      <result>
        <database>
          <name>Nombre de la base de datos</name>
          <colecciones>
            <name>Nombre de la colección 1</name>
            <ejemploColeccion>
              <campo11>Tipo de campo (String,Number,Boolean,Array,Objeto,null, tipos BSON)</campo11>
              <campo12>Tipo de campo (String,Number,Boolean,Array,Objeto,null, tipos BSON)</campo12>
              <campo13>Tipo de campo (String,Number,Boolean,Array,Objeto,null, tipos BSON)</campo13>
              <campo14>Tipo de campo (String,Number,Boolean,Array,Objeto,null, tipos BSON)</campo14>
              ...
            </ejemploColeccion>
          </colecciones>
          <colecciones>
            <name>Nombre de la colección 1</name>
            <ejemploColeccion>
              <campo21>Tipo de campo (String,Number,Boolean,Array,Objeto,null, tipos BSON)</campo21>
              <campo22>Tipo de campo (String,Number,Boolean,Array,Objeto,null, tipos BSON)</campo22>
              <campo23>Tipo de campo (String,Number,Boolean,Array,Objeto,null, tipos BSON)</campo23>
              ...
            </ejemploColeccion>
          </colecciones>
          <colecciones>
            <name>Nombre de la colección 1</name>
            <ejemploColeccion>
              <campo31>Tipo de campo (String,Number,Boolean,Array,Objeto,null, tipos BSON)</campo31>
              <campo32>Tipo de campo (String,Number,Boolean,Array,Objeto,null, tipos BSON)</campo32>
              <campo33>Tipo de campo (String,Number,Boolean,Array,Objeto,null, tipos BSON)</campo33>
              ...
            </ejemploColeccion>
          </colecciones>
          ...
        </database>
      </result>
    </body>
  </encripted>
</root>
```

Figura 5.20: *Mensaje 005 para MongoDB*

Los campos de los mensajes son:

- Message type: tipo de mensaje.
- Source: nodo cliente receptor.
- Destination: nodo cliente emisor.
- Encripted: contiene todos los siguientes campos:
  - Db name: nombre de la base de datos.
  - Db type: tipo de base de datos.

- Body: en el cuerpo del mensaje nos encontramos con el esquema de la base de datos.

En el apéndice A puedes encontrar ejemplos concretos de mensajes 005.

5.3.6. Mensaje 006

Este mensaje es enviado por el cliente receptor IDNetDaemon como respuesta al mensaje 003 enviado por el cliente emisor IDNetSoftware. Tiene la función principal de responder con los resultados de la consulta requerida en el mensaje 003. Análogamente al mensaje 005, se encripta simétricamente gran parte del mensaje. Ver Figuras 5.21 y 5.22.

```
<root>
  <message_type>006</message_type>
  <source>Juan</source>
  <destination>Lorenzo</destination>
  <encripted>
    <db_name>Nombre de la base de datos</db_name>
    <db_type>mysql</db_type>
    <body>
      <result table="Nombre de la tabla">
        <row campo1="Valor campo11" campo2="Valor campo12" campo3="Valor campo13" />
        <row campo1="Valor campo21" campo2="Valor campo22" campo3="Valor campo23" />
        <row campo1="Valor campo31" campo2="Valor campo32" campo3="Valor campo33" />
        ...
      </result>
    </body>
  </encripted>
</root>
```

Figura 5.21: *Mensaje 006 para MySQL*

```
<root>
  <message_type>006</message_type>
  <source>Cliente emisor</source>
  <destination>Cliente destino</destination>
  <encripted>
    <db_name>Nombre de la base de datos</db_name>
    <db_type>mongodb</db_type>
    <body>
      <result>
        <row>
          <campo11>Valor campo11</campo11>
          <campo12>Valor campo12</campo12>
          <campo13>Valor campo13</campo13>
          ...
        </row>
        <row>
          <campo21>Valor campo21</campo21>
          <campo22>Valor campo22</campo22>
          <campo23>Valor campo23</campo23>
          ...
        </row>
        ...
      </result>
    </body>
  </encripted>
</root>
```

Figura 5.22: *Mensaje 006 para MongoDB*

Como se puede comprobar en las imágenes, tenemos los siguientes campos:

- Message type: tipo de mensaje.
- Source: nodo cliente receptor.
- Destination: nodo cliente emisor.
- Encripted: contiene los siguientes campos:
  - Db name: nombre de la base de datos.
  - Db type: tipo de base de datos.
  - Body: en el cuerpo del mensaje nos encontramos con los resultados de la consulta a la base de datos.

En el apéndice A puedes encontrar ejemplos concretos de mensajes 006.

5.3.7. Mensaje 010

El mensaje 010, junto con el mensaje 011, pertenecen a la comunicación con los nodos Gatekeeper.
El mensaje 010 es enviado por el cliente emisor IDNetDaemon al nodo Gatekeeper que tiene

asignado, con el fin de enviar al nodo Gatekeeper la información concerniente al registro en la aplicación de dicho cliente. Ver Figura 5.23.

```
<root>
    <message_type>010</message_type>
    <source>Cliente emisor</source>
    <ip>IP del cliente emisor</ip>
    <destination>IP del GateKeeper asignado</destination>
    <code>Código de registro</code>
</root>
```

Figura 5.23: *Mensaje 010*

Contiene los siguientes campos:

- Message type: tipo de mensaje.
- Source: nodo cliente emisor.
- IP: dirección IP del nodo cliente emisor.
- Destination: nodo Gatekeeper.
- Code: código numérico aleatorio enviado al cliente. Este código es el mismo que queda registrado en la base de datos del servidor de la aplicación. El Gatekeeper comprobará internamente esta condición mencionada.

En el apéndice A puedes encontrar ejemplos concretos de mensajes 010.

5.3.8. Mensaje 011

El mensaje 011 es enviado por el cliente emisor IDNetSoftware al nodo Gatekeeper que tiene asignado, con el fin de solicitar al nodo Gatekeeper la información concerniente a los vecinos de la aplicación. Ver Figura 5.24.

```
<root>
  <message_type>011</message_type>
  <source>Cliente emisor</source>
  <ip>IP del cliente emisor</ip>
  <destination>IP del GateKeeper asignado</destination>
</root>
```

Figura 5.24: *Mensaje 011*

El mensaje contiene los siguientes campos:

- Message type: tipo de mensaje.

- Source: nodo cliente emisor.
- IP: dirección IP del nodo cliente emisor.
- Destination: IP del nodo Gatekeeper asignado.

5.3.9. Mensaje 012

El mensaje 012 es enviado por el GateKeeper en respuesta al nodo Cliente que se acaba de conectar e identificar. Devuelve cada uno de los nodos Cliente de la red con los que el cliente con el que se está comunicando puede contactar. Ver Figura 5.25.

```
<?xml version="1.0" encoding="UTF-8"?>
<routes>
  <route>
    <name></name>
  </route>
</routes>
```

Figura 5.25: *Mensaje 012*

El mensaje contiene los siguientes campos:

- routes: Nodo Raíz.
- route: Indica una Ruta (otro cliente en la red).
- name: Nombre del cliente.

5.4. Securización de las arquitecturas

A través de las secciones 5.1 y 5.2 hemos visto las arquitecturas internas, tanto del nodo cliente como en la nube. Los puntos críticos de estas arquitecturas se encuentran en los mensajes intercambios entre los nodos que componen nuestra red. Por ello, vamos a ver qué medidas se han tomado para securizar los mensajes intercambiados. Por otro lado, se van a hacer mención otras medidas de seguridad implantadas en el proyecto.

5.4.1. Securización en el protocolo de comunicación

Las medidas tomadas para la securización del protocolo de comunicación en relación a los mensajes entre nodos cliente son las siguientes:

- **Encriptación con criptografía asimétrica**: en los primeros mensajes de una comunicación se comparten las claves públicas, de tal manera que una vez compartidas las claves puedan empezar a enviar información sensible. En los mensajes 001a y 004a

se realiza tal intercambio. Los mensajes 001b y 004b son encriptados con las claves públicas del receptor; son desencriptados con las claves privadas del cliente receptor. La información sensible se encuentra dentro de la etiqueta XML *<encripted>*. En relación a la implementación, se ha usado el algoritmo de cifrado asimétrica RSA.

- **Encriptación con criptografía simétrica**: una vez se haya realizado la encriptación asimétrica mencionada, es la hora de pasar la información necesaria para que los posteriores mensajes sean encriptados con criptografía simétrica. Para ello, en el mensaje 001b se transmite tanto la clave como el vector de inicialización para las futuras encriptaciones simétricas. Los restantes mensajes, es decir, los mensajes 002,003,004,005 y 006, tienen encriptada la información sensible dentro de la etiqueta XML *<encripted>*, de forma similar que en la criptografía asimétrica. En relación a la implementación, se ha usado el algoritmo de cifrado simétrico Rijndael, comúnmente conocida como AES.

- **Doble encriptación**: no solo los mensajes contendrán todos los elementos XML dentro del elemento *<encripted>*, sino que el propio elemento *<source>* se encripta asimétricamente. Por lo tanto, en los mensajes 002, 003, 004, 005 y 006 hay una parte encriptada simétricamente y otra asimétricamente, llegando a realizar una criptografía híbrida.

Estas medidas tomadas nos proporcionan las siguientes ventajas para una comunicación segura:

- **Protección de información sensible: dentro de la etiqueta *<encripted>***. Información relativa a nombre de base de datos, tipo de bases de datos, esquema de una base de datos y valores reales de esas bases de datos.

- **Protección contra ataques *MITM*** [1]: mediante la encriptación asimétrica y simétrica evitamos que una tercer persona sea capaz de obtener información sobre el mensaje y suplante la identidad de uno de ellos.

- **Anonimización de la fuente de comunicación**: de esta forma ni los propios GateKeeper conocen de dónde viene el mensaje, ni tampoco aquellos que intercepten el mensaje.

En la siguiente imagen se puede ver un ejemplo de la criptografía híbrida anteriormente mencionada.

Figura 5.26: *Criptografía híbrida*

[1] Man in the Middle

5.4.2. Securización de la base de datos IDNet

La base de datos de IDNet, como se ha comentado en la sección 5.2.3, se encuentra alojada en Amazon Web Services gracias al servicio Amazon RDS. Este servicio nos proporciona la ejecución de nuestra instancia de la base de datos en Amazon Virtual Private (Amazon VPC), permitiéndonos aislar nuestra instancia en una red virtual privada.

Una medida de seguridad de la base de datos es la salvaguarda de contraseñas con la función Hash SHA-256. El proceso es el siguiente:

1. El nodo cliente IDNetSoftware en el momento de registrarse su contraseña es hasheada. Tanto la contraseña, como los campos del nombre de usuario y el código, se insertan mediante una consulta en la base de datos.

2. A la hora de iniciar sesión, se cifra de nuevo la contraseña en el nodo cliente IDNetSoftware con el fin de comprobar que coincide tanto la contraseña hasheada como la contraseña almacenada en la base de datos.

3. En ningún momento se puede volver a obtener la contraseña en texto plano de la base de datos, ya que la función Hash nos garantiza la integridad de la contraseña.

En la siguiente imagen se puede ver un ejemplo de una fila de la base de datos.

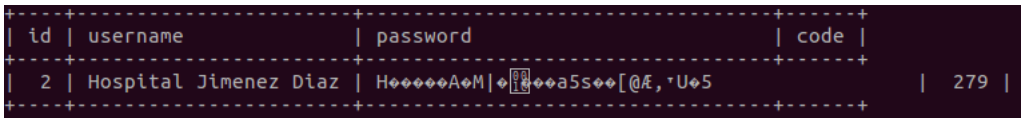


Figura 5.27: *Ejemplo de una fila de la base de datos*

5.4.3. Protección de información de las BBDD en local

Como se ha comentado en la sección relacionada con la arquitectura interna del cliente (5.1), gestionamos la información necesaria para conectarnos a las bases de datos existentes, guardándola en un fichero de configuración. Un campo de esta información es la contraseña, solo en aquellos casos en el que el acceso a la base de datos se haga mediante un usuario y contraseña. Por ello, se ha vuelto a usar criptografía simétrica para cifrar la contraseña, evitando que se encuentre en texto plano.

Capítulo 6

Tecnologías

En las siguientes secciones se enumeran las tecnologías utilizadas y descartadas, así como las razones para llegar a tales conclusiones.

6.1. Tecnologías utilizadas

A continuación vamos a enumerar las tecnologías empleadas en el proyecto.

6.1.1. C#

Nos decantamos por usar este lenguaje de programación debido a la amplia gama de librerías que se pueden usar con este lenguaje. Además, los integrantes del proyecto queríamos aprender un lenguaje nuevo y de esta forma ampliar nuestros conocimientos. Entendemos que es un lenguaje sólido y, usando el Framework .Net [8], podríamos desarrollar fácilmente la aplicación.

6.1.2. Mono

Mono es un proyecto de código abierto basado en GNU/Linux, compatible con .NET e independiente de la plataforma. El proyecto decidimos desarrollarlo en Linux porque vimos que los administradores de sistemas usaban este sistema operativo para administrar sus bases de datos. Por todo ello, hemos usado Mono para desarrollar en Linux nuestro framework. Además, para desarrollar el software en sí nos hemos ayudado de un entorno de desarrollo llamado MonoDevelop.

Figura 6.1: *Logos de MonoDevelop y Mono respectivamente*

Adicionalmente, hemos usado un componente de Mono llamado "mono-service". Este componente nos permite desarrollar un demonio de un ejecutable [3] desarrollado con Mono.

6.1.3. MySQL

Determinamos usar MySQL porque era la base de datos relacional de código abierto más popular del mundo. También, porque los integrantes ya habíamos usado esta base de datos relacional. Adicionalmente, al ser un referente en lo que a este tipo de base de datos se refiere, consideramos que la cantidad de documentación y desarrollo por parte de la comunidad será extremadamente grande. Esto nos facilita mucho nuestro propio desarrollo y nos olvidamos del tema de compatibilidades ya que casi todos los lenguajes tienen soporte para MySQL[2].

6.1.4. MongoDB

Los integrantes del proyecto solo conocíamos como bases de datos no relacionales MongoDB [4], por lo que decidimos usar esta base de datos para demostrar que el tipo de bases de datos es independiente del framework. Además, pensamos que la estructura de documentos JSON de MongoDB puede ser de extrema utilidad a la hora de realizar las conversiones entre consultas.

6.1.5. XML

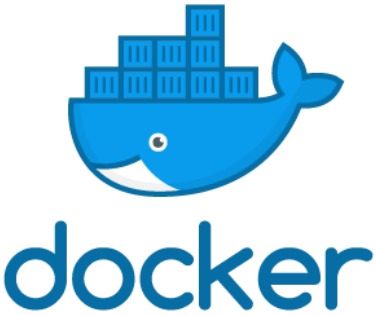
El presente lenguaje de marcas lo hemos usado para los mensajes de comunicación transmitidos a través del framework. Debido a su amplia utilidad en las comunicaciones entre aplicaciones hoy en día, no hemos dudado en usarla. También es una ventaja la facilidad con la que la información representada en xml puede ser tratada desde .NET o transformada a JSON.

6.1.6. Git

Tecnología utilizada para el control de versiones de la aplicación. Los integrantes ya habíamos usado con antelación esta herramienta en otras asignaturas, por lo que teníamos una mayor facilidad para usarlo. En particular, usamos un repositorio privado de la plataforma GitHub, basada en la tecnologia Git.

6.1.7. Docker

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de Virtualización a nivel de sistema operativo en Linux. Se trata de una herramienta de reciente creación pero de indudable utilidad en cuanto a los despliegues se refiere. Vimos el potencial que tenía esta tecnlogía y nos decididmos a usarla para el despliegue de los nodos en cloud.



6.1.8. Amazon Web Services

Como proveedor de Cloud hemos elegido Amazon Web Services porque los integrantes del proyecto lo hemos usado en una asignatura de la carrera. Entre los servicios que ofrece, hemos optado por usar Amazon RDS y Amazon EC2 [1]. El primer servicio nos permite tener una base de datos relacional en la nube, mientras que el segundo servicio nos permite desplegar contenedores Docker en la nube.

6.2. Tecnologías descartadas

A continuación vamos a enumerar aquellas tecnologías que hemos ido descartando durante la realización del proyecto.

6.2.1. Python

En un principio teníamos pensado usar este lenguaje de programación debido a su versatilidad y facilidad para conectarse a las bases de datos. Sin embargo, veíamos un incremento de la complejidad usar dos lenguajes a la vez, junto con C# , por lo que al final descartamos utilizar Python.

Capítulo 7

Casos de uso

Los dos casos de uso recogen las principales funcionalidades existentes dentro de la aplicación.

Para los casos de uso vamos a suponer que nuestra aplicación está formada por usuarios pertenecientes al ámbito sanitario, más concretamente, por administrador de sistemas pertenecientes a diferentes hospitales.

El usuario de referencia para los casos de uso es el "Hospital Jiménez Díaz".

7.1. Configuración de una base de datos propia

Vamos a suponer que el administrador de sistemas del Hospital Jiménez Díaz quiere configurar sus bases de datos en la aplicación, con el fin de que los demás hospitales pertenecientes a la aplicación puedan interactuar con sus bases de datos.
En primer lugar, accede a la aplicación e inicia sesión en ella. Una vez acceda, puede añadir/borrar/actualizar las bases de datos que posea gracias a la barra de iconos situada a la sección superior a la izquierda (Figura 7.1) o en el menú superior.

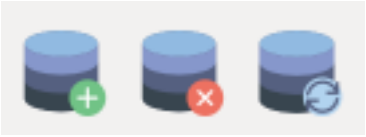


Figura 7.1: *Barra de iconos para las configuraciones de las bases de datos propias*

En las figuras 7.2 y 7.3 se muestran los diálogos de adicción y borrado de bases de datos.

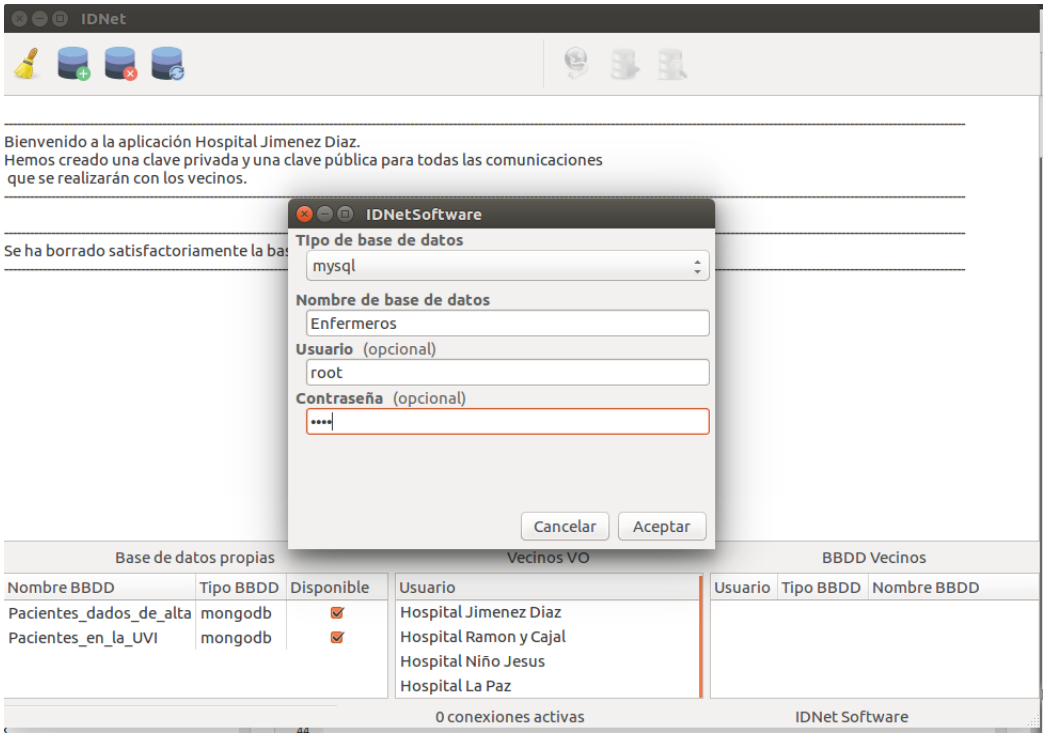


Figura 7.2: *Diálogo de adicción*

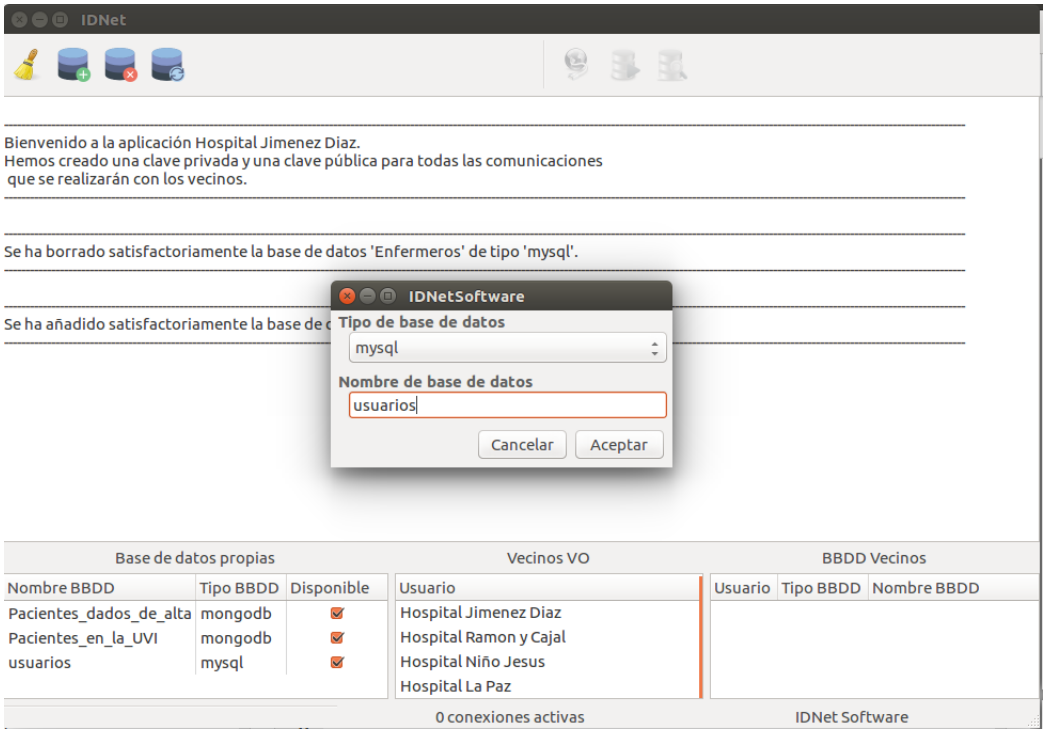


Figura 7.3: *Diálogo de borrado*

A continuación de realizar cualquier configuración, se mostrará un mensaje informativo sobre el éxito de ella. (Figura 7.4)

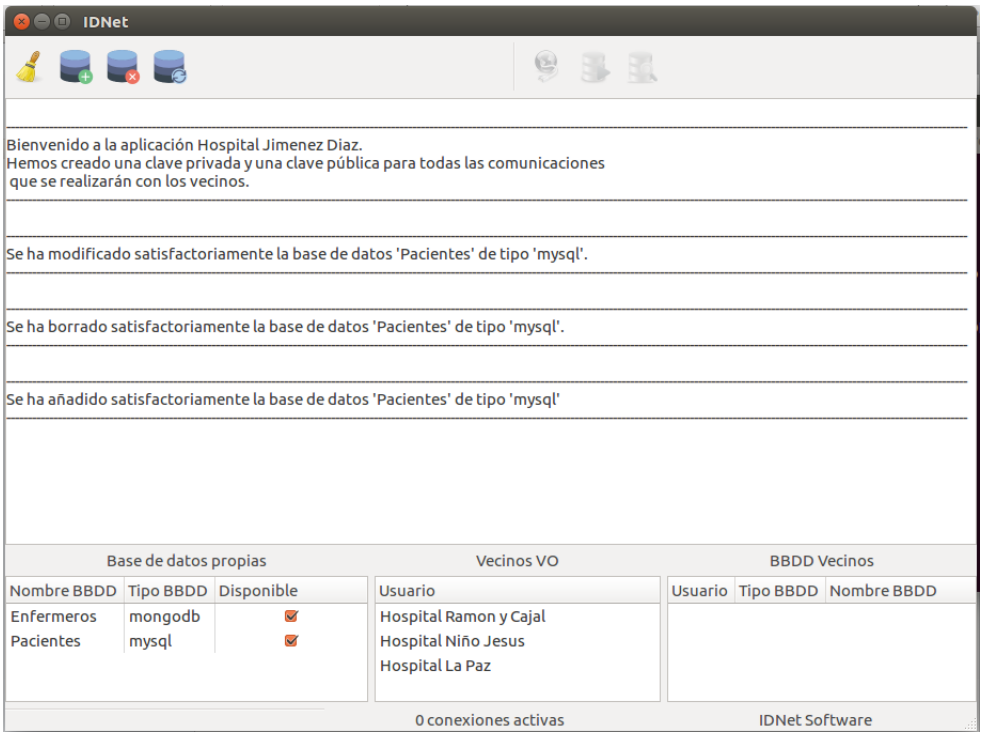


Figura 7.4: *Ejemplos de mensajes informativos*

7.2. Obtención de la información de una base de datos

Nuestro usuario de referencia quiere obtener información relativa a una base de datos del Hospital Ramón y Cajal, por ejemplo, quiere obtener los pacientes registrados del hospital. Para ello, partiendo del menú principal (Figura 7.6) deberá conectarse primero a dicho vecino (Figura 7.7), para posteriormente solicitar el esquema de la base de datos "Pacientes" (Figura 7.9).
De forma similar al caso de uso anterior, se dispone de un barra de iconos dedicada exclusivamente a la conexión, solicitud del esquema y a la realización de una consulta. (Figura 7.5)

Figura 7.5: *Barra de iconos para las comunicaciones en la red*

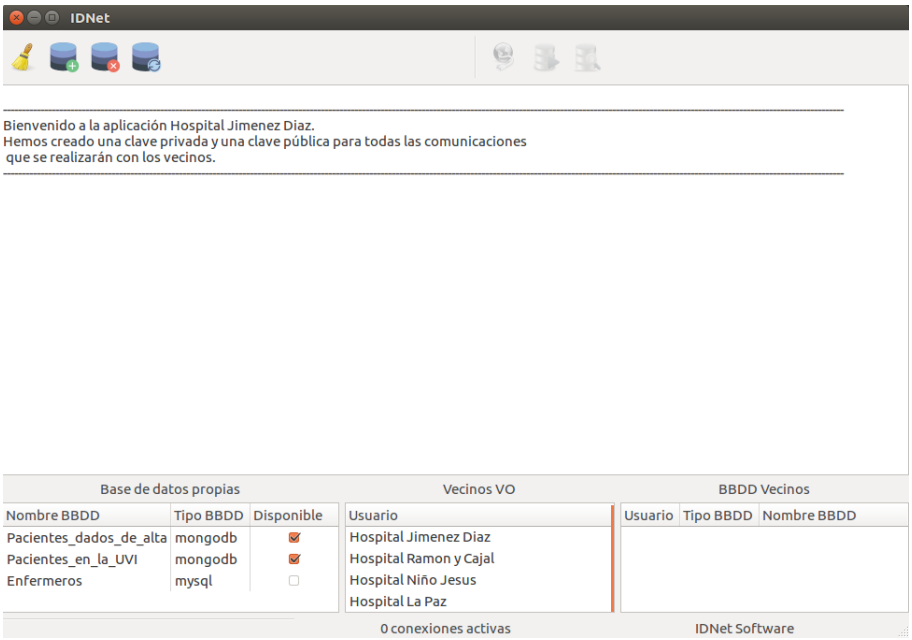


Figura 7.6: *Menú Principal*

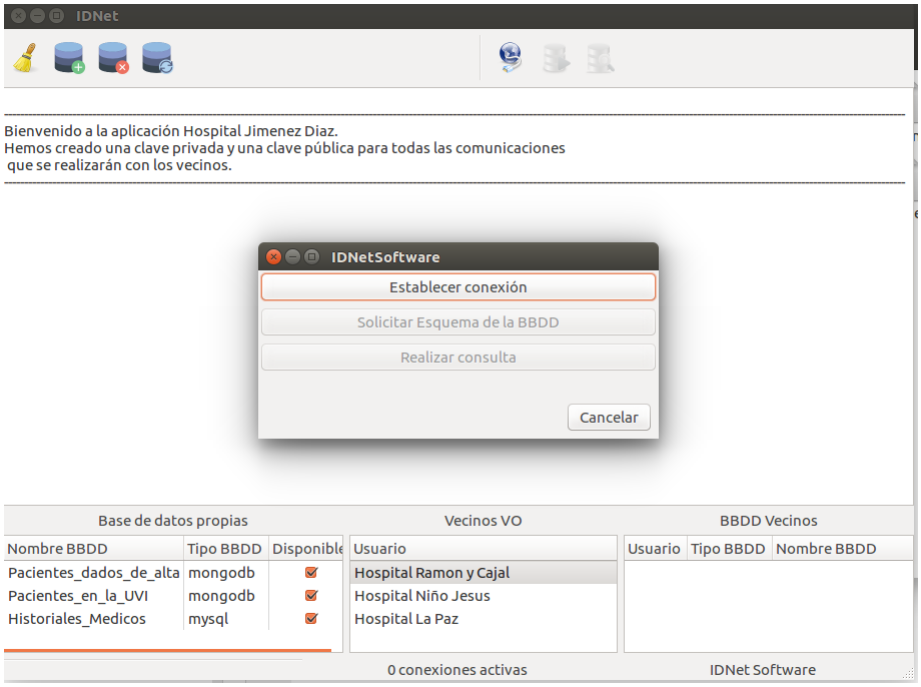


Figura 7.7: *Diálogo para solicitar conexión a un vecino*

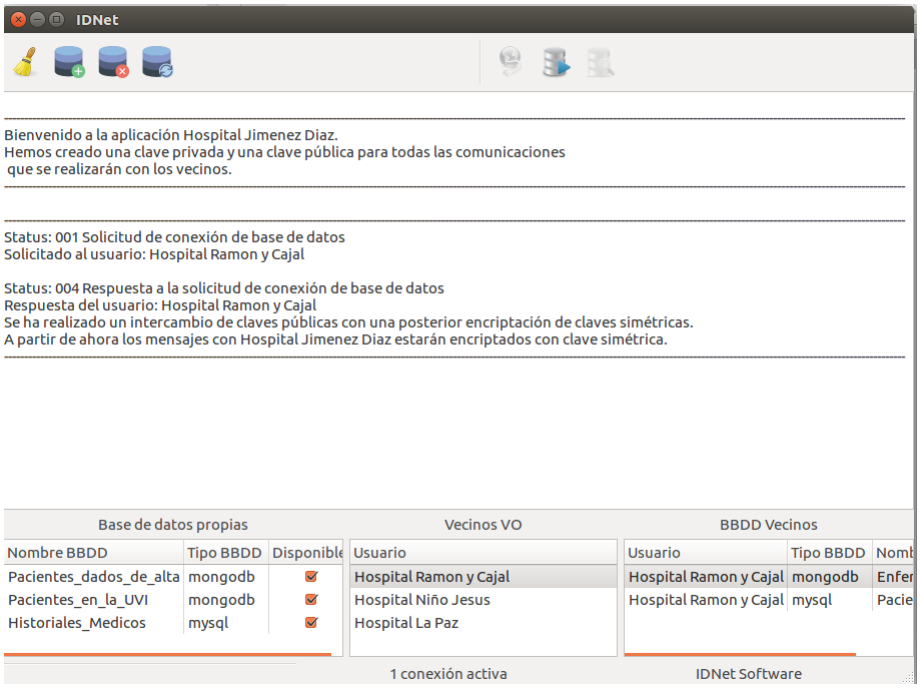


Figura 7.8: *Resultado de la conexión*

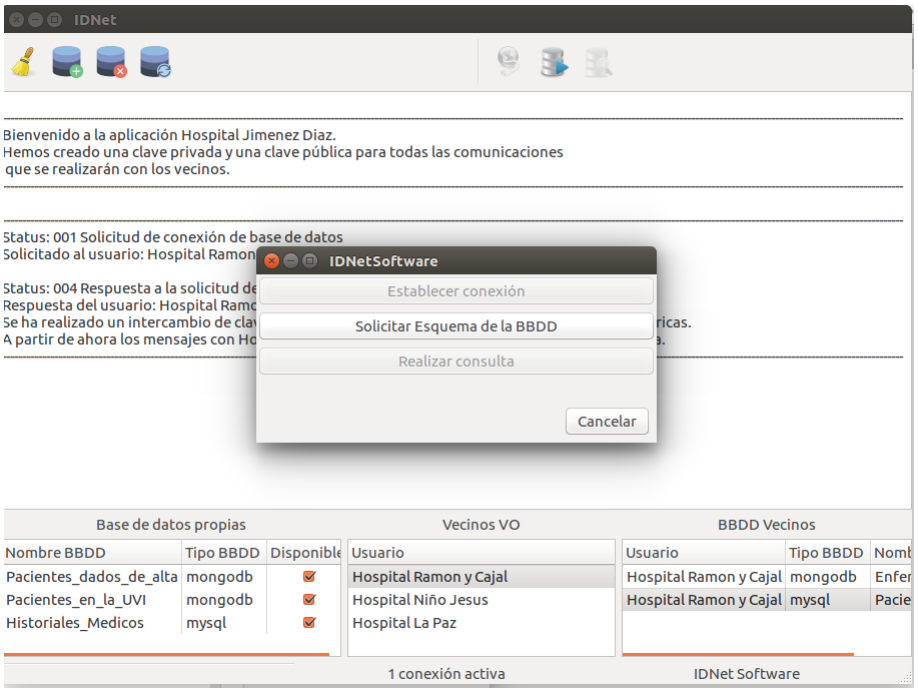


Figura 7.9: *Dialogo para solicitar el esquema de la Base de Datos*

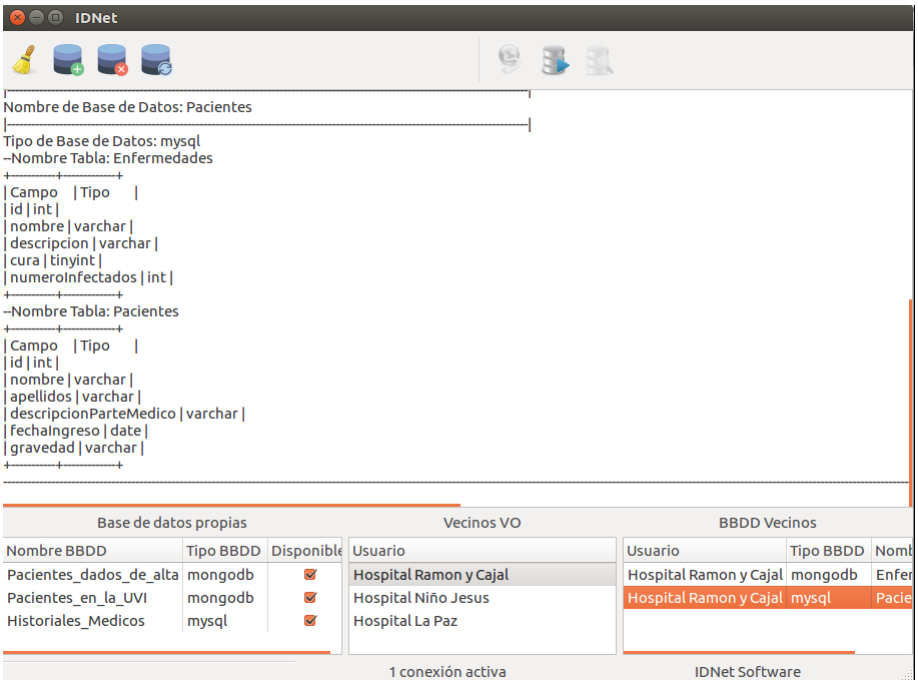


Figura 7.10: *Selección de la Base de Datos*

Una vez obtenido el esquema de la base de datos y meditado qué información quiere obtener, se dispone a realizar una consulta. (Figura 7.11).

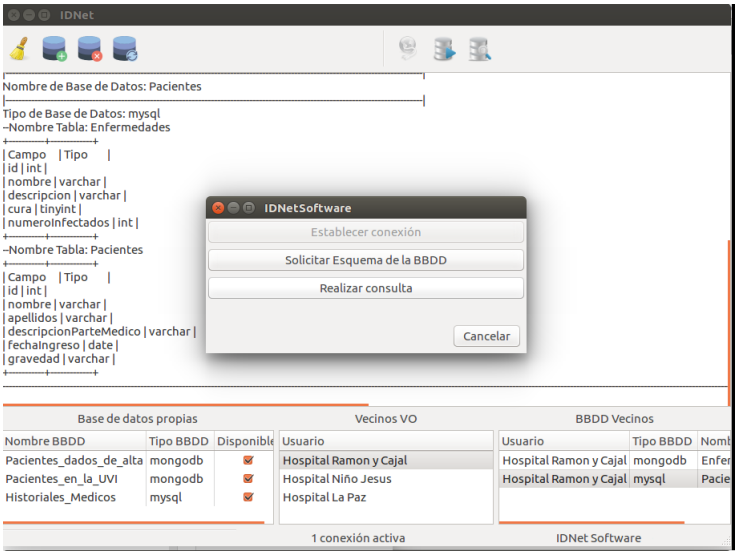


Figura 7.11: *Diálogo para iniciar una consulta*

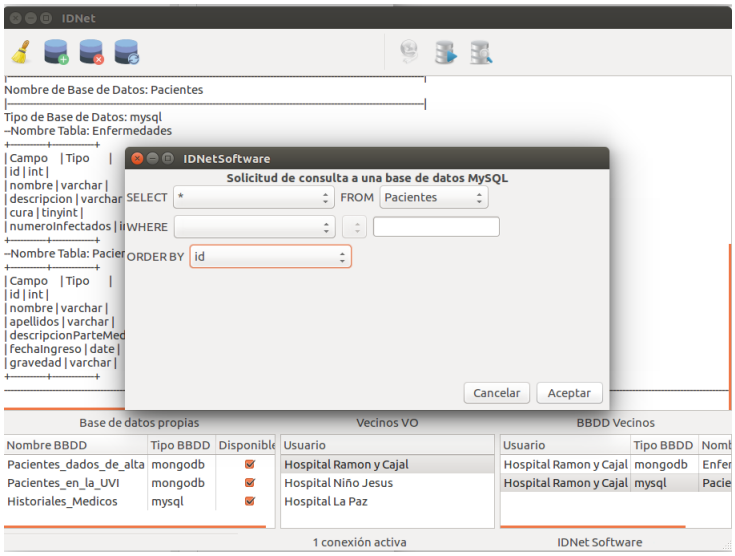


Figura 7.12: *Diálogo para definir la consulta*

Finalmente, obtiene la información requerida (Figura 7.13).

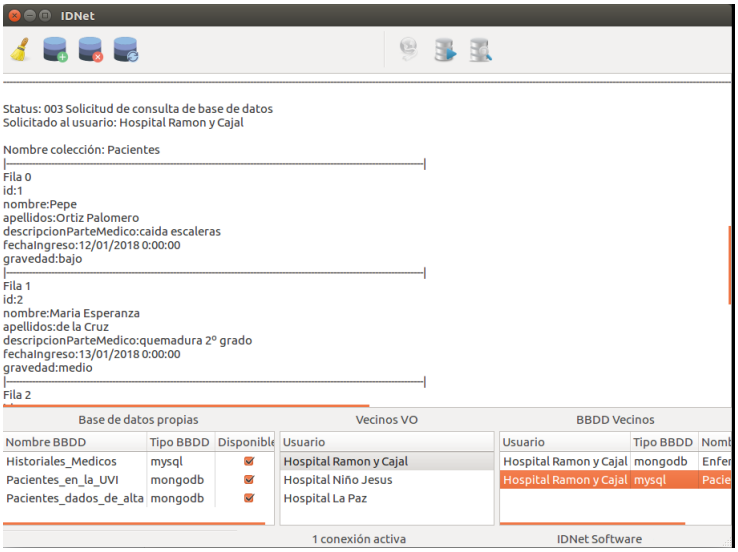


Figura 7.13: *Resultado de la Consulta*

Capítulo 8

Conclusiones

Nuestro objetivo principal ha sido permitir el intercambio de información entre bases de datos a través de un red anónima que asegure la integridad y privacidad de los datos transmitidos. Los usuarios podrán realizar consultas a otras bases de datos sin temor a que dichos datos puedan ser expuestos o interceptados por un agente tercero.

Existen muchas tecnologías que ya favorecen el intercambio de información —Freenet por ejemplo— pero ninguna que se aventure a abordar la comunicación directa entre bases de datos de distinta índole. Es por eso que nosotros hemos querido dar este primer paso en esta dirección.

IDNet es una primera aproximación a esta problemática. Con él, queremos abrir las infraestructuras críticas y favorecer que puedan aprovechar los numerosos recursos que Internet puede poner a su disposición. En esta versión dejamos el camino preparado para que IDNet pueda establecer Organizaciones Virtuales, conexiones a través de VPN, etc.

Personalmente, nos sentimos orgullosos de poder "iniciar" esta línea de trabajo. El proyecto nos ha satisfecho tanto personal como técnicamente y consideramos que, aunque el proyecto está finalizado, dejamos allanado el camino para que pueda crecer y poder soportar muchas más funcionalidades. Desde nuestro punto de vista, consideramos que ha sido muy provechoso. Hemos aprendido a usar .NET y su versión libre, Mono. También hemos aprendido a manejar contenedores Docker y a crear procesos independientes e interrelacionados que puedan funcionar en un entorno de Producción adecuado.

Conclusions

During this project, we aimed to allow the information exchange between databases through an anonymous network while guaranteeing the integrity and privacy of the transmitted data. Users will be able to query other databases without fearing data to be exposed or intercepted by any third agent.

There already exist many other technologies which assist the information exchange —e.g., Freenet— but no one establishes direct communication between different kinds of databases. That is why we wanted to take the first step in this direction.

IDNet is the first approximation to this problem. With it, we want to open critical infrastructures and allow them to take advantage of the huge number of resources that Internet can provide them. Therefore, this version lets the path opened to IDNet to grow even more by adding new functionalities such as Virtual Organizations, VPN connections, etc.

Personally, we feel proud for starting this working line. The project has satisfied us both, personally and professionally. We consider that, although the project is already finished, it still has a lot of potential. From our point of view, we think that it has been fruitful. We have learnt how to use .NET and its open version, Mono. Moreover, we have also learnt about Docker containers and creating independent and interrelated processes which could work within a proper Production environment.

Capítulo 9

Ampliación futura

A través del presente capítulo se van a detallar aquellas mejoras que se pueden aplicar al proyecto en el futuro. Muchas de ellas nos han ido surgiendo durante el desarrollo del proyecto y otras han sido fruto de un razonamiento posterior al desarrollo.

9.1. Creación de Organizaciones Virtuales

Para este trabajo no ha sido posible establecer un protocolo claro y una implementación precisa de las Organizaciones Virtuales. No obstante, sí que fue una de las ideas originales del proyecto y uno de los puntos que más podrían reforzar la aplicación.

Cuando hablamos de Organizaciones Virtuales nos referimos básicamente a redes privadas de acceso restringido. Esto favorece el uso controlado de la red y el aislamiento de la misma. Esta característica está especialmente pensada para, por ejemplo, infraestructuras críticas u organizaciones que traten datos de carácter sensible. Mucha gente considera que, en un mundo hiperconectado como el que tenemos hoy en día, la forma más eficaz de proteger nuestros datos es no tener acceso a Internet. Si yo no tengo acceso a Internet, Internet no tiene acceso a mí. Como es lógico dejamos a un lado las vulnerabilidades analógicas, humanas y a nivel hardware pues se encuentran más allá del ámbito de este trabajo.[5]

Sin embargo, nuestra idea es permitir aprovechar la información. Mediante las organizaciones virtuales somos capaces de comunicar los nodos sin que estos deban exponerse directamente Internet. Literalmente estamos creando un entorno *"sandbox"* privado para que puedan comunicarse de forma segura.

Al ser una de las ideas iniciales del proyecto, la topología de la red y la arquitectura de los nodos están preparados para soportarlas. Por otra parte, esto plantea nuevos retos que explicaremos a continuación.

9.2. Nodos Administradores

Actualmente, muchas de funcionalidades secundarias de securización y coordinación están programadas sobre los propios nodos GateKeeper. Esto es poco eficiente y presentaría un cuello de botella importante si atendemos a la escalabilidad de la red. Además serían necesarios para una correcta implementación de las Organizaciones Virtuales.

Para eso diseñamos la idea de los Nodos Administradores. La red es, a todos los efectos, P2P, pero usa una topología mixta. Con esto nos referimos a que los nodos clientes conforman por definición una red descentralizada. Sin embargo, por otro lado, la coordinación de los nodos GateKeeper dentro de la red debería ser monitorizada y controlada por un Nodo Administrador privado que serviría como "apoyo logístico" de los GateKeeper. Algunas de las funciones que asumiría sería la autorización de nuevos nodos cliente en la Organización Virtual, la identificación de los mismos al inicio de la conexión o la definición de las propias Organizaciones. Los nodos Administradores serían completamente independientes unos de otros, aislando de esta forma las redes por completo.

9.3. BlackList

Una de las mejoras que mejorarían sensiblemente la seguridad de los accesos a las BBDD y la flexibilidad del servicio consistirían en la creación de una lista negra. De esta forma, los propios clientes pueden definir accesos clasificados para cada una de las bases de datos que tengan registradas en IDNet.

La lista negra permitiría a un administrador poder seleccionar que miembros de su Organización Virtual tienen acceso a qué bases de datos aumentando así el nivel de control.

Esta idea es también aplicable a la creación de un sistema de roles o perfiles determinados de usuarios que usen la aplicación. La idea es poder dar al cliente el soporte y la capacidad necesarios para definir su propio sistema de permisos y tener controlado en todo momento que identidades tienen acceso a la base de datos.

9.4. Aumento de las bases de datos compatibles

IDNet no está pensado para limitarse a conectar MySQL con MongoDB y viceversa, sino para solventar ese gran problema que resulta la comunicación entre bases de datos de distinta índole. Es por eso que el diseño modular del nodo Cliente está especialmente diseñado para facilitar la inserción de nuevos tipos de base de datos. A su vez, el uso de lenguajes semiestructurados de transmisión de datos como XML o JSON, favorece enormemente la comunicación. En definitiva, la mayor mejora que IDNet puede obtener es el soporte para más tipos de bases de datos.

9.5. Virtual Private Net

Inicialmente, consideramos que las conexiones entre un nodo Cliente y un nodo GateKeeper debían ir a través de una VPN prefijada entre ambos nodos. Esto resolvería multitud de vulnerabilidades que se pudieran encontrar, eliminando completamente aquellas que tuvieran que ver con ataques MITM[1].

Por desgracia, tuvimos que aplazar la implementación de esta idea debido a la dificultad que entrañaba el uso de VPN's con Mono.

9.6. Servicio Web

Con el tiempo, nos dimos cuenta de que debíamos procurar que la configuración de los nodos Cliente fuera lo más transparente posible para el usuario. Es por eso que ciertas configuraciones incialmente debían ser llevadas a cabo por un servicio de configuración externo.

Este servicio permitiría a los usuarios dar de alta Organizaciones Virtuales y nuevos usuarios asociados a ellas. Además, permitiría configurar un cliente con los módulos de conexión a BBDD que ellos prefieran (no tiene sentido que un cliente tenga un módulo de conexión a MongoDB si no usa dicha base de datos). Estas medidas fueron pensadas con la escalabilidad en mente.

Al mismo tiempo, permitiría comunicarse con los nodos Administrados y asignar dinámicamente una conexión fija a un nodo GateKeeper de la red. De esta forma el usuario no tiene que configurar la conexión, sino que desde el primer momento, dicho nodo ya tiene configurado y asignado un nodo con el que comunicarse.

[1] Man in the Middle

Capítulo 10

División del trabajo

En este capítulo describiremos el trabajo que ha desempeñado cada miembro del equipo IDNet en la elaboración de este trabajo.

Previamente a lo que a continuación expondremos, ambos integrantes han trabajado en conjunto a lo largo de todo el proceso de concepción, desarrollo, implementación y documentación del proyecto. En ese sentido, el trabajo se ha repartido de forma equitativa y, en muchas ocasiones, se ha trabajado en conjunto.

10.1. Lorenzo José de la Paz Suárez

Durante el comienzo del proyecto Lorenzo José aportó ideas relacionadas tanto con las bases de datos, como con la forma en que la que se podrían comunicar entre ellas.

Una vez que la idea del proyecto fue fijada por ambos integrantes junto con el director del Trabajo de Fin de Grado, Lorenzo José comenzó con las conexiones a las bases de datos desde el IDE que hemos utilizado: MonoDevelop. Se encargo posteriormente de todo lo relacionado con la comunicación entre el nodo y la base de datos.

Lorenzo José se encargó de terminar el nodo Cliente ya que tenía una mayor familiaridad con la comunicación con las bases de datos, una de las funcionalidades más importantes del nodo Cliente.

La implementación de la encriptación en la aplicación fue llevada a cabo por Lorenzo José. Se encargó de codificar los diferentes sistemas de encriptación y los módulos correspondientes.

Lorenzo José realizó a nivel de implementación el protocolo de comunicación y todo lo concerniente a los diferentes mensajes enviados entre los dos tipos de nodos en la red.

Junto con Juan, Lorenzo José ha trabajado en el despliegue de los nodos GateKeeper usando contenedores Docker en Amazon Web Services.

Y, por último, Lorenzo José ha aportado de forma equitativa a la redacción de la memoria, escribiendo más concretamente en las secciones en las que ha trabajado.

Figura 10.1: *Lorenzo José de la Paz Suárez*

10.2. Juan Mas Aguilar

Desde el comienzo del proyecto Juan aportó ideas que ayudaron a la concepción del proyecto en su fase inicial. También investigó sobre los tipos de tecnologías que se deberían llevar a cabo y propuso ideas para los diseños iniciales.

Juan diseñó, junto con Lorenzo José, los protocolos iniciales de comunicación entre nodos y estableció los protocolos de encriptación y el establecimiento seguro de la comunicación entre nodos. Trabajó en el nodo Cliente hasta desarrollar la primera versión estable.

Desarrolló a su vez el nodo GateKeeper con ayuda de las ideas y consejos aportados por Lorenzo José. Estableció el protocolo de enrutado de mensajes y los diferentes archivos de configuración. Junto con Lorenzo José, se encargó de acordar el sistema de alias para los nodos Cliente y sus correspondencias en el GateKeeper.

Por último, Juan trabajó en la memoria cumplimentando los apartados que había realizado y otros que le correspondieron en la división de los capítulos. Ayudó y trabajo a su vez en la investigación sobre el uso de Docker y la creación de las imágenes que luego se subirían a AWS.

Figura 10.2: *Juan Mas Aguilar*

Capítulo 11

Código Fuente

El código fuente de la implementación de IDNet se ha almacenado en un repositorio público en Github. En la siguiente dirección web aparece la página de información sobre el proyecto:
https://lorenpaz.github.io/IDNet/

En la siguiente dirección web se encuentra el código de la implementación:
https://github.com/lorenpaz/IDNet/releases

Bibliografía

[1] Inc Amazon Web Services. Documentacion de aws ecs. https://aws.amazon.com/es/documentation/ecs/. Accessed: 2018-05-09.

[2] Mikael Chudinov. Connect c to mysql. https://www.codeproject.com/Articles/43438/Connect-C-to-MySQL. Accessed: 2018-05-09.

[3] Mikael Chudinov. Demonisation of a .net mono application on linux. http://blog.chudinov.net/demonisation-of-a-net-mono-application-on-linux/. Accessed: 2018-05-09.

[4] Rubén Fernández. Tutorial mongodb y c: Conexión a la base de datos. http://charlascylon.com/2013-10-23-tutorial-mongodb-y-c-conexion-a-la-base-de-datos. Accessed: 2018-05-09.

[5] Ian Foster, Carl Kesselman, and Steven Tuecke. "The Anatomy of the Grid". Technical report, Information Science Institute, 05 2001.

[6] Inc. MongoDB. C# ecosystem for mongodb. https://docs.mongodb.com/ecosystem/drivers/csharp/. Accessed: 2018-05-09.

[7] Smashicons.com. Licencias de los iconos de la aplicación diseñadas por smashicons desde flaticon. https://smashicons.com/. Accessed: 2018-05-09.

[8] Bill Wagner, Olprod, OpenLozalizationService, and Saisang Cai. C# programming guide. https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/. Accessed: 2018-05-09.

Apéndice A

Ejemplos de mensajes de comunicación

Para los ejemplos de mensajes de comunicación, se va a suponer el siguiente escenario: Tenemos un nodo Cliente que ha iniciado sesión con el usuario "Hospital Jiménez Díaz".

En la aplicación tenemos a tres usuarios aparte del anterior mencionado:

- Hospital Ramón y Cajal: con una base de datos mysql llamada "Pacientes" y otra "Enfermeros".
- Hospital Niño Jesús.
- Hospital La Paz.

En los siguientes mensajes se va a poder comprobar como el usuario solicita información a un vecino de la aplicación.

Mensajes 001

Mensaje 001a

```
<root>
  <message_type>001a</message_type>
  <source>Hospital Jiménez Díaz</source>
  <destination>Hospital Ramón y Cajal</destination>
  <key>-----BEGIN PUBLIC KEY-----
    MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhz6QUpCw7dmPIUaUUqAI
    U9fb044GOnkbGvhI84/InIgsb+fH9umEZGinfwrNb68KCAufjLtWjNlSEqXfXLe/
    S1e93f88UzgIhZ428pnpvNMhJhpfUVxbllKX9OdD3CHVf1dEDaPOIMZ/vrVPOg7h
    7w0YXF9Mlns6Wl/CBBTDmaYKwK8Jvb12kSlG2Duwxce2JgSIbc5CnCw6576mbQu8
    1gZUBI5vAH3o260yhEW2jPkHQm7siHQJUbG5lwcBEjYwEK3XRpEWB/bptm5hbGJM
    HiZ2ils+bdjHKP74Es59WCdm7CBZOhl6EGFz4/kgXMQh86l0Er7t6KYuFDW0LkzV
    uwIDAQAB
    -----END PUBLIC KEY-----
  </key>
</root>
```

Figura A.1: *Ejemplo de mensaje 001a*

Mensaje 001b

```
<root>
  <message_type>001b</message_type>
  <source>Hospital Jiménez Díaz</source>
  <destination>Hospital Ramón y Cajal</destination>
  <encripted>
    <key>ZxpVD6Kh6O6XKm/UAapSmiDOo78r8frb2DvaxqJCHI0=</key>
    <IV>lgk+a737CDL+Al5iktsmRw==</IV>
  </encripted>
</root>
```

Figura A.2: *Ejemplo de mensaje 001b*

Mensaje 002

```xml
<root>
  <message_type>002</message_type>
  <source>Hospital Jiménez Díaz</source>
  <destination>Hospital Ramón y Cajal</destination>
  <encripted>
    <db_name>Pacientes</db_name>
    <db_type>mysql</db_type>
  </encripted>
</root>
```

Figura A.3: *Ejemplo de mensaje 002*

Mensajes 003

Mensaje 003 formato MySQL

```xml
<root>
  <message_type>003</message_type>
  <source>Hospital Jiménez Díaz</source>
  <destination>Hospital Ramón y Cajal</destination>
  <encripted>
    <db_name>Pacientes</db_name>
    <db_type>mysql</db_type>
    <body>
      <query>
      <select>*</select>
      <from>pacientes</from>
      <where></where>
      <orderby>id</orderby>
      </query>
    </body>
  </encripted>
</root>
```

Figura A.4: *Ejemplo de mensaje 003 para MySQL*

Mensaje 003 formato MongoDB

```
<root>
  <message_type>003</message_type>
  <source>Hospital Ramón y Cajal</source>
  <destination>Hospital Jiménez Díaz</destination>
  <encripted>
    <db_name>Enfermeros</db_name>
    <db_type>mongodb</db_type>
    <body>
      <query>
        <collection>Enfermeros Planta Primera</collection>
        <filter />
        <projection nombre="1" apellidos="1" contrato="0" telefono="1" />
        <sort>apellidos</sort>
        <limit>10</limit>
      </query>
    </body>
  </encripted>
</root>
```

Figura A.5: *Ejemplo de mensaje 003 para MongoDB*

Mensajes 004

Mensaje 004a

```
<root>
  <message_type>004a</message_type>
  <source>Hospital Ramón y Cajal</source>
  <destination>Hospital Jiménez Díaz</destination>
  <key>-----BEGIN PUBLIC KEY-----
  MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwnfuskE50kCzxkKxZwGS\ne/jU
  sf4KIfj5tnDbyAx1YvyCA+jETgCO4bxaUsjYiEIYoBiexo+ehTeJLF50QeUx\nyLo0D++v
  w3Fjj3aiQfIo1mABS86eg85hUwKvtP4TrYj42QDNaA9CD/Vm16uGhXQO\nwSthzvOv2Gmz
  qjnz/JWEG8cyudFJDJBlnZERygGK8eCi26p6X90+Ru7ZFtQykzDq\n0Vbud4DYd9Nnz9XH
  cqNl6E6RsobW/iZjRzmTX4ULj0bGgIBuG4Hr04LlMtzk4/s7\nEUc5bKH+e7dxYypgVQmv
  tvVFtmTnJf0fBPhjCYcavzKZc2lUyyp1VU6ELbOFBi0h\nkQIDAQAB
  -----END PUBLIC KEY-----
  </key>
</root>
```

Figura A.6: *Ejemplo de mensaje 004a*

Mensaje 004b

```xml
<root>
  <message_type>004b</message_type>
  <source>Hospital Ramón y Cajal</source>
  <destination>Hospital Jiménez Díaz</destination>
  <encripted>
    <databases>
      <database db_type="mysql">Pacientes</database>
      <database db_type="mongodb">Enfermeros</database>
    </databases>
  </encripted>
</root>
```

Figura A.7: *Ejemplo de mensaje 004b*

Mensajes 005

Mensaje 006 para MySQL

```xml
<root>
  <message_type>005</message_type>
  <source>Hospital Ramón y Cajal</source>
  <destination>Hospital Jiménez Díaz</destination>
  <encripted>
    <db_name>Pacientes</db_name>
    <db_type>mysql</db_type>
    <body>
      <database name="pacientes">
        <table name="pacientes">
          <col name="id" type="int" />
          <col name="nombre" type="varchar" />
          <col name="apellidos" type="varchar" />
          <col name="descripcionParteMedico" type="varchar" />
          <col name="fechaIngreso" type="date" />
          <col name="gravedad" type="varchar" />
        </table>
        <table name="enfermedades">
          <col name="id" type="int" />
          <col name="nombre" type="varchar" />
          <col name="descripcion" type="varchar" />
          <col name="cura" type="boolean" />
          <col name="numeroInfectados" type="int" />
        </table>
        <table name="relacionPacientesEnfermedades">
          <col name="idPaciente" type="int" />
          <col name="idEnfermedad" type="int" />
        </table>
      </database>
    </body>
  </encripted>
</root>
```

Figura A.8: *Ejemplo de mensaje 005 para MySQL*

Mensaje 006 para MongoDB

```xml
<root>
  <message_type>005</message_type>
  <source>Hospital Ramón y Cajal</source>
  <destination>Hospital Jiménez Díaz</destination>
  <encripted>
    <db_name>Enfermeros</db_name>
    <db_type>mongodb</db_type>
    <body>
      <result>
        <database>
          <name>Enfermeros</name>
          <colecciones>
            <name>Enfermeros Planta Primera</name>
            <ejemploColeccion>
              <nombre>String</nombre>
              <apellidos>String</apellidos>
              <contrato>Boolean</contrato>
              <telefono>Object</telefono>
            </ejemploColeccion>
          </colecciones>
          <colecciones>
            <name>Enfermeros Planta Segunda</name>
            <ejemploColeccion>
              <nombre>String</nombre>
              <apellidos>String</apellidos>
              <contrato>Boolean</contrato>
            </ejemploColeccion>
          </colecciones>
        </database>
      </result>
    </body>
  </encripted>
</root>
```

Figura A.9: *Ejemplo de mensaje 005 para MongoDB*

Mensajes 006

Mensaje 006 para MySQL

```xml
<root>
  <message_type>006</message_type>
  <source>Hospital Ramón y Cajal</source>
  <destination>Hospital Jiménez Díaz</destination>
  <encripted>
    <db_name>Pacientes</db_name>
    <db_type>mysql</db_type>
    <body>
      <result table="pacientes">
        <row id="1" nombre="Pepe" apellidos="Ortiz Palomero" descripcionParteMedico="caida escaleras" fechaIngreso="12-01-2018" gravedad="bajo" />
        <row id="2" nombre="Maria Esperanza" apellidos="de la Cruz" descripcionParteMedico="quemadura 2º grado" fechaIngreso="13-02-2018" gravedad="medio" />
        <row id="3" nombre="Marino" apellidos="Hortelano Suárez" descripcionParteMedico="routa talón de Aquiles" fechaIngreso="09-03-2018" gravedad="bajo" />
        <row id="4" nombre="Lorenzo" apellidos="de la Cruz" descripcionParteMedico="parada cardiaca" fechaIngreso="17-04-2018" gravedad="alto" />
        <row id="5" nombre="Juan" apellidos="Ortiz Palomero" descripcionParteMedico="pulmón dañado" fechaIngreso="22-05-2018" gravedad="alto" />
        <row id="6" nombre="David" apellidos="Aguilar " descripcionParteMedico="costillas rotas" fechaIngreso="02-03-2018" gravedad="medio" />
      </result>
    </body>
  </encripted>
</root>
```

Figura A.10: *Ejemplo de mensaje 006*

Mensaje 006 para MongoDB

```
<root>
  <message_type>006</message_type>
  <source>Hospital Ramón y Cajal</source>
  <destination>Hospital Jiménez Díaz</destination>
  <encripted>
    <db_name>Enfermeros</db_name>
    <db_type>mongodb</db_type>
    <body>
      <result>
        <row>
          <nombre>Juan</nombre>
          <apellidos>de la Vega Torrejo</apellidos>
          <telefono>
            <fijo>915472612</fijo>
            <movil>678432156</movil>
          </telefono>
        </row>
        <row>
          <nombre>María</nombre>
          <apellidos>Pedrolas Saavedra</apellidos>
          <telefono>
            <fijo>915492511</fijo>
            <movil>722342776</movil>
          </telefono>
        </row>
      </result>
    </body>
  </encripted>
</root>
```

Figura A.11: *Ejemplo de mensaje 006 para MongoDB*

Mensaje 010

```
<root>
  <message_type>010</message_type>
  <source>Hospital Ramón y Cajal</source>
  <ip>88.13.193.37</ip>
  <destination>172.16.0.23</destination>
  <code>5674967</code>
</root>
```

Figura A.12: *Ejemplo de mensaje 010*

Mensaje 011

```
<root>
  <message_type>011</message_type>
  <source>Hospital Ramón y Cajal</source>
  <ip>88.13.193.37</ip>
  <destination> 172.16.0.23</destination>
</root>
```

Figura A.13: *Ejemplo de mensaje 011*

Apéndice B

Ejemplos de ficheros de soporte

Los ficheros de soporte los vamos a dividir en función de la arquitectura a la que pertenecen.

De forma similar al Anexo A, se va a suponer que tenemos un nodo Cliente que ha iniciado sesión con el usuario "Hospital Jiménez Díaz". Presente tres bases de datos:

- Pacientes_dados_de_alta: MongoDB
- Pacientes_en_la_UVI: MongoDB
- Historiales_Médicos: MySQL

En la aplicación tenemos a tres usuarios aparte del anterior mencionado:

- Hospital Ramón y Cajal: con una base de datos MySQL llamada "Pacientes" y otra base de datos MongoDB "Enfermeros".
- Hospital Niño Jesús: con 4 bases de datos: "Pacientes Planta 1", "Pacientes Planta 2", "Pacientes Planta 3" y "Pacientes Planta 4". Todas son de tipo MySQL.
- Hospital La Paz: con una base de datos llamada "Pacientes en urgencias" .

Ficheros de la arquitectura Cliente

En esta sección se mostrarán los ficheros de configuración tanto de IDNetDaemon como de IDNetSoftware. Como ya se ha comentado en las secciones 5.1.1 y 5.1.2, ambos comparten los siguientes ficheros: *info.conf* y *databases.conf*.

Ejemplo fichero *info.conf*

```
nombre=administrador Hospital Jiménez Díaz|code:3456789;
```

Figura B.1: *Ejemplo de fichero de configuración info.conf*

Ejemplo fichero *databases.conf*

```
mongodb*Pacientes dados de alta|administrador*AozJRMGruo2R7y+UfNwUbA==;
mongodb*Pacientes en la UVI|administrador*wv9ta4GXZ3hVAUxjT5hS9lE9zHusJnQu+vFiLh8OP5A=;
mysql*Historiales médicos|administrador*aqoifJViwfm0loYbjXpanMwQA==;
```

Figura B.2: *Ejemplo de fichero de configuración databases.conf*

Ejemplo fichero *neighbours.conf*

```
Hospital Ramón y Cajal;
Hospital Niño Jesús;
Hospital La Paz;
```

Figura B.3: *Ejemplo de fichero de configuración neighbours.conf*

Ejemplo fichero *neighboursDatabases.conf*

```
Hospital Ramón y Cajal=mysql,Pacientes;
Hospital Ramón y Cajal=mongodb,Enfermeros;
Hospital Niño Jesús=mysql,Pacientes Planta 1;
Hospital Niño Jesús=mysql,Pacientes Planta 2;
Hospital Niño Jesús=mysql,Pacientes Planta 3;
Hospital Niño Jesús=mysql,Pacientes Planta 4;
Hospital La Paz=mongodb,Pacientes de urgencias;
```

Figura B.4: *Ejemplo de fichero de configuración neighboursDatabases.conf*

Ficheros de la arquitectura en la Nube

Ejemplo fichero *neighbours.xml*

```
<?xml version="1.0" encoding="utf-8" ?>
<neighbours>
  <node>192.168.0.1</node>
  <node>192.168.0.2</node>
</neighbours>
```

Figura B.5: *Ejemplo de fichero de configuración neighbours.xml*

Ejemplo fichero *routes.xml*

```
<routes>
  <route>
    <d_node>176.96.0.1</d_node>
    <d_hop>192.168.0.1</d_hop>
    <name>Hospital Quirón</name>
    <distance>3</distance>
  </route>
</routes>
```

Figura B.6: *Ejemplo de fichero de configuración routes.xml*

Apéndice C

Diagramas de Clases y de módulos

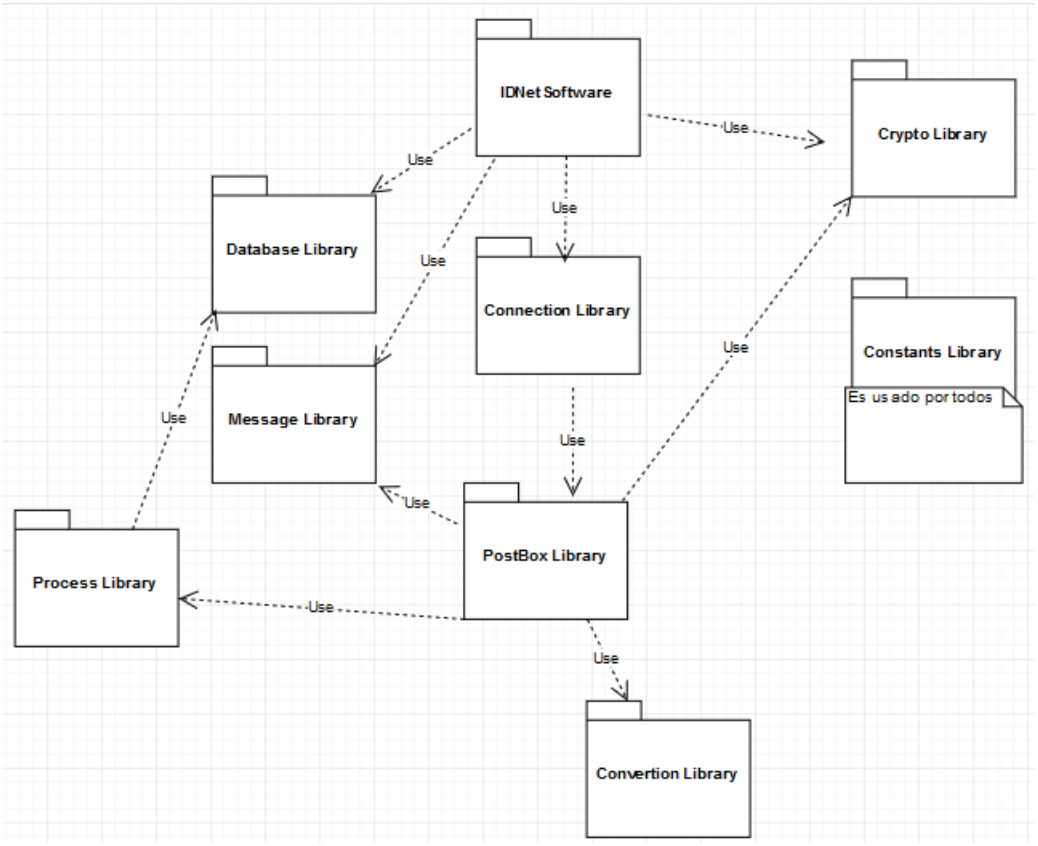


Figura C.1: *Diagrama de módulos IDNetSoftware*

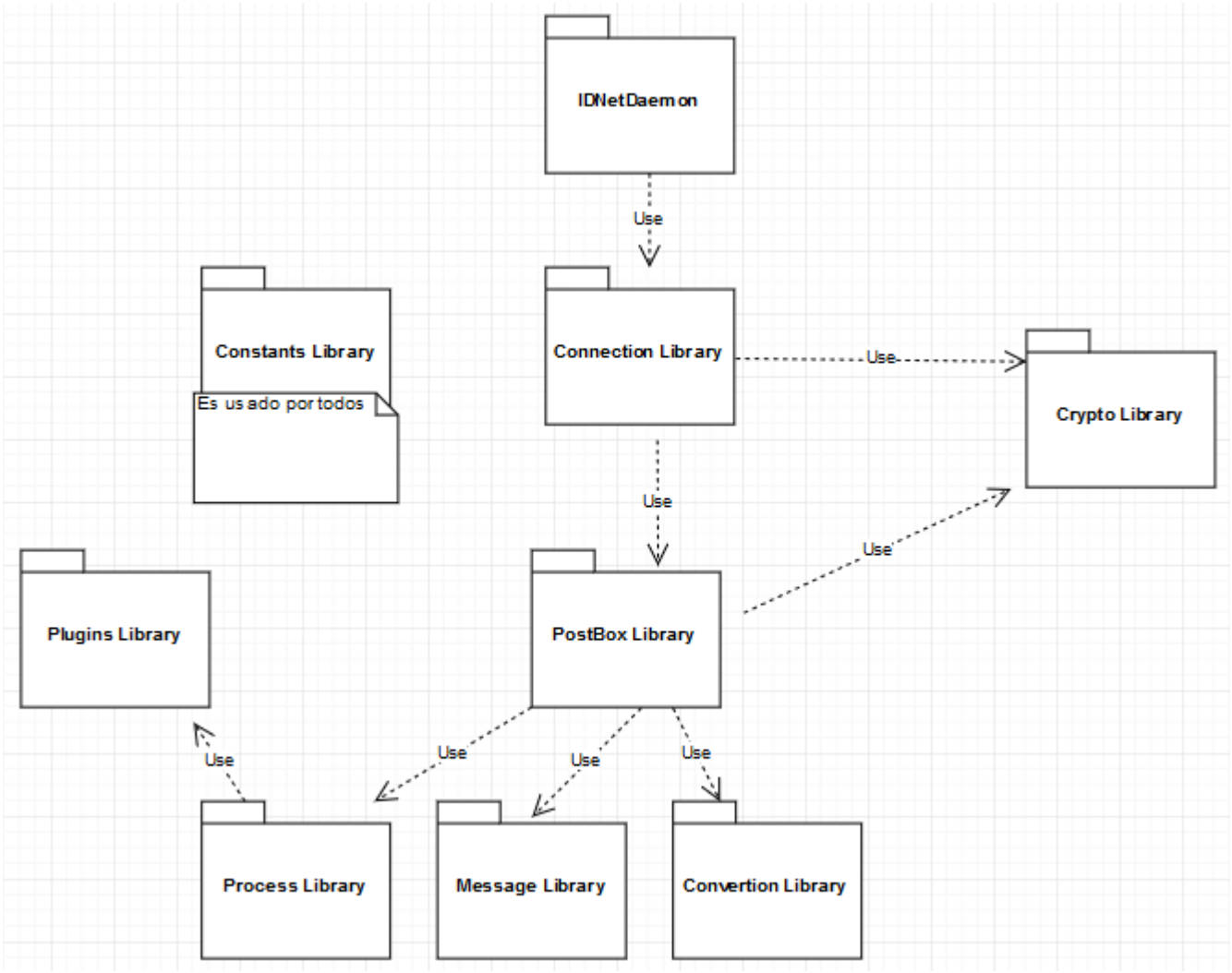


Figura C.2: *Diagrama de módulos IDNetDaemon*

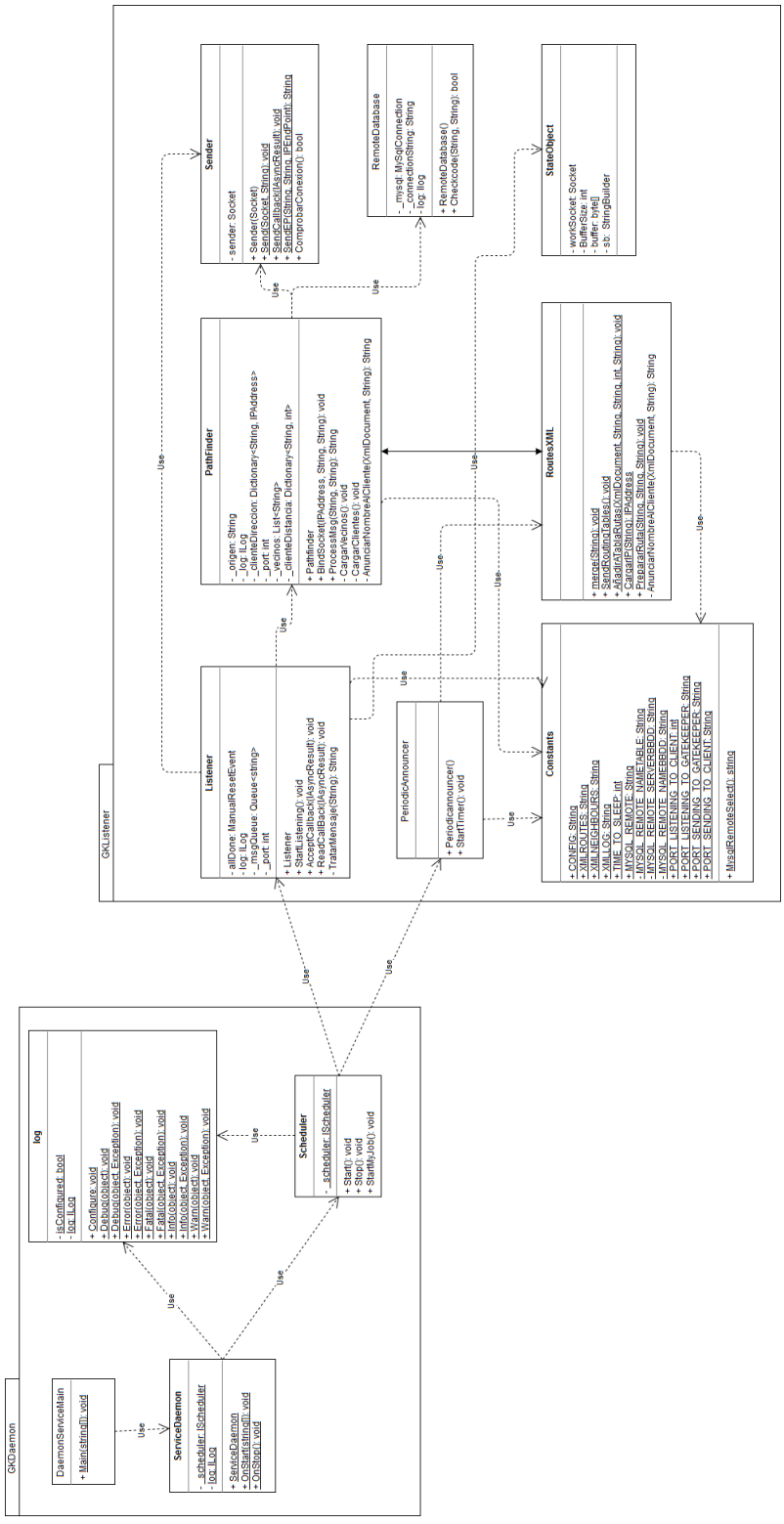


Figura C.3: *Esquema General de la arquitectura interna de los GateKeeper*

Apéndice D

Manual de usuario

La ejecución de la aplicación se realiza ejecutando el script launch.sh, el cuál lanza IDNetSoftware. IDNetDaemon se lanza sólo si no ha sido lanzado en previas ejecuciones.
A continuación, después de la ventana informativa del proyecto (Figura D.1), aparecerá el login de la aplicación (Figura D.2).

Figura D.1: *Ventana informativa sobre el proyecto*

Figura D.2: *Login de la aplicación*

Adicción de una base de datos en la aplicación

Para añadir una base de datos en la aplicación, en primer lugar se debe de acceder al diálogo de adicción de una base de datos. El acceso al diálogo mencionado se puede realizar mediante la barra de iconos (Figura D.3) o mediante el menú superior.
En el dialogo (Figura 7.2) se deberán rellenar los campos:

- Nombre de la base de datos
- Tipo de base de datos
- Usuario (Opcional)
- Contraseña (Opcional)

Figura D.3: *Icono de adicción*

Como se puede ver en la figura 7.4, se mostrará en el cuadro de mensajes si se ha realizado con éxito la adicción de la base de datos.

Borrado de una base de datos en la aplicación

Para el borrado una base de datos en la aplicación, en primer lugar se debe de acceder al diálogo de borrado de una base de datos. El acceso al diálogo mencionado se puede realizar mediante la barra de iconos (Figura D.4) o mediante el menú superior.
En el dialogo (Figura 7.3) se deberán rellenar los campos:

- Nombre de la base de datos
- Tipo de base de datos

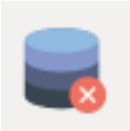


Figura D.4: *Icono de borrado*

Como se puede ver en la figura 7.4, se mostrará en el cuadro de mensajes si se ha realizado con éxito el borrado de la base de datos.

Actualización del estado de los servidores de la aplicación

Debido a que las bases de datos añadidas en la aplicación deben de estar disponibles, se comprueban si existen dichas bases de datos en sus determinados servidores.
Pulsando en el icono de Actualización de la barra de iconos (Figura D.5), se podrá realizar la actualización del estado de los servidores, avisando la aplicación al usuario del resultado de la acción (Figura D.6)

Figura D.5: *Icono de actualización*

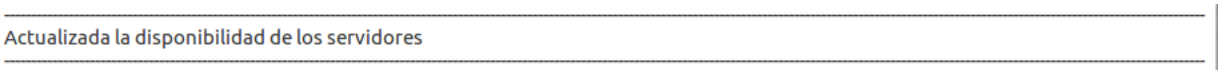


Figura D.6: *Mensaje informativo de la actualización del estado de los servidores*

Conexión a un vecino

Desde el menú principal (Figura 7.6), selecciona al vecinos de la lista "Vecinos OV" situada en la parte inferior central de la aplicación clickeando dos veces. A continuación aparecerá un cuadro de diálogo similar al de la figura D.7. Otra forma de llegar al cuadro de diálogo es acudiendo a la barra de herramientas y pulsar en el icono similar a la figura D.8, habiendo anteriormente pulsando en el vecino.

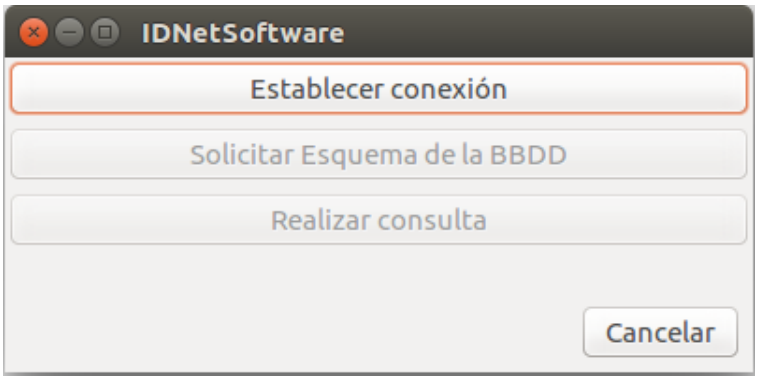


Figura D.7: *Diálogo de conexión a un vecino*

Figura D.8: *Icono de conexión*

A continuación pulsa en el botón "Conexión a la base de datos", tal y como en la figura 7.7.

Una vez realizado, aparecerá la información concerniente a las bases de datos del vecino en la lista situada abajo a la derecha.
En la figura 7.8 podemos ver las bases de datos disponibles del vecino.

Solicitud del esquema de una base de datos

Para solicitar un esquema de la base de datos, se requiere previamente la conexión con el vecino que posea dicha base de datos.

(Figura 7.6), selecciona al vecinos de la lista "Vecinos OV" situada en la parte inferior central de la aplicación clickeando dos veces. A continuación aparecerá un cuadro de diálogo similar al de la figura D.7. Otra forma de llegar al cuadro de diálogo es acudiendo a la

barra de herramientas y pulsar en el icono similar a la figura D.9, habiendo anteriormente pulsando en el vecino.

Figura D.9: *Icono para la solicitud del esquema de la base de datos*

A continuación selecciona "Solicitar Esquema de la BBDD" y se mostrará el esquema de la base de datos requerida. (Figura 7.9)

Información sobre los mensajes transmitidos

Tanto los mensajes enviados como recibidos a través de la aplicación pueden ser vistos gracias a nuestra ventana informativa de mensajes.

Para el acceso a dicha ventana, solo se tiene que acudir al menú superior y pulsar en "OV", para a continuación pulsar en "Mensajes" . Una vez se haya realizado, aparecerá una ventana con todos los mensajes que han atravesado la red, siendo origen o destino el propio usuario que ha iniciado sesión.

En la figura D.10 se muestra la ventana de mensajes junto con un ejemplo de la visualización de un determinado mensaje. Estos ejemplos se corresponden con las figuras D.11 y D.12.

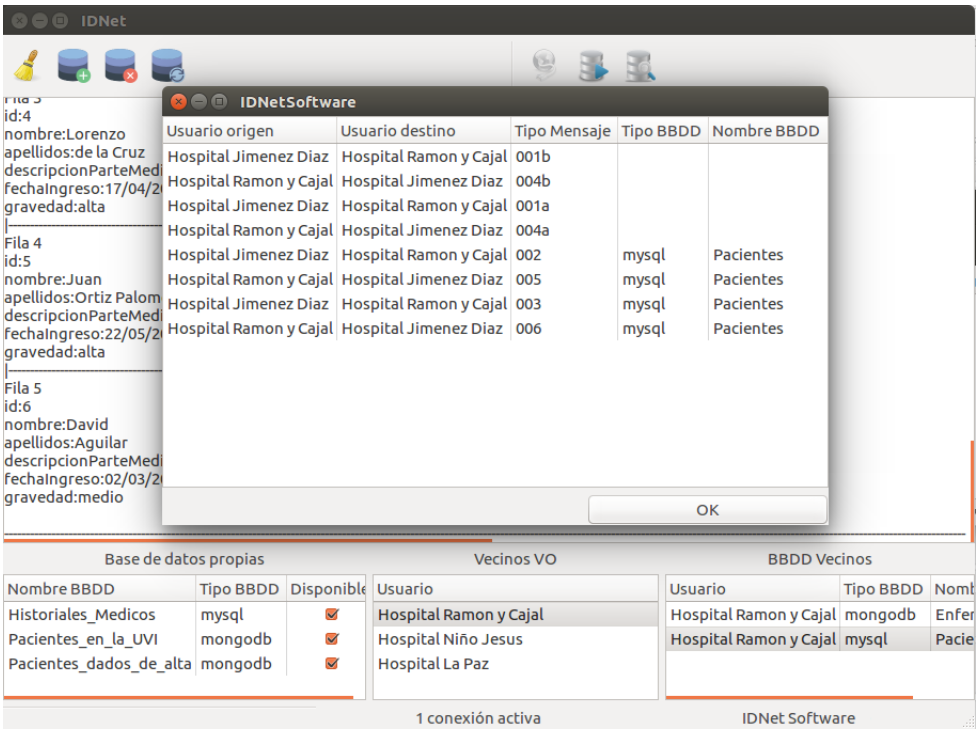


Figura D.10: *Ventana informativa de mensajes*

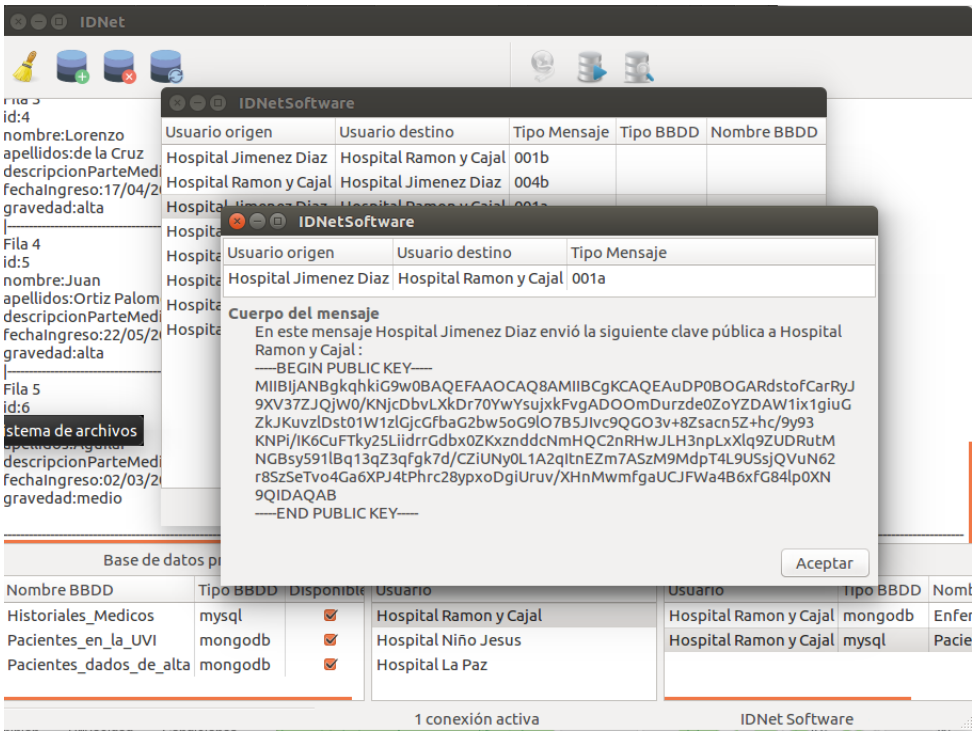


Figura D.11: *Información sobre un mensaje transmitido: 001a*

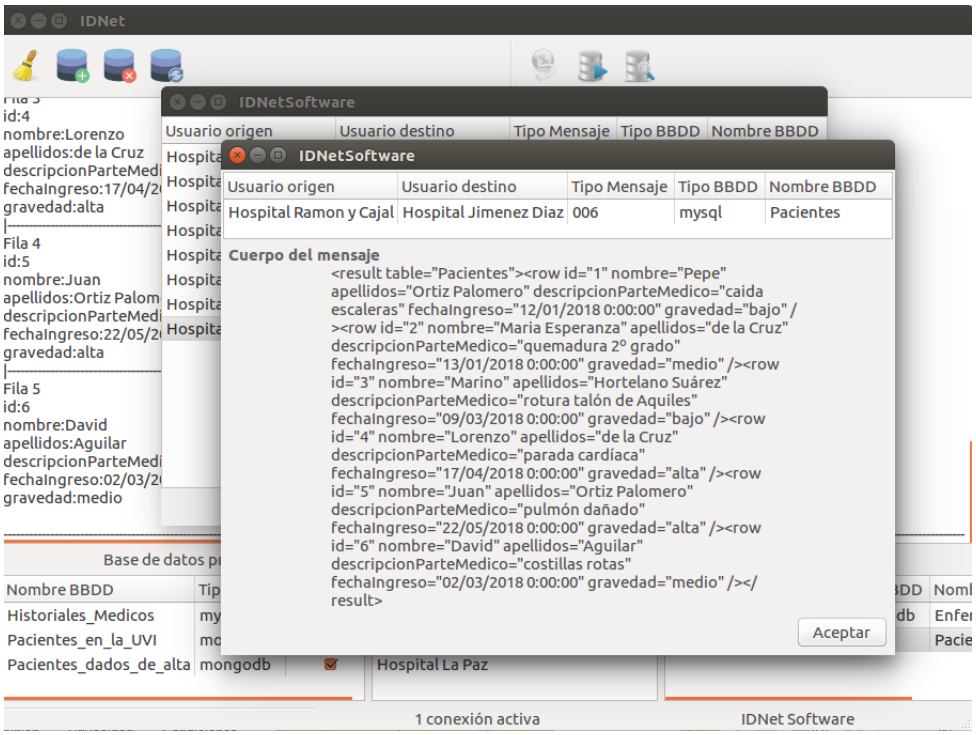


Figura D.12: *Información sobre un mensaje transmitido: 006*

Apéndice E

Glosario de términos

E.1. Bases de datos

Las bases de datos son los componentes esenciales de IDNet. IDNet permite dos tipos de bases de datos: MySQL y MongoDB. Cada una de ellas es la base de datos de referencia en su ámbito: MySQL es la referente de las bases de datos relacionales y MongoDB es la base de datos referente dentro del mundo de las bases de datos NoSQL.

E.2. Peer-to-Peer

Define un tipo de red cuyos nodos son, al mismo tiempo, cliente y servidor. Se suelen utilizar para redes anónimas y de transmisión de ficheros. Todos los nodos en IDNet implementan este concepto.

E.3. Organización Virtual

En IDNet, una Organización Virtual se define como un conjunto de nodos que pueden comunicarse entre sí y que tienen un ámbito definido y delimitado con acceso restringido.

E.4. Cloud

"Cloud", conocida como "Cloud Computing" y su comparativa en la lengua española "Computación en la nube", proporciona a IDNet la distribución de los nodos GateKeeper en la red gracias a los servicios disponibles que proporcionan. El proveedor de servicios IaaS elegido para el despliegue ha sido Amazon Web Services, el cual nos permite usar la tecnología Docker, además de proporcionarnos una base de datos relacional en la nube gracias a su servicio Amazon RDS.

E.5. Docker

Docker es un proyecto abierto que permite el despliegue automático de aplicaciones dentro contenedores, es decir, dentro de sistemas operativos con una capa de abstracción y de virtualización. Los nodos desplegados por IDNet se ejecutan en contenedores con sistema operativo Linux. Como bien en el término "Cloud" hemos comentado, el servicio EC2 de Amazon Web Services nos proporciona su despliegue en la nube.