



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

TELECOM ESCUELA
TÉCNICA **VLC** SUPERIOR
DE **UPV** INGENIEROS
DE TELECOMUNICACIÓN

DISEÑO DE UN SISTEMA PARA EL PROCESAMIENTO DE DATOS PROCEDENTES DE AIS

Dario Alandes Codina

Tutor: Carlos Enrique Palau Salvador

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2017-18

Valencia, 26 de noviembre de 2017

Agradecimientos

A mi novia, Daniela, por su inestimable apoyo, porque sin ella no habría podido hacerlo todo.

Resumen

El objetivo de este proyecto es el tratamiento de los datos recogidos por un sistema AIS. Para ello se ha diseñado un servidor sobre la versión 16.04.2 de Ubuntu y una sencilla aplicación en Android nativo compatible con versiones 6.0 o superiores.

El sistema AIS permite de forma standard el seguimiento de barcos de toda clase debido a que en tierra los puertos tienen receptores que conectan con sus sistemas de gestión para tener localizados a todos los barcos que haya en las costas.

La función principal del servidor nombrado anteriormente es la de recoger mediante Apache Storm los datos obtenidos del AIS y almacenarlos en MySQL (base de datos), preparándolos así para ser mostrados en la aplicación.

Dicha aplicación requiere permisos al usuario para conectarse a internet y una vez obtenidos le muestra la información disponible, que aparece en forma de un mapa donde se ve la localización, el nombre, el tamaño y hasta 14 características sobre las embarcaciones.

Resum

L'objectiu d'aquest projecte és el tractament de les dades recollides per un sistema AIS. Per a això s'ha dissenyat un servidor sobre la versió 16.04.2 de Ubuntu i una senzilla aplicació en Android natiu compatible amb versions 6.0 o superiors.

El sistema AIS permet de forma estàndard el seguiment de vaixells de tota classe a causa que en terra els ports tenen receptors que connecten amb els seus sistemes de gestió per tenir localitzats a tots els vaixells que hi hagi en les costes.

La funció principal del servidor anomenat anteriorment és la de recollir mitjançant Apatxe Storm les dades obtingudes del AIS i emmagatzemar-los en MySQL (base de dades), preparant-los així per ser mostrats en l'aplicació.

Aquesta aplicació requereix permisos a l'usuari per connectar-se a internet i una vegada obtinguts li mostra la informació disponible, que apareix en forma d'un mapa on es pot veure la localització, el nom, la grandària i fins a 14 característiques sobre les embarcacions.

Abstract

This project aims at the treatment of data collected by an AIS system. As to it, a server on the version 16.04.2 of Ubuntu and a simple application in native Android compatible with versions 6.0 or higher have been designed.

The AIS system allows standard tracking of all sort of ships because on the ground the ports have receivers that connect with their management systems to localize all vessels on the coast.

The main function of the server named above is to collect by Apache Storm the data obtained from the AIS and store them in MYSQL (database), preparing them at this way to be displayed in the application.

This application requires permissions to the user to connect to internet and once obtained shows the information available, which appears in form of a map where you can see the location, name, size and up to 14 features of the vessels.

Índice

Capítulo 1. Introducción y motivación	1
1.1. Motivación	1
1.2. Estructura del documento	3
Capítulo 2. Objetivos	4
Capítulo 3. Tecnologías.....	5
3.1. Sistema Automático de Identificación (AIS)	5
3.2. Ubuntu.....	6
3.2.1. Historia.....	6
3.2.2. Características	6
3.2.3. Versiones para diferentes dispositivos	6
3.3. Apache Storm.....	8
3.3.1 Componentes	8
3.3.2. Arquitectura.....	9
3.4. Java	10
3.4.1. Características principales	10
3.5. Base de datos, MySQL	12
3.5.1. MySQL.....	12
3.5.2. Lenguaje SQL	12
3.6. Android	13
3.6.1. Historia.....	13
3.6.2. Características	13
3.6.3 Arquitectura.....	14
Capítulo 4. Desarrollo	15
4.1. Arquitectura Global.....	15
4.2. Instalación del servidor.....	16
4.3. Base de datos.....	16

4.4. Apache Storm.....	19
4.4.1. Configuración.....	19
4.4.2. Topología	20
4.4.3. Spout	22
4.4.4. Bolt – InsertBolt	26
4.4.5. Bolt – HistorialBolt	30
4.5 Aplicación Android	34
4.5.1. activity_maps.xml	34
4.5.2. MainActivity.java	35
4.5.3. Resultados	36
Capítulo 5. Conclusiones y trabajo futuro	37
5.1. Conclusiones	37
5.2. Líneas futuras	38
Bibliografía	39

Capítulo 1. Introducción y motivación

1.1. Motivación

La gran cantidad de barcos que navegan por los mares hace necesario un mecanismo para que estos puedan detectarse unos a otros, en 1904, Christian Hülsmeyer fue el primero en usar ondas de radio para detectar la presencia de objetos metálicos distantes. Antes de la segunda guerra mundial estaban desarrollando tecnologías que acabarían siendo el actual sistema de radar, consiste en el uso de ondas electromagnéticas para medir distancias entre buques para evitar colisiones. Este también se usa para navegar, pudiéndose fijar la posición en el mar cuando se tiene dentro del rango del radar referencias fijas como islas, boyas o buques faro.

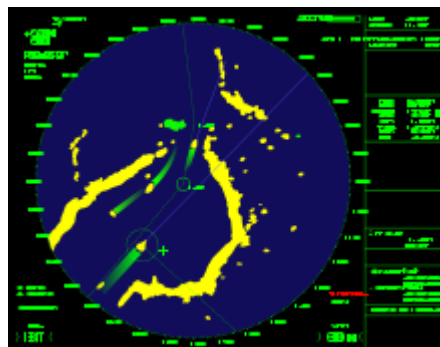


Figura 1. Ejemplo señal de radar

En 1990 se desarrolló AIS (Automatic Identification System) como una red de identificación y seguimiento de corto alcance y alta intensidad. Este sistema permite la identificación desde no solo barcos sino desde otros como aviones o estaciones costeras. El sistema se basa en un dispositivo instalado a bordo que transmite la información relativa al barco para identificarse con otros barcos cercanos y estaciones de tierra, debidamente equipadas. En comparación con el radar, el sistema AIS además de servir para evitar colisiones también sirve como control de tráfico marítimo. Aunque tiene más funciones que un radar, AIS no puede reemplazar al radar, puesto que si el otro barco no tiene equipado un dispositivo adecuado no se podrá ver y podría haber complicaciones, ambos sistemas se complementan y ninguno puede ser sustituido por otro.

En 2002, el AIS fue aprobado con un calendario de implementación en función de las características del barco. A partir de 2007 AIS es obligatorio para barcos adheridos al convenio SOLAS que cumplan alguna de las características siguientes:

“Ámbito de aplicación

- 1. La presente Directiva se aplicará a los buques de un arqueo bruto igual o superior a 300 toneladas, siempre que no se establezca otra cosa.*
- 2. La presente Directiva no se aplicará a:*
 - a) los buques de guerra, unidades navales auxiliares u otros buques propiedad de un Estado miembro o que estén a su servicio y presten servicios públicos de carácter no comercial;*
 - b) los barcos de pesca, los buques tradicionales y las embarcaciones de recreo de una eslora inferior a 45 metros;*
 - c) los buques cisterna de menos de 5000 toneladas, las provisiones de a bordo y el equipo a bordo de los buques.”*

Mas adelante se modificó para que los barcos pesqueros estén obligados a usarlo. El resto de embarcaciones no sometidas al Convenio SOLAS y que no sean pesqueras no están obligadas a usar AIS, pero sí que se les recomienda usarlo.

Con la llegada del AIS a la mayoría de los barcos se hace posible la investigación sobre los océanos que antes no, todos los datos procedentes de estos terminales viajan a través de internet, se pueden tratar como cualquier información y sacar las conclusiones pertinentes.



Figura 2. Rutas marítimas sin mapa en 1 año

En la figura dos se puede observar como las rutas marítimas son tantas que prácticamente se pueden ver los continentes sin estar estos dibujados, todo esto se puede usar para realizar los análisis.

Este proyecto tiene como objetivo el analizar los datos obtenidos de la red AIS, estos datos se pueden utilizar para realizar diversos estudios de los océanos, un análisis que se hará es la ruta que toman los barcos, concretamente, sus últimas 4 actualizaciones de posición. Para ello montaremos un servidor que recoja todos estos datos y con la ayuda de Apache Storm realizar los análisis pertinentes, además de una aplicación para Android para poder visualizar los datos. [1]
[2]

1.2. Estructura del documento

A continuación, se explicará la estructura del documento para guiar al lector de este Trabajo Final de Grado:

❖ **Introducción**

En la introducción se explicará de manera breve la motivación del proyecto, el problema que supone y una posible solución a implementar, además de la explicación del documento.

❖ **Objetivos**

Como el nombre del apartado indica, aquí se describirán los objetivos finales de este proyecto.

❖ **Tecnologías**

Aquí se explicarán las tecnologías y herramientas necesarias para llevar a cabo el proyecto. En pocas palabras, teoría.

❖ **Desarrollo del sistema**

En este apartado se procederá a explicar el desarrollo completo del sistema, comenzando con la instalación básica del sistema operativo que llevará el servidor hasta llegar a la aplicación móvil.

❖ **Conclusiones y líneas futuras**

Para finalizar se expondrán las conclusiones tras la realización del proyecto y las líneas futuras para mejorar el proyecto.

Capítulo 2. Objetivos

En este capítulo se pasarán a describir los objetivos que se intentan alcanzar en este Trabajo de Fin de Grado. Como ya indica el propio título del trabajo, el objetivo principal de este es el desarrollo de un sistema capaz de recoger información de un terminal AIS, analizar esta información y tratarla, guardarla en una base de datos y mostrar la ubicación de los barcos en una aplicación para Android.

Los objetivos generales durante la realización del proyecto son:

- Estudio de las bases de datos MySQL, su implementación en el entorno de desarrollo y la creación de una base de datos MySQL en un servidor.
- Conocer el lenguaje SQL y las bases de datos relacionales.
- Estudio del lenguaje de programación Java y de un entorno de desarrollo basado en este lenguaje.
- Estudio del sistema operativo Android, la programación de aplicaciones en Android y su entorno de desarrollo Android Studio.
- Estudio del sistema Apache Storm, y desarrollo de un cluster para el análisis de datos.
- Familiarizarse con el tratamiento de grandes volúmenes de datos, así como de las herramientas para tal fin.

El objetivo implementar Apache Storm son los siguientes:

- Estudio y familiarización con una herramienta capaz de tratar con grandes volúmenes de datos en tiempo real.
- Uso del lenguaje Java para el tratamiento de grandes volúmenes de datos.
- Conexión a una base datos MySQL ubicada en un servidor.
- Tratamiento de los datos para poder realizar análisis de los mismos, así como su presentación en una aplicación.

Los objetivos de realizar una aplicación en Android son:

- Conectarse desde cualquier dispositivo Android al servidor que proporcionara los datos.
- Capacidad de mostrar la posición de los barcos recibidas del servidor.
- Mostrar detalles del barco seleccionado por el usuario.
- Mostrar el análisis que se ha hecho de los datos en un mapa.

Capítulo 3. Tecnologías

En este capítulo haré una breve introducción de la tecnología usada para realizar el proyecto. Primero se mencionará por encima como está compuesto el proyecto y después se explicará con algo más de profundidad cada componente.

El servidor es una maquina con el sistema operativo Ubuntu 16.04.2, el más reciente hasta la fecha que se empezó a realizar el proyecto, al cual se le instaló Apache Storm para el tratamiento de datos, Storm admite varios lenguajes de programación, eligiéndose Java por la familiaridad con él, y una base de datos MySQL que almacena los datos, en cuanto a la aplicación, esta está programada en Android.

3.1. Sistema Automático de Identificación (AIS)

AIS es un sistema que permite conocer automáticamente y en tiempo real, la situación de barcos, su nombre, velocidad, rumbo y otra información que podría resultar de interés. El funcionamiento está basado en la señal de GPS y de la transmisión digital en la banda marítima de VHF como sistema de difusión y comunicación entre barcos, esto permite aumentar la seguridad de las embarcaciones, sobre todo para evitar las limitaciones del radar evitando las “zonas ciegas” de este que se producen a la hora de identificar ecos detrás de montañas, islas, canales, etc.

El objetivo prioritario del sistema AIS es garantizar la prevención de los accidentes, mejorando así la seguridad en el mar, además ayuda al control del tráfico marítimo, así como hacer más eficiente la navegación. Esto es posible gracias al intercambio automático entre barcos y estaciones costeras de control de tráfico. Usando la banda marítima de VHF, se es capaz de manejar más de 4500 mensajes por minuto, además de sus actualizaciones cada 2 segundos. Para ello es necesario una tecnología capaz de ordenar los mensajes y sus datos, AIS utiliza SOTDMA (*Self Organizing Time Division Multiple Access*).

El sistema AIS consta de un equipo con un transmisor y tres receptores VHF, dos receptores para señales TDMA y el otro para señales DSC, los datos de posicionamiento se obtienen del GPS del barco que deberá estar conectado al sistema AIS y los equipos electrónicos de navegación. Las transmisiones se efectúan de forma digital y con modulación FM con una señal de 9.6 Kbytes de ancho de banda, utilizando el protocolo HDLC, aunque AIS puede funcionar con un solo canal de VHF en la práctica se utilizan dos para evitar problemas de interferencia que pudiesen aparecer en alguno de ellos.

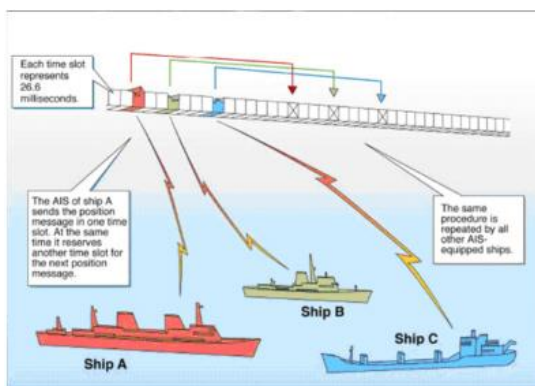


Figura 3. Esquema transmisión AIS

Los datos más críticos, como el ratio de giro expresado en grados por minuto, la velocidad con una resolución de una décima de nudo, la posición GPS con exactitud DGPS y el rumbo, se actualizan en intervalos de dos a diez segundos dependiendo del cambio de estos. Otros datos fijos, o no tan variables, se actualizan cada 6 minutos, estos datos pueden ser el MSI, el número IMO del barco, la señal de llamada por radioteléfono, el nombre y el tipo de barco, así como sus dimensiones en metros, el calado actual, su posición de destino y la hora y fecha aproximada de llegada al destino. [3]

3.2. Ubuntu

Ubuntu es un sistema operativo basado en GNU/Linux que se distribuye como software libre. Es una de las más importantes distribuciones de Linux a nivel mundial, la cuota de mercado dentro las distribuciones Linux esta aproximadamente sobre el 49%. Destinado al usuario promedio, concentra su objetivo en la facilidad y libertad de uso, la fluida instalación y los lanzamientos regulares.



Figura 4. Ubuntu

3.2.1. Historia

El proyecto nació por iniciativa de algunos programadores de los proyectos Debian, Gnome, porque se encontraban decepcionados con la manera de operar del proyecto, este no ponía énfasis en estabilizar el desarrollo de sus versiones de prueba y sólo proporcionaba auditorías de seguridad a su versión estable. El 8 de Julio de 2004 se anunció la creación de la distribución Ubuntu por parte del sudafricano Mark Shuttleworth y la empresa Canonical Ltd. Varios meses más tarde la primera versión de Ubuntu se lanzó, el 20 de octubre de 2004.

Ubuntu 4.10 – *Warty Warthog*, fue la primera publicación de Ubuntu, y recibió ese nombre (jabalí verrugoso) porque fue publicado “*warts and all*” (con verrugas y todo), el número de versión se debe a la fecha en la que se lanzó, el 4 simboliza el año en el que se publicó; 2004, y el 10 se debe al mes; octubre. El escritorio era de un color bastante oscuro, el naranja y el marrón simbolizaban las tribus sudafricanas. Un motivo por el que Ubuntu se puso en la cabeza desde su comienzo fue el Live CD, que ofrecía a los usuarios el poder probar Ubuntu sin necesidad de instalarlo, algo que ninguna otra distribución de Linux ofrecía a los usuarios de escritorio.

La siguiente versión salió 6 meses después, tal y como prometieron, Ubuntu 5.04 *Hoary Hedgehog*, y así cada 6 meses sale una nueva versión, hasta la actual 17.04, la próxima versión será la 17.10 que saldrá el mes de octubre de 2017.

[4]

3.2.2. Características

Como característica principal cabe destacar que los usuarios pueden participar en el desarrollo de Ubuntu de muchas maneras, escribiendo el código, probando versiones inestables o simplemente aportando sus ideas y votando las del resto.

Ubuntu posee muchas aplicaciones para diferentes ámbitos, como el entretenimiento, el desarrollo y la configuración de todo el sistema. *Ubuntu* posee una interfaz predeterminada, pero además existen versiones extraoficiales que pueden descargarse e instalarse sin ningún problema, también posee varias aplicaciones ya instaladas por defecto, vendrían a ser *Mozilla Firefox*, *Empathy*, *Thunderbird*, *Totem*... también incluye funciones avanzadas de seguridad, entre ellas está el no activar procesos latentes al momento de instalarse, por tanto, no sería necesario que hubiera un cortafuegos, se supone que no hay servicios peligrosos.

3.2.3. Versiones para diferentes dispositivos

A partir del núcleo se han desarrollado versiones enfocadas a múltiples dispositivos, están serian:

- Ubuntu Desktop, pensado para ordenadores.
- Ubuntu Phone, pensado para smartphone, cuenta con la posibilidad para iniciar el escritorio desde un dock con monitor externo.

- Ubuntu Tablet, para tabletas, cuenta con una interfaz multitarea para ejecutar dos aplicaciones al mismo tiempo, además de la posibilidad de multiusuario.
- Ubuntu TV, orientado a smartTV, con interfaz simple para organizar contenidos para TV.
- Ubuntu for Andorid, como Ubuntu Phone pero destianada a Android.

Ubuntu Server, orientado a servidores, básicamente instala Ubuntu sin la interfaz gráfica de usuario. [5]

3.3. Apache Storm

Apache Storm es un sistema de computación en tiempo real distribuido, gratuito y de código abierto, escalable, tolerante a fallos y con alta disponibilidad, fácil de montar y operar con él. Storm está pensado para trabajar con datos que deben ser analizados en tiempo real, como sensores, datos de redes sociales, etc. en nuestro caso los datos de los barcos.



Figura 5. Apache Storm

3.3.1 Componentes

Apache Storm está compuesto básicamente de dos componentes *spouts* y *bolts*, para decir cómo están conectados estos se usa una topología, la figura muestra cómo sería una topología, básicamente lo dice que flujo de datos va con que componente.

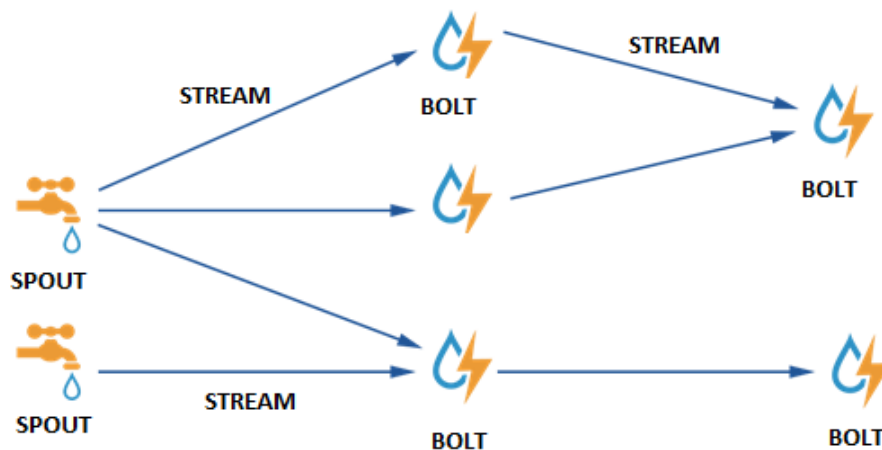


Figura 6. Ejemplo topología

- Spouts

Los *Spouts* son la fuente de datos de la topología, coge los datos de fuentes externas y los emite dentro de la topología, estos pueden funcionar de dos maneras, *reliable* o *unreliable*, en modo *reliable* el *spout* es capaz de volver a enviar los datos si estos se han perdido en el proceso, en el otro modo, el *spout* olvida los datos tan pronto los envía por la topología, siendo incapaz de recuperar los datos si se pierden.

También pueden emitir más de un stream a la vez, para hacerlo basta con declarar varios y especificar el stream a enviar. El método principal en un *spout* es **nextTuple**, este puede emitir los datos a la topología o devuelve si no hay nada más que emitir, es muy importante que el método no esté bloqueado por nada, ya que Storm llama a todos los métodos del *spout* en el mismo hilo. Otros métodos importantes son **ack** y **fail**, son llamados cuando los datos son procesados correctamente a través de la topología o no se han podido completar.

- Bolts

Los Bolts son el lugar donde se realiza todo el procesamiento de los datos, estos pueden hacer cualquier cosa, conectarse con bases de datos, funciones, filtrar, etc. Un bolt puede hacer operaciones simples sobre un stream de datos, para las operaciones más complejas se suelen necesitar varios bolts. Del mismo modo que los *spouts* pueden emitir más de un stream a la vez.

El método principal de los bolts es **execute**, este método toma como entrada los datos recibidos, en este método se hacen todas las operaciones, el bolt puede emitir otro stream o no, para ello el objeto *OutputCollector*. Los bolts deberían llamar al método *ack* cada vez que termina

de procesar los datos recibidos, para que Storm pueda saber cuándo se terminan de procesar los datos.

- Stream grouping

Un aspecto importante en Storm es la forma en la que se van a compartir los datos entre los diferentes componentes de la topología. Como modelo de datos Storm utiliza tuplas, básicamente son listas de valores con un nombre, el valor asociado puede ser un objeto de cualquier tipo. Hay 8 agrupaciones de datos en Storm, también se pueden hacer agrupaciones propias.

- Shuffle grouping: Storm decide de forma aleatoria la tarea a la que se va a enviar la tupla de manera que la distribución se realiza equitativamente entre todos los nodos
- Fields grouping: Se agrupan los streams por un determinado campo de manera que se distribuyen los valores que cumplen una determinada condición a la misma tarea.
- All grouping: El stream se pasa a todas las tareas del cluster haciendo multicast.
- Global grouping: El stream se envía al bolt con ID más bajo.
- None grouping: Bastante similar a shuffle grouping donde el orden no es importante.
- Direct grouping: La propia tarea es la encargada de decidir a qué bolt emitir la tupla indicando el ID de ese emisor. Esta forma dota de mayor lógica de distribución en los nodos para que puedan decidir hacia donde redirigir los streams.
- Local grouping: Se utiliza el mismo bolt si tiene una o más tareas en el mismo proceso.

3.3.2. Arquitectura

Storm, tiene una arquitectura sencilla, se divide en tres componentes:

- **Master node**, encargado de realizar la asignación y monitorización de las tareas en las distintas máquinas del cluster, además ejecuta el Nimbus daemon, responsable de distribuir la información a través del cluster.

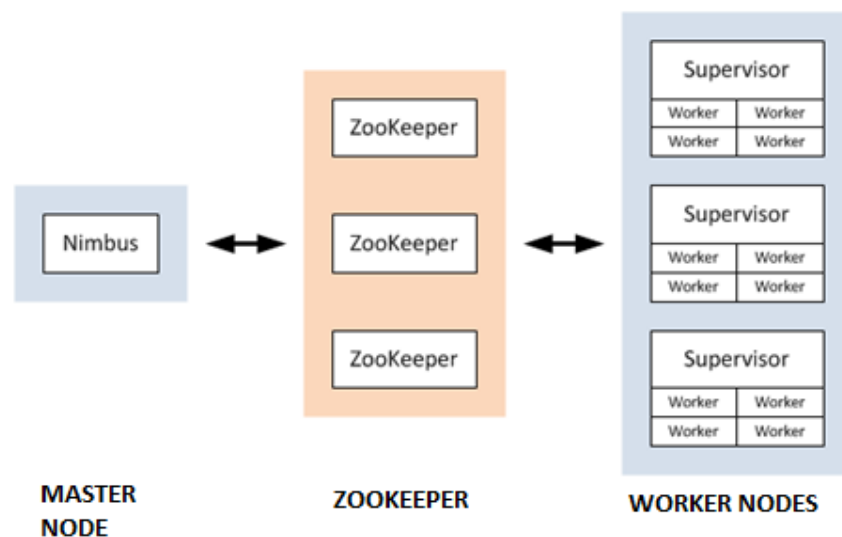


Figura 7. Arquitectura Apache Storm

- **Worker node**, este componente ejecuta el Supervisor daemon encargado de recoger y procesar los trabajos asignados a la máquina donde está funcionando. Cada nodo ejecuta un fragmento de la topología para así poder distribuir todos los trabajos por todo el cluster. Si por alguna razón un nodo dejara de funcionar, el Nimbus daemon redirigiría el trabajo a otro.

Zookeeper, este no exactamente un componente de Storm, pero sí es necesario para la coordinación entre el Nimbus y los Supervisors, y para mantener el estado, puesto que los otros no tiene estado (stateless). [6] [7] [8]

3.4. Java

Java es un lenguaje de programación y una plataforma informática, comercializada por primera vez en 1995 por Sun Microsystems, con la intención de permitir la ejecución de aplicaciones en cualquier dispositivo sin la necesidad de volver a compilar el código.

El lenguaje *Java* fue creado como un componente fundamental de la *plataforma Java*. La forma en la que se escribe código es muy similar a *C* y *C++*, pero sin tantas utilidades de bajo nivel como estos y sin soportar extensiones de código ensamblador.



Figura 8. Icono de *Java*

3.4.1. Características principales

- ❖ **Orientación a objetos:** la programación orientada a objetos es un paradigma de programación, se llama objeto a las entidades utilizadas como elementos fundamentales a la hora de hacer una solución. El objetivo principal de este tipo de programación es la reutilización de objetos procedentes de otros proyectos, y así poder reducir el tiempo de desarrollo y el número de proyectos que no llegan a finalizarse. *Java* tiene todos los conceptos de esta técnica de programación, tales como: abstracción, herencia, polimorfismo, etc.
- ❖ **Independencia de la plataforma:** todo software escrito en *Java* está preparado para ser ejecutado en cualquier plataforma, con que se compile una vez es suficiente. Cuando se compila el código *Java* se genera un código *bytecode*, este se ejecuta en una *JVM (Java Virtual Machine)*, este es el encargado de interpretar y ejecutar el código.
- ❖ **Gestión automática de la memoria:** *Java* usa un recolector de basura para la gestión de la memoria. *Java runtime*, el entorno en tiempo de ejecución de *Java* se encarga de la vida de los objetos junto con el recolector para así liberar la memoria que estos ocupaban. El programador no ha de preocuparse la gestión de la memoria, tan solo tiene que decidir cuándo se crean y destruyen los objetos.
- ❖ **Multihilo:** *Java* soporta el *multithreading*, sincronización de múltiples hilos de ejecución, y por tanto puede de hacer varias funciones al mismo tiempo.
- ❖ **Alto Rendimiento:** *Java* se considera un lenguaje de alto rendimiento por la velocidad de ejecución y ser capaz de ahorrar en líneas de código de los programas.
- ❖ **Seguridad:** Se implementaron medidas de seguridad en el lenguaje y en sistema de ejecución.

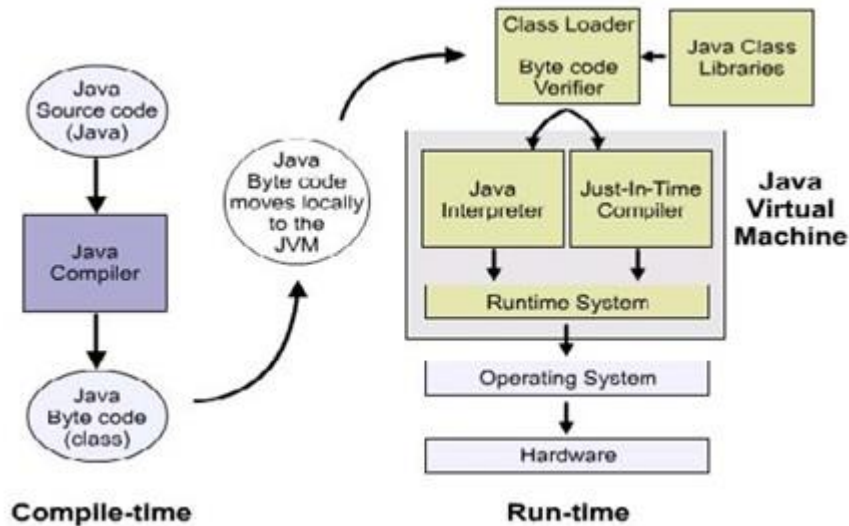


Figura 9. Estructura de ejecución de un programa en Java

La plataforma Java es un conjunto de programas para hacer más sencillo el desarrollo y ejecución de programas escritos en Java. Formada por una máquina virtual, el motor de ejecución, un compilador y librerías.

La figura 6 nos muestra cómo se ejecuta un código Java. Los componentes más importantes que hay que destacar son:

JVM (Java Virtual Machine): es el núcleo de la plataforma Java, aquí es donde se ejecutan los *bytecode* generados por el compilador, los *bytecode* son lo que permite la ejecución en cualquier maquina sin importar que dispositivo sea o que sistema operativo tenga, puesto que es el JVM el que se encarga de ejecutar el código.

JRE (Java Runtime Environment): se trata del software necesario para ejecutar cualquier tipo de aplicación desarrollada en Java, dentro de este se encuentra la JVM. [9]

3.5. Base de datos, MySQL

Las bases de datos son lugares donde se guarda un conjunto determinado de información relacionada entre si para poder ordenarla y clasificarla según algunos criterios. Las bases de datos informáticas tienen que tener un fácil acceso a su información además de ser dinámicas.

Hay varios tipos de bases de datos, entre ellos están: bases de datos jerárquicas, de red, orientadas a objetos, etc. el que a nosotros nos interesa es la base de datos relacionales.

Las bases de datos relacionales cumplen el modelo de datos relacional, modelo creado por *Dr. Edgar F. Codd* en *IBM* en los 70, estas bases de datos están formadas por conjuntos de tablas, estas tablas están formadas por campos (filas) y registros (columnas), donde se cruzan un campo y un registro hay el dato correspondiente a ese campo y ese registro.

En las bases de datos no puede haber diferentes tablas con el mismo nombre ni registro, cada tabla necesita de una clave primaria, se debe cumplimentar correctamente los datos para que no haya registros erróneos, hay relaciones entre tablas, pueden ser *1-1*, *1-n* y *n-n*, las relaciones se establecen entre el registro primario de una y un registro, no hace falta que sea primario, de la segunda, los registros relacionados tendrán el mismo valor en las dos tablas.

3.5.1. MySQL

El sistema de gestión de base de datos *MySQL* es el que vamos a utilizar, ya que *MySQL* permite almacenar y tratar los datos con mucha eficiencia de las aplicaciones del concepto *IoT*, se posiciona como la opción perfecta para el desarrollo de estas.

MySQL es un sistema de bases de datos relacionales, multihilo y multiusuario, de código abierto, tiene alta disponibilidad gracias a la creación de muchos marcos de trabajo por parte de los usuarios de este sistema, desarrollado por *Oracle Corporation*. Además, utiliza el lenguaje *SQL*, el lenguaje estándar de las bases de datos relacionales.



Figura 10. Icono MySQL

3.5.2. Lenguaje SQL

Structured Query Language (lenguaje SQL), es el lenguaje que se usa a la hora de definir manipular y controlar bases de datos relacionales. Los principales gestores de bases de datos usan este lenguaje. Fue creado en los 70 en un laboratorio de investigación de *IBM*, se desarrolló junto con el modelo de base de datos relacionales. En 1986 fue nombrado estándar oficial del *ANSI* y un año más tarde, en 1987, del *ISO*.

Con el lenguaje *SQL*, el programador especifica un conjunto de preposiciones, afirmaciones, restricciones, etc. que describen el problema y la solución al mismo, ya que el lenguaje *SL* es un lenguaje de programación declarativo.

Existen varios tipos de sentencias *SQL* que describen las acciones que se harán sobre la base de datos, estos son:

- Definición de datos (DDL)

Estas sentencias se encargan de modificar los objetos de la base de datos, desde su creación, su modificación o su eliminación, los objetos a los que puede afectar pueden ser tablas, vistas, índices, etc. Estas instrucciones son: *CREATE*, *DROP*, *ALTER*, *TRUNCATE*.

- Manipulación de datos (DML)

Este grupo es el que se encarga de la consulta, gestión o modificación de los datos almacenados. Estas instrucciones serían las de: *SELECT*, *INSERT*, *UPDATE*, *DELETE*.

- Control de datos (DCL)

Estas son las instrucciones relacionadas con el control de acceso a la base de datos, permiten configurar el tipo de acceso a la misma. Las instrucciones SQL son: GRANT o REVOKE

[10] [11]

3.6. Android

Android es un sistema operativo en el núcleo de Linux, está pensado principalmente para dispositivos con pantalla táctil, como los smartphones, relojes inteligentes, tabletas... es el sistema que más se ha popularizado por encima de otros como *Symbian*, *Firefox OS*, *iOS*. De todos estos los únicos consolidados en el mercado son *Android* y *iOS*. Actualmente *Android* posee una cuota de mercado en España del 91%, mientras que es segundo es *iOS* con solo 8.4%. Esto se puede deber a que android es un sistema operativo de código abierto y no está limitado a ningún dispositivo concreto, a diferencia de su principal competidor *iOS*, que está limitado a los dispositivos fabricados por la marca *Apple*.

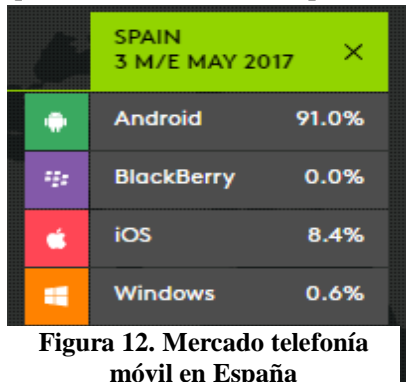


Figura 12. Mercado telefonía móvil en España

En 2007, se crea la *Open Handset Alliance*, y el mismo día se anuncia la primera versión: *Android 1.0 Apple Pie*. El primer dispositivo tardó un año en llegar al mercado, así fue como en 2008 se lanzó *HTC Dream* con la versión 1.5 de *Android* y a un precio de 179 dólares. Junto con el lanzamiento del primer dispositivo con *Android* se estrenó *Android Market*, lo que actualmente se conoce como *Google Play*.

3.6.2. Características

Hay muchas características que hacen que *Android* se líder en el mercado, la característica más importante a destacar es la de que *Android* es un sistema operativo de código abierto, lo que significa que cualquier puede ver el código y modificarlo, llegando a tener el sistema totalmente personalizado.

Otra característica importante por destacar es el amplio abanico de dispositivos en los que puede funcionar, soporta prácticamente todo tipo de pantallas y resoluciones, eso podría llegar a ser un inconveniente, pero el kit de desarrollo de *Android* ofrece facilidades para que los desarrolladores adapten sus aplicaciones a diferentes configuraciones de pantallas, para así poder visualizar bien el contenido de su aplicación.



Figura 11. Logo Android

[12]

3.6.1. Historia

Android se empezó a desarrollar en 2003 a manos de una empresa llamada *Android Inc*, esta empezó a desarrollar un sistema operativo para dispositivos móviles basada en Linux, en 2005 la empresa *Google* adquirió esta pequeña compañía, interesada el mercado de la telefonía móvil.

A partir de entonces, y reuniendo a los fabricantes de hardware, desarrolladores de software y operadores de telefonía para el proyecto de *Android*, dos años más tarde, en

3.6.3 Arquitectura

La arquitectura de *Android* está formada por 4 capas como se puede ver en la figura 8, todas ellas están basadas en software libre. El núcleo se trata de la versión 2.6 del sistema operativo *Linux*, este proporciona seguridad, manejo de memoria, multiproceso y soporte de drivers de los dispositivos. Esta es la única capa que depende del *hardware*, esta capa actúa como capa de abstracción entre el resto de capas y el *hardware*.

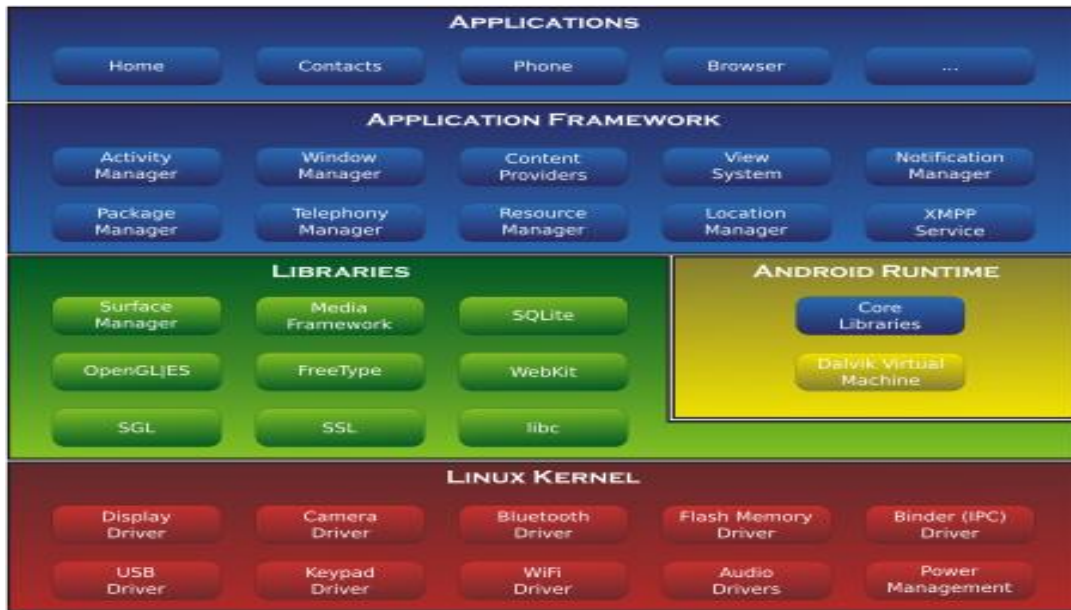


Figura 13. Arquitectura *Android*

El entorno de ejecución de *Android (Runtime)* es parecido a la máquina virtual que utiliza *Java*, directamente está basado en el mismo concepto. E trata de una máquina virtual basada en registros que ejecuta las aplicaciones compiladas en formato *.dex*, se eligió este formato para optimizar recursos, puesto que *Android* se diseñó para dispositivos móviles que funcionan con batería, esto es algo muy importante, en la versión *Android 5.0* y superiores se ha cambiado por otra máquina virtual llamada *ART*, con el que se mejoró el tiempo de ejecución de un código *Java* un 33%.

Pasando a la segunda capa nos encontramos con las librerías de *C/C++* de las cuales hacen uso diferentes componentes del sistema *Android*, estas librerías están compiladas en código nativo del procesador.

El *framework* de *Android* proporciona una plataforma de desarrollo con múltiples opciones, da a los desarrolladores la oportunidad de reutilizar con facilidad los componentes. Incluye servicios como; *Activity Manager*, *Content Provider*, etc. [13]

Capítulo 4. Desarrollo

Este capítulo se detallará el desarrollo del proyecto final de carrera, así como el funcionamiento de cada uno de sus componentes. Primero se hará una vista global de la arquitectura, y después se pasará uno por uno detallando el funcionamiento.

4.1. Arquitectura Global

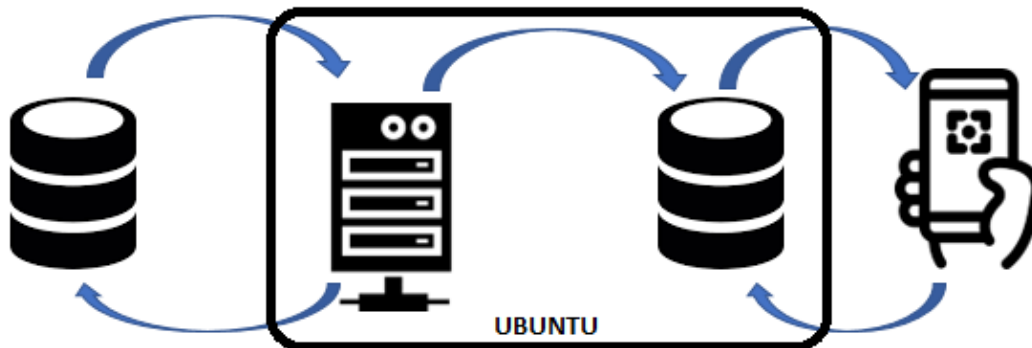


Figura 14. Arquitectura global

El sistema está formado principalmente por 4 componentes:

1. Una base de datos AIS, aquí es donde están todos los datos obtenidos por el sistema AIS, esta información corresponde a todos los barcos los cuales su señal llega a las antenas AIS.
2. El sistema Apache Storm, encargado de recoger la información de la base de datos AIS, analizar y tratarlos para posteriormente guardar esta información en otra base de datos.
3. Una base datos MySQL, que sirve para almacenar los datos recibidos por el sistema Apache Storm y tenerlos a disposición de la aplicación
4. Una aplicación desarrollada en Android que coge la información de la base de datos MySQL y muestra un mapa con la posición de los barcos.

El orden cronológico del desarrollo del proyecto es el mismo orden que se seguirá en la explicación detallada del funcionamiento, a cada paso se han realizado pruebas para comprobar el correcto funcionamiento.

El entorno donde se ha realizado todo el proyecto es:

- Portátil Asus (2.2 GHz Intel Core i7-2670QM, 8GB DDR3)
- Oracle VM Virtual Box 5.0
- Oracle Java SE Development Kit 8
- Apache Storm 1.1.0
- MySQL server 5.7
- Android Studio 2.2.3
- Móvil bq M5

Cabe decir que todo el proyecto se ha desarrollado en un entorno local sin conexión alguna a internet por parte de los componentes, por tanto, se ha creado una base de datos para emular la base de datos del sistema AIS.

4.2. Instalación del servidor

El primer paso antes de hacer nada es crear la máquina virtual donde se pondrá el sistema Apache Storm y la base de datos que recogerá la información, para ello, se instalará Oracle VM Virtual Box, y se montará el sistema operativo Ubuntu 16.04.2, para que no falten recursos en la máquina virtual se le asignará la mitad de los recursos disponibles de la máquina, quedándose con 4 núcleos, 4 GB de RAM y 80GB de HDD.

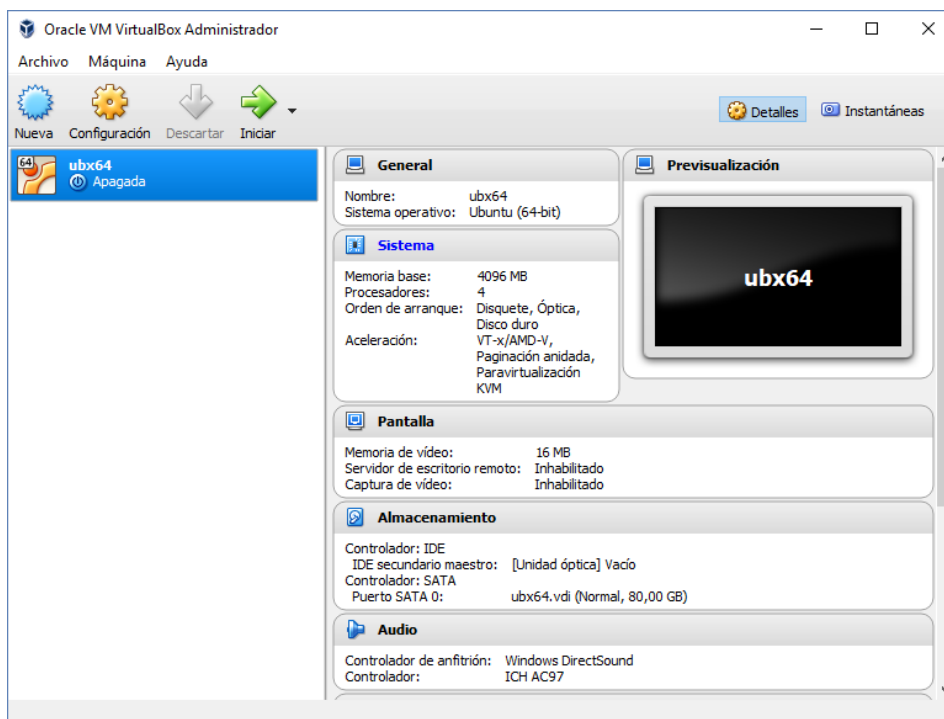


Figura 15. Oracle VM Virtual Box

La instalación de Ubuntu es muy sencilla, la única dificultad está en seguir los pasos correctamente y el sistema se instalará.

4.3. Base de datos

El siguiente paso para que todo funcione es crear la base de datos acorde a las necesidades, lo primero, antes de nada, se investigará como almacena la información AISHUB, para ello se entrará en su página web y verá que información guarda de los barcos.

TAI RAN,			
DETAILS			
Destination:	HONG KONG		
MMSI:	373685000	ETA:	n/a
CallSign:	3FFW7	Coordinates :	22.6082, 120.2412
IMO:	8784846	Course:	230°
Size:	84 x 14 (m)	Speed:	0.1 kn
Draught:	4,8	Heading	232
Last update:	07-09-2017 18:30 UTC	Stations	2187 Monterey-CA, USA

Figura 16. Información de los barcos en AISHUB

Tras ver la información que muestran en la página web diseñamos la base de datos acorde con ello, quedando de la siguiente manera:

Barcos	
Campo	Tipo
MMSI	int(11)
vessel	varchar(45)
callsign	varchar(45)
IMO	varchar(45)
size	varchar(45)
draught	varchar(45)
lastupdate	varchar(45)
destination	varchar(45)
ETA	varchar(45)
coordinates	varchar(45)
course	varchar(45)
speed	varchar(45)
heading	varchar(45)
stations	varchar(45)

Tabla 1. Estructura de la tabla barcos

varchar(45), size varchar(45), draught varchar(45), lastupdate varchar(45), destination varchar(45), ETA varchar(45), coordinates varchar(45), course varchar(45), speed varchar(45), heading varchar(45), stations varchar(45), PRIMARY KEY (MMSI));

Para evitar problemas de seguridad se configurará un usuario para que lo puedo usar el sistema Apache Storm, a este solo se le dará permisos para manipular la base de datos que se ha creado anteriormente, para ello se usará el comando:

```
> CREATE USER 'storm'@'localhost' IDENTIFIED BY 'ap.storm';
```

El dominio localhost no se modificará puesto que, tanto Storm como MySQL están en la misma máquina y no habría ningún inconveniente, en principio. Si no se hace nada más el usuario no tiene permiso alguno, para que este los tenga se usará los siguientes comandos:

```
> GRANT ALL PRIVILEGES ON barcosdb.* TO 'storm'@'localhost';
```

```
> FLUSH PRIVILEGES;
```

Para comprobar que el usuario se ha creado correctamente se volverá a iniciar sesión por consola usando el usuario recién creado en vez del usuario root. Si todo ha ido bien no se tendrá ningún problema a la hora de entrar.

Como paso final en la creación de la base de datos, se creará la base de datos que se utilizará para simular la base de datos del sistema AIS, puesto que, como se ha dicho anteriormente, no se ha realizado conexión alguna con internet para los componentes del proyecto.

Esta base de datos se creará en el sistema operativo anfitrión, es decir, en Windows 10 del portátil, para ello lo más sencillo es descargar XAMPP, puesto que solo interesa MySQL, a la hora de instalarlo se desmarcarán todas las opciones dejando solo la opción de MySQL.

Los siguientes pasos que seguir son los mismos que para crear la base de datos en Ubuntu, se accederá a la base de datos, se creará la base de datos y un usuario con privilegios de acceso a la base creada, con la diferencia de que se cambiará el parámetro del 'host_name' a la

Como el sistema operativo es Ubuntu la instalación de MySQL es muy sencilla, tan solo hay que usar un par de comandos:

```
$ sudo apt-get upgrade
```

```
$ sudo apt-get install mysql-server
```

Una vez instalado mysql, se pasará a crear la base de datos, para ello simplemente se entrará en MySQL con el comando:

```
$ mysql -u root -p
```

Por defecto, el usuario root tiene como contraseña root, cuando se ejecuta el comando la pedirá, se introduce y ya se está dentro.

En este caso, el nombre de la base de datos será barcosdb, para su creación, el comando a utilizar es:

```
> create database barcosdb;
```

El siguiente paso es crear la tabla, para ello se usará el comando create table:

```
> CREATE TABLE barcosdb.barcos (MMSI int(11) NOT NULL, vessel varchar(45), callsign varchar(45), IMO
```


hora de crear el usuario, si se quiere que ese usuario sea válido para cualquier maquina se sustituirá por '%', en este caso se pondrá la IP correspondiente a la máquina virtual '192.168.1.200'. El comando quedaría de la siguiente manera:

> **CREATE USER 'storm'@'192.168.1.200' IDENTIFIED BY 'ap.storm';**

Para finalizar esta última base de datos se irá a la página de AISHUB y se cogerá toda la información de un par de barcos, como por ejemplo dos de la siguiente figura:



SEA CARGO EXPRESS,		CHEMICAL MARKETER, Malta	
DETAILS		DETAILS	
MMSI:	229061000	MMSI:	249814000
CallSign:	9HA3034	CallSign:	9HA2018
IMO:	9358060	IMO:	9304291
Size:	119 x 18 (m)	Size:	134 x 20 (m)
Draught:	5.2	Draught:	6
Last update:	08-09-2017 15:36 UTC	Last update:	08-09-2017 15:36 UTC
Destination:	BERGEN	Destination:	ALGICERAS
ETA:	n/a	ETA:	n/a
Coordinates :	60.3923, 5.3081	Coordinates :	49.9318, -3.2755
Course:	237	Course:	254°
Speed:	0	Speed:	14.2 kn
Heading	44	Heading	254
Stations	 2535 Trondheim, Norway	Stations	 2472 Devon, United Kingdom

Figura 17. Barcos de ejemplo

Para ver la ruta que hacen los barcos se creará otra tabla en la base de datos, esta recogerá el historial de posiciones de los barcos, como se está trabajando sin una base de datos real y los datos se actualizan manualmente se pondrá un máximo de 5 posiciones.

Historial	
Campo	Tipo
MMSI	Int (11)
vessel	varchar (45)
posicion1	varchar (45)
posicion2	varchar (45)
posicion3	varchar (45)
posicion4	varchar (45)
posicion5	Varchar (45)

Tabla 2. Estructura de la tabla historial

Esta tabla tendrá como nombre historial y será una tabla muy básica donde solo tendremos el identificador MMSI, el nombre del barco y sus posiciones, como anteriormente se ha detallado la forma de crear una base de datos, así como lo necesario para que sea accesible no se volverá a detallar aquí.

Esta tabla tan solo se creará en la parte del servidor, es decir, en el sistema Ubuntu ya que está se irá actualizando gracias a Apache Storm.

4.4. Apache Storm

Como nexo central del proyecto tenemos es sistema *Apache Storm*, encargado de realizar la recogida de información, realizar análisis y manipular los datos. La instalación de este es bastante sencilla, puesto que solo se necesita descargarlo y descomprimirlo.

Para descargarlo vamos a la página web oficial de *Apache Storm*, <http://storm.apache.org/downloads.html>, y se elegirá la versión pertinente, en este caso, se descargará la versión 1.1.0, la más reciente hasta el momento.

Una vez descargado, se descomprimirá el archivo en una carpeta, esta carpeta contendrá todo lo necesario para poner marcha un *cluster Storm*, excepto *zookeeper* que lo se tendría que descargar aparte, al igual que con *Storm*, desde a su página web, <http://apache.rediris.es/zookeeper/>, se descargara la versión pertinente.

4.4.1. Configuración

Como primer paso antes de montar el sistema *Storm* hay que configurarlo, primero *zookeeper*, para ello se abrirá el archivo *zoo.cfg* y se añadirán los siguientes parámetros:

```
tickTime=2000
```

```
dataDir=/zookeeper/data
```

```
clientPort=2181
```

```
initLimit=5
```

```
syncLimit=2
```

Lo siguiente es configurar *Storm*, al igual como con *zookeeper* se abrirá el archivo de configuración *storm.yaml* y se añadirán los siguientes parámetros:

```
storm.zookeeper.servers:
```

```
- “localhost”
```

```
storm.local.dir: “/apache-storm/data”
```

```
nimbus.host: “localhost”
```

```
supervisor.slots.ports:
```

```
- 6700
```

```
- 6701
```

```
- 6702
```

```
- 6703
```

En la configuración se indica donde guardar los datos generados por *zookeeper* y *Storm* además de puertos activos y los diferentes componentes de los mismos.

4.4.2. Topología

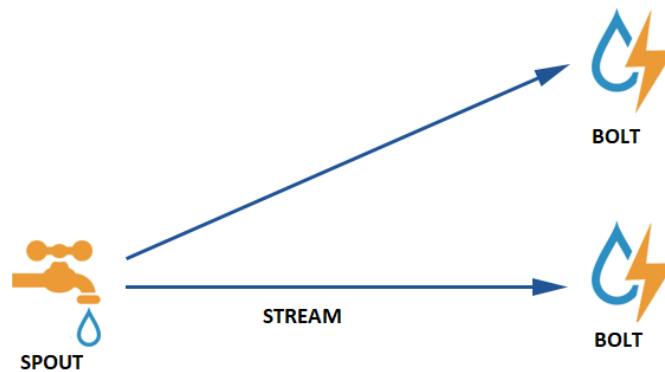


Figura 18. Topología de Storm

Tras configurar el sistema lo primero que se hará es definir la topología, en este caso la topología es sencilla basta con un *spout* y dos *bolt*, el *spout* recogerá la información de la base de datos AIS y se la pasará a ambos *bolt*, uno será el encargado de guardar la información en la base de datos, concretamente en la tabla barcos, para que esté disponible para nuestra aplicación *Android*, y el otro será el que irá actualizando la tabla historial, guardando así la ruta que seguirán estos barcos.

```
TopologyBuilder builder = new TopologyBuilder();
```

Para ello, se creará un *TopologyBuilder*, llamado *builder*, este sirve para definir cada *spout* y cada *bolt*, además de decir la relación entre ellos, en este caso se creará el *spout* y a su flujo se llamará *data-spout*.

```
builder.setSpout("data-spout", new DataSpout());
```

Seguidamente se crearán los *bolt*, el primero llamado *InsertBolt* y el segundo *HistorialBolt*, como se quiere que ambos reciban la información del *spout* indistintamente se agruparán los flujos mediante *shuffleGrouping*.

```
builder.setBolt("insert-bolt", new InsertBolt())  
    .shuffleGrouping("data-spout");  
builder.setBolt("historial-bolt", new HistorialBolt())  
    .shuffleGrouping("data-spout");
```

Como el sistema está montado en local, la topología se ejecutará sin llegar a subirse al cluster, además este método está pensado para desarrollo, solo ejecutará una única JVM por tanto se ahorrarán recursos.

```
LocalCluster cluster = new LocalCluster();  
cluster.submitTopology("Topology", config, builder.createTopology());
```

Como se quiere comprobar su funcionamiento y no que este todo el tiempo en marcha, la topología se apagará pasado un tiempo, para ello se usará:

```
Thread.sleep(100000);  
cluster.shutdown();
```

Con ello la topología estará activa por tan solo 100 segundos, tiempo más que suficiente para comprobar su correcto funcionamiento, con las entradas disponibles en la base de datos simulada.

4.4.3. Spout

El siguiente paso por realizar es programar el *Spout*, este es el encargado de recoger la información y pasarla a la topología, este, se conectará con la base de datos que se preparó para simular una base de datos AIS y recogerá toda la información, pasándola a ambos *bolt*.

Creamos la clase *DataSpout*, esta implementa interfaz *IRichSpout*, esta interfaz contiene los métodos *declareOutputFields*, *getComponentConfiguration*, *ack*, *activate*, *close*, *deactivate*, *fail*, *nextTuple* y *open*.

open – provee al *spout* de un ambiente para ejecutarse. Este método es al que se llama para inicializar el *spout*.

nextTuple – envía la información recibida a la topología.

close – este método es llamado cuando se quiere apagar el *spout*.

declareOutputFields – declara el esquema del flujo saliente del *spout*.

ack – método llamado cuando la información ha sido completamente procesada a través de la topología.

fail – al igual que *ack* salvo que es llamado cuando la información no ha podido ser completamente procesada.

Lo primero es completar el método *open* que quedaría de la siguiente manera:

@Override

```
public void open (Map conf, TopologyContext context, SpoutOutputCollector collector)
{
    this.context = context;
    this.collector = collector;
}
```

conf – proporciona la configuración de Storm al *spout*

context – proporciona la información completa acerca del *spout* dentro de la topología, como su ID e información de entrada y salida.

collector – nos permite enviar la información a los bolts.

Vamos a usar una conexión JDBC con la base de datos, con lo que declaramos el nombre de la base de datos, dirección y credenciales de acceso.

```
// JDBC driver name and database URL
static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
static final String DB_URL = "jdbc:mysql://192.168.1.150/barcosdb";

// Database credentials
static final String USER = "storm";
static final String PASS = "ap.storm";
```

Como segundo paso vamos al método donde se enviará la información a la topología, *nextTuple*, este método es llamado periódicamente con el mismo bucle que el método *ack* y *fail*. Declaramos todos los datos que vamos a recibir de la base de datos y nos preparamos para realizar la conexión:

```
@Override  
public void nextTuple() {  
    Connection conn = null;  
    Statement stmt = null;  
    Integer MMSI = 0 ;  
    String vessel = "n/a" ;  
    String callsign = "n/a" ;  
    String IMO = "n/a" ;  
    String draught = "n/a" ;  
    String lastupdate = "n/a" ;  
    String destination = "n/a";  
    String ETA = "n/a" ;  
    String coordinates = "n/a" ;  
    String course = "n/a" ;  
    String speed = "n/a" ;  
    String heading = "n/a" ;  
    String stations = "n/a" ;  
    String size = "n/a";
```

Lo siguiente será realizar la conexión, para ello lo primero es registrar el driver JDBC con el método *Class.forName*, acto seguido abrimos la conexión con la base de datos con el método *DriverManager.getConnection*:

```
try{  
    //STEP 2: Register JDBC driver  
    Class.forName("com.mysql.jdbc.Driver");  
    //STEP 3: Open a connection  
    conn = DriverManager.getConnection(DB_URL,USER,PASS);
```

Preparamos y ejecutamos la sentencia SQL para recibir los datos:

```
//STEP 4: Execute a query  
stmt = conn.createStatement();  
String sql;  
sql = "SELECT * FROM barcos";  
ResultSet rs = stmt.executeQuery(sql);
```

Una vez tenemos los datos, lo siguiente será ponerlos en las variables y enviarlos, los enviamos todos juntos por comodidad y porque nos interesa que sea así, ya que solo tenemos un *bolt* que recibirá los datos, si tuviéramos otros *bolts* a los que enviar información podríamos hacerlo, primero tendríamos que declarar que información queremos enviar, ponerles una etiqueta y enviando al *bolt* correspondiente:

```
//STEP 5: Extract data from result set
while(rs.next()){
    //Retrieve by column name
    MMSI = rs.getInt("MMSI");
    vessel = rs.getString("vessel");
    callsign = rs.getString("callsign");
    IMO = rs.getString("IMO");
    draught = rs.getString("draught");
    lastupdate = rs.getString("lastupdate");
    destination = rs.getString("destination");
    ETA = rs.getString("ETA");
    coordinates = rs.getString("coordinates");
    course = rs.getString("course");
    speed = rs.getString("speed");
    heading = rs.getString("heading");
    stations = rs.getString("stations");
    size = rs.getString("size");

    this.collector.emit(new Values(MMSI, vessel, callsign, IMO, size, draught,
lastupdate, destination, ETA, coordinates, course, speed, heading, stations));
} //end while
```

Para finalizar con el método cerramos la conexión y limpiamos el entorno:

```
rs.close();
stmt.close();
conn.close();
}catch(SQLException se){
    //Handle errors for JDBC
    se.printStackTrace();
}catch(Exception e){
    //Handle errors for Class.forName
    e.printStackTrace();
```

```

}finally{
    //finally block used to close resources
    try{
        if(stmt!=null)
            stmt.close();
    }catch(SQLException se2){
    }// nothing we can do
    try{
        if(conn!=null)
            conn.close();
    }catch(SQLException se){
        se.printStackTrace();
    }//end finally try
} //end try
} //end next tuple

```

Los métodos *close*, *activate*, *desactivate*, *ack* y *fail* no les haremos nada, por tanto, los dejaremos así:

```

@Override
public void close() {}

@Override
public void activate() {}

@Override
public void deactivate() {}

@Override
public void ack(Object msgId) {}

@Override
public void fail(Object msgId) {}

```

Para finalizar completaremos el método *declareOutputFields*, este método es llamado para especificar como serán los datos que salen del *spout*:

```

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("MMSI", "vessel", "callsign", "IMO", "size", "draught",
    "lastupdate", "dastination", "ETA", "coordinates", "course", "speed", "heading",
    "stations"));
}

```

declarer – se usa para declarar el ID de la información enviada por el *spout*, los campos, etc.

4.4.4. Bolt – InsertBolt

Seguimos, con el sistema *Apache Storm*, ahora se implementará el *InsertBolt*, este componente es el que se encarga de recibir la información dada por el *Spout*, conectarse a la base de datos y guardar la información recibida.

Para empezar, creamos la clase *InsertBolt*, esta implementa la interfaz *IRichBolt*, que contiene los métodos *cleanup*, *execute*, *prepare*, *declareOutputFields* y *getComponentConfiguration*.

prepare – al igual que el método *open* del interfaz *IRichSpout*, este le da un ambiente al *bolt* para ejecutarse, y también lo llaman para inicializar el *bolt*

execute – este método procesa la información recibida, aquí es donde se realizan todas las operaciones

cleanup – cuando se va a apagar el *bolt* se llama a este método

Los otros dos métodos hacen lo mismo que los de la interfaz *IRichSpout* explicados anteriormente.

Se comenzará con el método *prepare*, este es muy similar al método *open* por lo que no se entrara en detalle.

```
public void prepare (Map conf, TopologyContext context, OutputCollector collector) {  
this.collector = collector; }
```

Seguidamente, detallamos el método *execute*, aquí se realizará todo lo necesario para poner los datos en la base de datos del servidor, lo primero será recoger la información que proporciona el *spout*:

@Override

```
public void execute (Tuple tuple) {  
Integer MMSI = tuple.getInteger(0);  
String vessel = tuple.getString(1);  
String callsign = tuple.getString(2);  
String IMO = tuple.getString(3);  
String size = tuple.getString(4);  
String draught = tuple.getString(5);  
String lastupdate = tuple.getString(6);  
String destination = tuple.getString(7);  
String ETA = tuple.getString(8);  
String coordinates = tuple.getString(9);  
String course = tuple.getString(10);  
String speed = tuple.getString(11);  
String heading = tuple.getString(12);  
String stations = tuple.getString(13);
```

Una vez recogida toda la información recibida, se puede hacer cualquier cosa con ella, en este caso tan solo se guardará en otra base de datos para tenerla lista para para la aplicación de *Android*.

AL igual que con el *spout* se hará una conexión a la base de datos mediante *JDBC*, por tanto, declaramos el nombre de la base de datos, dirección y credenciales de acceso.

```
// JDBC driver name and database URL
static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
static final String DB_URL = "jdbc:mysql://localhost/barcosdb";

// Database credentials
static final String USER = "storm";
static final String PASS = "ap.storm";
```

A continuación, se prepara la conexión y se realiza con un bloque *try-catch*:

```
Connection conn = null;
Statement stmt = null;
try{
    //Register JDBC driver
    Class.forName("com.mysql.jdbc.Driver");

    // Open a connection
    conn = DriverManager.getConnection(DB_URL, USER, PASS);
```

Una vez realizada la conexión se preparará la sentencia *SQL* para guardar los datos en la base de datos, lo primero será comprobar si hay algo con la misma clave primaria, es decir si ese barco ya está guardado en la base de datos, si lo está entonces se actualizará esa información:

```
stmt = conn.createStatement();
String sql;
sql = "SELECT * FROM barcos WHERE MMSI='"+MMSI+"'";
ResultSet rs = stmt.executeQuery(sql);
if(rs.next()){ rs.close(); //si hay algo -> update... si no hay -> insert
    String update = "UPDATE barcos SET vessel='"+vessel"',
callsign='"+callsign"', IMO='"+IMO"', size='"+size"', draught='"+draught"',
lastupdate='"+lastupdate"', destination='"+destination"', ETA='"+ETA"',
coordinates='"+coordinates"', course='"+course"', speed='"+speed"',
heading='"+heading"', stations='"+stations"' WHERE MMSI='"+MMSI+"'";
    stmt.executeUpdate(update);
```

```
}else{ rs.close();
```

```
String insert = "INSERT INTO barcos (MMSI, vessel, callsign, IMO, size, draught, lastupdate, destination, ETA, coordinates, course, speed, heading, stations) VALUES('"+MMSI+"', '"+vessel+"', '"+callsign+"', '"+IMO+"', '"+size+"', '"+draught+"', '"+lastupdate+"', '"+destination+"', '"+ETA+"', '"+coordinates+"', '"+course+"', '"+speed+"', '"+heading+"', '"+stations+"')";
```

```
stmt.executeUpdate(insert); }
```

A continuación, se cierra la conexión y se informa de que la información ha sido correctamente procesada, por tanto:

```
stmt.close();
```

```
conn.close();
```

```
}catch(SQLException se){
```

```
//Handle errors for JDBC
```

```
se.printStackTrace();
```

```
}catch(Exception e){
```

```
//Handle errors for Class.forName
```

```
e.printStackTrace();
```

```
}finally{
```

```
//finally block used to close resources
```

```
try{
```

```
if(stmt!=null)
```

```
stmt.close();
```

```
}catch(SQLException se2){
```

```
// nothing we can do
```

```
try{
```

```
if(conn!=null)
```

```
conn.close();
```

```
}catch(SQLException se){
```

```
se.printStackTrace();
```

```
//end finally try
```

```
//end try
```

```
collector.ack(tuple);
```

Antes de cerrar el método se añadirá un tiempo de espera para que no esté realizando conexiones con demasiada frecuencia, en un sistema real los barcos actualizan información cada pocos segundos, como no es este caso, se pondrá un tiempo de esper de unos 1000 milisegundos.

```
try{Thread.sleep(1000);}catch(Exception e){e.printStackTrace();}
```

Los otros métodos no se van a usar por lo que simplemente se hace lo mismo que con el *Spout*:

```
@Override
```

```
public void cleanup() {}
```

```
@Override
```

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {}
```

Ahora ya se puede compilar y ejecutar la topología, al estar en modo local no hace falta subirlo al sistema *Storm*, simplemente se ejecuta la clase principal con *Storm* en marcha y se podrá ver si todo funciona o no.

Como estamos usando una base de datos simulada para saber si todo funciona en tiempo real tenemos que cambiar los datos en la base de datos mientras la topología se está ejecutando. Al hacerlo se ve como todo funciona perfectamente y que el sistema actualiza nuestra base de datos en tiempo real.

4.4.5. Bolt – HistorialBolt

Para finalizar con el sistema *Apache Storm*, se implementará el *HistorialBolt*, este componente también recibirá la información dada por el *Spout*, y se conectará a la base de datos, este se encargará de actualizar la tabla historial poniendo las últimas 5 posiciones registradas de los barcos.

Al igual que con el anterior, para empezar, creamos la clase *HistorialBolt*, esta implementa la interfaz *IRichBolt*, que contiene los métodos *cleanup*, *execute*, *prepare*, *declareOutputFields* y *getComponentConfiguration*.

Estos métodos son los mismos que los anteriores y por tanto no se entrará aquí en detalles de los mismos. Así pues, comenzamos directamente a detallar el método *prepare*, este es el método que variará con respecto al otro *Bolt*.

```
public void prepare (Map conf, TopologyContext context, OutputCollector collector) {  
this.collector = collector; }
```

Seguimos con el método *execute*, donde se realizará lo necesario para poner los datos en nuestra base, lo primero será recoger la información que nos proporciona el *spout*:

@Override

```
public void execute (Tuple tuple) {  
    Integer MMSI = tuple.getInteger(0);  
    String vessel = tuple.getString(1);  
    String coordinates = tuple.getString(9);
```

En este caso solo necesitaremos de tres datos dados por el *spout*, una vez recogidos, los trataremos para hacer que sea se guarde como las últimas 5 posiciones recibidas.

Como hicimos con el *spout* y el bolt anterior, haremos una conexión a la base de datos mediante *JDBC*, por tanto, declaramos el nombre de la base de datos, dirección y credenciales de acceso.

```
// JDBC driver name and database URL  
static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";  
static final String DB_URL = "jdbc:mysql://localhost/barcosdb";  
  
// Database credentials  
static final String USER = "storm";  
static final String PASS = "ap.storm";
```

A continuación, preparamos la conexión y la realizamos con un bloque *try-catch*:

```
Connection conn = null;  
Statement stmt = null;  
  
try{  
    //Register JDBC driver
```

```
Class.forName("com.mysql.jdbc.Driver");
```

```
// Open a connection
```

```
conn = DriverManager.getConnection(DB_URL, USER, PASS);
```

Una vez realizada la conexión se preparará la sentencia *SQL* para recoger los datos que pueda haber en la tabla historial, lo primero será comprobar si hay algo con la misma clave primaria, es decir si ese barco ya está guardado en la base de datos, si lo está entonces se actualizará esa información, si hay algo se recogerá en la variable posición [], esta se creó como un vector de strings para guardar las 5 posiciones de los barcos, y realizar el movimiento de los datos.

```
stmt = conn.createStatement();
```

```
String sql;
```

```
sql = "SELECT * FROM historial WHERE MMSI='"+MMSI+"'";
```

```
ResultSet rs = stmt.executeQuery(sql);
```

```
//Extract data from result set
```

```
if(rs.next()){
```

```
    posicion[0] = rs.getString("posicion1");
```

```
    posicion[1] = rs.getString("posicion2");
```

```
    posicion[2] = rs.getString("posicion3");
```

```
    posicion[3] = rs.getString("posicion4");
```

```
    posicion[4] = rs.getString("posicion5");
```

```
    if(!coordinates.equals(posicion[0])){
```

```
        posicion[4]=posicion[3];
```

```
        posicion[3]=posicion[2];
```

```
        posicion[2]=posicion[1];
```

```
        posicion[1]=posicion[0];
```

```
        posicion[0]=coordinates;
```

```
    }
```

```
rs.close(); //si hay algo -> update... si no hay -> insert
```

```
String update = "UPDATE historial SET vessel='"+vessel+''',  
posicion1='"+posicion[0]+'',                posicion2='"+posicion[1]+'',
```

```

    posicion3="'+posicion[2]+'',                posicion4="'+posicion[3]+'',
    posicion5="'+posicion[4]+' WHERE MMSI="'+MMSI+'";

        stmt.executeUpdate(update);

    }

    else{

        rs.close();

        String insert = "INSERT INTO historial (MMSI ,vessel , posicion1,
posicion2,          posicion3,          posicion4,          posicion5) VALUES
('"+MMSI+"', '"+vessel+"', '"+posicion[0]+'', '"+posicion[1]+'', '"+posicion[2]+'', '"+p
osicion[3]+'', '"+posicion[4]+'')";

        stmt.executeUpdate(insert);

    }

```

Antes de salir hay que cerrar la conexión e informar de que la información ha sido correctamente procesada, por tanto:

```

    stmt.close();
    conn.close();
}catch(SQLException se){
    //Handle errors for JDBC
    se.printStackTrace();
}catch(Exception e){
    //Handle errors for Class.forName
    e.printStackTrace();
}finally{
    //finally block used to close resources
    try{
        if(stmt!=null)
            stmt.close();
    }catch(SQLException se2){
        // nothing we can do
    }
    try{
        if(conn!=null)
            conn.close();
    }catch(SQLException se){

```

```
        se.printStackTrace();  
  
        }//end finally try  
  
    }//end try  
  
    collector.ack(tuple);
```

Los otros métodos no se van a usar por lo que simplemente hacemos lo mismo que anteriormente:

```
    @Override  
  
    public void cleanup() {}  
  
    @Override  
  
    public void declareOutputFields(OutputFieldsDeclarer declarer) {}
```

Para comprobar que todo funciona se compilará y ejecutará la topología, al estar en modo local no hace falta subirlo al sistema *Storm*, al igual que se hizo anteriormente, simplemente se ejecutará la clase principal, *Topology*, con *Storm* en marcha y se podrá ver su funcionamiento.

Al igual que antes, como se está usando una base de datos simulada para saber si todo funciona en tiempo real se tendrán que cambiar los datos en la base de datos mientras la topología está en ejecución. Al hacerlo se verá como todo funciona perfectamente y que el sistema actualiza la base de datos en tiempo real.

4.5 Aplicación Android

Esta es la última parte del proyecto, en esta se desarrolla una aplicación para *Android*, para ello vamos a utilizar la herramienta *Android Studio*, esta es la herramienta que más utilizan los desarrolladores puesto que la herramienta oficial para *Android* e incluye muchas facilidades para desarrollar aplicaciones.

La aplicación a desarrollar es simple, lo que hará es mostrar en un mapa la posición de los barcos recogido en la base de datos, para ello se necesitará permisos para conectarse a internet, pues, aunque sea una red local necesita estos permisos para comunicarse con otros dispositivos que no sean él mismo.

En cuanto a los elementos de la aplicación hay fundamentalmente 3:

- *AndroidManifest.xml*, este archivo describe la aplicación, es decir, indica las actividades, los permisos que se necesitan, las intenciones y la versión mínima de Android que se necesita para ejecutar nuestra aplicación, lo ideal sería coger la versión mínima que más porcentaje de dispositivos abarque, por eso se elige la versión *4.4 KitKat*, para ser compatible con el 90.1% de los dispositivos. Como se ha dicho antes nuestra aplicación solo necesita permisos para conectarse a internet, por tanto, lo declaramos de la siguiente manera:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

- *Layouts*, aquí es donde se define la vista de la aplicación, es decir, como se muestra por pantalla nuestra aplicación, por cada actividad se hace un *layout*, como la que se está desarrollando solo tiene una actividad, solo tiene un *layout* llamado *activity_maps.xml*, en este se muestra un mapa y se ajusta al ancho de nuestra pantalla, en nuestro caso se trata de una pantalla de 5 pulgadas, correspondería a la de un Nexus 5.
- *Clases Java*, por último, las clases de java, por cada actividad se hace una clase, en este caso se tiene una sola actividad y por tanto una sola clase llamada *MapsActivity.java*, en ella se realizará la conexión a base de datos y se recogerá toda la información necesaria para la aplicación, que para este caso es el nombre del barco y su posición, además se mostrará en el mapa dicha información.

4.5.1. activity_maps.xml

El *layout* es muy sencillo, como se ha dicho anteriormente solo pone el mapa y lo ajusta a toda la pantalla, el código es el siguiente:

```
<fragment
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:map="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:id="@+id/map"

android:name="com.google.android.gms.maps.SupportMapFragment"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="com.example.dario.mapappbq.MainActivity" />
```

4.5.2 *MapsActivity.java*

La clase principal que contiene la actividad es donde se va a realizar la conexión a la base de datos y muestra los datos en el mapa, para ello, aunque no es muy recomendable por motivos de compatibilidad y seguridad, vamos a realizar una conexión *JDBC* con la base datos. Primeramente, se declarará las variables a utilizar además de la sentencia *SQL* a ejecutar:

```
Connection connexion;  
String vessel;  
String coordinates;  
Statement stmt;  
ResultSet rs;  
String sql = "SELECT vessel, coordinates FROM barcos";
```

El siguiente paso es realizar la conexión y ejecutar la sentencia *SQL* dentro de un bloque *try-catch*:

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
    connexion = DriverManager.getConnection  
("jdbc:mysql://192.168.1.200/barcosdb", "android", "app.android")  
;  
    stmt = connexion.createStatement();  
    rs = stmt.executeQuery(sql);
```

Una vez realizada la conexión, se recogen los datos en las variables, y se muestra la posición de los barcos en el mapa. Aquí hay pequeño inconveniente, para mostrar los datos en el mapa se necesita tener las coordenadas como dos *double*, uno para la longitud y otro para la latitud, y la base de datos lo tiene guardado como un solo *string*, separando las coordenadas por una coma, por ello se realiza el cambio en la misma aplicación. El código final sería el siguiente:

```
while (rs.next()) {  
    vessel = rs.getString("vessel");  
    coordinates = rs.getString("coordinates");  
    //separar y convertir en double  
    String[] parts = coordinates.split(",");  
    Double coordinate1 = Double.parseDouble(parts[0]);  
    Double coordinate2 = Double.parseDouble(parts[1]);  
    //mostrar posición de los barcos  
    LatLng point = new LatLng(coordinate1, coordinate2);  
    mMap.addMarker(new  
MarkerOptions().position(point).title(vessel));  
}
```

Para finalizar se cierra la conexión y liberan recursos:

```
rs.close();  
stmt.close();  
connexion.close();
```

4.5.3. Resultados

Tras compilar y ejecutar la aplicación quedaría así:

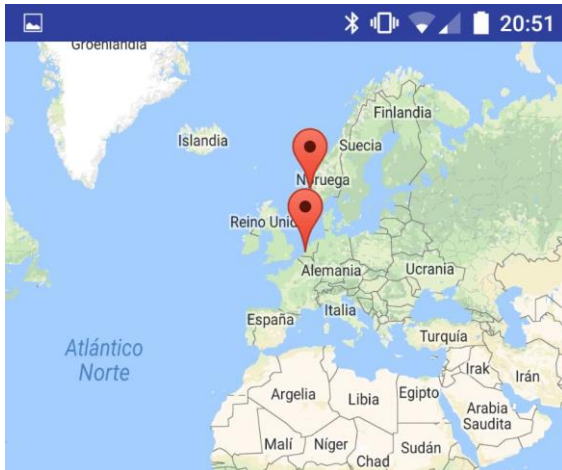


Figura 19. Aplicación final 1

Como se observa en la aplicación el sistema en conjunto de todo el proyecto funciona perfectamente, si tocamos encima de cualquier marcador nos indica que barco es.



Figura 21. Aplicación final 3

Como era de esperar, la aplicación recoge la posición de los barcos que se añadieron a la base de datos.

Para ver si funciona correctamente todo el proyecto, se añade la información de dos barcos más.

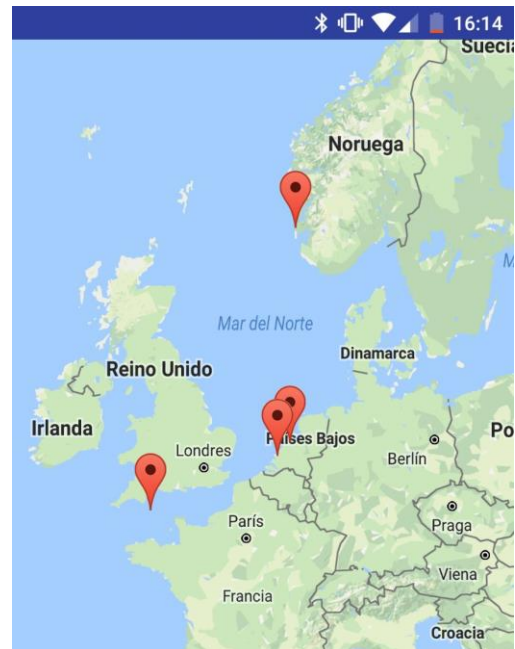


Figura 20. Aplicación final 2

Capítulo 5. Conclusiones y trabajo futuro

5.1. Conclusiones

En este proyecto final de carrera hemos realizado un sistema que sería capaz de realizar un análisis de los datos, como se ha desarrollado y probado en un entorno local no se ha llegado a comprobar su funcionamiento en un entorno real, con una cantidad de información superior, y con la posibilidad de hacer algún tipo de análisis sobre los datos, puesto que con 4 barcos no se pueden realizar demasiados.

La parte más desafiante del proyecto ha sido la creación de todo el sistema de *Apache Storm*, puesto que nunca lo había visto y tenía que empezar desde 0. Como se ha simulado una base de datos que contenía los datos y todo el entorno ha estado siempre muy controlado no he tenido muchos problemas a la hora de ver si todo funcionaba puesto que si algo fallaba sabía casi inmediatamente donde estaba el fallo.

La parte más desafiante del proyecto también es la más interesante, el sistema *Apache Storm* es una gran herramienta capaz de procesar una cantidad de información increíble en tiempo real, y con una gran cantidad de herramientas, que no he sido capaz de verlas todas, aunque me hubiese gustado. Lo que más leí cuando buscaba información es que era muy similar a *Hadoop*, pero que este no se comportaba tan bien en entornos con información en tiempo real, cosa que me ayudó mucho a decidir que herramienta iba a usar para realizar análisis de datos en el proyecto.

Una complicación añadida ha sido tener que usar *Ubuntu*, pues la poca familiaridad con este ha retrasado el desarrollo del proyecto, pero realizarlo en *Ubuntu* también ha tenido sus ventajas, por ejemplo, la instalación del sistema *Apache Storm* a resultado mucho más sencilla que en *Windows 10*, sistema con el que lo intenté, pero al final desistí perdiendo así bastante tiempo.

La lección más importante que aprendí realizando el proyecto y a la cual no le di mucha importancia es a tener una buena planificación y metodología a seguir. Cuando empecé a realizar el proyecto no tenía una idea clara de lo que quería hacer e iba haciendo conforme se me ocurrían las cosas, hasta que llegué a un punto en el que no podía avanzar y me di cuenta de que lo estaba haciendo de una manera que no podía salir nada bueno. Fue entonces cuando me decidí por planificar todo mejor y seguir una metodología.

Así pues, me puse a realizar el proyecto de nuevo desde otra perspectiva e intentando reutilizar todo lo posible para no perder demasiado tiempo, y así fue como llegué a realizar todo lo descrito en la memoria.

En cuanto a la aplicación realizada en Android me ha servido para familiarizarme más con el desarrollo de las mismas, puesto que durante se había visto algo. Lo más complicado de esta ha sido realizar con éxito una conexión *JDBC* con la base de datos, entendiéndolo por qué no es recomendable.

En cuanto a experiencia personal, el proyecto me ha parecido una oportunidad perfecta para abrirme paso a través del mundo del desarrollo de servidores y aplicaciones para Android, así como dar los primeros pasos con el análisis de datos en tiempo real con *Apache Storm*. Aunque me hubiera gustado poder probar mi proyecto en un entorno real con muchísimos más datos en tiempo real, no ha podido ser. Me quedo con la experiencia de haber realizado todo un sistema que este interconectado entre si todo haga su función tal y como lo planee desde un principio.

5.2. Líneas futuras

Soy consciente que este proyecto tiene muchas mejoras que se le podrían hacer, y también que hay mucho trabajo por delante, también sé que esto puede llegar a ser algo muy grande y bastante útil.

Una de las mejoras principales que se haría sería quitar las conexiones *JDBC* de la aplicación y realizar otro tipo de recogida de datos, como sería poner un servicio web en el servidor y que la aplicación recoja de ahí los datos. Otra cosa por realizar en la aplicación es la mejora de la mima en cuanto a contenido, se podría poner otra actividad que nos muestre la información detallada de cada barco y no solo su posición. También se pondría un botón de refresco de mapa, los barcos están en movimiento y ahora mismo el único modo de ver esos cambios sería reabrir la aplicación. Para que la aplicación le resulte interesante a más gente convendría poner más idiomas, además de hacer que sea compatible con otros dispositivos como los de la marca *Apple*.

Sin embargo, lo primero que habría que hacer es probar todo el sistema en un entorno real, puesto que como está montado ahora no tiene ninguna utilidad, aparte de servirme a mí para practicar. Para lanzarlo finalmente deberíamos ver cómo podríamos recoger la información de la base de datos real, la cual contiene la información de los barcos. Al hacer esto ya podríamos comenzar a realizar algunos análisis interesantes acerca de los barcos. También tendríamos que adaptar nuestra base de datos local para optimizar el almacenamiento de dicha información. Ahora mismo todas las entradas excepto el identificador del servicio móvil marítimo (MMSI).

Otra mejora sería la de hacer un *cluster* distribuido de *Apache Storm*, ya que este permite tener sus componentes distribuidos en diferentes servidores, con ello podríamos tener más tolerancia ante fallos, puesto que si un servidor fallara el propio Storm redistribuiría la carga entre los otros servidores.

Bibliografía

- [1] https://en.wikipedia.org/wiki/Automatic_identification_system#How_AIS_works
- [2] http://www.elfarodeluisu.es/el_sistema_ais.html
- [3] <https://www.navcen.uscg.gov/?pageName=AISworks>
- [4] <https://tuxpepino.wordpress.com/2007/04/17/historia-de-ubuntu-y-sus-versiones/>
- [5] https://es.wikipedia.org/wiki/Ubuntu#Inicio_de_Ubuntu
- [6] <http://storm.apache.org/releases/1.1.1/Concepts.html>
- [7] <https://www.adictosaltrabajo.com/tutoriales/introduccion-storm/#01>
- [8] https://www.tutorialspoint.com/apache_storm/
- [9] [https://es.wikipedia.org/wiki/Java_\(lenguaje_de_programaci%C3%B3n\)](https://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n))
- [10] <https://es.wikipedia.org/wiki/MySQL>
- [11] <https://www.mysql.com/>
- [12] <https://www.kantarworldpanel.com/global/smartphone-os-market-share/>
- [13] <https://es.wikipedia.org/wiki/Android>