

Universidad Politécnica de Madrid  
Escuela Técnica Superior de Ingenieros de Telecomunicación



**CÁLCULO DE RUTAS ÓPTIMAS EN UN SISTEMA DE  
EVACUACIÓN DE EMERGENCIA**

**TRABAJO FIN DE MÁSTER**

**Andreína Montoya Pow Chon Long**

2017

Universidad Politécnica de Madrid  
Escuela Técnica Superior de Ingenieros de Telecomunicación

**Máster Universitario en Ingeniería de Redes y Servicios  
Telemáticos**

**TRABAJO FIN DE MÁSTER**

**CÁLCULO DE RUTAS ÓPTIMAS EN UN SISTEMA DE  
EVACUACIÓN DE EMERGENCIA**

Autor  
**Andreína Montoya Pow Chon Long**

Director  
**Joaquín Salvachúa Rodríguez**

Departamento de Ingeniería de Sistemas Telemáticos

2017

## Resumen

El presente proyecto consiste en el desarrollo e implementación de un sistema que realice el cálculo de rutas dinámicas y óptimas para la evacuación de un edificio cuando se detecta una alerta de emergencia, considerando ciertas áreas que pueden encontrarse afectadas debido a este evento.

El desarrollo de este trabajo se basa en un sistema de evacuación a implementar en un edificio de la Escuela Superior Politécnica del Litoral (ESPOL), que se encuentra ubicada en la ciudad de Guayaquil, Ecuador. Siendo este proyecto un complemento del estudio "A Distributed System Model for Managing Data Ingestion in a Wireless Sensor Network" en el cual se define una red de comunicación distribuida que recibe, clasifica y transmite los datos que una red de sensores inalámbrica recopila del ambiente continuamente. Por lo cual este sistema debe consumir esa información y en caso de que se presente una alerta poder calcular las rutas de evacuación considerando sólo las áreas hábiles por donde las personas pueden evacuar.

El trabajo se encuentra estructurado en 5 capítulos, donde el primer capítulo expone una introducción, la motivación del desarrollo de este proyecto, el objetivo general y los objetivos específicos definidos.

En el segundo capítulo se encuentra el estado del arte, donde se describe detalladamente cada una de las tecnologías utilizadas para llevar a cabo este trabajo.

En el tercer capítulo se presenta la arquitectura planteada para el desarrollo del proyecto, la cual ha sido dividida en tres partes: Adquisición de Datos, Procesamiento de Datos y Visualización, para una mejor descripción, explicando el uso e integración de todas las tecnologías y librerías que se detallan en el segundo capítulo.

En el cuarto capítulo se describe el desarrollo e implementación del trabajo, detallando paso a paso el análisis y todo lo realizado para cumplir con los objetivos propuestos. Además se muestran los resultados de las

pruebas realizadas con todo el sistema integrado, donde se pueden ver las rutas generadas según los datos que se van recibiendo desde la red de sensores.

Finalmente, en el último capítulo se encuentran las conclusiones que se determinan a lo largo del desarrollo del proyecto y en base a los resultados obtenidos luego de su implementación. De igual manera se plantean recomendaciones para trabajos futuros.

# Abstract

The present project is about the development and implementation of a system that performs the calculation of dynamic and optimal routes for the evacuation of a building when an emergency alert is detected, considering certain areas that may be affected due to this event.

The development of this work is based on an evacuation system to be implemented in a building of the Escuela Superior Politecnica del Litoral (ESPOL), located in the city of Guayaquil, Ecuador. This project is a complement to the study "A Distributed System for Data Management in a Wireless Sensor Network" in which a distributed communication network is defined to receive, classify and transmit the data that a wireless sensor network continuously collects from the environment. Therefore, this system should consume this information and in case of an alert is able to calculate the routes of evacuation considering only the available areas where people can evacuate.

The work is structured in 5 chapters, where the first chapter presents an introduction, the motivation of the development of this project, the general objective and specific objectives defined,

In the second chapter is the state of the art, describing in detail each of the technologies used to carry out this work.

The third chapter presents the architecture proposed for the development of the project, which has been divided into three parts for a better description, explaining the use and integration of all the technologies and libraries detailed in the previous chapter.

The fourth chapter describes the development and implementation of this work, detailing step by step the analysis and everything done to get the proposed objectives. In addition, the results of the tests performed with the whole integrated system are shown, where you can see the routes generated according to the data that it is receiving from the sensor network.

Finally, the last chapter contains the conclusions that are determined throughout the development of the project and based on the results obtained after its implementation. Likewise, recommendations are made for future works.

# Índice General

Resumen	iii
Abstract	v
Índice General	vi
Índice de Figuras	ix
Índice de Tablas	xiii
Siglas	xiv
<b>1 Introducción</b>	<b>1</b>
1.1 Motivación . . . . .	2
1.2 Objetivos . . . . .	2
1.2.1 Objetivo General . . . . .	2
1.2.2 Objetivos Específicos . . . . .	2
<b>2 Estado del Arte</b>	<b>3</b>
2.1 Docker . . . . .	3
2.1.1 Características . . . . .	4
2.1.2 Arquitectura . . . . .	4
2.1.3 Componentes . . . . .	7
2.2 Apache Kafka . . . . .	9

2.2.1	Características . . . . .	9
2.2.2	Arquitectura . . . . .	10
2.2.3	Componentes . . . . .	14
2.3	Python . . . . .	15
2.4	Neo4j . . . . .	16
2.4.1	Características . . . . .	18
2.4.2	Base de Datos de Grafos vs. Base de Datos Relacional	20
2.4.3	Base de Datos de Grafos vs. Base de Datos NoSQL . .	23
2.4.4	Consultando Grafos con Cypher . . . . .	25
2.4.5	Cálculo de rutas . . . . .	26
2.4.6	Casos de Uso . . . . .	29
2.5	HTLM5 . . . . .	30
2.6	D3.js . . . . .	32
<b>3</b>	<b>Arquitectura</b>	<b>35</b>
3.1	Adquisición de Datos . . . . .	36
3.2	Procesamiento de Datos . . . . .	41
3.3	Visualización . . . . .	42
<b>4</b>	<b>Desarrollo e Implementación</b>	<b>45</b>
4.1	Simulación de Red de Sensores Inalámbrica . . . . .	45
4.2	Modelado de Datos en Neo4j . . . . .	48
4.3	Desarrollo del Consumidor de Kafka . . . . .	55

4.4	Visualización . . . . .	62
4.5	Pruebas . . . . .	67
<b>5</b>	<b>Conclusiones y Trabajos Futuros</b>	<b>70</b>
5.1	Conclusiones . . . . .	70
5.2	Trabajos Futuros . . . . .	71
	<b>Bibliografía</b>	<b>73</b>
	<b>Anexos</b>	<b>76</b>
	Anexo A . . . . .	76
	Anexo B . . . . .	78
	Anexo C . . . . .	82
	Anexo D . . . . .	88
	Anexo E . . . . .	89



# Índice de Figuras

Figura 1.	Docker Engine. . . . .	5
Figura 2.	Arquitectura de Docker. . . . .	6
Figura 3.	Máquinas virtuales vs Docker. . . . .	7
Figura 4.	Imagen base de Docker. . . . .	8
Figura 5.	Nueva imagen creada a partir de la imagen base de Docker. . . . .	8
Figura 6.	Nodo único - clúster con un único broker. . . . .	12
Figura 7.	Nodo único - clúster con múltiples brokers. . . . .	13
Figura 8.	Nodo múltiple - clúster con múltiples brokers. . . . .	13
Figura 9.	Clúster nodo único - único broker con topic con 3 particiones. . . . .	14
Figura 10.	Ejemplo de grafo en Neo4j. . . . .	17
Figura 11.	Nodos en Neo4j. . . . .	17
Figura 12.	Nodos en Neo4j. . . . .	18
Figura 13.	Relación muchos-a-muchos entre Personas y Departamentos en una base de datos relacional. . . . .	21
Figura 14.	Relación entre claves foráneas de las tablas Personas y Departamentos en una base de datos relacional. . . . .	21
Figura 15.	Modelo de datos de grafos de Personas y Departamentos. . . . .	23
Figura 16.	Relacionar bases de datos clave-valor con las bases de datos de grafos. . . . .	24
Figura 17.	Navegar bases de datos de documentos con bases de datos de grafos. . . . .	25

Figura 18.	Ejemplo de consulta con Cypher. . . . .	26
Figura 19.	Algoritmo de Búsqueda en Anchura. . . . .	27
Figura 20.	Gráfico de barras con D3. . . . .	33
Figura 21.	Perspectiva general de la arquitectura del sistema. . . .	36
Figura 22.	Arquitectura de una red de sensores inalámbricos con un clúster centralizado. . . . .	38
Figura 23.	Nodo de comunicación entre la WSN y el clúster Kafka.	39
Figura 24.	Arquitectura distribuida utilizando un clúster de Kafka.	40
Figura 25.	Esquema utilizado para el procesamiento de datos. . . .	42
Figura 26.	Esquema para visualización de rutas. . . . .	44
Figura 27.	Plano del área del edificio y ubicación de sensores. . . .	46
Figura 28.	Simulación de la red de sensores inalámbrica. . . . .	46
Figura 29.	Nodo de simulación. . . . .	47
Figura 30.	Diseño del primer grafo. . . . .	49
Figura 31.	Sentencia para crear nodos sensores. . . . .	49
Figura 32.	Sentencia para crear nodos de señalización. . . . .	50
Figura 33.	Sentencia para crear relación entre nodos sensores y señalización. . . . .	50
Figura 34.	Sentencia para crear relación entre nodos sensores. . . .	51
Figura 35.	Grafo de nodos sensores y señalización. . . . .	51
Figura 36.	Diseño del segundo grafo. . . . .	52
Figura 37.	Sentencia para crear nodos de puertas conectadas al corredor. . . . .	52

Figura 38.	Sentencia para crear nodos de intersecciones en el corredor.	53
Figura 39.	Sentencia para crear nodos de escaleras. . . . .	53
Figura 40.	Sentencia para crear relaciones entre nodos. . . . .	53
Figura 41.	Grafo de la primera planta del edificio. . . . .	54
Figura 42.	Diseño de asociación de los dos grafos. . . . .	54
Figura 43.	Sentencia para crear etiqueta para relacionar los dos grafos.	55
Figura 44.	Se importa librería para desarrollar consumidor. . . . .	56
Figura 45.	Código para consumir del clúster de Kafka. . . . .	56
Figura 46.	Se importa librería para conectarse con Neo4j. . . . .	56
Figura 47.	Código para conectarse con Neo4j. . . . .	56
Figura 48.	Código para verificar si el nodo ya se encuentra en alerta.	57
Figura 49.	Código para verificar si un nodo se encuentra adyacente a otro ya alertado. . . . .	58
Figura 50.	Código para calcular rutas con Neo4j. . . . .	58
Figura 51.	Código para calcular rutas con Dijkstra en Neo4j. . . . .	59
Figura 52.	Resultado en formato JSON. . . . .	60
Figura 53.	Código para reiniciar valores en propiedades. . . . .	61
Figura 54.	Importar librerías para trabajar con Flask y Flask-SocketIO. . . . .	61
Figura 55.	Iniciar aplicación. . . . .	61
Figura 56.	Ejecución de método en segundo plano. . . . .	61
Figura 57.	Método que consume del clúster de Kafka. . . . .	62

Figura 58. Método para parámetros modificados durante la generación de las rutas. . . . .	62
Figura 59. Elementos de la página web. . . . .	63
Figura 60. Función JavaScript para presentar las rutas. . . . .	64
Figura 61. Creación de enlaces en grafos para visualización. . . . .	64
Figura 62. Creación de nodos en grafos para visualización. . . . .	65
Figura 63. Se agregan nodos y enlaces a la simulación. . . . .	65
Figura 64. Conexión con el consumidor de Python desde JavaScript. . . . .	66
Figura 65. Interfaz web para simulación de señalización. . . . .	66
Figura 66. Imágenes de Docker. . . . .	67
Figura 67. Señalización cuando se alerta el nodo sensor S5 mientras el nodo S4 ya está alertado. . . . .	68
Figura 68. Rutas generadas cuando se alerta el nodo sensor S5 mientras el nodo S4 ya está alertado. . . . .	68
Figura 69. Señalización cuando se alerta el nodo sensor S1 mientras los nodos S4, S5 y S2 ya están alertados. . . . .	69
Figura 70. Rutas generadas cuando se alerta el nodo sensor S1 mientras los nodos S4, S5 y S2 ya están alertados. . . . .	69

## Índice de Tablas

Tabla 1.	Descripción de sensores. . . . .	47
Tabla 2.	Relación entre nodos sensores y señalización. . . . .	50
Tabla 3.	Asociación entre nodos sensores y grafo de la planta del edificio. . . . .	55

## Siglas

ACID	Atomicidad, Consistencia, Aislamiento y Durabilidad
API	Interfaz de Programación de Aplicaciones
CEO	Chief Executive Officer
CEP	Procesamiento de Eventos Complejos
CERN	Conseil Européen pour la Recherche Nucléaire
CLI	Command Line Interface
CSS	Cascading Style Sheets
Etc	Etcétera
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
MB	MegaBytes
PaaS	Platform as a Service
SaaS	Software as a Service
SVG	Scalable Vector Graphics
TBs	TeraBytes
TCP	Protocolo de Control de Transmisión
VM	Virtual Machine
WPAN	Red Inalámbrica de Área Personal
WSN	Wireless Sensor Network, Red de Sensores Inalámbrica

# 1 Introducción

Los sistemas de evacuación que se encuentran implementados actualmente en diferentes infraestructuras ya sean públicas o privadas cuentan en su mayoría con señalización estática ubicada en lugares estratégicos donde puedan ser claramente visibles para dirigir a las personas que se encuentran dentro del lugar hacia las salidas de emergencias en caso de que se presente algún evento de alerta.

Sin embargo, puede no ser suficiente o correcta la aplicación de este tipo de señalización para ciertas situaciones donde la ruta de evacuación que se debe indicar depende de las salidas que se encuentren habilitadas y de las áreas que no hayan sido afectadas por el evento de emergencia, ya que podría provocar confusión e incluso dirigir a las personas a salidas donde puedan estar expuestas a situaciones de peligro.

Debido a esto y considerando el avance tecnológico continuo es posible plantear nuevos modelos que permitan mejorar los sistemas de evacuación, valiéndose de sensores que recopilen información sobre el ambiente del edificio como la temperatura, el nivel de sustancias tóxicas, entre otros, para poder conocer el estado de las diferentes áreas del edificio y junto con el uso de nuevas tecnologías se pueda automatizar el sistema, generando de forma más rápida y dinámica las rutas óptimas con el fin de minimizar el tiempo de evacuación.

El desarrollo de este trabajo se enfoca en el cálculo y generación de rutas más cortas hacia las salidas de evacuación a partir de la información que se obtiene del modelo definido en el estudio "A Distributed System Model for Managing Data Ingestion in a Wireless Sensor Network", donde se plantea el diseño de una red de comunicación distribuida para recolectar y transmitir de manera continua los datos que se recopilan de una red de sensores inalámbrica a implementar en el edificio administrativo de la Escuela Superior Politécnica del Litoral (ESPOL), que cuenta con 1692 m<sup>2</sup> y se encuentra ubicado en la ciudad de Guayaquil, Ecuador. [1]

Para realizar el cálculo de rutas se utilizarán técnicas de big data que permitan trabajar con la información que se obtiene continuamente de la red de sensores, además de modelar los datos necesarios para generar las rutas de evacuación de una forma dinámica y automática descartando las áreas que se encuentren inhabilitadas al detectar una alerta en un nodo sensor de la red. De igual manera se propone el desarrollo de una interfaz gráfica donde se

puedan visualizar las rutas que se obtienen como resultado del cálculo según las alertas presentadas.

## **1.1 Motivación**

Los sistemas de evacuación forman parte fundamental del plan de seguridad que se implementa en las diferentes infraestructuras con el fin de salvaguardar la vida de las personas que trabajan o visitan estos edificios, por lo cual la motivación de este trabajo es el poder aportar por medio de nuevas tecnologías y procedimientos a la mejora de estos sistemas para reducir el número de víctimas que por motivos de desconocimiento y falta de información sobre rutas de evacuación puedan quedar expuestos a situaciones que pongan en peligro su vida.

## **1.2 Objetivos**

### **1.2.1 Objetivo General**

El objetivo general de este trabajo es comprender el uso de las diferentes tecnologías de big data para realizar el cálculo de rutas de evacuación más cortas, dinámicas y óptimas dentro de un edificio.

### **1.2.2 Objetivos Específicos**

- Comprender y utilizar tecnologías que permiten trabajar con flujos de datos continuos recopilados por la red de sensores para poder realizar su análisis y procesamiento.
- Analizar y modelar los datos necesarios para contar con un diseño que permita calcular las rutas dentro del edificio.
- Desarrollar una aplicación que permita calcular las rutas de evacuación de forma dinámica, considerando las áreas que se encuentran habilitadas para la evacuación.
- Diseñar y desarrollar una interfaz gráfica donde se puedan visualizar las rutas calculadas de forma dinámica.



## 2 Estado del Arte

En este capítulo se encuentra una descripción y características principales de las tecnologías que se utilizan para el desarrollo e implementación de este trabajo.

### 2.1 Docker

Docker comenzó como un proyecto de código abierto en el año 2013 en dotCloud, una compañía de plataforma como servicio (PaaS), siendo esto una plataforma y un entorno que permiten a los desarrolladores crear aplicaciones y servicios que funcionen a través de Internet, alojados en la nube. Inicialmente, Docker era una extensión de la tecnología que la compañía había desarrollado para ejecutar su negocio en la nube en miles de servidores. Está escrito en Go, un lenguaje de programación desarrollado por Google con su sintaxis basada en C. Su avance fue rápido de seis a nueve meses, la compañía contrató a un nuevo director ejecutivo (CEO), se unió a la Fundación Linux, cambió su nombre a Docker Inc. y anunció que cambió su enfoque al desarrollo de Docker y al ecosistema de Docker. [2]

Docker es una plataforma para desarrolladores y administradores de sistemas que permite desarrollar, enviar y ejecutar aplicaciones, proporciona la facilidad de separar las aplicaciones de la infraestructura para así poder entregar software de una forma más rápida. Aprovechando las metodologías de Docker para enviar, probar e implementar código rápidamente, se puede reducir significativamente el retraso entre escribir código y ejecutarlo en entorno de producción.

Provee la capacidad de empaquetar y ejecutar una aplicación en un entorno aislado llamado contenedor. El aislamiento y la seguridad permiten ejecutar muchos contenedores simultáneamente en un host determinado. Los contenedores son ligeros porque no necesitan la carga adicional de un hipervisor como es el caso de las máquinas virtuales, sino que se ejecutan directamente dentro del núcleo de la máquina host. Esto significa que se puede ejecutar más contenedores en una combinación de hardware dada que si se estuviera utilizando máquinas virtuales. Incluso se puede ejecutar contenedores Docker dentro de máquinas host que son en realidad máquinas virtuales. [3]

### 2.1.1 Características

Las principales características de Docker son:

- **Portabilidad:** Las imágenes de Docker pueden ser desplegadas en cualquier otro sistema que soporte esta tecnología, de esta manera no es necesario volver a instalar y configurar todas las aplicaciones que se utilizan, ejecutándose todo de la misma manera en cualquier entorno.
- **Ligereza:** Los procesos de Docker son mucho más ligeros, debido a que no emula o virtualiza una máquina y su sistema operativo como se realiza en el caso de la virtualización y no es necesario el sistema de archivos completo del sistema operativo invitado, es por esto que se puede aprovechar de mejor manera el hardware ya que aumenta la cantidad de servicios que se pueden tener en una máquina utilizando una fracción de espacio de almacenamiento del que es necesario en la virtualización.
- **Autosuficiencia:** Un contenedor Docker es autosuficiente ya que sólo se necesita de la imagen del contenedor para poder desplegar los servicios que éste contiene. [4]
- **Estándar:** Los contenedores de Docker se basan en estándares abiertos y se ejecutan en todas las principales distribuciones de Linux, Microsoft Windows y en cualquier infraestructura incluyendo máquinas virtuales, bare-metal y en la nube.
- **Seguridad:** Los contenedores Docker aíslan aplicaciones entre sí y de la infraestructura subyacente. Docker proporciona el aislamiento por defecto más fuerte para que en caso de que se presente un problema en alguna aplicación se limite a un único contenedor en lugar de toda la máquina. [5]

### 2.1.2 Arquitectura

Docker usa una arquitectura cliente-servidor y consiste de:

- Docker Engine que es una ligera y potente tecnología de contenedorización de código abierto combinada con un flujo de trabajo para construir y contenedorizar aplicaciones.

- Docker Hub que proporciona software como un servicio (SaaS) donde se puede compartir y administrar las aplicaciones.

Docker Engine es la parte principal de Docker, es una aplicación cliente-servidor que contiene:

- Un servidor que es un tipo de programa de larga ejecución llamado proceso "daemon" (demonio).
- Una API REST que especifica las interfaces que los programas pueden usar para hablar con el daemon e instruirle qué hacer.
- Un cliente que es una interfaz de línea de comandos (CLI).

En la Figura 1 se muestran los elementos mencionados anteriormente, donde se puede observar que la CLI utiliza la API REST de Docker para controlar o interactuar con el daemon Docker a través de scripts o comandos directos de la CLI. Además se puede ver que el daemon crea y administra objetos de Docker, como imágenes, contenedores, redes y volúmenes. [3]

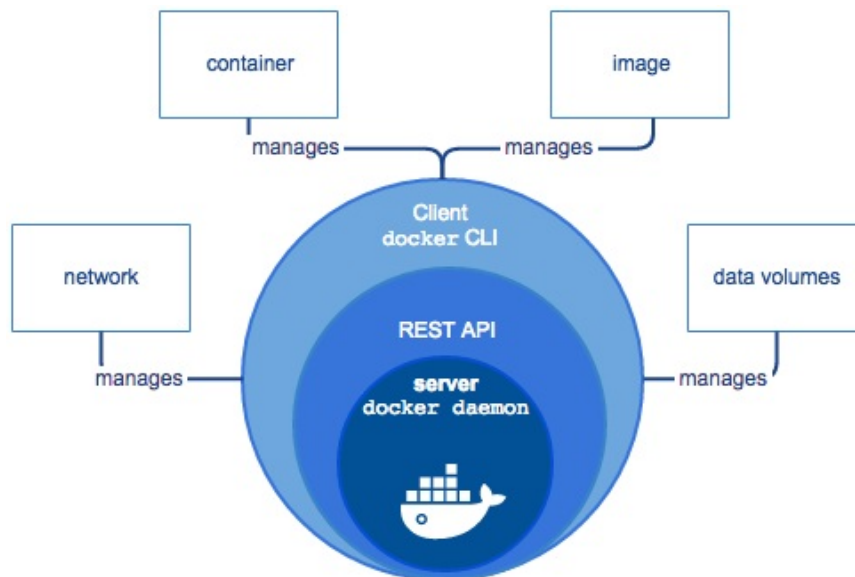


Figura 1. Docker Engine. [3]

El cliente de Docker habla con el daemon que hace el trabajo de crear, correr y distribuir los contenedores. El cliente y el daemon pueden ejecutarse en el mismo sistema, o se puede conectar un cliente Docker a un daemon Docker remoto.

En la Figura 2 se puede observar la arquitectura detallada de Docker, donde el daemon Docker escucha las solicitudes del API de Docker y administra objetos de Docker como imágenes, contenedores, redes y volúmenes. Un daemon también puede comunicarse con otros daemons para administrar servicios de Docker. Además se puede apreciar el cliente Docker que es la principal forma en que muchos usuarios de Docker interactúan con Docker. Y la parte del registro de Docker que es donde se almacenan imágenes de Docker. Docker Hub y Docker Cloud son registros públicos que cualquier persona puede usar y Docker está configurado para buscar imágenes en Docker Hub de forma predeterminada. [3]

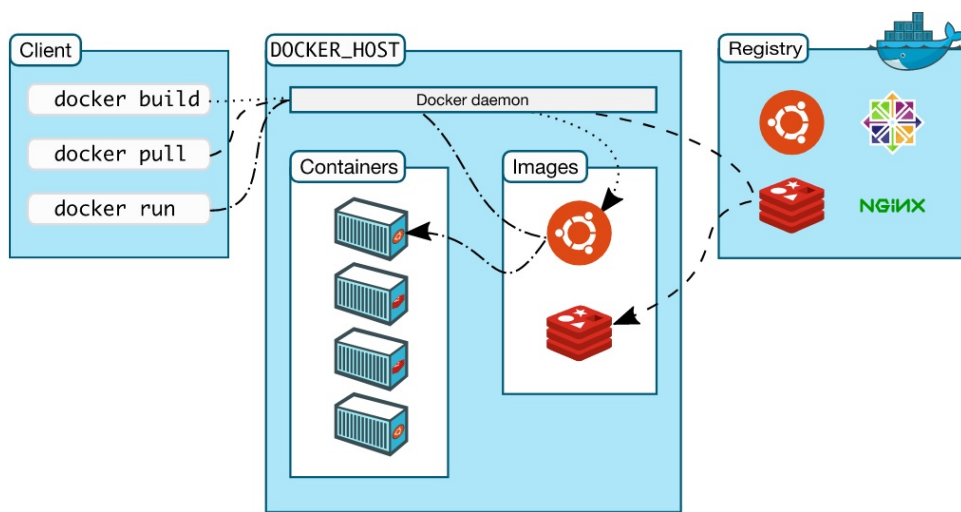


Figura 2. Arquitectura de Docker. [3]

Los contenedores y las máquinas virtuales tienen beneficios similares de aislamiento y asignación de recursos, pero funcionan de manera diferente porque los contenedores virtualizan el sistema operativo en lugar del hardware, los contenedores son más portátiles y eficientes, en la Figura 3 se muestran los esquemas que manejan las máquinas virtuales y Docker. Donde se puede observar que las máquinas virtuales (VM) son una abstracción del hardware físico que convierte a un servidor en muchos servidores. El hipervisor permite que múltiples máquinas virtuales se ejecuten en una sola máquina. Cada VM incluye una copia completa de un sistema operativo, una o más aplicaciones, binarios y bibliotecas necesarios, ocupando decenas de GB y pueden presentar lentitud para arrancar. Mientras que los contenedores son una abstracción en la capa de la aplicación que agrupa código y dependencias. Múltiples contenedores pueden ejecutarse en la misma máquina y compartir el núcleo del sistema operativo con otros contenedores, cada uno ejecutándose como procesos aislados en el espacio del usuario. Los contenedores ocupan menos espacio que las máquinas virtuales,

por lo general las imágenes de contenedores suelen tener decenas de MB de tamaño y se inician casi al instante. [5]

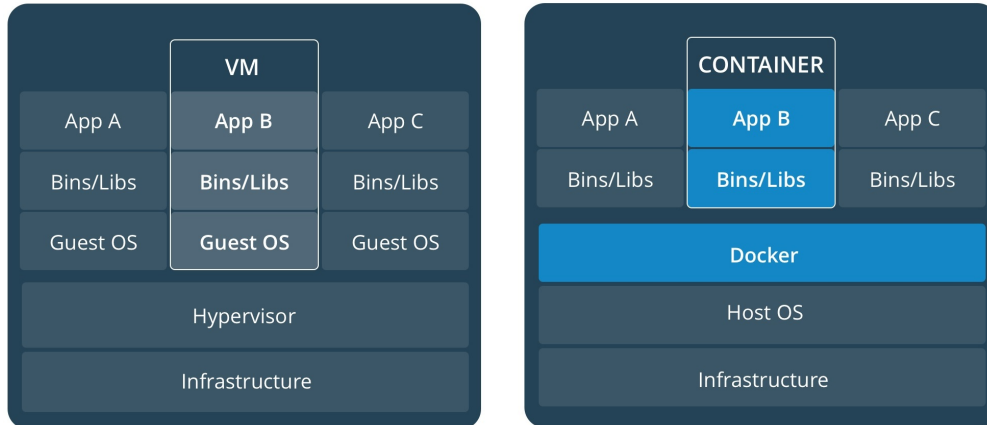


Figura 3. Máquinas virtuales vs Docker. [5]

### 2.1.3 Componentes

Entre los componentes de Docker se encuentran:

- **Imágenes:** Una imagen Docker es una colección de todos los archivos que componen una aplicación de software ejecutable, esto incluye la aplicación, más todas las bibliotecas, binarios y otras dependencias, como descriptores de despliegue, etc. Estos archivos en la imagen Docker son de sólo lectura, por lo tanto, el contenido de la imagen no se puede alterar. Si se desea modificar el contenido de una imagen, la única opción que Docker permite es añadir otra capa con los nuevos cambios. La arquitectura de imagen de Docker aprovecha de manera efectiva este concepto de capas para añadir perfectamente capacidades adicionales a las imágenes existentes para satisfacer las diferentes necesidades del negocio y aumentar la reutilización de imágenes. En otras palabras, se pueden agregar capacidades a las imágenes existentes agregando capas adicionales encima de esa imagen y derivando una nueva imagen. Las imágenes Docker tienen una relación padre e hijo y la imagen inferior se llama imagen base que no tiene ningún padre como se muestra en la Figura 4. [6]



Figura 4. Imagen base de Docker. [6]

Y en la siguiente Figura 5 se puede ver como todo empieza con una imagen base y cada adición que se hace a la imagen base original se almacena en una capa separada como una nueva imagen que hace referencia a una imagen padre.

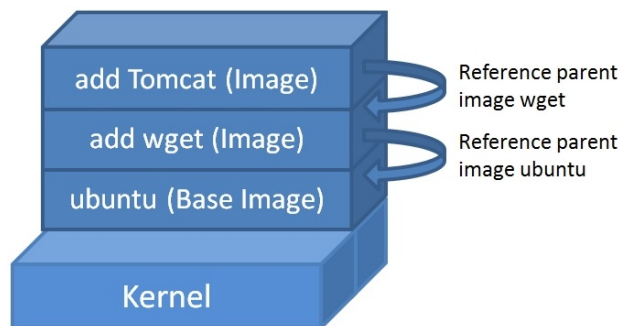


Figura 5. Nueva imagen creada a partir de la imagen base de Docker. [6]

- **Contenedor:** Un contenedor es una instancia ejecutable de una imagen. Se puede crear, ejecutar, detener, mover o eliminar un contenedor mediante la API de Docker o la CLI. Además se puede conectar un contenedor a una o más redes, adjuntarle almacenamiento o incluso crear una nueva imagen basada en su estado actual. [3]
- **Registro:** Un Registro de Docker es un lugar donde las imágenes de Docker pueden ser almacenadas de forma pública o privada para que se pueda acceder a ellas y puedan ser utilizadas por los desarrolladores de software en todo el mundo para crear rápidamente aplicaciones nuevas y compuestas sin riesgos.
- **Docker Hub:** Es el repositorio oficial que contiene todas las imágenes cuidadosamente curadas que son creadas y depositadas por la comunidad de desarrollo de Docker en todo el mundo. Esta llamada curación se promulga para asegurar que todas las imágenes almacenadas en el Docker Hub son seguras y de alta calidad.

## 2.2 Apache Kafka

Disponible bajo la licencia de Apache Software Foundation, es una plataforma de código abierto que permite tratar información en tiempo real producida por diferentes tipos de sistemas y aplicaciones, conocidos como productores y enrutarla a múltiples consumidores, proporcionando una integración transparente entre la información generada por los productores y los consumidores, sin bloquear a los productores y sin dejar que los productores conozcan quiénes son los consumidores finales. [7]

Apache Kafka es un sistema de mensajería distribuido, particionado y replicado, con un esquema de publicación-suscripción, en el cual los mensajes son enviados de manera asíncrona por los publicadores (productores), clasificados en diferentes clases o temas y se envían a los suscriptores (consumidores) según los temas de su interés. [8]

### 2.2.1 Características

Apache Kafka fue diseñado principalmente con las siguientes características:

- **Distribuido:** El diseño centralizado del clúster permite la distribución y el particionamiento de mensajes sobre los miembros del clúster manteniendo la semántica, por lo cual el clúster de Kafka puede crecer horizontalmente, de manera elástica y transparente sin ningún tiempo de inactividad que afecte a los consumidores.
- **Alto rendimiento:** Puede manejar cientos de operaciones de lectura y escritura por segundo de un gran número de clientes.
- **Múltiples clientes:** Soporta fácil integración con clientes de diferentes plataformas como python, java, scala, entre otros.
- **Persistencia:** Diseñado con eficiencia  $O(1)$ , por lo cual las estructuras de datos proporcionan un acceso con un rendimiento de tiempo constante, incluso con volúmenes muy grandes de mensajes almacenados que están en el orden de TBs. Con Kafka, los mensajes se persisten en disco, además de ser replicados dentro del clúster para evitar la pérdida de datos.

- **Tiempo real:** Los mensajes producidos por los hilos productores deben estar inmediatamente visibles para los consumidores, ésta característica es esencial para los sistemas basados en eventos, como los sistemas de procesamiento de eventos complejos (CEP). [7][8]

### 2.2.2 Arquitectura

Apache Kafka se ejecuta como un clúster en uno o más servidores, los principales componentes de la arquitectura de un clúster Kafka son:

- **Broker:** Kafka se ejecuta como un clúster con uno o más servidores físicos, donde cada uno puede tener uno o más procesos de servidor ejecutándose conocidos como broker.
- **Topic:** Es una categoría o un nombre en el cual se publican los registros. Un topic en Kafka siempre es multi-suscriptor puede tener cero, uno o muchos consumidores que se suscriben a los datos escritos en él.

Para cada topic, el clúster de Kafka mantiene un log particionado, cada partición es una secuencia ordenada e inmutable de registros que se agrega continuamente a un log estructurado. A cada registro de las particiones se les asigna un número de identificación secuencial llamado offset, que identifica de forma única cada registro dentro la partición.

El clúster de Kafka conserva todos los registros publicados, hayan sido o no consumidos, durante un periodo de tiempo de retención configurable, luego del cual los registros se descartan para liberar espacio.

- **Productor:** Publica los datos en los topics o temas de su elección, siendo responsable de asignar los registros en la partición apropiada dentro del topic, lo cual se puede realizar de manera round-robin para el balanceamiento de carga o definiendo alguna función personalizada, como puede ser algún tipo de algoritmo basado en una clave del registro.
- **Consumidor:** Los consumidores son las aplicaciones o procesos que se suscriben a los topics para recibir mensajes de su interés y se auto-etiquetan con un nombre de grupo de consumidores.

El concepto de grupo de consumidores generaliza los dos modelos de mensajería tradicional: colas (queuing) y publicación-suscripción



(publish-subscribe). Ya que al igual que un sistema de mensajería de colas, el grupo de consumidores permite dividir el procesamiento en una colección de procesos (los miembros del grupo de consumidores) para poder escalar el procesamiento, por lo cual cada registro publicado es enviado sólo a una de las instancias de consumidor dentro de cada grupo de consumidores; y al igual que el modelo de publicación-suscripción, Kafka permite difundir los mensajes a varios grupos de consumidores suscritos a un mismo topic. [7][8][9]

- **ZooKeeper:** Sirve como interfaz de coordinación entre el broker de Kafka y los consumidores.

Apache ZooKeeper forma parte de los proyectos de Apache Software Foundation y es un servicio de coordinación distribuido y de código abierto para aplicaciones distribuidas. Diseñado para almacenar datos de coordinación como información de estado, configuración, información de ubicación, etc., presenta un conjunto simple de primitivas en las que aplicaciones distribuidas pueden basarse para implementar servicios de nivel superior para sincronización y mantenimiento de configuración, liberando a las aplicaciones distribuidas de la responsabilidad de implementar servicios de coordinación desde cero.

Permite que los procesos distribuidos se coordinen entre sí a través de un espacio de nombres jerárquico compartido que se organiza de forma similar a un sistema de archivos estándar, el espacio de nombres consiste en registros de datos, denominados znodes, similares a archivos y directorios, que pueden tener datos asociados, están diseñados para almacenar datos sobre coordinación y están limitados a la cantidad de datos que pueden tener. A diferencia de un sistema de archivos típico que es diseñado para el almacenamiento, los datos de ZooKeeper se mantienen en la memoria, por lo cual puede alcanzar un alto rendimiento y baja latencia. [7][10]

La comunicación entre los clientes (productores y consumidores) y servidores se realiza mediante un protocolo simple, de alto rendimiento y lenguaje agnóstico, TCP, por lo cual existen clientes disponibles en diferentes lenguajes de programación como java, python, etc. [9]

Con Kafka se pueden crear los tipos de clúster que se detallan a continuación, donde los productores y consumidores pueden estar conectados a uno o mas brokers y éstos a su vez se conectan a ZooKeeper al igual que los consumidores para su coordinación, obteniendo información del estado y realizando el seguimiento de los offsets de los mensajes respectivamente [7]:

- **Nodo único - clúster con un único broker:** En la Figura 6 se puede observar un ejemplo de este sencillo tipo de clúster Kafka, donde sólo existe un nodo y a su vez éste contiene un sólo broker que se encuentra conectado a ZooKeeper.

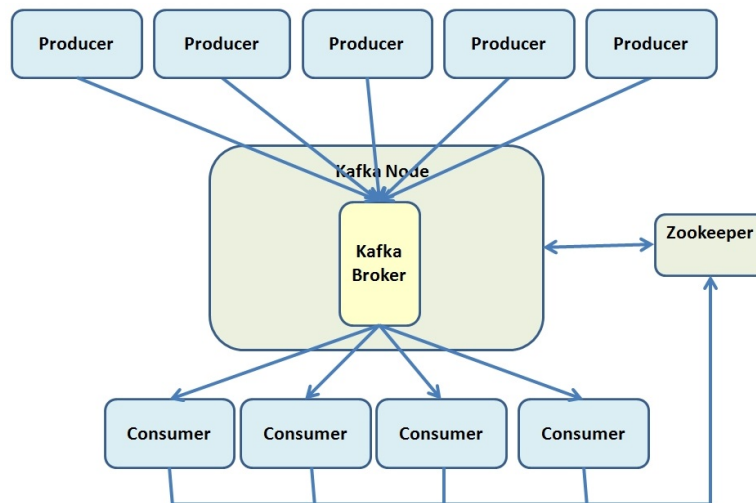


Figura 6. Nodo único - clúster con un único broker. [8]

- **Nodo único - clúster con múltiples brokers:** Este tipo de clúster cuenta con un sólo nodo, el cual contiene más de un broker que se conectan al mismo ZooKeeper como se muestra en la Figura 7.

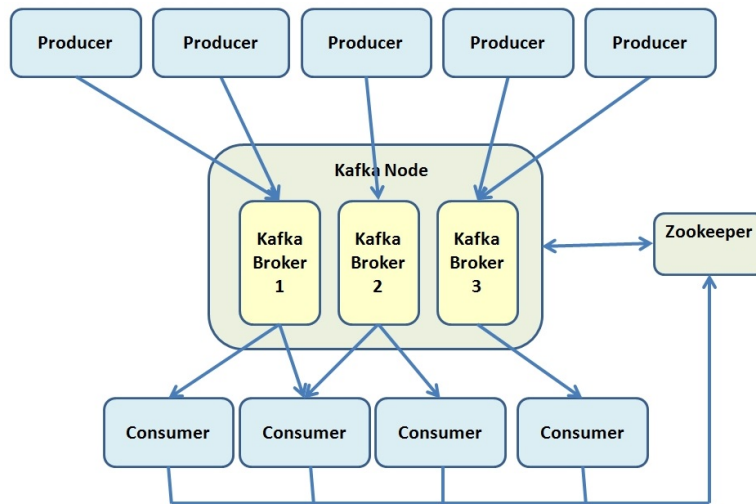


Figura 7. Nodo único - clúster con múltiples brokers. [8]

- **Nodo múltiple - clúster con múltiples brokers:** El clúster de la Figura 8 es un tipo más complejo, en el cual se debe instalar Kafka en cada nodo del clúster y todos los brokers que se encuentran en los diferentes nodos deben estar conectados al mismo ZooKeeper.

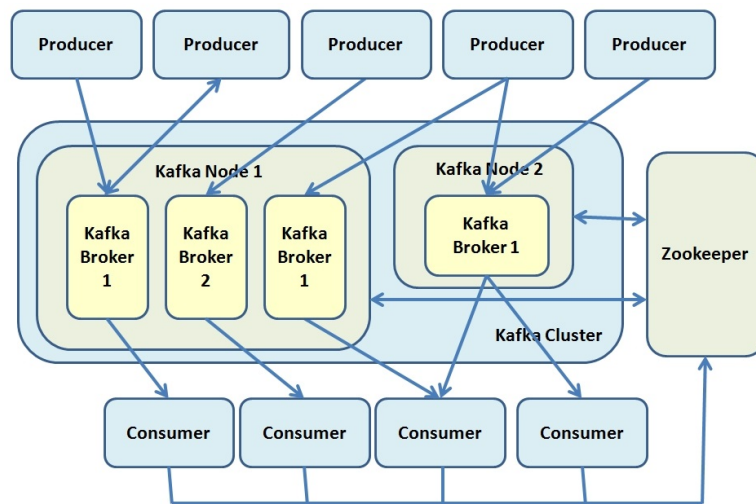


Figura 8. Nodo múltiple - clúster con múltiples brokers. [8]

En la Figura 9 se pueden observar los cinco componentes principales de la arquitectura de un clúster Kafka con un único nodo y un único broker, que tiene configurado un topic con tres particiones.

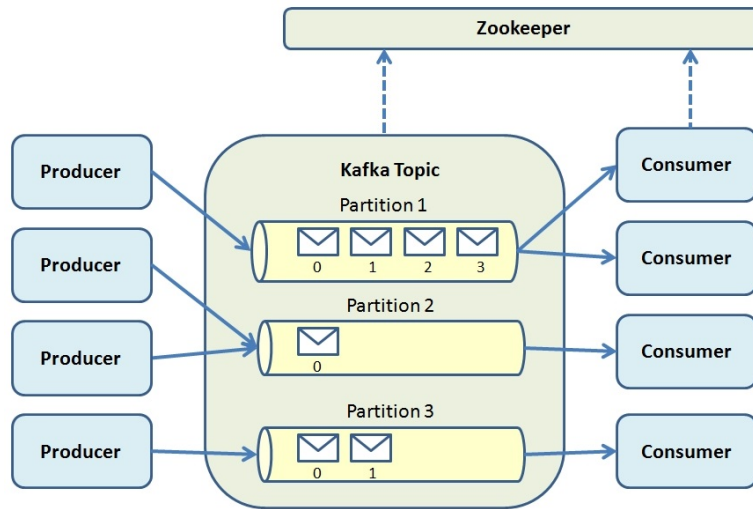


Figura 9. Clúster nodo único - único broker y un topic con 3 particiones. [8]

### 2.2.3 Componentes

Como se describió en la sección de arquitectura, uno de los principales componentes es el broker, en una máquina con Kafka instalado se pueden crear uno o más brokers según la arquitectura que se necesite, a los cuales se conectan los productores y consumidores para publicar y leer los mensajes respectivamente. Todos los brokers, ya sea que se encuentren configurados en una o en diferentes máquinas dentro de un clúster, deben estar conectados al mismo ZooKeeper.

Otro de los componentes principales es el topic, al crearlo se pueden configurar parámetros como el factor de replicación para indicar cuantos servidores replicarán cada mensaje publicado por el productor y el número de particiones que determina el paralelismo para los consumidores, reflejando el número de logs en los que se fragmentará el topic, por lo que es importante configurarlo según como se consumirán los datos. [7]

Además Kafka incluye cuatro APIs principales:

- **API del productor:** permite a aplicaciones que pueden ser de diferente naturaleza, como aplicaciones front-end, servicios back-end, aplicaciones proxy, etc., e implementadas en diferentes lenguajes de programación, enviar flujos o streams de datos a los topics configurados en el clúster de Kafka para su consumo.
- **API del consumidor:** permite a aplicaciones que al igual que los

productores pueden ser de diferente naturaleza, como aplicaciones de análisis en tiempo real, aplicaciones con soluciones NoSQL, servicios back-end, etc., e implementadas en diferentes lenguajes de programación, leer streams de datos desde los topics definidos en el clúster de Kafka. [7][9]

- **API de streams:** permite transformar flujos de datos de topics de entrada a topics de salida, es decir, permite procesar y analizar datos almacenados en Kafka para luego escribir los datos resultantes de vuelta a Kafka o enviar la salida final a algún sistema externo.
- **API de conexión:** permite la transmisión escalable y confiable de datos entre Apache Kafka y otros sistemas, facilitando la definición rápida de conectores que mueven grandes colecciones de datos dentro y fuera de Kafka. Esta API puede importar bases de datos enteras o recopilar métricas de servidores de aplicaciones a topics de Kafka, haciendo que los datos estén disponibles para el procesamiento de flujos con baja latencia. Muchos usuarios no necesitarán utilizar esta API directamente, ya que pueden utilizar conectores pre-construidos sin necesidad de escribir ningún código. [9]

## 2.3 Python

Python es un lenguaje de programación potente y fácil de aprender, cuenta con estructuras de datos eficientes de alto nivel y un enfoque simple pero efectivo para la programación orientada a objetos. Es un lenguaje ideal para el desarrollo de scripting y aplicaciones rápidas en muchas áreas sobre la mayoría de plataformas ya que posee una elegante sintaxis y tipado dinámico, junto con su naturaleza interpretada. [11]

Se puede utilizar Python como un lenguaje de scripting para ejecutar código simplemente, como un lenguaje procedimental para organizar un programa en una colección de funciones que se llaman entre sí, o como un lenguaje orientado a objetos que usa clases, herencia y módulos para crear una jerarquía. Esta flexibilidad permite elegir el estilo de programación más adecuado para un proyecto en particular.

Cuando se desarrolla utilizando un lenguaje más tradicional como C o C++, se tiene que compilar y vincular el código antes de poder ejecutarlo. Con Python, se puede ejecutar directamente después de editarlo, por lo que

en la práctica, el proceso de modificar y ejecutar el código una y otra vez es menos complicado con Python.

La sintaxis breve, concisa y expresiva de Python hace que sea fácil realizar operaciones complejas con sólo unas pocas líneas de código. Además existen formas estándar para llamar código C/C++ desde Python y viceversa, en fin como se puede encontrar librerías para hacer casi cualquier cosa en Python, es fácil combinar Python con módulos desarrollados en otros lenguajes en proyectos grandes, esta es la razón por la que Python se considera un lenguaje de pegamento "glue language", que facilita la combinación de diversos componentes de software.[12]

## 2.4 Neo4j

Desarrollada por la compañía Neo Technology, Neo4j es una base de datos NoSQL de código abierto orientada a grafos, implementada en java y scala. Con su desarrollo iniciado a partir del 2003, se puso a disposición del público desde el 2007, siendo actualmente utilizada por cientos de miles de empresas y organizaciones en casi todas las industrias.

Como una base de datos orientada a grafos almacena los datos en grafos, donde un grafo es un conjunto de vértices conocidos como nodos y aristas que representan las relaciones que los conectan. El grafo de Neo4j se basa en el Modelo de Grafo de Propiedad para representar la información, el cual se refiere a un grafo con peso, con etiquetas y donde se pueden asignar propiedades tanto a nodos como a relaciones y más de una relación entre dos nodos, en la Figura 10 se muestra un ejemplo de cómo se pueden representar los datos en un grafo de Neo4j. [13][14]

La Figura 10 presenta los siguientes elementos que contiene un grafo de propiedad en Neo4j:

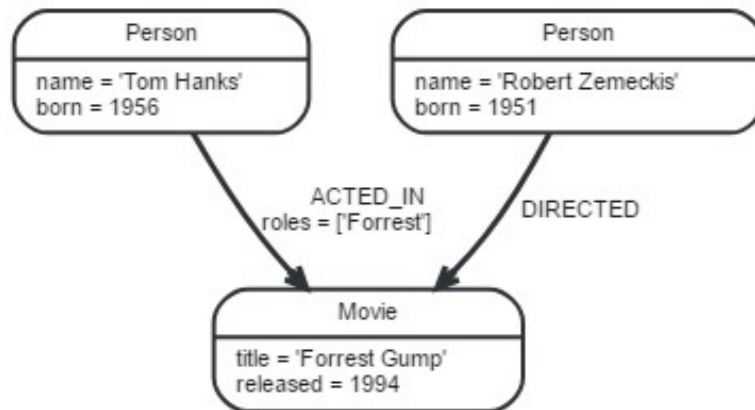


Figura 10. Ejemplo de grafo en Neo4j. [13]

- **Nodos:** las entidades, que representan objetos o conceptos en el mundo real, son modeladas como nodos, en la imagen anterior se muestran 3 nodos indicados en la siguiente Figura 11.



Figura 11. Nodos en Neo4j. [13]

- **Relaciones:** representan conexiones directas y semánticamente relevantes entre dos nodos. Una relación en Neo4j siempre tendrá un nodo de origen y un nodo destino válidos. En el ejemplo de la Figura 10 se pueden ver los tipos de relación "ACTED\_IN" y "DIRECTED", donde una relación de tipo "ACTED\_IN" es la que tiene al nodo "Tom Hanks" como origen y "Forrest Gump" como destino. Las relaciones necesitan ser creadas con una dirección, pero podemos ignorar la dirección mientras se las recorre, es decir, no hay necesidad de añadir relaciones duplicadas en la dirección opuesta (con respecto a la transversalidad o rendimiento).
- **Propiedades:** son pares clave-valor que contienen información sobre el nodo o la relación, donde la clave es una cadena de texto y el valor puede ser de tipo numérico (float o integer), cadena de texto (String), booleano (boolean) o una lista (array) de cualquiera de los tipos anteriores, en la Figura 10 se tienen las propiedades "name" y "born" en el nodo "Tom Hanks" y en la relación de tipo "ACTED\_IN" se tiene la propiedad "roles" con un array como valor.

- **Etiquetas:** una etiqueta es la construcción de un grafo etiquetado, se utilizan para representar el papel del nodo en nuestro dominio, es decir, para agrupar los nodos en un conjunto, todos los nodos etiquetados con la misma etiqueta pertenecen al mismo conjunto, un nodo puede o no tener varias etiquetas al mismo tiempo, ya que son opcionales en el grafo. Además de agregar más significado a los nodos, las etiquetas también se utilizan para agregar restricciones e índices que son locales a la etiqueta en particular, en el ejemplo de la Figura 10 se tienen las etiquetas "Person" y "Movie".
- **Recorrido (Traversal):** es cómo se consulta un grafo, navegando desde nodos iniciales hasta nodos relacionados. Recorrer un grafo significa visitar sus nodos, siguiendo las relaciones según algunas reglas. En la mayoría de los casos sólo se visita un subgrafo, ya que se conoce dónde se encuentran en el grafo los nodos y relaciones de interés. Por ejemplo en el caso de que se desee averiguar en qué películas actuó Tom Hanks según el grafo de la Figura 10, el recorrido comenzaría desde el nodo de "Tom Hanks", seguiría cualquier relación "ACTED\_IN" conectada a este nodo y terminaría con "Forrest Gump" como resultado.
- **Rutas:** una ruta representa un camino a través de un grafo de propiedad y consiste en una secuencia de nodos y de relaciones alternas, siempre comienza y termina en un nodo. La ruta más pequeña posible contiene un sólo nodo y se llama ruta vacía. Una ruta tiene una longitud, la cual es un entero mayor o igual que cero, que es igual al número de relaciones en la ruta. En el ejemplo anterior, el recorrido obtenido, que se muestra en la Figura 12, puede ser devuelto como ruta de acceso, donde la longitud de la ruta es 1. [13][15]

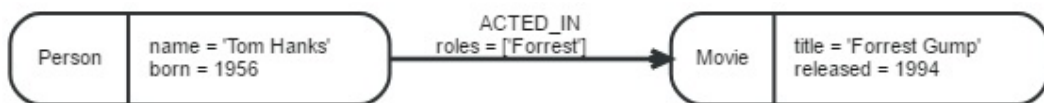


Figura 12. Ruta en Neo4j. [13]

### 2.4.1 Características

Algunas de las características particulares de Neo4j son:

- Proporciona un modelo orientado a grafos junto con un framework de visualización para la representación de datos y resultados de consultas.



- Materializar las relaciones en el momento de creación, no afecta las consultas con tiempo de ejecución complejo.
- Recorridos de tiempo constante para las relaciones en el grafo tanto en profundidad como en amplitud debido a la eficiente representación de nodos y relaciones.
- Todas las relaciones en Neo4j son igualmente importantes y rápidas, lo que hace posible materializar y utilizar nuevas relaciones más adelante para "acortar" y acelerar los datos del dominio cuando surgen nuevas necesidades.
- Compatibilidad con la mayoría de lenguajes, incluyendo Python, Java, Clojure y Ruby. La interfaz REST es un método recomendado para el acceso a la base de datos.
- Incluye nativamente un gestor de almacenamiento basado en disco que ha sido completamente optimizado para almacenar grafos para proporcionar escalabilidad y un rendimiento mejorado.
- Es altamente escalable, una sola instancia de Neo4j puede manejar grafos que contienen miles de millones de nodos y relaciones.[13][16]
- Proporciona seguridad de datos a través de transacciones ACID, lo cual implica:
  - Atomicidad: significa que los cambios en la base de datos deben seguir la regla de todo o nada, es decir si una transacción está compuesta por una serie de operaciones, si alguna de esas operaciones falla, no se ejecuta la transacción.
  - Consistencia: significa que sólo los datos consistentes o válidos se permitirán ingresar en la base de datos. El requisito de consistencia principal en Neo4j es en realidad que las relaciones deben tener un inicio y un nodo final.
  - Aislamiento: requiere que si varias transacciones que se ejecuten en paralelo en la misma instancia de base de datos no se afecten entre sí.
  - Durabilidad: almacenamiento persistente, una vez realizada la transacción no se podrá deshacer aunque falle el sistema. [17]

Existen dos ediciones de Neo4j, la edición comunitaria que es gratuita y de código abierto, la cual ofrece una base de datos de grafos básica, totalmente

funcional, transaccional ACID de alto rendimiento; y la edición empresarial que proporciona toda la funcionalidad de la edición comunitaria, además de clustering (para alta disponibilidad y balanceo de carga), monitoreo avanzado, almacenamiento en caché avanzado y copias de seguridad en línea. [13][17]

#### **2.4.2 Base de Datos de Grafos vs. Base de Datos Relacional**

Durante varias décadas las bases de datos relacionales han sido las más utilizadas en el desarrollo de aplicaciones de software, almacenando datos altamente estructurados en tablas con columnas (campos) predeterminadas y muchas filas (registros) del mismo tipo de información, basándose en el modelo relacional de datos propuesto por Edgar Frank Codd en 1970 y para lo cual requieren que los desarrolladores y aplicaciones estructuren estrictamente los datos utilizados. Cada registro tiene su identificador único (clave primaria) y las relaciones entre registros se realizan por medio de referencias (a otras filas y tablas), básicamente haciendo referencia a los atributos que forman la clave primaria del registro a través de las columnas de clave foránea. Las combinaciones (joins) entre tablas se calculan en el momento de la consulta haciendo coincidir las claves primarias y foráneas de las muchas filas (potencialmente indexadas) de las tablas que se van a unir. Estas operaciones requieren mucho cálculo y memoria y tienen un costo exponencial.

En el caso de utilizar relaciones muchos-a-muchos, se debe introducir una tabla de unión o join, que contenga claves foráneas de ambas tablas participantes como se puede observar en la Figura 13, lo cual aumenta aún más los costos de operación de join. Esas costosas operaciones de unión se tratan generalmente desnormalizando datos para reducir el número de uniones necesarias. [13][14]

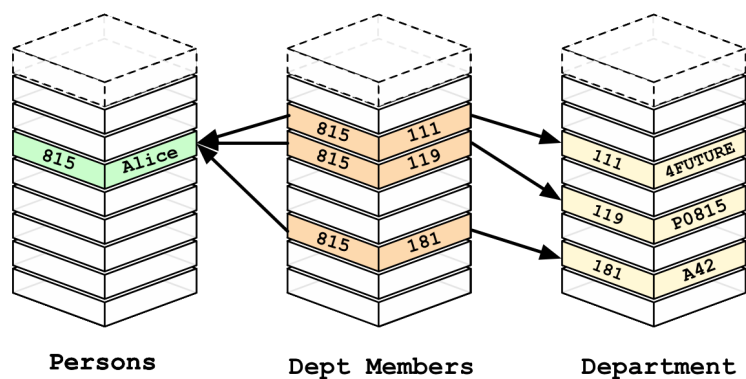


Figura 13. Relación muchos-a-muchos entre Personas y Departamentos en una base de datos relacional. [13]

En la Figura 14 se puede observar explícitamente la relacionan entre las tablas Personas y Departamentos a través de los datos de clave foránea en una base de datos relacional.

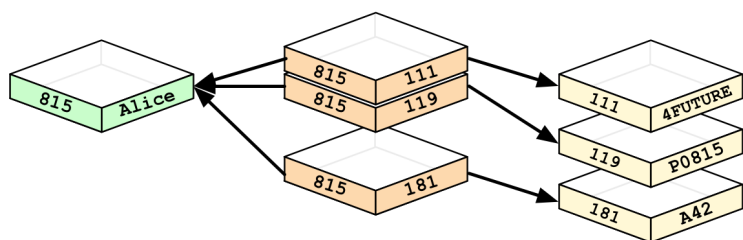


Figura 14. Relación entre claves foráneas de las tablas Personas y Departamentos en una base de datos relacional. [13]

A medida que se multiplican los datos y la estructura general del conjunto de datos se vuelve más compleja y menos uniforme, el modelo relacional se carga con grandes tablas de join, filas escasamente pobladas y mucha lógica de comprobación nula. El incremento de conexiones se traduce en el mundo relacional en el incremento de joins, lo que disminuye el rendimiento y dificulta la evolución de una base de datos existente en respuesta a las necesidades cambiantes del negocio.

Aunque no todos los casos de uso son adecuados para este tipo de modelo de datos, en el pasado, la falta de otras alternativas viables y el gran soporte a las bases de datos relacionales ha dificultado la entrada de nuevos modelos alternativos a la corriente principal. [13][14]

Con el uso de base de datos orientadas a grafos se pueden construir modelos sofisticados que se correlacionan estrechamente con el dominio del

problema, al reunir las simples abstracciones de nodos y relaciones en estructuras conectadas. Cada nodo del modelo de base de datos de grafos contiene directa y físicamente una lista de registros de relaciones que representan sus relaciones con otros nodos. Estos registros de relación se organizan por tipo y dirección y pueden contener atributos adicionales. Cada vez que se ejecute el equivalente de una operación join, la base de datos sólo utiliza esta lista y tiene acceso directo a los nodos conectados, eliminando la necesidad de una costosa búsqueda.

Esta capacidad de pre-materializar las relaciones en las estructuras de la base de datos permite a Neo4j ofrecer ejecuciones de varias órdenes de magnitud, especialmente para las consultas de join pesadas, la ventaja de minutos a milisegundos que muchos usuarios aprovechan. Los modelos de datos resultantes son mucho más sencillos y al mismo tiempo más expresivos que los producidos usando bases de datos relacionales tradicionales u otras bases de datos NoSQL.

Las bases de datos de grafos admiten un modelo de datos muy flexible y detallado que permite modelar y administrar muchos dominios de forma fácil e intuitiva. Manteniendo los datos como lo es en el mundo real: entidades pequeñas, normalizadas, pero fuertemente conectadas, lo cual permite consultar y ver los datos desde cualquier punto de interés imaginable, soportando muchos casos de uso diferentes. En la Figura 15 se muestra como el modelo relacional que se tenía anteriormente se puede representar de una manera mas sencilla y mas entendible por medio de un grafo, donde los nodos y las relaciones etiquetados han reemplazado las tablas, claves foráneas y tabla de join. [13][15]

La flexibilidad y la agilidad son consideraciones importantes en el mundo de hoy donde las necesidades del negocio están en constante evolución. Los desarrolladores necesitan tener una herramienta que les permita pensar de forma incremental en el modelo en lugar de bloquear el modelo de datos antes de comenzar a codificar. Las bases de datos de grafos permiten agregar relaciones, tipos de nodos y propiedades sin realizar cambios en las consultas existentes. Podemos conectar el modelo de forma incremental, lo que permite una consulta más sofisticada. Esta flexibilidad también significa menos migraciones. Incluso en caso de cambios en el modelo de datos, las migraciones se pueden realizar sin tener la base de datos sin conexión durante mucho tiempo, lo que ayuda a los equipos a entregar software más rápido al concentrarse en el dominio en lugar de gestionar la infraestructura y la comunicación.

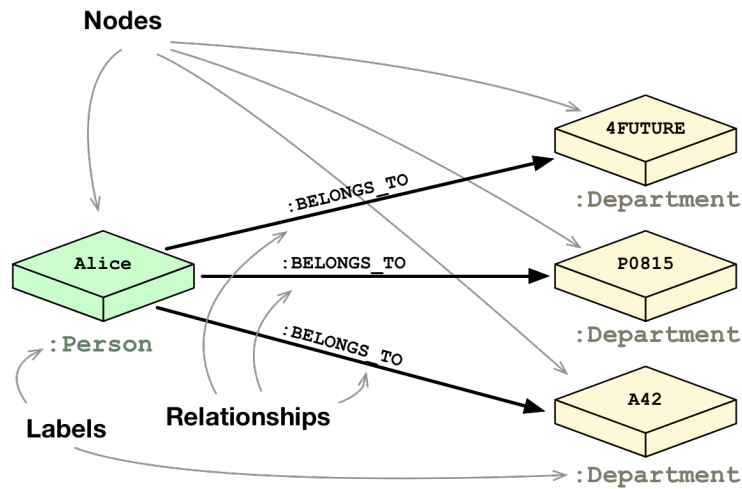


Figura 15. Modelo de datos de grafos de Personas y Departamentos. [13]

Dado que las bases de datos de grafos son sin esquema, el esquema es dictado por la aplicación y por lo tanto es mejor validado. Permite un mejor diseño de los desarrolladores ya que no hay ambigüedad del modelo de dominio en comparación con la forma en que se almacena en las tablas y una menor ambigüedad conduce a mejores modelos.

Además, el modelo detallado también implica que no hay límite fijo alrededor de los agregados, por lo que el alcance de las operaciones de actualización es proporcionado por la aplicación durante la operación de lectura o escritura. El bien conocido y probado concepto de transacciones agrupa un conjunto de actualizaciones de nodos y relaciones en una operación atómica, consistente, aislada y duradera (ACID). Las bases de datos de grafos como Neo4j admiten totalmente los conceptos transaccionales, incluidos los logs de escritura anticipada y la recuperación después de una terminación anormal. Por lo tanto, nunca se pierden los datos que se han enviado a la base de datos. [13][15]

### 2.4.3 Base de Datos de Grafos vs. Base de Datos NoSQL

NoSQL que significa "No sólo SQL" reúne muchas soluciones interesantes que ofrecen diferentes modelos de datos y sistemas de bases de datos, cada uno adecuado para diferentes casos de uso. Con la llegada del NoSQL, se sustituyó el uso de base de datos relaciones como solución para todos los sistemas por el hecho de tomar decisiones conscientes acerca de encontrar la herramienta

adecuada para el trabajo.

La mayoría de los sistemas NoSQL están orientados a agregados, proporcionando sólo operaciones atómicas dentro de su agregado natural, lo cual proporciona sólo una vista dedicada de los datos, donde otras proyecciones tienen que ser calculadas duplicando los datos. Sin embargo, las bases de datos de grafos manejan redes de información de grano fino, proporcionando cualquier perspectiva sobre los datos que se adapte a los casos de uso con los que se trabaja.

Con NoSQL, las empresas de todos los tamaños tienen una variedad de opciones modernas desde las cuales construir soluciones relevantes para sus casos de uso, como por ejemplo:

- Cálculo del ingreso promedio: Se puede utilizar una base de datos relacional.
- Construir un carrito de compras: Se puede utilizar un almacenamiento de tipo clave-valor.
- Almacenamiento de información estructurada del producto: Puede ser almacenada como un documento.
- Describir cómo un usuario llegó del punto A al punto B: Utilizar un grafo.

El modelo de clave-valor es altamente eficiente para búsquedas de grandes cantidades de valores simples o incluso complejos. Cuando los valores están interconectados entre sí, se tiene como resultado un grafo como se muestra en la Figura 16, Neo4j permite recorrer rápidamente todos los valores conectados y encontrar ideas en las relaciones.

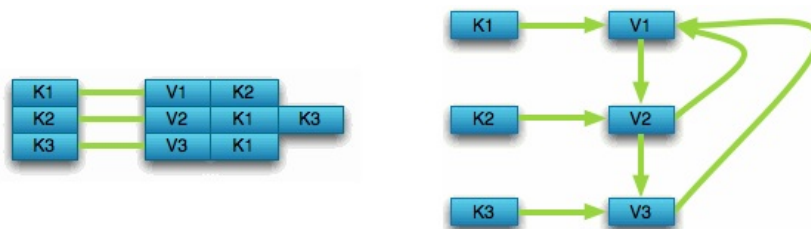


Figura 16. Relacionar bases de datos clave-valor con las bases de datos de grafos. [13]

De igual manera, la jerarquía estructurada de un documento contiene una gran cantidad de datos libres de esquemas que pueden representarse fácilmente como un árbol. Aunque los árboles son un tipo de grafo, un árbol representa sólo una proyección de los datos, al consultar otros documentos (o elementos contenidos) dentro de ese árbol, se tendrá una representación más expresiva de los mismos datos que se pueden navegar fácilmente con Neo4j como en la Figura 17. Un modelo de datos de grafos permite que más de una representación natural surja dinámicamente según sea necesario. [13]

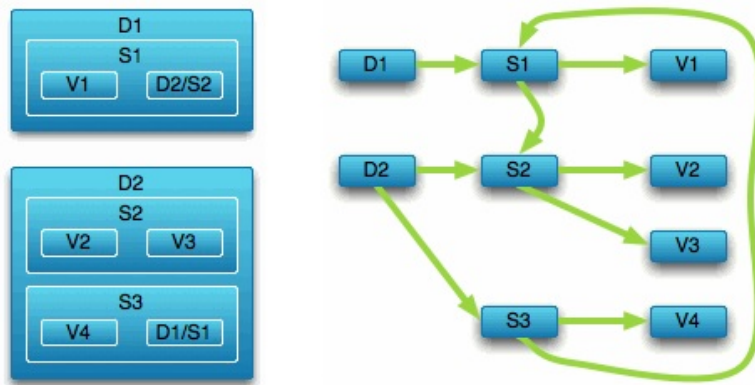


Figura 17. Navegar bases de datos de documentos con bases de datos de grafos. [13]

#### 2.4.4 Consultando Grafos con Cypher

Cypher es un lenguaje declarativo, específico de Neo4j, inspirado en SQL para describir patrones en grafos visualmente usando una sintaxis de arte ASCII, es un lenguaje expresivo (pero compacto) y fácil de aprender para realizar consultas en bases de datos de grafos. Permite indicar lo que se desea seleccionar, insertar, actualizar o eliminar de los datos de grafos sin que sea necesario describir exactamente cómo hacerlo.

Cypher está diseñado para ser fácilmente leído y comprendido por desarrolladores, profesionales de bases de datos y grupos de interés empresariales. Su facilidad de uso se deriva del hecho de que está de acuerdo con la forma en que intuitivamente se describen los grafos utilizando diagramas.

Como la mayoría de los lenguajes de consulta, Cypher se compone de cláusulas. Las consultas más simples consisten en una cláusula MATCH seguida de una cláusula RETURN como se muestra en el ejemplo de la

Figura 18, donde se desea encontrar los amigos mutuos de un usuario llamado Jim: [13][14]

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]->(c),  
        (a)-[:KNOWS]->(c)  
RETURN b, c
```

Figura 18. Ejemplo de consulta con Cypher. [14]

Algunas de las cláusulas más importantes son:

- **MATCH**: está en el corazón de la mayoría de las consultas Cypher. Usando caracteres ASCII para representar nodos y relaciones, se dibujan los datos de interés, se dibujan los nodos con paréntesis y relaciones usando pares de guiones con signos mayor o menor que (-> y <-). Los signos "<" y ">" indican la dirección de la relación. Entre los guiones, dentro de corchetes y prefijados por dos puntos, se puede colocar el nombre de la relación, las etiquetas de nodo tienen el mismo prefijo de dos puntos y los pares clave-valor de las propiedades de nodo y relación se especifican dentro de llaves.
- **RETURN**: especificar qué nodos, relaciones y propiedades de los datos coincidentes con la consulta se deben devolver.
- **WHERE**: utilizada para proporcionar criterios para filtrar los resultados de concordancia de patrones.
- **CREATE** y **CREATE UNIQUE**: Crear nodos y relaciones.
- **MERGE**: asegura que el patrón suministrado existe en el grafo, ya sea reutilizando nodos existentes y relaciones que coincidan con los predicados suministrados, o mediante la creación de nuevos nodos y relaciones.
- **DELETE**: eliminar nodos, relaciones y propiedades.
- **: SET**: establecer valores de propiedades. [14]

#### 2.4.5 Cálculo de rutas

Calcular las rutas en un grafo, implica determinar el mejor camino que se puede recorrer para llegar de un nodo a otro, para lo cual existen diferentes algoritmos que utiliza Neo4j como los que se detallan a continuación:



- **Algoritmo de Búsqueda en Profundidad:** El algoritmo Depth-First Search llamado en inglés, se desplaza desde un nodo inicial hasta un nodo final expandiendo cada nodo que se va encontrando en el camino antes de repetir la búsqueda por una ruta diferente desde el mismo nodo inicial hasta que se responde a la consulta. Generalmente, este algoritmo es una buena opción al intentar descubrir piezas discretas de información. También son una buena estrategia para realizar recorridos generales de grafos.

La búsqueda clásica del algoritmo de profundidad es desinformada, es decir, simplemente busca un camino hasta encontrar el final del grafo, una vez al final, retrocede al nodo de inicio e intenta un camino diferente. Debido a que las bases de datos de grafos son semánticamente ricas, se pueden tener búsquedas informadas que permiten terminar pronto una búsqueda a lo largo de una rama en particular si se encuentra un nodo sin relaciones de salidas compatibles, o se ha recorrido "lo suficientemente lejos", estas búsquedas pueden resultar en tiempos de ejecución más bajos. Las consultas Cypher generalmente realizan búsquedas informadas.

- **Algoritmo de Búsqueda en Anchura:** Este algoritmo realiza la búsqueda explorando el grafo una capa a la vez, primero se visita cada nodo en profundidad 1 desde el nodo de inicio, luego cada uno de ellos a profundidad 2, luego profundidad 3, etc., hasta que se ha visitado todo el grafo. Esta progresión se visualiza fácilmente comenzando en el nodo etiquetado 0 (para el origen) y progresando hacia fuera una capa a la vez, como se muestra en la Figura 19. [14][18] La búsqueda en anchura

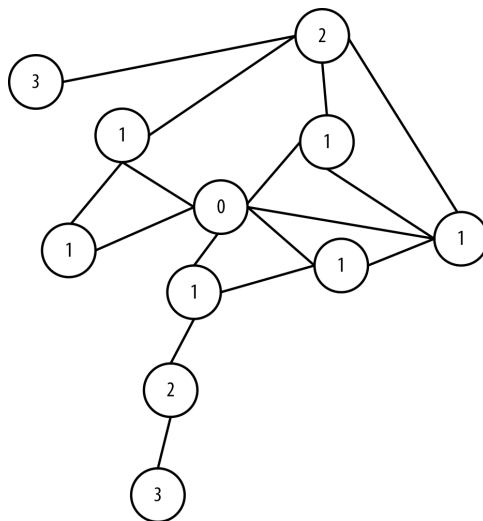


Figura 19. Algoritmo de Búsqueda en Anchura. [14]

se utiliza a menudo en algoritmos de búsqueda de rutas o cuando el grafo completo necesita ser buscado sistemáticamente.

- **Algoritmo Dijkstra:** El objetivo del algoritmo de Dijkstra es realizar una búsqueda de anchura o amplitud con un nivel de análisis más alto para encontrar la ruta más corta entre dos nodos en un grafo. Lo hace de la siguiente manera:

1. Se eligen los nodos de inicio y fin y se agrega el nodo de inicio al conjunto de nodos resueltos con un valor de 0. Los nodos resueltos son el conjunto de nodos con la ruta más corta conocida desde el nodo de inicio. El nodo de inicio tiene un valor 0 porque su ruta tiene tamaño 0 lejos de sí mismo.
2. Recorrer en anchura desde el nodo de inicio a sus vecinos más cercanos y se registra la longitud de la ruta contra cada nodo vecino.
3. Elegir el camino más corto a uno de estos vecinos y marcar ese nodo como resuelto. En caso de empate, el algoritmo de Dijkstra elegirá al azar.
4. Visitar a los vecinos más cercanos al conjunto de nodos resueltos y anotar las longitudes de ruta desde el nodo de inicio contra estos nuevos vecinos. No visitar los nodos vecinos que ya han sido resueltos, ya que ya se saben los caminos más cortos a ellos.
5. Repetir los pasos tres y cuatro hasta que el nodo de destino se haya marcado como resuelto.

El algoritmo de Dijkstra se utiliza a menudo para encontrar los caminos más cortos del mundo real, como para la navegación y la logística. [14][18]

Consultar los caminos más cortos en Cypher puede conducir a diferentes planes de consulta dependiendo de los predicados que se evalúan. Neo4j ofrece funciones para obtener la ruta mas cortas llamadas "shortestPath" y "allShortestPaths", donde internamente, Neo4j utiliza un algoritmo de búsqueda rápida bidireccional en anchura si los predicados pueden ser evaluados mientras se busca la ruta. Por lo tanto, este algoritmo rápido siempre devolverá la respuesta correcta cuando hay predicados universales en el camino, por ejemplo, al buscar la ruta más corta donde todos los nodos tienen la etiqueta "Person".

Si los predicados necesitan inspeccionar toda la ruta antes de decidir si es válida o no, no se puede confiar en este algoritmo rápido para encontrar el camino más corto, por lo que Neo4j puede tener que recurrir a un algoritmo más lento de búsqueda en profundidad para encontrar el camino, esto significa que los planes de consulta para las consultas de ruta más cortas con predicados no universales incluirán una alternativa de ejecutar la búsqueda exhaustiva para encontrar la ruta si el algoritmo rápido no tiene éxito. Por ejemplo, dependiendo de los datos, la respuesta a una consulta de ruta más corta con predicados como el requisito de que al menos un nodo contenga el nombre de propiedad = "Charlie Sheen" puede no ser encontrada por el algoritmo rápido. En este caso, Neo4j volverá a usar la búsqueda exhaustiva para enumerar todas las rutas y potencialmente devolver una respuesta.

La búsqueda exhaustiva sólo se ejecuta cuando el algoritmo rápido no logra encontrar rutas coincidentes. El algoritmo rápido siempre se ejecuta primero, ya que es posible que encuentre una ruta válida aunque no se pueda garantizar en el momento de la planificación. [18]

Para este trabajo el uso de las funciones "shortestPath" y "allShortestPaths" definidas en Neo4j no es factible debido a que estas funciones no tienen en cuenta el peso de cada relación, caracterizada por la propiedad de la distancia en este caso, de hecho, una ruta más corta, significa la ruta que tiene el menor número de relaciones, la cual puede tener una distancia total más alta que una relación más larga pero más ligera [19]. Es por esto que se utiliza el algoritmo Dijkstra para poder considerar las distancias o un peso en las relaciones, Neo4j soporta y ofrece un procedimiento para utilizarlo.

#### 2.4.6 Casos de Uso

- El enrutamiento es un problema de grafos y se han hecho muchas investigaciones al respecto. Hoy en día, muchas aplicaciones de grafos aprovechan la extraordinaria capacidad de los grafos y los algoritmos de grafos para calcular la ruta óptima entre dos nodos en una red.
- El campo geoespacial es el caso de uso del grafo original. Euler resolvió el problema de los Siete Puentes de Königsberg al plantear un teorema matemático que posteriormente llegó a formar la base de la teoría de grafos. Las aplicaciones geoespaciales de las bases de datos de grafos abarcan desde el cálculo de rutas entre ubicaciones en una red abstracta como una red de carreteras o ferrocarril, una red de espacio aéreo o una

red logística a operaciones espaciales como encontrar todos los puntos de interés en un área limitada, encontrar el centro de una región y calcular la intersección entre dos o más regiones.

- Las redes sociales son problemas adecuados para grafos, ya que se pueden aprovechar las conexiones de los usuarios para obtener datos y decidir lo que es accesible y lo que no. Facebook, en particular, utiliza su búsqueda de grafos y la ha expuesto a los usuarios para permitirles realizar mejores búsquedas. Facebook se basa en gran medida en los grafos de las personas y sus amigos para asociar su información.
- La recomendación es un problema de grafos que puede resolverse utilizando bases de datos de grafos. Mientras que compañías como eBay originalmente dependían de MySQL, finalmente se volcaron a Neo4j.
- Las búsquedas en la web pueden no parecer un problema de grafos. Sin embargo, Google utiliza su grafo de conocimiento para dar resultados de búsqueda basados en lo bien conectado que se encuentra una parte del contenido con el término que se busca. Más recientemente, Facebook ha aprovechado su grafo social para mejorar las búsquedas.
- La investigación médica es otro campo en el que se utilizan grafos, debido a que los datos médicos están altamente interconectados y por lo tanto pueden beneficiarse enormemente del uso de bases de datos de grafos. [14][15]

## 2.5 HTML5

Hypertext Markup Language, o HTML, es un lenguaje que permite crear y representar visualmente el contenido de una página web. Se origina a finales de 1989 debido a que Tim Berners-Lee consciente de la necesidad de encontrar una mejor manera de compartir información en línea, propuso un proyecto piloto que emplearía el hipertexto que es texto con referencias a otra información que el usuario puede activar para obtener acceso inmediato a esa información, esto incluye información contenida en el mismo documento, así como información en documentos externos u otras fuentes de datos, estos enlaces se denominan hipervínculos. Además emplearía un simple lenguaje de marcado como base, el cual proporciona una manera de anotar o etiquetar (o "marcar" como un editor marca un documento en

revisión) un documento de tal manera que las anotaciones son sintácticamente distintas al contenido principal del documento. [20]

Durante sus primeros cinco años (1990-1995), HTML pasó por una serie de revisiones y experimentó una serie de extensiones, principalmente alojado primero en la Organización Europea para la Investigación Nuclear conocida comúnmente como CERN por sus siglas en francés y luego en el Grupo de Trabajo de Ingeniería de Internet (IETF). Después, con la creación del World Wide Web Consortium (W3C), el desarrollo de HTML cambió de lugar nuevamente. [21]

Las versiones de HTML antes de la 5 sólo definen etiquetas de marcado para el contenido con listas, párrafos, encabezados, tablas, etc. HTML5 define muchas características nuevas a la especificación existente del HTML y de igual manera descarta el uso de algunas antiguas, tiene nuevas etiquetas de contenido (como `<audio>` y `<video>`), pero también define interacciones complejas como arrastrar y soltar, nuevas interfaces de red como eventos de servidor e incluso tiene nueva funcionalidad asíncrona como los web workers que permiten ejecutar procesos JavaScript en segundo plano. Generalmente, las referencias a HTML5 se refieren a un conjunto de características y tecnologías que rodean el uso no sólo de HTML, sino también de hojas de estilo en cascada (CSS) que permiten controlar el aspecto de documentos electrónicos escritos con HTML, así como el lenguaje de programación JavaScript que permite crear página web dinámicas. Sin el uso de CSS y JavaScript, los documentos HTML sencillos, incluso los documentos HTML5, seguirán siendo extremadamente simplistas en cuanto a funcionalidad y apariencia. [20][22]

Entre las principales ventajas de utilizar HTML5 se encuentran:

- Accesibilidad al entorno de desarrollo, dado que HTML5 no requiere un compilador especial o un software para escribir código, los desarrolladores son libres de elegir su configuración favorita para escribir y probar sus proyectos. Las aplicaciones pueden ejecutarse y probarse fácilmente en cualquier navegador web compatible y probarse localmente sin necesidad de un servidor web. Esto convierte a HTML5 en una de las tecnologías más accesibles y fáciles de usar de la web.
- Escribir una vez, desplegar en todas partes, debido a que cualquier dispositivo con un navegador web moderno puede interactuar con el contenido web HTML5, con la ayuda de CSS para el

redimensionamiento dinámico del contenido visible, las aplicaciones HTML5 pueden utilizarse sin necesidad de instalación de aplicaciones o dependencias en plataformas de escritorio o móviles lo que permite llegar fácilmente a todo tipo de usuario.

- Nuevas características, que permiten mejorar la experiencia del usuario como es el caso de:
  - API de geolocalización para que los usuarios permitan que su posición actual se envíe a un documento HTML5 para su uso en JavaScript, los valores de geolocalización pueden agregar muchas nuevas características a documentos web que permiten una experiencia de usuario más interactiva.
  - Arrastrar y soltar, lo que permite que los elementos HTML5 tengan la capacidad de ser arrastrables, simple pero importante experiencia de usuario.
  - Almacenamiento local, que permite que las aplicaciones web pueden almacenar datos localmente en el navegador del usuario, antes de HTML5, los datos de la aplicación debían almacenarse en cookies, el almacenamiento local es más seguro y se pueden almacenar grandes cantidades de datos sin afectar el rendimiento del sitio web. El almacenamiento local es por origen, es decir, todas las páginas, desde un origen, pueden almacenar y acceder a los mismos datos.

Estas y muchas más funcionalidades que hacen de HTML5 un estándar más versátil, que permitirá realizar una programación web dinámica, simple y sencilla. [22]

## 2.6 D3.js

Data-Driven Documents (D3.js) es una librería de JavaScript que es un lenguaje de programación que permite realizar páginas web dinámicas e interactivas. D3 es una librería de código abierto, escrita por Mike Bostock, que permite manipular documentos basados en datos, es decir, elementos de una página web en el contexto de un conjunto de datos. [23][24]

La biblioteca D3 permite crear cualquier representación, empezando por los elementos gráficos más básicos, como círculos, líneas, cuadrados, etc. Ciertamente, este enfoque complica mucho la implementación de un gráfico, pero al mismo tiempo, permite desarrollar representaciones gráficas

completamente nuevas, sin tener que seguir los patrones preestablecidos que proporcionan otras librerías gráficas. [25]

D3 ayuda a traer datos a la vida, enlazando datos arbitrarios a un modelo de objetos de documento (DOM), lo cual es la representación en árbol de los elementos jerárquicos de un documento HTML y fue especificado por el World Wide Web Consortium (W3C), estos elementos en el DOM se llaman nodos (por ejemplo: html, head, body, etc.), que pueden tener atributos (por ejemplo: class = "header") y contenido (por ejemplo: "Mi Aplicación"); y con esto poder aplicar transformaciones basadas en datos al documento.

Además de ser una librería de datos para la manipulación de DOM, es un conjunto de herramientas gráficas con máxima flexibilidad, compatibilidad, accesibilidad y rendimiento que utiliza plenamente las capacidades de los navegadores modernos y estándares web como HTML, CSS y SVG, siendo SVG o gráficos vectoriales escalables un estándar web para gráficos vectoriales especificado por el W3C directamente en el DOM, donde un gráfico vectorial es una imagen representada únicamente por los primitivos geométricos (forma) y los atributos (tamaño, márgenes, apariencia, etc.) de sus elementos que lo contienen, estos elementos pueden ser formas primitivas (como líneas, círculos, triángulos, etc.) o formas complejas que están compuestas por estas primitivas, todos los elementos están incluidos en el gráfico. Por ejemplo, se puede utilizar D3 para generar una tabla HTML de una matriz de números, o bien, se pueden usar los mismos datos para crear un gráfico de barras SVG interactivo con transiciones e interacciones suaves como se muestra en la Figura 20. [23][26]

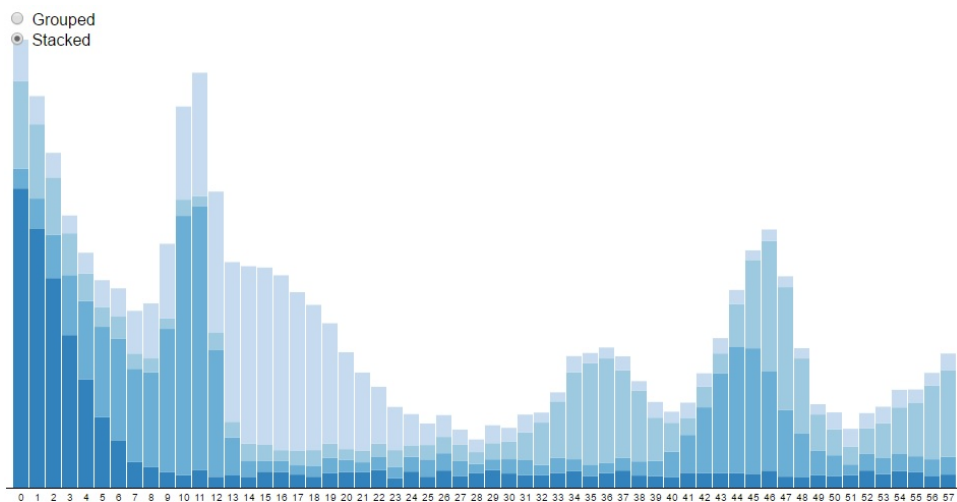


Figura 20. Gráfico de barras con D3. [23]

Con una sobrecarga mínima, D3 es extremadamente rápida, soportando grandes conjuntos de datos y comportamientos dinámicos para la interacción y la animación de los mismos. El estilo funcional de D3 permite la reutilización de código a través de una diversa colección de módulos oficiales que han sido desarrollados por la comunidad. [23]



### 3 Arquitectura

En este capítulo se describe la arquitectura utilizada en el desarrollo de este trabajo para realizar el cálculo, recomendación y señalización de rutas óptimas en un sistema de evacuación de emergencia, la cual comprende el uso de componentes que permitan obtener y clasificar la información de una red de sensores inalámbrica (WSN) para determinar las áreas afectadas y poder deducir las rutas más cortas dentro de la primera planta del edificio administrativo de la ESPOL para una pronta y efectiva evacuación.

Se definen tres fases para describir los componentes que forman parte de este sistema de evacuación, la primera es la Adquisición de Datos, donde se detallan los componentes utilizados para la recopilación de los datos que provienen de los nodos sensores y el esquema y tecnologías necesarias para la transmisión y clasificación de esa información. La segunda fase, en la cual se enfoca el desarrollo de este trabajo, es el Procesamiento de Datos, donde se describen los componentes utilizados para calcular rutas óptimas con la ayuda de la información clasificada previamente. Y por último se tiene la fase de Visualización donde se indica cómo se muestran las rutas calculadas en la fase anterior para que puedan ser visualizadas por los usuarios. Todos los componentes que conforman la arquitectura del sistema de evacuación deben funcionar conjuntamente para manejar adecuadamente la congestión en cada momento durante un evento de desastre.

En la Figura 21 se muestran los componentes principales que forman parte de la arquitectura utilizada en este proyecto divididos según las tres fases mencionadas anteriormente, los cuales serán detallados a continuación. Además, se puede observar que se utilizan contenedores de Docker para el despliegue de las tecnologías involucradas en cada fase y el uso de los recursos disponibles en una nube de Microsoft Azure para la implementación.

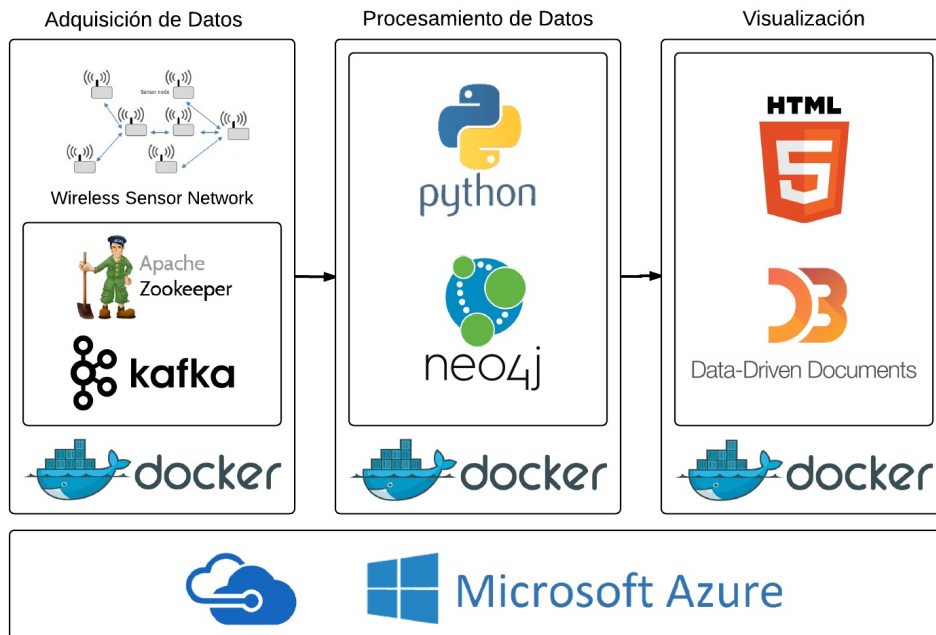


Figura 21. Perspectiva general de la arquitectura del sistema

### 3.1 Adquisición de Datos

En esta fase es donde se realiza la recopilación y transmisión de datos, para lo cual se trabaja con el modelo planteado en el estudio "A Distributed System Model for Managing Data Ingestion in a Wireless Sensor Network" [1], donde se propone un modelo de sistema distribuido que permite recolectar y gestionar flujos continuos de datos de alta velocidad obtenidos desde una red de sensores inalámbrica (WSN), utilizando una red de comunicación distribuida y técnicas de big data para realizar el procesamiento de streaming e integración de los datos, con el fin de minimizar el tiempo de alerta y evacuación para salvaguardar la vida de las personas.

Este estudio en referencia [1] propone el uso de una WSN compuesta por nodos sensores que manejan una interfaz de comunicación en el estándar 802.15.4, el cual define el protocolo y la interconexión compatible para dispositivos de comunicación de datos que utilizan transmisiones de radiofrecuencia de corto alcance, de baja tasa de datos, baja potencia y baja complejidad en una red inalámbrica de área personal (WPAN). Este estándar define las especificaciones de la capa física y de la capa media de control de acceso para la conectividad inalámbrica de baja velocidad de datos con dispositivos fijos, portátiles y móviles sin batería o con un consumo de batería muy limitado.

Los nodos sensores son responsables del envío de información que permita detectar anomalías en el ambiente como es el caso del nivel de temperatura (TMP), monóxido de carbono (CO), dióxido de carbono (CO<sub>2</sub>), gas líquido de petróleo (PLG) y nivel de líquido vertical (VLL), por lo cual deben estar distribuidos dentro del edificio de tal manera que cubran todos los puntos de entrada de los bloques en él.

En el caso de un sistema de evacuación es importante tener en cuenta que los equipos electrónicos utilizados deben protegerse para evitar daños durante algún evento de desastre, considerando el uso de una cubierta para los componentes que puede ser de dos tipos de materiales: cerámica en la superficie externa para proteger los dispositivos electrónicos de altas temperaturas producidas en caso de incendio. Y el corcho para proteger el dispositivo del agua, además de crear un aislamiento entre el material cerámico y los componentes electrónicos del nodo. El uso de estos materiales puede garantizar que cada nodo sensor continúe operando por lo menos durante los primeros 15 minutos según la situación de emergencia que se presente. [1][27]

El desarrollo de este proyecto se enfoca principalmente en el procesamiento de los datos recibidos en esta fase para realizar el cálculo de las rutas más cortas para la evacuación, por lo cual para la implementación de este trabajo, esta fase de Adquisición de Datos se realiza por medio de una simulación de la red de sensores inalámbrica y los datos que ésta recolecta de la primera planta del edificio administrativo de la ESPOL.

Además de la recopilación de los datos, esta fase implica la clasificación y transmisión de los datos recolectados por los nodos sensores, siguiendo con el esquema planteado en el estudio en referencia [1], se trabaja con un clúster de Kafka, la instalación de Kafka se puede realizar siguiendo las instrucciones que se encuentran en la página oficial, donde se puede trabajar con una versión de ZooKeeper instalada de forma independiente o trabajar con la versión que viene por defecto en Kafka, la cual ofrece una instancia de un sólo nodo de ZooKeeper. Así mismo se trabaja con un nodo llamado nodo de distribución, para el cual a nivel físico se pueden considerar tres interfaces de comunicación, la interfaz Ethernet utilizada para comunicar datos con el clúster de Kafka, un enlace de comunicación móvil inalámbrico 3G como un enlace de respaldo para ser utilizado en caso de que la interfaz Ethernet se encuentre inactiva, de esta manera se garantiza el correcto funcionamiento del sistema de evacuación incluso cuando la interfaz de comunicación puede verse afectada por alguna situación de emergencia; y por último se tiene un

módulo Zigbee para la comunicación con la WSN.

En la Figura 22, se presenta un esquema de la arquitectura donde las secciones de interés para este proyecto son: la sección A que contiene los nodos sensores que recolectan la información y la envían a la sección B, donde se encuentra este nodo de distribución, en esta sección se muestran dos nodos que contienen el mismo esquema y se comunican entre sí para proporcionar redundancia en la transmisión, teniendo a N1 como el nodo principal y N2 como el nodo de respaldo, los cuales interactúan con la sección C para la transmisión de los datos y su posterior procesamiento. [1]

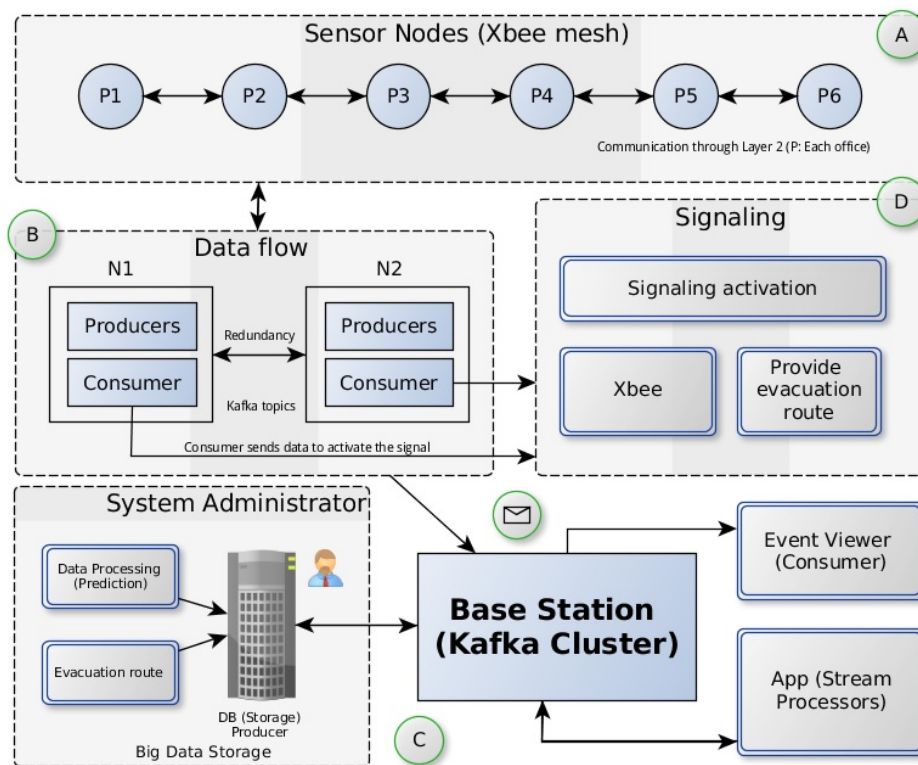


Figura 22. Arquitectura de una red de sensores inalámbricos con un clúster centralizado. [1]

El nodo de distribución contiene las aplicaciones de software que trabajan como productores y consumidores del clúster de Kafka como se puede observar en la Figura 23, donde los productores se encargan de recibir la información proveniente de los nodos sensores y enviarla a los brokers de Kafka clasificándola según los tópicos definidos, para luego ser procesada; y los consumidores reciben información de los brokers de Kafka para procesarla y realizar alguna acción determinada, como en este caso, para generar las rutas de evacuación más cortas cuando se presenta una alarma o algún comportamiento anormal en los datos que los nodos sensores recolectan del

ambiente que pueda representar algún peligro para las personas que se encuentran dentro del edificio.

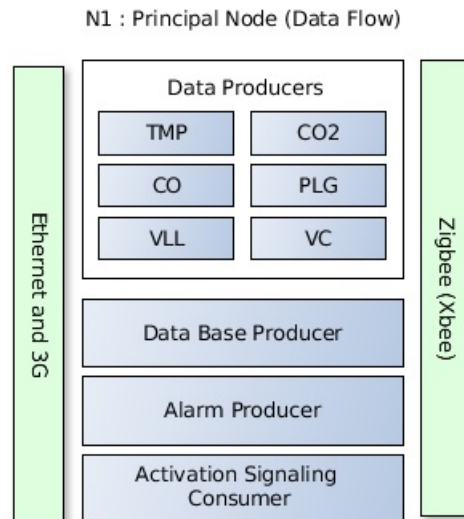


Figura 23. Nodo de comunicación entre la WSN y el clúster Kafka. [1]

A continuación se describen los componentes del nodo N1 que se muestran en la Figura 23:

- Productores de Datos (Data Producers): reciben los datos de los nodos sensores (WSN) y envían la información a los tópicos definidos en el clúster de Kafka.
- Productores de Base de Datos (Data Base Producer): se encarga de transmitir la información recibida de los nodos sensores, al igual que los productores de datos, pero a la base de datos para poder contar con un registro e historial de la actividad de los sensores. Además de evitar la sobrecarga ya que las aplicaciones de streaming (procesadores de flujos de datos) utilizarán información de los productores de datos.
- Productor de Alarma (Alarm Producer): cuando se obtienen datos de los nodos sensores, este nodo los procesa para determinar si se encuentran anomalías con los umbrales configurados para cada métrica, en caso de existir alguna anomalía, se envía un paquete de "Alarma" a un tópico definido para este fin en el clúster de Kafka.
- Consumidor de Activación de Señalización (Activating Signaling Consumer): es un proceso consumidor que se encargará de transmitir la información para la activación de la señalética. [1]

Los datos de los nodos sensores distribuidos en la planta del edificio son recolectados y transmitidos a un nodo centralizado, donde estos datos se clasifican por el tipo de información que transmiten, cada una de las métricas de interés tiene definido un tópico específico y un tópico de replica en el clúster de Kafka, es decir, si un nodo sensor envía valores de temperatura, enviará la información a dos tópicos: al tópico "TMP" y al tópico "Réplica TMP". Entre los tópicos definidos en el clúster de Kafka tenemos: monóxido de carbono "CO", dióxido de carbono "CO2", gas líquido de petróleo "PLG", nivel de líquido vertical (VLL), temperatura "TMP" y alarma "AL" que indica si existe una alerta, cada uno con su respectivo tópico de réplica como se muestra en la Figura 24. En esta figura se puede observar el esquema detallado anteriormente, desde la información recopilada por los nodos sensores que es transmitida por los productores a los brokers de Kafka, hasta el módulo de procesamiento que por medio de los consumidores genera la visualización de las rutas recomendadas para la evacuación. Este diseño contempla las diferentes aplicaciones que pueden interactuar con esta información ya que implementa un tópico de gestión de base de datos para la interacción con técnicas de big data, lo que ayudará en el procesamiento de datos sin interferir con los procesos de las aplicaciones.

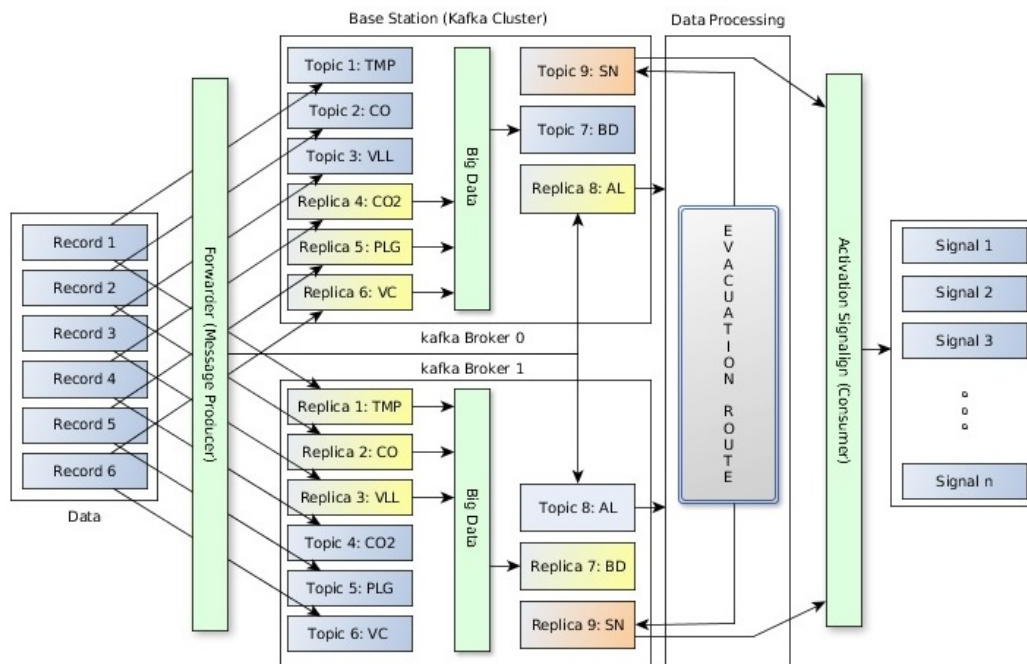


Figura 24. Arquitectura distribuida utilizando un clúster de Kafka. [1]

## 3.2 Procesamiento de Datos

Para realizar el procesamiento de datos recopilados por los nodos sensores y clasificados en los diferentes tópicos que se encuentran definidos en el clúster de Kafka como se describió en la fase anterior, se utiliza un proceso consumidor desarrollado en lenguaje Python y la base de datos orientada a grafos Neo4j.

El desarrollo de este consumidor se realiza por medio del cliente Python para el sistema de procesamiento de flujo distribuido de Apache Kafka, el cual es una librería de Python llamada "kafka-python" que se encuentra diseñada para trabajar como el cliente java oficial, que es el único que se mantiene como parte del proyecto principal de Kafka, los demás clientes se encuentran disponibles como proyectos independientes de código abierto, como es el caso de "kafka-python". Este cliente Python puede utilizarse para definir tanto productores como consumidores de Kafka, en este proyecto se utiliza para definir un consumidor aprovechando las ventajas que ofrece como son ciertas interfaces de Python como los iteradores de consumo que devuelven registros que son simples tuplas nombradas (namedtuples) con los atributos básicos del mensaje: tópico (topic), partición (partition), identificador de registro dentro la partición (offset), clave (key) y valor (value), a los cuales se puede acceder con la facilidad que brindan las tuplas nombradas, ya sea con el índice o con el nombre de los campos utilizando dot notation, que es la notación típica de la programación orientada a objetos. [28]

El consumidor de Python se suscribe a los tópicos de interés que se encuentran en el clúster de Kafka, en este caso puede ser al tópico de alarma "AL" que indica específicamente si se presenta una alerta en la planta del edificio o a los demás tópicos (que contienen la información más detallada que envían cada uno de los sensores como "TMP", "CO", etc.) que por medio de valores umbrales definidos se pueda determinar si sobrepasan el nivel normal de operación identificando una posible anomalía en el ambiente. Una vez recibido estos datos y detectado una alerta se realizan las consultas a la base de datos Neo4j sobre las rutas más cortas de evacuación.

Para la instalación de Neo4j se pueden seguir los pasos que se detallan en la página oficial, es un proceso sencillo y rápido después del cual por medio de la interfaz de usuario del navegador se puede trabajar con la base de datos directamente y para poder interactuar con la base de datos Neo4j desde Python se utiliza "Py2neo" que es una librería cliente y un conjunto de herramientas que permiten trabajar con esta base de datos desde aplicaciones Python e

incluso desde la línea de comandos. La librería central no tiene dependencias externas y ha sido cuidadosamente diseñada para ser fácil e intuitiva de usar [29].

En la Figura 25 se puede observar el esquema detallado en esta sección, sin olvidar que para la implementación se despliega todo en contenedores Docker. Este esquema se puede resumir en que una vez recibidos los datos de la fase de Adquisición de Datos por medio del consumidor de Python cada cierto tiempo, se realiza la interacción con la base de datos Neo4j para el cálculo de las rutas más cortas dentro de la planta del edificio.

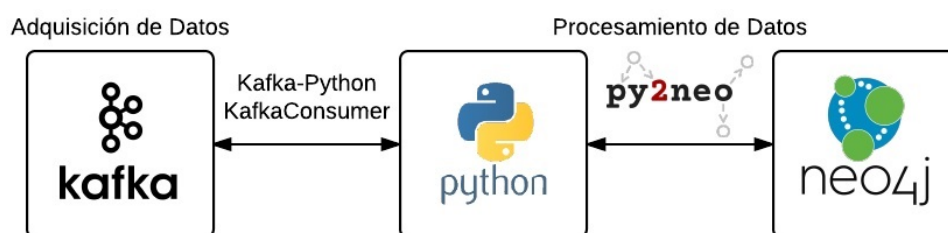


Figura 25. Esquema utilizado para el procesamiento de datos.

### 3.3 Visualización

Esta fase implica presentar de forma visual las rutas calculadas en la fase anterior para que el usuario pueda entenderlas fácilmente sobretodo en un evento de alerta o desastre para agilizar la evacuación del edificio.

Para implementar esta fase en un sistema de evacuación se debe considerar el uso de señales visuales que se encuentren distribuidas, en este caso, en la planta del edificio administrativo de la ESPOL, con el fin de mostrar a los usuarios las rutas óptimas, más rápidas y más cercanas a su ubicación para evacuar en situaciones de emergencia, esta señalización debe interactuar con los componentes de la fase de Procesamiento de Datos para que se activen de inmediato y de forma dinámica las rutas calculadas al ser detectada una alerta o una anomalía en el ambiente según las áreas habilitadas para realizar la evacuación.

Para el desarrollo de este trabajo se realiza una interfaz gráfica con el fin de poder simular y presentar de alguna manera la señalización que se obtiene como resultado del cálculo de rutas. Esta interfaz gráfica se basa en una página web en HTML5 que muestra las rutas calculadas utilizando la librería de JavaScript "D3.js".



Considerando el hecho de que en un sistema de evacuación de emergencia, la señalización debe encenderse automáticamente cuando se detecta una alerta en el ambiente, se utiliza el microframework de Python "Flask" y la extensión "Flask-SocketIO" para simular la generación de rutas de la misma forma evitando el comportamiento usual de las páginas web, es decir, sin tener que esperar una interacción de algún usuario con la página web para que se actualicen o se muestren las rutas.

Flask es uno de los frameworks que existen que facilitan trabajar con Python, un framework es una colección de paquetes o módulos que permite desarrollar aplicaciones web o servicios sin tener que manejar detalles de bajo nivel como protocolos, sockets o gestión de procesos/hilos. Aunque para referirse a Flask con más exactitud, se dice que Flask es un microframework para Python, "micro" no significa que toda la aplicación web tiene que encajar en un solo archivo Python (aunque ciertamente puede), ni significa que Flask carezca de funcionalidad. El "micro" en microframework significa que Flask tiene como objetivo mantener el núcleo simple pero extensible. De forma predeterminada, Flask no incluye una capa de abstracción de bases de datos, validación de formularios o cualquier otra cosa donde ya existan librerías diferentes que puedan manejarlo. En su lugar, Flask admite extensiones para agregar dicha funcionalidad a la aplicación como si se implementara en el propio Flask. [30][31]

Y Flask-SocketIO es una extensión muy fácil de usar que permite las comunicaciones WebSocket en aplicaciones de Flask. WebSocket es un nuevo protocolo de comunicación introducido con HTML5, principalmente para ser implementado por clientes y servidores web, aunque también puede ser implementado fuera de la web. A diferencia de las conexiones HTTP o protocolo de transferencia de hipertexto que tienen un esquema petición-respuesta, una conexión WebSocket es un canal de comunicación bidireccional permanente entre un cliente y el servidor, donde cualquiera puede iniciar un intercambio. Una vez establecida, la conexión permanece disponible hasta que una de las partes se desconecte de ella. Las conexiones de WebSocket son útiles para juegos o sitios web que necesitan mostrar información en vivo con muy poca latencia. En el lado del cliente, la librería de JavaScript SocketIO abstrae la aplicación cliente del protocolo de transporte real, para los navegadores modernos se utiliza el protocolo WebSocket, pero para los navegadores antiguos que no tienen WebSocket, SocketIO emula la conexión usando una de las soluciones más antiguas, la mejor disponible para cada cliente dado. [32][33]

Usando principalmente estas dos librerías se logra mantener una conexión abierta entre el consumidor Python y la página web donde se mostrarán las rutas. De esta manera cuando el consumidor Python detecte una alerta y consulte las rutas de evacuación en la base de datos Neo4j, como se detalló en la fase anterior, este mismo proceso envía o emite una petición a la página web para que se actualicen las rutas y se puedan observar las rutas calculadas. En la Figura 26 se presenta un esquema que muestra lo descrito anteriormente, el cual para la implementación será desplegado en contenedores Docker.

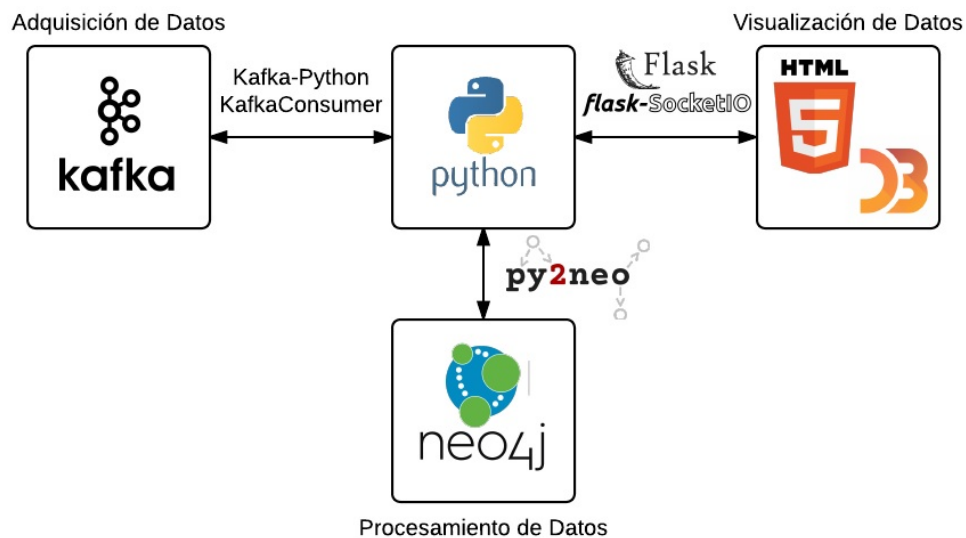


Figura 26. Esquema para visualización de rutas.

## 4 Desarrollo e Implementación

En este capítulo se describe el uso y desarrollo de todos los componentes necesarios para la implementación y ejecución de este proyecto de acuerdo a la arquitectura descrita anteriormente, comenzando por el entorno empleado para la simulación de la red de sensores inalámbrica y los datos que ésta recopila y transmite para generar las rutas de evacuación. Se describe detalladamente el diseño y modelado de los datos necesarios para poder calcular las rutas más cortas en la base de datos Neo4j. Además de explicar el desarrollo y funcionamiento del consumidor para la interacción con el clúster de Kafka y Neo4j; y cómo se presentan las rutas generadas a través de una interfaz gráfica. Finalmente se muestran los resultados que se obtienen en las pruebas del sistema.

### 4.1 Simulación de Red de Sensores Inalámbrica

Para el desarrollo de este proyecto se utiliza una simulación de la red de sensores inalámbrica para poder generar la información que debe ser recopilada por los sensores que se encuentren distribuidos en la primera planta del edificio administrativo de la ESPOL.

En base al área de interés del edificio para este proyecto se considera el uso de siete nodos sensores y cada nodo a su vez cuenta con cinco sensores para medir y transmitir información de diferentes parámetros del ambiente, estos nodos se deben encontrar distribuidos de manera estratégica para cubrir todos los espacios de la planta y poder detectar cualquier anomalía que se presente en el ambiente. En la Figura 27 se presenta un plano del área del edificio, donde se muestran todas las subáreas que contiene y las puertas o salidas conectadas directamente al corredor que deben considerarse al colocar la señalización que indique las rutas de evacuación, el corredor se encuentra conectado a tres escaleras que son las salidas de evacuación a tomar en cuenta para el cálculo de rutas. Además en esta figura se puede observar la ubicación determinada para los nodos sensores y la sección que cubre cada uno, con el fin de que toda el área se encuentre cubierta.

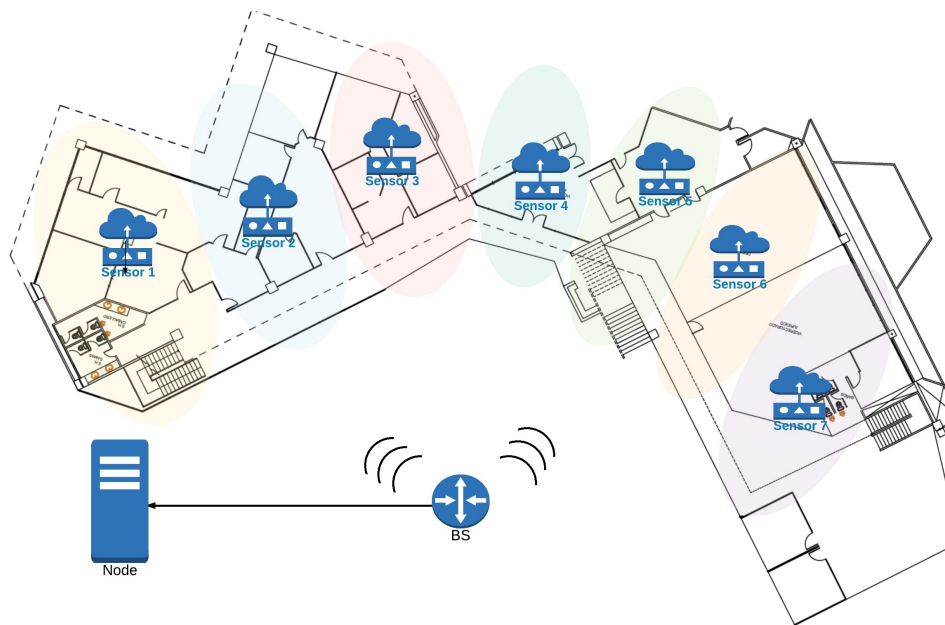


Figura 27. Plano del área del edificio y ubicación de sensores.

La simulación que se utiliza para representar el esquema mostrado en la figura anterior se encuentra desarrollada en el lenguaje de programación Java, la cual permite generar de forma dinámica la cantidad de nodos que se necesitan para simular la red de sensores inalámbrica, en este caso son siete nodos sensores y cada nodo con cinco sensores. En la Figura 28 se muestra la ejecución de la simulación donde se puede observar una interfaz gráfica de siete ventanas, cada una representa un nodo sensor y dentro de éstas se encuentran cinco pestañas una por cada sensor que contiene el nodo.

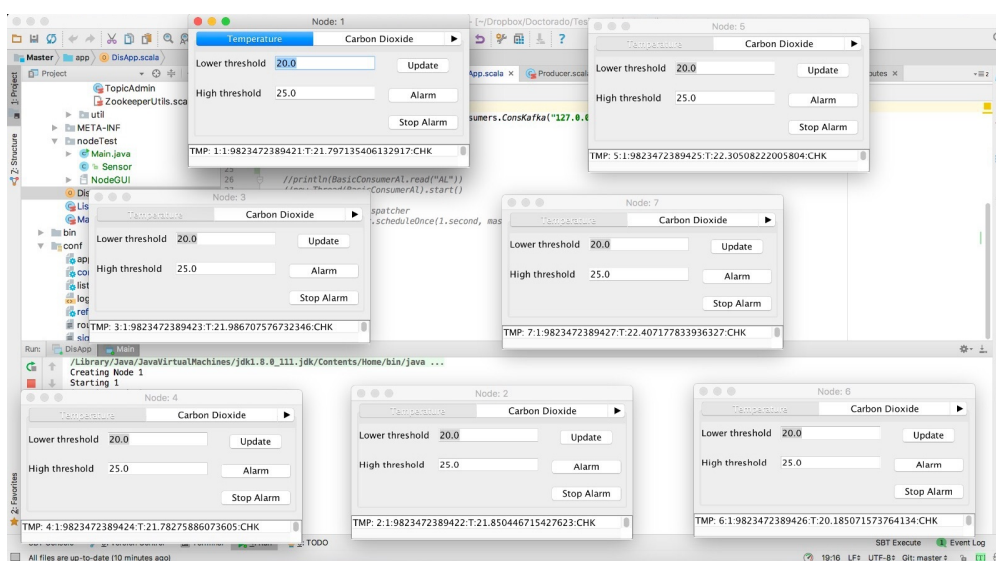


Figura 28. Simulación de la red de sensores inalámbrica.

Por medio de este programa se puede simular el envío continuo de información que deben realizar los nodos por todos los sensores que contienen y que recopilan datos del ambiente, enviando datos que son generados aleatoriamente con una cierta frecuencia por cada sensor de cada nodo simulado, en la Tabla 1 se detallan los cinco sensores con los que se trabaja en este proyecto, donde también se muestran los valores umbrales por defecto a partir de los cuales un dato generado puede considerarse como una alerta para el sistema de evacuación.

Tabla 1. Descripción de sensores.

Sensor	Descripción	Valor de Activación
TMP	Sensor de temperatura.	$\geq 40$
CO	Sensor de monóxido de carbono.	$\geq 200$
CO2	Sensor de dióxido de carbono.	$\geq 400$
PLG	Sensor de gas licuado de petróleo.	$\geq 750$
VLL	Sensor medidor de líquido vertical.	$\geq 40$

Para facilitar la configuración de la simulación se tiene la interfaz gráfica que comprende las ventanas que representan cada nodo, donde se puede ver y configurar cada sensor a través de las pestañas de navegación, pudiendo modificar los valores umbrales para la activación de alarma en caso de que sea necesario, además se puede generar una alerta por sensor presionando un botón para que el sensor comience a generar datos que pasen los valores umbrales y de igual manera se puede detener la alerta para regresar los datos generados a los valores normales. En la Figura 29 se puede observar la ventana de uno de los nodos de la simulación.

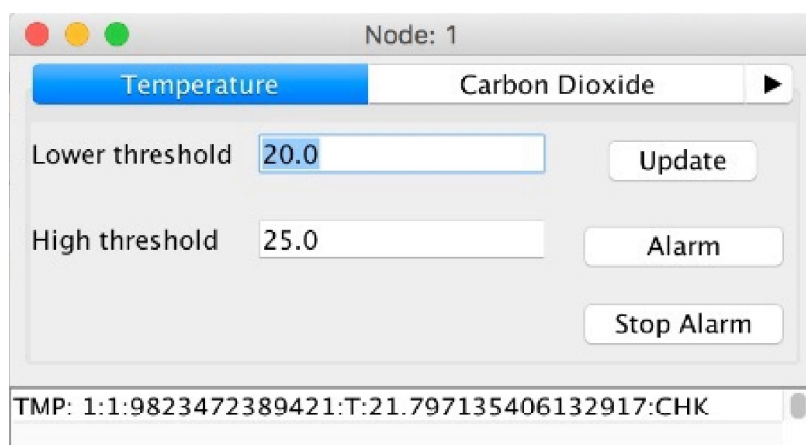


Figura 29. Nodo de simulación.

## 4.2 Modelado de Datos en Neo4j

Para realizar el cálculo y generación de rutas dentro de la primera planta del edificio administrativo de la ESPOL, se utiliza la base de datos orientada a grafos Neo4j, ya que brinda la facilidad de poder representar y acceder a los datos de una manera más sencilla.

Se comienza por la definición del grafo que se va a utilizar, en este caso después de realizar el análisis respectivo se determina que la mejor forma de modelar los datos para el cálculo de rutas es por medio del diseño de dos grafos, uno para representar los nodos sensores que deben estar asociados a la señalización que se va a activar en situaciones de alerta, es decir, cada sensor debe estar relacionado directamente con la señalización que se encuentra ubicada en su área de cobertura; y otro grafo para representar exclusivamente el mapa de la planta del edificio para poder generar las rutas de evacuación. Estos dos grafos deben estar conectados de alguna manera para poder asociar el área de cobertura del sensor con la respectiva área en el plano de la planta del edificio y poder activar la señalización que se encuentre ubicada en esa área en eventos de alerta.

Para realizar el diseño del primer grafo se debe determinar cual es la mejor ubicación para la señalización considerando las puertas o salidas que den acceso directo al corredor que se encuentra conectado a las escaleras, que en este caso son las salidas de evacuación, de tal forma que la señalización sea fácil de entender y claramente visible para las personas en eventos de desastre. Además se debe considerar la ubicación definida para los nodos sensores dentro del plano de la planta del edificio y así poder asociar según el área de cobertura del nodo sensor, la señalización que debe ser activada.

En la siguiente Figura 30 se muestra el plano del área de interés del edificio para el desarrollo de este proyecto, donde se pueden observar los nodos sensores (círculos celestes) y la señalización (círculos verdes) que se ha considerado, la cual se encuentra relacionada al nodo que da cobertura en esa área y por lo tanto será el encargado de activarla y emitir la información correspondiente que define la dirección y el color con el que se debe de encender.

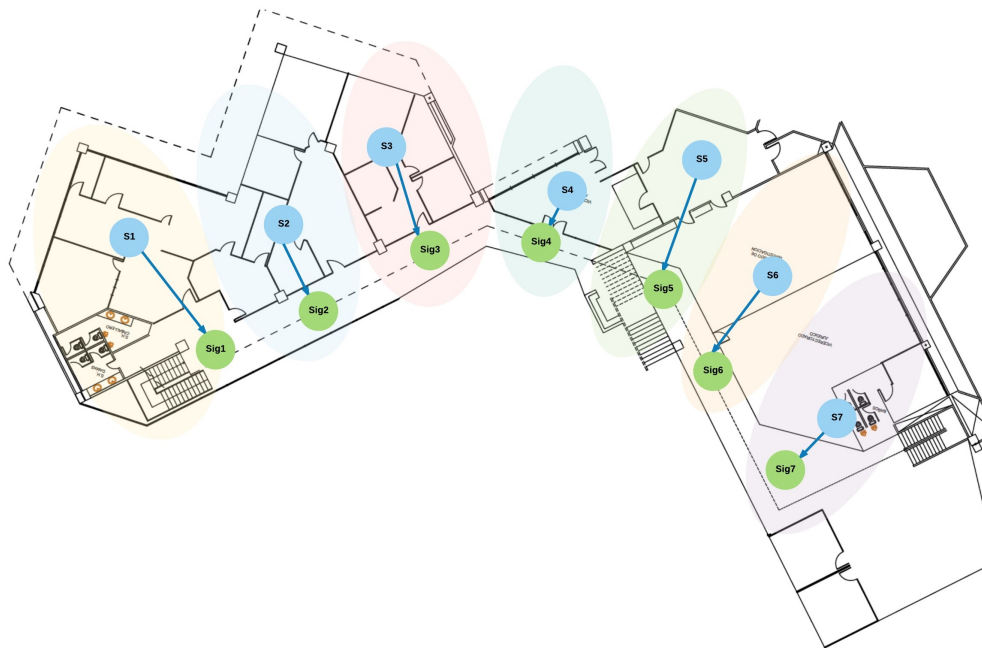


Figura 30. Diseño del primer grafo.

Teniendo en cuenta el diseño de la figura anterior se crea el primer grafo, comenzando por la creación de los nodos del grafo que representan a los nodos sensores de la red identificados por medio de la etiqueta "Sensor", para lo cual se utiliza la sentencia que se muestra en la Figura 31.

```

1 CREATE (:Sensor {name:"S1"})
2 CREATE (:Sensor {name:"S2"})
3 CREATE (:Sensor {name:"S3"})
4 CREATE (:Sensor {name:"S4"})
5 CREATE (:Sensor {name:"S5"})
6 CREATE (:Sensor {name:"S6"})
7 CREATE (:Sensor {name:"S7"})

```

Figura 31. Sentencia para crear nodos sensores.

Luego se procede a crear los nodos que representan la señalización, los cuales se identifican con la etiqueta "Signal" como se puede observar en la Figura 32.

```

1 CREATE (:Signal {name:"Sig1"})
2 CREATE (:Signal {name:"Sig2"})
3 CREATE (:Signal {name:"Sig3"})
4 CREATE (:Signal {name:"Sig4"})
5 CREATE (:Signal {name:"Sig5"})
6 CREATE (:Signal {name:"Sig6"})
7 CREATE (:Signal {name:"Sig7"})

```

Figura 32. Sentencia para crear nodos de señalización.

Y por último se crean las relaciones que existen entre los nodos sensores y la señalización como se describen en la Tabla 2.

Tabla 2. Relación entre nodos sensores y señalización.

Nodo Sensor	Conectado a
S1	Sig1
S2	Sig2
S3	Sig3
S4	Sig4
S5	Sig5
S6	Sig6
S7	Sig7

En la Figura 33 se muestra la sentencia que se utiliza para la creación de las relaciones entre los nodos con etiqueta "Sensor" y los nodos "Signal" según lo descrito en la tabla anterior, esta relación se define con la etiqueta de "CONNECTED\_TO".

```

1 MATCH (a:Sensor{name:"S1"}),(b:Signaling{name:"Sig1"})
2 CREATE (a)-[:CONNECTED_TO]->(b)

```

Figura 33. Sentencia para crear relación entre nodos sensores y señalización.

Además se relacionan los nodos entre sí por medio de la etiqueta "NEXT", definiendo cuales son los nodos sensores vecinos o mas cercanos como se aprecia en la Figura 34.





Figura 34. Sentencia para crear relación entre nodos sensores.

Una vez realizado todo lo descrito anteriormente se obtiene el primer grafo mostrado en la Figura 35.

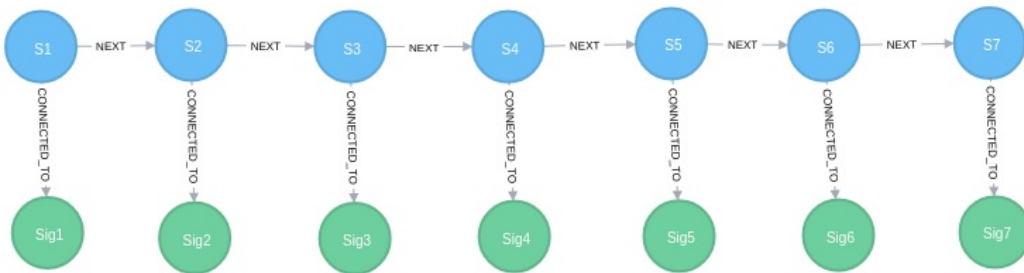


Figura 35. Grafo de nodos sensores y señalización.

Para realizar el cálculo de rutas de evacuación se determina diseñar otro grafo que represente únicamente el mapa de la planta del edificio y asociarlo a cada nodo sensor según su área de cobertura, de esta manera si un nodo sensor se alerta porque detecta alguna anomalía, se podrá conocer el área que se encuentra afectada para la generación de las rutas de evacuación y en caso de que se alerte un nodo adyacente a este, se pueda suponer una propagación del evento de desastre considerando como inhabilitada el área alertada previamente.

El segundo grafo se basa en el diseño mostrado en la Figura 36, donde se pueden observar que por cada puerta que de acceso directo al corredor, por cada intersección que se encuentra en el corredor y por cada escalera como salida de evacuación, se considera la creación de un nodo del grafo para así poder representar de una mejor manera un mapa de la planta del edificio.

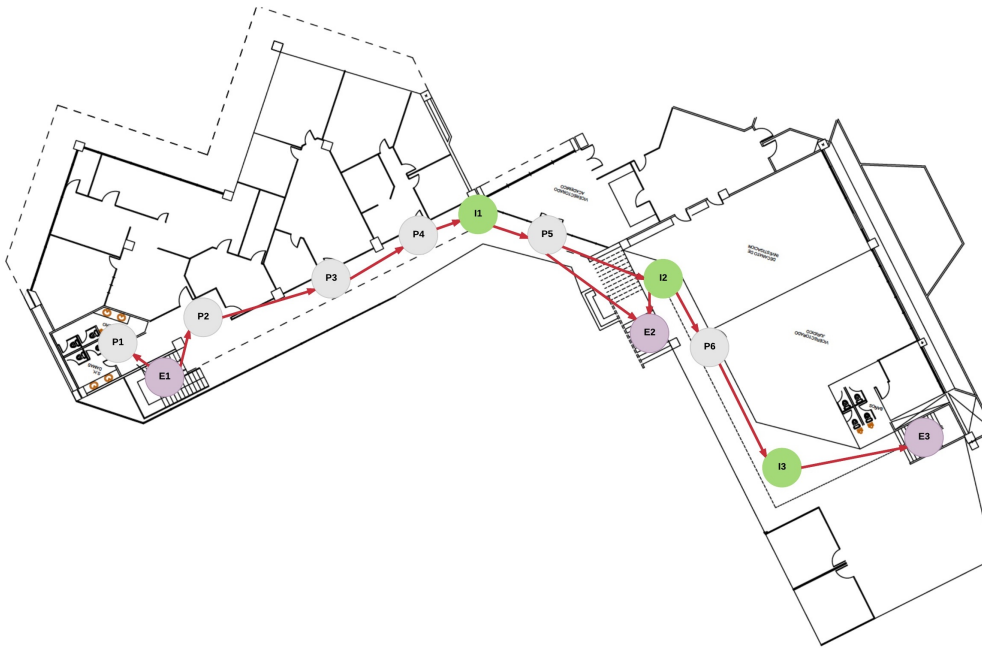


Figura 36. Diseño del segundo grafo.

Considerando el diseño de la figura anterior se crea el segundo grafo, donde se comienza por la creación de los nodos del grafo que representan las puertas identificados por medio de la etiqueta "Door", para lo cual se utiliza la sentencia mostrada en la Figura 37.

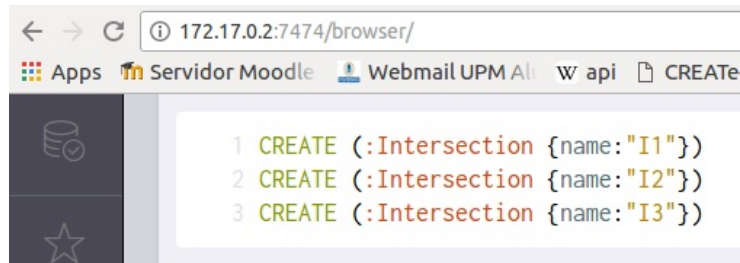
```

1 CREATE (:Door {name:"P1"})
2 CREATE (:Door {name:"P2"})
3 CREATE (:Door {name:"P3"})
4 CREATE (:Door {name:"P4"})
5 CREATE (:Door {name:"P5"})
6 CREATE (:Door {name:"P6"})

```

Figura 37. Sentencia para crear nodos de puertas conectadas al corredor.

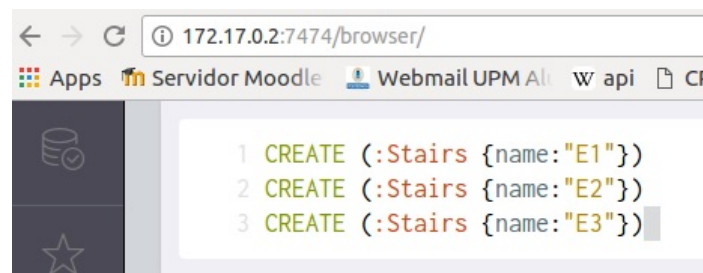
Luego se crean los nodos que representan las intersecciones del corredor con la etiqueta "Intersection", lo cual se realiza con la sentencia que puede observar en la Figura 38.



```
1 CREATE (:Intersection {name:"I1"})
2 CREATE (:Intersection {name:"I2"})
3 CREATE (:Intersection {name:"I3"})
```

Figura 38. Sentencia para crear nodos de intersecciones en el corredor.

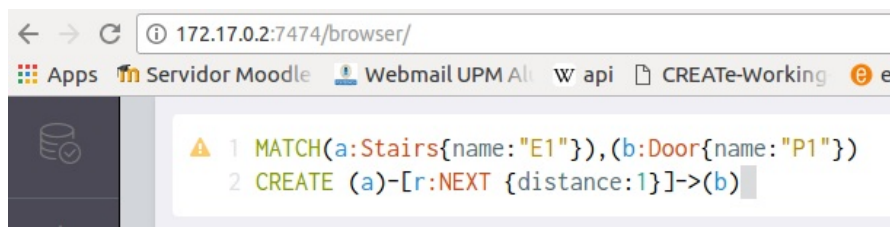
Se realiza la creación de los nodos que representan las escaleras con la etiqueta "Stairs" como se muestra en la Figura 39.



```
1 CREATE (:Stairs {name:"E1"})
2 CREATE (:Stairs {name:"E2"})
3 CREATE (:Stairs {name:"E3"})
```

Figura 39. Sentencia para crear nodos de escaleras.

Finalmente se relacionan los nodos del grafo formando una ruta genérica que recorre todo el corredor hasta llegar a todas las escaleras como se muestra en la Figura 36 con las flechas rojas, además se agrega una propiedad a esta relación entre nodo y nodo, debido que los nodos de este grafo representan puntos físicos del plano de la planta del edificio se agrega la distancia que existe entre nodo y nodo, como la propiedad "distance" en la relación, para así poder calcular cuál es la ruta más corta hacia las diferentes escaleras de evacuación, para lo cual se utiliza la sentencia de la Figura 40.



```
1 MATCH(a:Stairs{name:"E1"},(b:Door{name:"P1"}))
2 CREATE (a)-[r:NEXT {distance:1}]->(b)
```

Figura 40. Sentencia para crear relaciones entre nodos.

Después de realizar lo detallado previamente se obtiene el segundo grafo mostrado en la Figura 41.

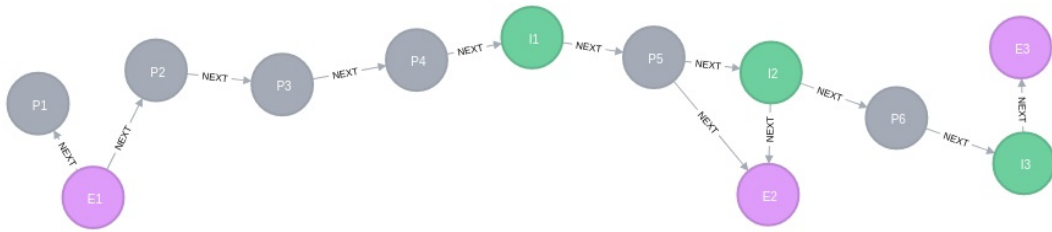


Figura 41. Grafo de la primera planta del edificio.

Ya se cuenta con los dos grafos, pero falta relacionarlos de alguna manera para que cuando se alarme un nodo sensor se pueda determinar el área afectada en el segundo grafo, calcular la ruta y activar la señalización respectiva, para esto se considera el diseño de la Figura 42, donde se delimita el área de cobertura de cada nodo sensor en el diseño del segundo grafo, para así poder definir cuáles nodos de este grafo deben ir asociados a determinado nodo sensor.



Figura 42. Diseño de asociación de los dos grafos.

Con este diseño se determina que los nodos van a estar asociados como se describe en la Tabla 3. Esta asociación se realiza agregando una etiqueta a los nodos del segundo grafo, se determina que esta etiqueta sea el nombre del nodo sensor al cual se encuentran asociados según lo detallado en la Tabla 3. En la Figura 43 se puede observar la sentencia que se utiliza para agregar esta nueva etiqueta, lo que debe ser realizado para cada nodo sensor.

Tabla 3. Asociación entre nodos sensores y grafo de la planta del edificio.

Nodo Sensor	Implica
S1	E1 P1 P2
S2	P3
S3	P4 I1
S4	P4
S5	E2 I2
S6	P6
S7	E3 I3

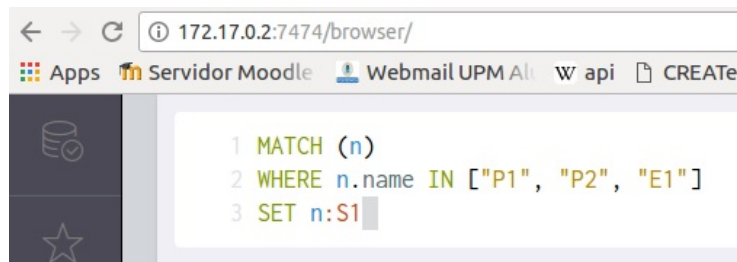


Figura 43. Sentencia para crear etiqueta para relacionar los dos grafos.

Y con esto se cuenta con los datos necesarios modelados para realizar el cálculo de las rutas de evacuación en este proyecto por medio de los dos grafos que se obtienen como resultado.

### 4.3 Desarrollo del Consumidor de Kafka

Para el desarrollo del consumidor de Kafka se utiliza el lenguaje de programación Python, este consumidor es una aplicación cliente encargada de obtener continuamente los datos que envía la simulación de la red de sensores inalámbrica desde los tópicos configurados en el clúster de Kafka y en caso de detectar una alarma generar las rutas de evacuación hacia las escaleras que se encuentran en la primera planta del edificio.

Comenzando por el consumo de los tópicos que se encuentran definidos en el clúster de Kafka, lo cual se realiza mediante el uso de la librería "kafka-

python" que permite crear un cliente de Kafka ya sea productor o consumidor, para este caso se define que este cliente consuma del t3pico "AL" (Alarma) repetidamente, en la Figura 44 se muestra como se importa la librería que se usa para el desarrollo del consumidor que es "KafkaConsumer" y luego en la Figura 45 como se conecta al cl3ster y se consume del t3pico "AL", mediante el bucle "for" se mantiene consumiendo la informaci3n de manera continua y adem3s se puede observar que una vez recuperada la informaci3n recibida del t3pico, se envía al m3todo "getPath" que es el encargado de calcular las rutas de evacuaci3n en caso de que se presente una alarma. En el Anexo A se encuentra el c3digo fuente completo donde se puede ver este m3todo.

```
from kafka import KafkaConsumer
```

Figura 44. Se importa librería para desarrollar consumidor.

```
consumerKafka= KafkaConsumer("AL",
                               bootstrap_servers='192.168.0.155:9092')
for msg in consumerKafka:
    value= eval(msg.value.decode("utf-8"))
    response= graph_db.getPath(value[0],
                                "S" + str(int(value[1][0])),
                                str(int(value[1][1])))
```

Figura 45. C3digo para consumir del cl3ster de Kafka.

Cuando el consumidor detecte una alerta, debe comunicarse con la base de datos Neo4j para calcular las rutas de evacuaci3n, esto se realiza mediante la librería "py2neo", por lo cual primero se debe importar la librería para luego establecer la conexi3n con la base de datos. En la Figura 46 se observa como se importa la librería y en la Figura 47 se aprecia el c3digo que se utiliza para establecer conexi3n con la base de datos obteniendo una instancia de la misma para poder acceder a los grafos. En el Anexo B se muestra el c3digo fuente completo donde se encuentran los m3todos utilizados para interactuar con la base de datos.

```
from py2neo import Graph, NodeSelector, Path
```

Figura 46. Se importa librería para conectarse con Neo4j.

```
graph = Graph("http://172.17.0.2:7474/db/data/")
```

Figura 47. C3digo para conectarse con Neo4j.

Entre los datos que se reciben del t3pico "AL" del cl3ster de Kafka se encuentra: si existe o no una alerta (1 o 0 respectivamente), el n3mero del nodo sensor y el n3mero de sensor que recopila esa informaci3n, siendo estos

los que se envían como parámetros de entrada al método "getPath" para el cálculo de rutas, donde con esta información se pueden presentar casos en los que se reciba una alerta de varios sensores del mismo nodo, sin embargo se debe considerar el hecho de que sólo con un sensor que presente una alerta, el nodo debe quedar en estado alertado, es por esto que se agrega la propiedad "alerted" en los nodos sensores del grafo para poder ir almacenando el estado del nodo alertado y así si ya se encuentra en estado de alerta el nodo, no es necesario volver a calcular la ruta porque ya se encuentra generada. Por lo cual lo primero que se realiza antes del cálculo de rutas es verificar si el nodo sensor ya se encuentra con el estado alertado mediante la propiedad "alerted" y en caso de que no exista o tenga valor 0 se modifica esta propiedad, como se puede observar en la Figura 48.

```
updateGraph= False
if(not alertedNode["alerted"]):
    updateGraph= True
    alertedNode["alerted"]= 1
    graph.push(alertedNode)
```

Figura 48. Código para verificar si el nodo ya se encuentra en alerta.

Cuando se detecta una alerta en los datos recibidos del clúster de Kafka y el nodo no se encuentra en estado de alerta, se consulta a la base de datos las rutas de evacuación disponibles según las alertas de los nodos sensores recibidas previamente, es decir, en caso de que un nodo sensor ya se encuentre alertado y luego se alerte un nodo adyacente, se puede suponer que el área que cubre el nodo anterior se encuentra inactiva o inhabilitada, ya que el evento se está propagando por lo cual no es recomendable tomar esa ruta.

Debido a esto antes de consultar las rutas de evacuación, se consultan los nodos que se encuentran adyacentes al nuevo nodo sensor alertado a través de las relaciones que éste tiene asociadas, ya sea como nodo de inicio o nodo final de la relación y se verifica si alguno de esos nodos se encuentra ya alertado, lo que se realiza por medio de la propiedad "alerted" de los nodos sensores donde se guarda el estado, de existir alguno alertado se consultan los nodos del segundo grafo (del plano de la planta del edificio) que se encuentren asociados a este nodo, es decir, se consultan los nodos que representan puntos físicos del plano que se encuentren dentro del área de cobertura del nodo sensor alertado previamente, mediante la etiqueta que corresponde al nombre de este nodo sensor (ver Tabla 3) y se les asigna el valor de 1 en una propiedad llamada "inactive", como se muestra en la Figura 49.



```

listTmp= list(graph.match(start_node=alertedNode, rel_type="NEXT"))
if(len(listTmp) > 0):
    nodoTmp= listTmp[0].end_node()
    if(nodoTmp["alerted"]):
        inactiveNodes= selector.select(nodoTmp["name"])
        for n in inactiveNodes:
            n["inactive"]= 1
            graph.push(n)
listTmp= list(graph.match(end_node=alertedNode, rel_type="NEXT"))
if(len(listTmp) > 0):
    nodoTmp= listTmp[0].start_node()
    if(nodoTmp["alerted"]):
        inactiveNodes= selector.select(nodoTmp["name"])
        for n in inactiveNodes:
            n["inactive"]= 1
            graph.push(n)

```

Figura 49. Código para verificar si un nodo se encuentra adyacente a otro ya alertado.

Una vez que se verifica lo descrito anteriormente, se procede a realizar la consulta de las rutas más cortas de evacuación en la base de datos, utilizando el código y la sentencia en Cypher que se muestra en la Figura 50. En este caso se utiliza una sentencia que busca las relaciones entre los nodos acumulando por medio de la función REDUCE la distancia total de las rutas que coinciden con los nodos buscados y luego se escoge la ruta con la distancia total menor.

```

for sensor in sensors:
    query= """MATCH (startNode:%s),
                (endNode:Stairs),
                p=(startNode)-[:NEXT*]-(endNode)
                WHERE ALL(n IN nodes(p)
                WHERE n.inactive IS NULL OR n.inactive = 0)
                RETURN p AS shortestPath,
                REDUCE(distance = 0, r IN relationships(p) |
                distance + COALESCE(r.distance, 0)) AS distance
                ORDER BY distance ASC
                LIMIT 1""" %(sensor["name"])
    cursor= graph.run(query)

```

Figura 50. Código para calcular rutas con Neo4j.

En esta sentencia se comienza definiendo el nodo de origen que es el punto de inicio desde el cual se quiere calcular la ruta más corta, luego se define el nodo final que en este caso son las tres escaleras para así poder determinar cuál es la que se encuentra más cerca y las relaciones que existen entre estos nodos, es decir, todos los nodos con sus relaciones que se encuentren en el camino hasta llegar al nodo final. Además se valida que sólo obtenga rutas que involucren nodos que no se encuentren inactivos, esto se aplica para el caso en que ya se encuentra un nodo sensor adyacente alertado, por lo tanto se encuentra inactiva su área de cobertura ya que se supone una propagación



del evento. En la siguiente línea se indica que sólo se retornen como resultado las rutas (relaciones) en una variable llamada "shortestPath", luego se utiliza la función de Neo4j "REDUCE" que sirve para ejecutar una expresión contra cada elemento individual de una lista y almacenar el resultado de la expresión en un acumulador, funciona como el método "reduce" en lenguajes funcionales como Lisp y Scala, por lo que se la usa para ir sumando la distancia, esto es el valor de la propiedad "distance" que se encuentra definido en todas las relaciones intermedias que se deben de recorrer para llegar al nodo final y así obtener la distancia total de ese camino, el valor acumulado se almacena en una variable o acumulador que en este caso se llama "distance" como se muestra en la sentencia. Una vez obtenida cada ruta con la distancia total, se ordena por el valor de la distancia en orden ascendente y se limita el resultado que retorna la sentencia a 1, de esta manera sólo se obtiene el primer resultado que sería el de menor valor de distancia. Se obtienen varias rutas debido a que se buscan las rutas más cortas hacia todos los nodos que representan las escaleras, que en este caso son 3 y luego de eso se debe escoger la más cercana que sería la que tiene menos distancia total.

Esto funciona muy bien en un pequeño conjunto de datos. Pero en grafos más grandes puede conducir a una explosión combinatoria. Por lo cual se realiza la consulta de las rutas más cortas utilizando el algoritmo Dijkstra ya que es uno de los algoritmos más utilizados para consultar las rutas entre dos nodos de un grafo debido a que considera un peso, en este caso la distancia entre las relaciones de estos nodos. Neo4j siempre ha tenido un paquete lleno de algoritmos de grafos disponibles los cuales sólo eran accesible desde la API JAVA, pero por medio del plugin APOC, que debe ser agregado en la carpeta "plugins" de Neo4j, se puede utilizar este algoritmo de Cypher. En la Figura 51 se puede ver la sentencia utilizando este algoritmo.

```

query= """MATCH (startNode:§s),
            (endNode:Stairs)
            CALL apoc.algo.dijkstra(startNode, endNode, 'NEXT', 'distance')
            YIELD path, weight
            WITH path, weight
            WHERE ALL(n IN nodes(path)
                    WHERE n.inactive IS NULL OR n.inactive = 0)
            AND weight > 0
            RETURN path AS shortestPath, weight AS distance
            ORDER BY distance ASC
            LIMIT 1""" §{sensor["name"]}
cursor= graph.run(query)

```

Figura 51. Código para calcular rutas con Dijkstra en Neo4j.

Donde luego de definir los nodos de inicio y fin de la ruta como se describe anteriormente, se llama el algoritmo Dijkstra por medio de una función que

recibe como parámetros los nodos, la relación y se envía la propiedad de la relación que se tomará como el peso, en este caso la propiedad "distance", definiendo que en las variables "path" y "weight" se obtenga cada ruta y la distancia total respectivamente. Luego se indica que los nodos considerados en la ruta no se encuentren inactivos, similar a la sentencia anterior. Es importante indicar que la distancia que se encuentra en la variable "weight" debe ser mayor a 0 ya que dijkstra define como 0 al mismo nodo que se busca como nodo final de la ruta. Y en las últimas tres líneas ya se define lo que se desea obtener como resultado, se ordenan por la distancia de forma ascendente ya que se obtienen las rutas más cortas hacia todos los nodos que representan las escaleras y de esas se obtiene la menor.

La ejecución de esta sentencia se realiza por cada nodo sensor debido a que cada nodo sensor se encuentra asociado a un nodo de señalización y así se puede obtener todas las rutas para luego obtener las direcciones en las que se deben activar todos los nodos de señalización, ya que en un evento de alerta toda la señalización de la planta del edificio debe activarse para realizar una evacuación no sólo el lado afectado. En caso de que un nodo ya se encuentre alertado y se alerta un nodo adyacente, el camino del nodo previo se inactiva, por lo que no se considera para calcular las demás rutas y no se obtendría ninguna dirección para la señalización asociada al área de ese nodo.

Con el resultado obtenido del cálculo de rutas se generan dos variables de resultado con una cadena de texto en formato JSON como se muestra en la Figura 52, que son utilizados en la parte de visualización que se describe más adelante, una de estas variables guarda las rutas y direcciones de los nodos de señalización y el otro contiene las rutas dentro del plano del edificio.

```
data["links"]= result
data1["links"]= result1
updateRes= {"graph1": json.dumps(data), "graph2": json.dumps(data1)}
```

Figura 52. Resultado en formato JSON.

Además se define un método llamado "reset" para reiniciar la generación de rutas para poder realizar algún otro caso de prueba, reiniciando los valores de las propiedades que se modifican en el proceso de cálculo, para esto se interactúa con la base de datos por medio de las sentencias que se muestran en la Figura 53.

```
def reset():
    graph.run("MATCH (n) WHERE n.alerted = 1 SET n.alerted = NULL")
    graph.run("MATCH (n) WHERE n.inactive = 1 SET n.inactive = NULL")
```

Figura 53. Código para reiniciar valores en propiedades.

Debido a que las rutas que se obtienen como resultado se presentan en una página web, por lo cual se debe actualizar constantemente la página web y con esto las rutas que se muestran en caso de que se generen alertas en los nodos sensores, sin necesidad de interacción del usuario, es decir, sin necesidad de que se realice primero una petición desde el lado del cliente (página web) hacia el lado del servidor (consumidor) como es generalmente, es por esto que se utiliza la librería "Flask-SocketIO" junto a "Flask" para que desde el lado del servidor se pueda actualizar la página web sin necesidad de interacción cada vez que el consumidor realice el proceso para calcular las rutas de evacuación.

En la Figura 54 se muestra como se importan las librerías para trabajar con ellas y en la Figura 55 se muestra como se inicia el servidor web de la aplicación.

```
from flask import Flask, render_template, session, request
from flask_socketio import SocketIO, emit, disconnect
```

Figura 54. Importar librerías para trabajar con Flask y Flask-SocketIO.

```
if __name__ == '__main__':
    socketio.run(app, host="0.0.0.0")
```

Figura 55. Iniciar aplicación.

Se define un método "test\_connect" que se ejecuta cuando se establece la conexión con el cliente, en el cual se va a configurar la ejecución de un método en segundo plano (background task) que será el encargado de consumir continuamente los datos del clúster de Kafka, en la Figura 56 se muestra el código de este método.

```
@socketio.on('connect', namespace='/tfm')
def test_connect():
    global thread
    if thread is None:
        thread = socketio.start_background_task(target=background_thread)
```

Figura 56. Ejecución de método en segundo plano.

En la Figura 57 se puede observar el código del método que se ejecuta en segundo plano, donde se consume de manera continua del clúster de Kafka y esto se envía al método "getPath" donde se realiza la consulta y generación de

rutas para luego enviar el resultado por medio del comando "socketio.emit" a la página web para que se actualice en caso de que se detecte una alerta o un cambio en las rutas generadas.

```
def background_thread():
    consumerKafka= KafkaConsumer("AL",
                                  bootstrap_servers='192.168.0.155:9092')
    for msg in consumerKafka:
        value= eval(msg.value.decode("utf-8"))
        response= graph_db.getPath(value[0],
                                    "S" + str(int(value[1][0])),
                                    str(int(value[1][1])))
        if(response):
            print(response["graph1"])
            socketio.emit('my_response',
                           {"data":response["graph1"], "graph":"graph1"},
                           namespace='/tfm')
            socketio.emit('my_response',
                           {"data":response["graph2"], "graph":"graph2"},
                           namespace='/tfm')
```

Figura 57. Método que consume del clúster de Kafka.

Además se define un método "reset\_request" mostrado en la Figura 58, que ejecuta el método "reset" para reiniciar desde la página web los parámetros modificados durante la generación de las rutas, en caso de que se deseen realizar otros escenarios de prueba.

```
@socketio.on('reset', namespace='/tfm')
def reset_request():
    response= graph_db.reset()
    emit('my_response',
          {"data":response["graph1"], "graph":"graph1"},
          namespace='/tfm')
    emit('my_response',
          {"data":response["graph2"], "graph":"graph2"},
          namespace='/tfm')
```

Figura 58. Método para parámetros modificados durante la generación de las rutas.

## 4.4 Visualización

La visualización de las rutas de evacuación generadas se realiza mediante el uso del lenguaje HTML5 y la librería de JavaScript D3.js con el fin de simular de una manera amigable la activación de la señalización en la primera planta del edificio administrativo de la ESPOl cuando se detecte una alarma en la información recopilada por los nodos sensores.

La librería D3.js brinda las facilidades para presentar en una interfaz web un grafo al igual como se ve en la interfaz de usuario del navegador que proporciona Neo4j que se encuentra basada en esta librería, por lo tanto la visualización de las rutas calculadas en la interfaz gráfica se basa en la presentación de los grafos de la red de sensores y del plano de la planta del edificio con sus nodos y enlaces de tal forma que se puedan representar la señalización que será activada dentro del plano del edificio y otro para mostrar la ruta en sí generada.

Se comienza el desarrollo con la creación de una página web en HTML5 que contenga principalmente dos elementos SVG que son contenedores para gráficos vectoriales escalables que se van a utilizar para la presentación de las rutas, estos gráficos tienen como imagen de fondo el plano de la planta del edificio para poder simular las rutas dentro del mismo. Además se considera la creación de un botón para que el usuario pueda reiniciar desde la página web los parámetros modificados durante el cálculo de rutas. Todo lo detallado anteriormente se puede observar en la Figura 59 y en el Anexo C se encuentra el código completo de la página web.

```

<div style="width:100%; padding:10px;">
  <form id="reset" method="POST" action="#">
    <button type="submit">Reset
      <i class="w3-margin-left fa fa-refresh"></i>
    </button>
  </form>
</div>
<svg id="graph1" width="960" height="600"
  onclick="showCoords(event)"></svg>
<svg id="graph2" width="960" height="600"
  onclick="showCoords(event)"></svg>

```

Figura 59. Elementos de la página web.

Además del estilo (style) de la página web, se agregan las librerías JQuery y D3.js que se van a utilizar, para comenzar con el desarrollo del código JavaScript que va a permitir presentar las rutas.

Se define una función JavaScript llamada "updateData" que recibe como parámetros el objeto JSON que contiene toda la información que se recibe del consumidor Python para generar las rutas en la página y además recibe el identificador ("id") del elemento SVG al que está asociado esa información. En la Figura 60 se puede observar esta función donde el identificador se utiliza para obtener el elemento SVG en el que se van a presentar los datos recibidos. Además se muestra como se crea una nueva simulación con la librería D3.js

```

function updateData(graph, graphId){
  var svg = d3.select("svg#" + graphId),
      width = +svg.attr("width"),
      height = +svg.attr("height");

  var color = d3.scaleOrdinal(d3.schemeCategory20);

  var simulation = d3.forceSimulation()

```

Figura 60. Función JavaScript para presentar las rutas.

En la siguiente Figura 61 se muestra el código utilizado para la creación de los enlaces del grafo, los cuales se presentan en diferentes formas según el grafo, ya que pueden representar la señalización que se activa en el edificio o las rutas.

```

var link = svg.append("g")
  .attr("class", "links")
  .selectAll("line")
  .data(graph.links)
  .enter().append("line")
  .attr("stroke-width", 7)
  .each(function(d) {
    if("direction" in d){
      d3.select(this).attr("marker-end", "url(#end)")
        .attr("class", "flowline");
    }else if("final" in d){
      d3.select(this).attr("marker-end", "url(#end)");
    }else{
      d3.select(this).style("stroke", "red");
    }
    if(d.color){
      d3.select(this).style("stroke", "#ffa1a1");
      svg.selectAll("marker").style("fill", "#FF5733");
    }
  });

```

Figura 61. Creación de enlaces en grafos para visualización.

Luego se definen los nodos de los grafos como se aprecia en la Figura 62, que en este caso para personalizar la presentación, sólo los nodos que representan a los nodos sensores en el plano se podrán visualizar, los demás nodos tendrán un tamaño mínimo para que no se visualicen pero sirven para poder mostrar la señalización o las rutas según corresponda.



```

var node = svg.append("g")
    .attr("class", "nodes")
    .selectAll("circle")
    .data(graph.nodes)
    .enter().append("circle")
    .attr("r", function(d) {return (d.width ? d.width : 1);})
    .attr("fill", function(d) {
        return (d.width ? d.color? d.color : "blue" : "transparent");
    })
    .each(function(d) {
        if(d.width){
            d3.select(this).style("filter", "url(#shadow)");
        }
    });

```

Figura 62. Creación de nodos en grafos para visualización.

Después de todo lo descrito anteriormente se procede a agregar los nodos y los enlaces en la simulación creada antes como se muestra en la Figura 63.

```

simulation
    .nodes(graph.nodes)
    .on("tick", ticked);

simulation.force("link")
    .links(graph.links);

```

Figura 63. Se agregan nodos y enlaces a la simulación.

Para realizar la conexión con el consumidor de Python se utiliza la librería Socket.IO de JavaScript, en la Figura 64 se muestra como se establece la conexión con el consumidor y la definición de los métodos y eventos para interactuar con el mismo, como es el caso del evento "my\_response" que se define con "socket.on" el cual se ejecuta cuando el consumidor envía los datos para generar y actualizar las rutas en la página web, como se puede ver este evento invoca la función "updateData" descrita anteriormente. Además se define la acción que se realiza cuando se presiona el botón de "Reset", el cual envía una petición al lado del servidor para que reinicie los parámetros modificados durante la generación de rutas, es decir, para reiniciar la simulación. Y por último se inicializa la simulación donde sólo se muestran los nodos sensores en el grafo en espera de que alguna alerta sea generada y se actualice la señalización y las rutas.

```

$(document).ready(function() {
  $.ajaxSetup({ cache: false });
  namespace = '/tfm';
  var socket = io.connect(location.protocol +
    '//' + document.domain + ':' +
    location.port + namespace);

  socket.on('connect', function() {
    //alert("connected");
  });

  socket.on('my_response', function(msg) {
    d3.selectAll("svg#" + msg.graph + " > *").remove();
    updateData(JSON.parse(msg.data), msg.graph);
  });

  $('form#reset').submit(function(event) {
    socket.emit('reset');
    return false;
  });

  $.getJSON("static/data.json", function( graph ){
    updateData(graph, "graph1");
    updateData(graph, "graph2");
  });
});

```

Figura 64. Conexión con el consumidor de Python desde JavaScript.

En la Figura 65 se puede observar como se ve la página web cuando se inicia la simulación.

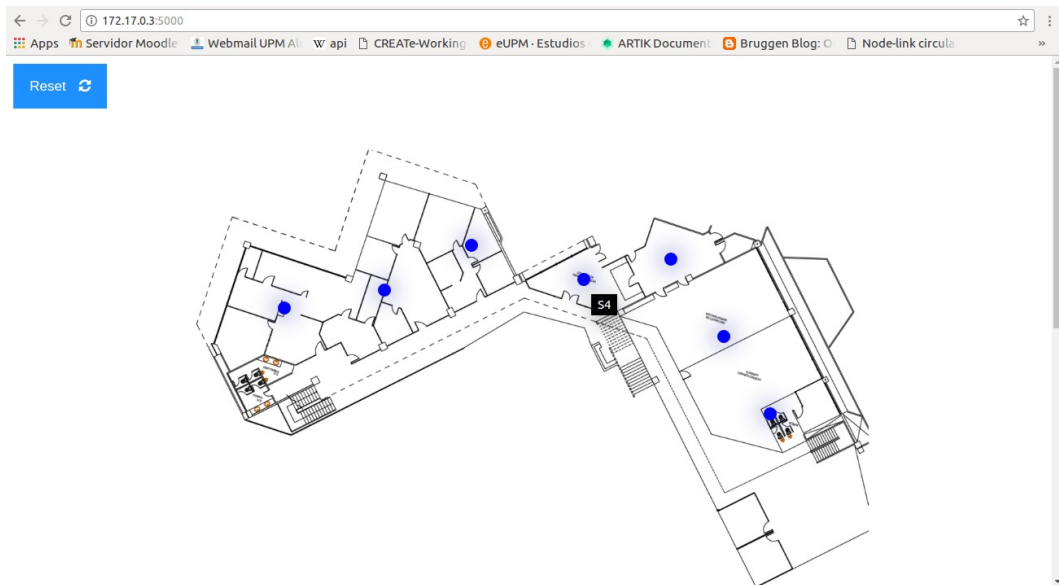


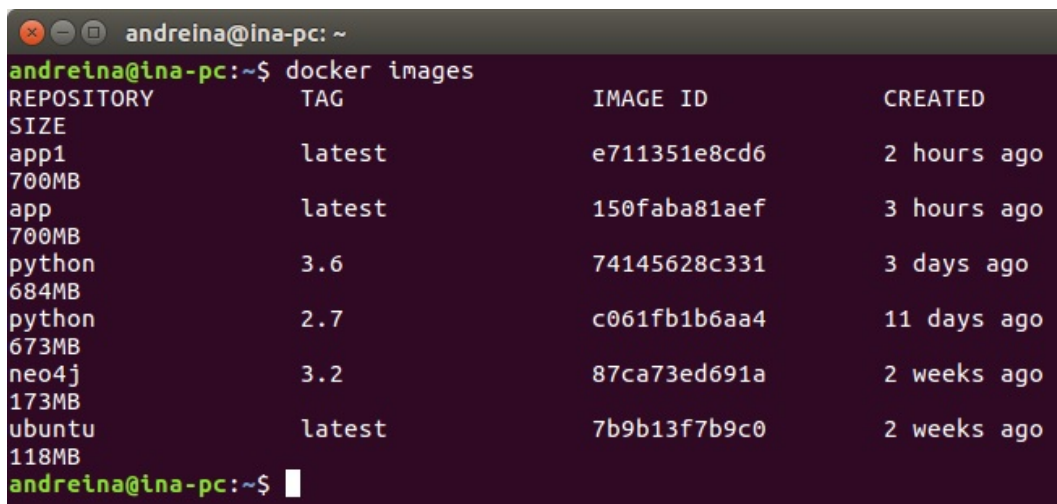
Figura 65. Interfaz web para simulación de señalización.



## 4.5 Pruebas

En esta sección se detallan las pruebas realizadas con todo el sistema integrado, donde se utiliza la simulación de la red de sensores que envía datos cada 5 segundos y por medio de la interfaz gráfica de los nodos sensores se generan alertas de diferentes sensores para que el consumidor detecte estas alarmas y genere las rutas por cada vez que se alerte un nodo sensor para así poder visualizarlas en la página web.

En la Figura 66 se muestran las imágenes creadas en Docker para el despliegue y prueba de la aplicación en un contenedor y la base de datos Neo4j en otro contenedor, donde la imagen de la aplicación se llama "app1" y la imagen de la base de datos es "neo4j". En los Anexos D y E se muestran los archivos llamados "Dockerfile" para la creación de estas imágenes.

A terminal window titled 'andreina@ina-pc: ~' showing the output of the 'docker images' command. The output is a table with columns: REPOSITORY, TAG, IMAGE ID, and CREATED. The rows list various Docker images including 'app1', 'app', 'python', 'neo4j', and 'ubuntu' with their respective sizes, tags, image IDs, and creation times.

```
andreina@ina-pc:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
app1                 latest             e711351e8cd6       2 hours ago
700MB
app                  latest             150faba81aef       3 hours ago
700MB
python               3.6                74145628c331       3 days ago
684MB
python               2.7                c061fb1b6aa4       11 days ago
673MB
neo4j                 3.2                87ca73ed691a       2 weeks ago
173MB
ubuntu               latest             7b9b13f7b9c0       2 weeks ago
118MB
andreina@ina-pc:~$
```

Figura 66. Imágenes de Docker.

A continuación se muestran las figuras donde se presentan los resultados de las pruebas, se muestra como se generan las rutas y se activa la señalización en la interfaz web.

En la Figura 67 se muestra el resultado de la señalización obtenida en la interfaz web cuando el nodo sensor "S4" se ha alertado y luego se alerta el nodo "S5", en este caso se inhabilita el camino que cubre el nodo "S4" debido a que el evento detectado inicialmente se está propagando al nodo sensor adyacente.

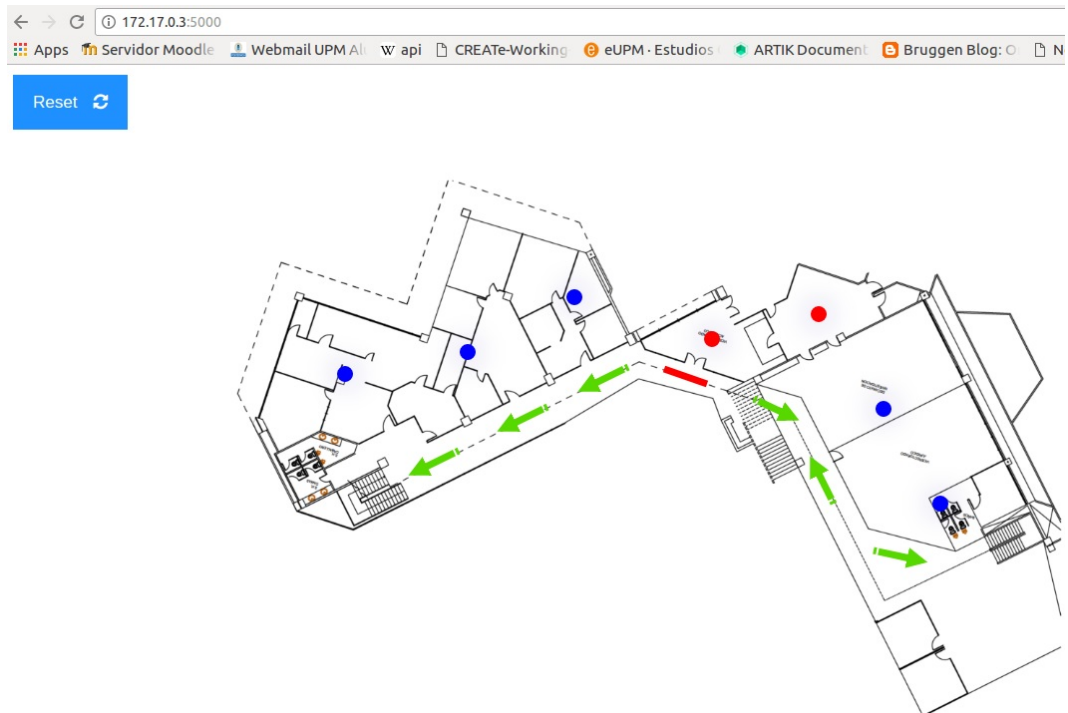


Figura 67. Señalización cuando se alerta el nodo sensor S5 mientras el nodo S4 ya está alertado.

Y en la Figura 68 se pueden ver las rutas que se generan en este mismo escenario de prueba.

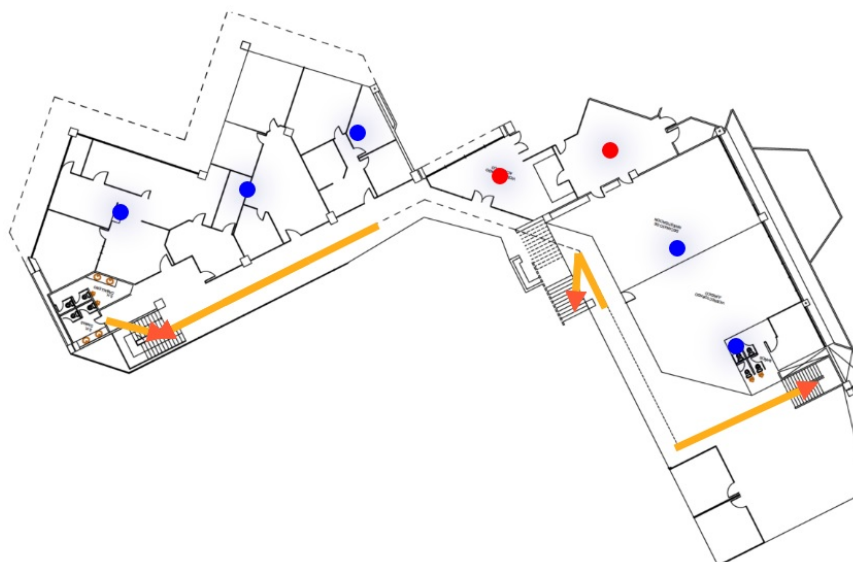


Figura 68. Rutas generadas cuando se alerta el nodo sensor S5 mientras el nodo S4 ya está alertado.

En el mismo escenario de prueba, en la Figura 69 se muestra el resultado

cuando se alerta el nodo sensor "S1" luego de que se alerta el nodo "S2".

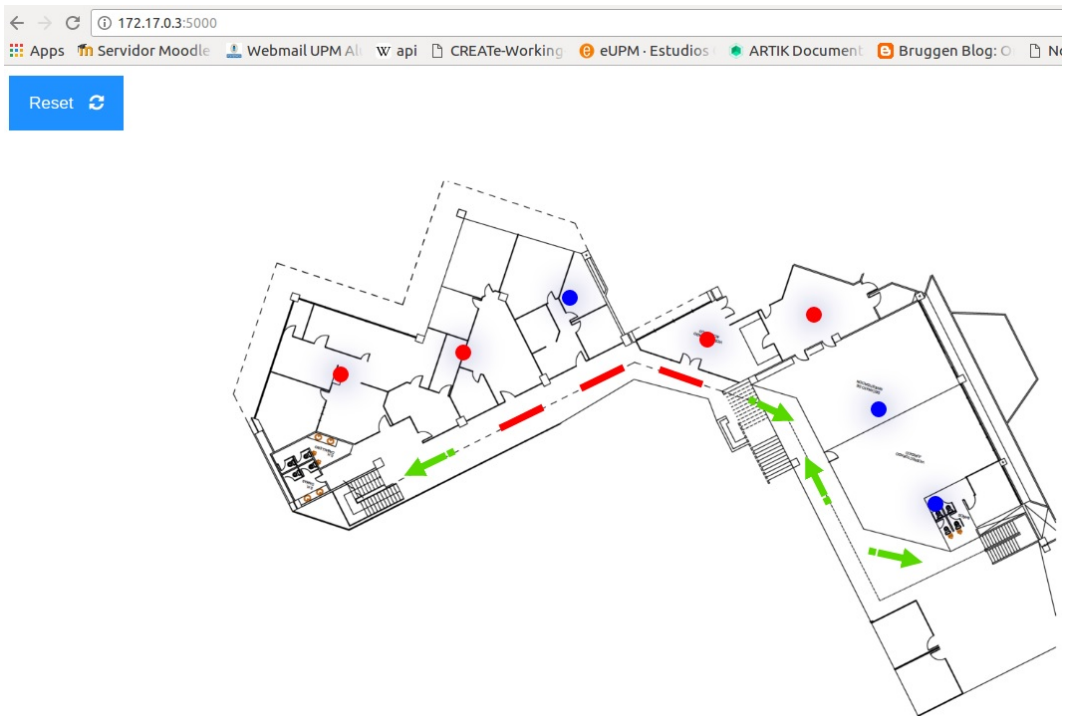


Figura 69. Señalización cuando se alerta el nodo sensor S1 mientras los nodos S4, S5 y S2 ya están alertados.

Y por último se muestran en la Figura 70 las rutas generadas cuando se alerta el nodo sensor "S1" en el mismo escenario de prueba anterior.

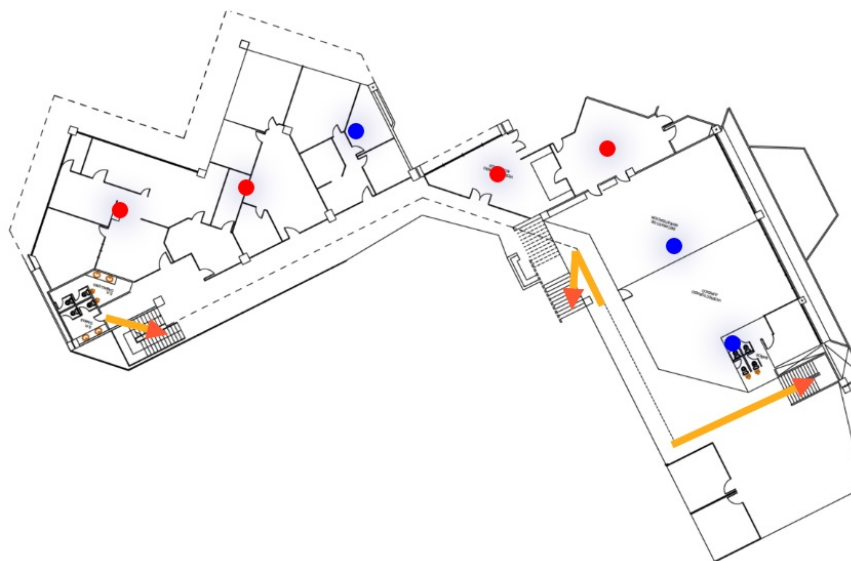


Figura 70. Rutas generadas cuando se alerta el nodo sensor S1 mientras los nodos S4, S5 y S2 ya están alertados.

## 5 Conclusiones y Trabajos Futuros

En este capítulo se detallan las conclusiones obtenidas luego de realizar todo el desarrollo e implementación del proyecto y además se exponen recomendaciones y posibles puntos en los que se pueda profundizar en trabajos futuros.

### 5.1 Conclusiones

La implementación de los sistemas de evacuación es muy importante en cualquier infraestructura para precautelar la vida y seguridad de todas las personas que se encuentran dentro del edificio, por lo cual es necesario considerar las nuevas tecnologías que surgen continuamente que permitan implementar mejoras en estos sistemas, con el fin de minimizar los tiempos y optimizar la evacuación.

Las bases de datos orientadas a grafos son una gran opción, aparte de su flexibilidad, escalabilidad y rendimiento, depende principalmente del modelo de negocio en el que se desee aplicar, ya que puede ser la mejor opción cuando se trata de aplicaciones que manejan datos sociales, recomendaciones basadas en las relaciones entre personas y cosas, cálculo de ruta entre localidades, como el caso de este trabajo, y muchas cosas más, pero es necesario analizar primero el tipo de aplicación para determinar si es óptimo su uso, ya que pueden presentarse casos en los que debido a lo que se desea modelar sería recomendable el uso de otro tipo de estructuras como es el caso de bases de datos relaciones.

Para realizar el modelado de los datos en bases de datos orientadas a grafos como Neo4j, es importante analizar cual es la mejor manera de representar los datos mediante nodos y relaciones para poder manipularlos eficientemente y obtener los resultados que se desean, aprovechando todas las ventajas que proporciona el utilizar este tipo de base de datos. Como en este caso donde se determina que la forma más sencilla de trabajar es con dos grafos separados, definiendo de alguna manera una asociación para poder referenciarlos.

Neo4j proporciona funcionalidades estándar que facilitan el manejo de los datos como en este caso para realizar el cálculo de rutas entre dos nodos utilizando el lenguaje Cypher, el cual es un lenguaje amigable, fácil de aprender y similar a SQL, por lo que para personas familiarizadas con SQL

es más sencillo adaptarse a este lenguaje. Además existen varias librerías que permiten trabajar con Neo4j desde una aplicación desarrollada en Python sin necesidad de realizar programación de tan bajo nivel.

Para la representación o visualización de los datos almacenados en base de datos orientadas a grafos, se tiene como una de las mejores opciones la librería de JavaScript D3.js, ya que ofrece gran variedad de alternativas y formas para presentar los datos en un entorno web de manera que se puedan entender fácilmente e incluso interactuar con ellos. En este trabajo es necesario la representación gráfica para poder visualizar las rutas que se van generando a medida que los nodos sensores van detectando alguna anomalía en el ambiente y de esta forma poder simular como debe ser el funcionamiento y activación de la señalización que se coloque dentro del edificio cuando se presente algún evento de emergencia.

En base al desarrollo realizado en este proyecto se puede concluir que se adquirieron habilidades y conocimientos en nuevas tecnologías, proporcionando una alternativa de mejora a implementar en los sistemas de evacuación de edificios, con el fin de reducir los tiempos de evacuación de las personas y brindarles información más acertada en situaciones de miedo y pánico donde muchas veces la falta de conocimiento puede poner en peligro su vida.

## **5.2 Trabajos Futuros**

Con el desarrollo de este trabajo se plantea una opción para mejorar los sistemas de evacuación actuales que cuentan con señalización estática, introduciendo el uso de nuevas tecnologías y técnicas de big data para automatizar estos sistemas y valerse de información que pueda ser recopilada por sensores para prevenir o disminuir las consecuencias que pueden ocasionar eventos de emergencia dentro de un edificio. Además para poder presentar las rutas calculadas de forma dinámica es necesario contar con señalización que pueda ser activada por medio del sistema, indicando el color y dirección hacia donde debe estar dirigida para guiar a las personas hacia las salidas de evacuación mas cercanas.

En este caso se han utilizado nodos sensores, donde cada nodo cuenta con cinco sensores que miden temperatura, monóxido de carbono, dióxido de carbono, gas líquido de petróleo y nivel del líquido vertical, sin embargo en

un sistema de evacuación se pueden utilizar más sensores que proporcionen otro tipo de métricas e incluso se pueden usar cámaras de video para técnicas de detección, analizar patrones de comportamiento, entre otras cosas que permitan obtener más información para mejorar el sistema de evacuación y minimizar los riesgos a los que pueden estar expuestas las personas que se encuentran dentro de los edificios.

En este proyecto se ha trabajado sobre el plano de la primera planta de un edificio de 1692 m<sup>2</sup>, teniendo como salidas de evacuación las escaleras, sin embargo poder probar este modelo en planos más grandes o incluso en todo el edificio ayudaría a comprobar el rendimiento de las tecnologías y del diseño utilizado con un mayor volumen de datos en situaciones de emergencia.

Además de la señalización dinámica que se debe utilizar en la implementación de este sistema, es recomendable el uso de una interfaz web que permita monitorear el comportamiento de los nodos sensores y las rutas que se generan para ayudar a la evacuación.

## Bibliografía

- [1] W. Velásquez, A. Munoz-Arcentales, and J. Salvachúa, “A Distributed System Model for Managing Data Ingestion in a Wireless Sensor Network,” 2016.
- [2] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment,” *Linux Journal*, vol. 2014, no. 2, 2014.
- [3] Docker Inc. (2017) Docker overview. [Online]. Available: <https://docs.docker.com/engine/docker-overview/>
- [4] Esaú A. (2014) Docker, Qué es y sus principales características. [Online]. Available: <https://openwebinars.net/blog/docker-que-es-sus-principales-caracteristicas/>
- [5] Docker Inc. (2017) What is a container. [Online]. Available: <https://www.docker.com/what-container>
- [6] R. McKendrick, P. Raj, J. S. Chelladhurai, and V. Singh, “Launching applications using docker,” in *Docker Bootcamp*. Packt Publishing, Apr. 2017, ch. 2.
- [7] N. Garg, *Learning Apache Kafka - Second Edition*. Packt Publishing Ltd., Feb. 2015.
- [8] R. Estrada, “The Broker - Apache Kafka,” in *Fast Data Processing Systems with SMACK Stack*. Packt Publishing Ltd., Dec. 2016, ch. 5.
- [9] (2016) Apache Kafka - Documentation. [Online]. Available: <https://kafka.apache.org/documentation/>
- [10] (2016) ZooKeeper. [Online]. Available: <https://zookeeper.apache.org/doc/trunk/zookeeperOver.html>
- [11] Python Software Foundation. (2017) The Python Tutorial. [Online]. Available: <https://docs.python.org/3/tutorial/index.html>
- [12] M. Venkitachalam, *Python Playground*. No Starch Press, Oct. 2015.
- [13] Neo Technology, Inc. (2017) Neo4j. [Online]. Available: <https://neo4j.com/>

- [14] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases, 2nd Edition*. O'Reilly Media, Inc., Jun. 2015.
- [15] M. Lal, *Neo4j Graph Data Modeling*. Packt Publishing, Jul. 2015.
- [16] S. Raj, *Neo4j High Performance*. Packt Publishing, Mar. 2015.
- [17] R. V. Bruggen, *Learning Neo4j*. Packt Publishing, Aug. 2014.
- [18] J. Chao. (2016, Aug.) Graph Databases for Beginners Graph Search Algorithm Basics. [Online]. Available: <https://neo4j.com/blog/graph-search-algorithm-basics/>
- [19] A. D. Angelis. (2016) GraphGist: Santa's shortest weighted path. [Online]. Available: <https://neo4j.com/graphgist/92064364-ae62-405b-bfa7-a46e5e85e22b>
- [20] J. Reid, "Welcome to html5," in *HTML5 Programmer's Reference*. Apress, Jul. 2015, ch. 1.
- [21] W3C. (2014) HTML5. [Online]. Available: <https://www.w3.org/TR/html5/introduction.html#a-quick-introduction-to-html>
- [22] M. Fisher, "Why html5?" in *HTML5 for Flash Developers*. Packt Publishing, Jul. 2013, ch. 1.
- [23] M. Bostock. (2017) Data-Driven Documents. [Online]. Available: <https://d3js.org/>
- [24] M. Dewar, *Getting Started with D3*. O'Reilly Media, Inc., Jul. 2012.
- [25] F. Nelli, "Working with d3," in *Create Web Charts with D3*. Apress, Dec. 2014, ch. 2.
- [26] C. Körner, "The magic of svg, d3.js, and angularjs," in *Data Visualization with D3 and AngularJS*. Packt Publishing, Apr. 2015, ch. 1.
- [27] W. Velásquez, A. Munoz-Arcentales, and W. Yáñez-Pazmino, "Proposal of a Communication Structure Model for Activating Reactive Signaling in an Emergency Evacuation Systems," 2016.
- [28] Python Software Foundation. (2017) kafka-python. [Online]. Available: <https://pypi.python.org/pypi/kafka-python>
- [29] N. Small. (2016) The Py2neo v3 Handbook. [Online]. Available: <http://py2neo.org/v3/>



- [30] Python Software Foundation. (2017) Web Frameworks for Python. [Online]. Available: <https://wiki.python.org/moin/WebFrameworks>
- [31] Armin Ronacher. (2017) Flask. [Online]. Available: <http://flask.pocoo.org/>
- [32] Miguel Grinberg. (2017) Flask-SocketIO. [Online]. Available: <https://flask-socketio.readthedocs.io/en/latest/>
- [33] ——. (2014) Easy WebSockets with Flask and Gevent. [Online]. Available: <https://blog.miguelgrinberg.com/post/easy-websockets-with-flask-and-gevent>

# Anexos

## Anexo A

Código fuente de la parte principal del programa.

```
1 #!/usr/bin/env python
2 from flask import Flask, render_template, session, request
3 from flask_socketio import SocketIO, emit, disconnect
4 from kafka import KafkaConsumer
5 import json
6 import time
7 import os
8 import graph_db
9
10 async_mode = None
11
12 app = Flask(__name__)
13 app.config['SECRET_KEY'] = 'secret!'
14 socketio = SocketIO(app, async_mode=async_mode)
15 thread = None
16
17 # Consumidor de Kafka
18 def background_thread():
19     consumerKafka= KafkaConsumer("AL", bootstrap_servers='
20     192.168.0.155:9092 ')
21     for msg in consumerKafka:
22         value= eval(msg.value.decode("utf-8"))
23         response= graph_db.getPath(value[0],
24             "S" + str(int(value[1][0])),
25             str(int(value[1][1])))
26     if(response):
27         print(response["graph1"])
28         socketio.emit('my_response',
29             {"data":response["graph1"], "graph":"graph1"},
30             namespace='/tfm')
31         socketio.emit('my_response',
32             {"data":response["graph2"], "graph":"graph2"},
33             namespace='/tfm')
34
35 @app.route('/')
36 def index():
37     return render_template('index.html', async_mode=socketio.
38         async_mode)
39
40 @socketio.on('disconnect_request', namespace='/tfm')
41 def disconnect_request():
```

```

    disconnect()
41
# Se inicia el metodo en segundo plano ,
43 # que es el que va a estar consumiendo de Kafka
@socketio.on('connect', namespace='/tfm')
45 def test_connect():
    global thread
47     if thread is None:
        thread = socketio.start_background_task(target=
            background_thread)
49
@socketio.on('disconnect', namespace='/tfm')
51 def test_disconnect():
    print('Client disconnected', request.sid)
53
# Reiniciar parametros de base de datos
55 @socketio.on('reset', namespace='/tfm')
def reset_request():
57     response= graph_db.reset()
    emit('my_response', {"data":response["graph1"], "graph": "graph1"
        }, namespace='/tfm')
59     emit('my_response', {"data":response["graph2"], "graph": "graph2"
        }, namespace='/tfm')
61
if __name__ == '__main__':
63     socketio.run(app, host="0.0.0.0")

```

app.py

## Anexo B

Código fuente de la parte que interactúa con la base de datos Neo4j.

```
1 # coding: utf-8
3 from py2neo import Graph, NodeSelector, Path
4 import json
5 import os
7 # Conexion base de datos Neo4j
8 graph = Graph("http://neo4j:admin@172.17.0.2:7474/db/data/")
9 selector = NodeSelector(graph)
11 # Reiniciar los datos de la base
12 def reset():
13     graph.run("MATCH (n) WHERE n.alerted = 1 SET n.alerted = NULL")
14     graph.run("MATCH (n) WHERE n.inactive = 1 SET n.inactive = NULL")
15
16     MYDIR = os.path.dirname(__file__)
17     with open(os.path.join(MYDIR, 'static/data.json')) as datafile:
18         data = json.load(datafile)
19         datafile.close()
21     with open(os.path.join(MYDIR, 'static/data1.json')) as datafile1:
22         :
23         data1 = json.load(datafile1)
24         datafile1.close()
25     return {"graph1": json.dumps(data), "graph2": json.dumps(data1)}
27 # Consultar las rutas en la base de datos
28 def getPath(alarm, sensorNode, sensor):
29     sensors= selector.select("Sensor")
30     sensors= list(sensors)
31     result, result1= [], []
32     color= ["red", "blue", "green", "yellow", "orange", "brown", "magenta"]
33     c= 0
34     updateRes= {}
35     for s in sensors:
36         if s["name"] == sensorNode:
37             alertedNode= s
38             break
39
40     updateGraph= False
41     if(not alertedNode["alerted"]):
42         updateGraph= True
```

```

43     alertedNode["alerted"]= 1
44     graph.push(alertedNode)
45
46     if(updateGraph):
47         listTmp= list(graph.match(start_node=alertedNode , rel_type="
NEXT"))
48         if(len(listTmp) > 0):
49             nodoTmp= listTmp[0].end_node()
50             if(nodoTmp["alerted"]):
51                 inactiveNodes= selector.select(nodoTmp["name"])
52                 for n in inactiveNodes:
53                     n["inactive"]= 1
54                     graph.push(n)
55         listTmp= list(graph.match(end_node=alertedNode , rel_type="NEXT
"))
56         if(len(listTmp) > 0):
57             nodoTmp= listTmp[0].start_node()
58             if(nodoTmp["alerted"]):
59                 inactiveNodes= selector.select(nodoTmp["name"])
60                 for n in inactiveNodes:
61                     n["inactive"]= 1
62                     graph.push(n)
63     for sensor in sensors:
64         # Sentencia sin algoritmo
65         #query= """MATCH (startNode:%s),
66             #             (endNode:Stairs),
67             #             p=(startNode)-[:NEXT*]-(endNode)
68             #             WHERE ALL(n IN nodes(p)
69             #             WHERE n.inactive IS NULL OR n.inactive = 0)
70             #             RETURN p AS shortestPath ,
71             #             REDUCE(distance = 0, r IN relationships(p) |
72             #             distance + COALESCE(r.distance, 0)) AS distance
73             #             ORDER BY distance ASC
74             #             LIMIT 1""" %(sensor["name"])
75         # Sentencia utilizando algoritmo Dijkstra
76         query= """MATCH (startNode:%s),
77             (endNode:Stairs)
78             CALL apoc.algo.dijkstra(startNode, endNode,
'NEXT', 'distance')
79             YIELD path, weight
80             WITH path, weight
81             WHERE ALL(n IN nodes(path)
82             WHERE n.inactive IS NULL OR n.inactive = 0)
83             AND weight > 0
84             RETURN path AS shortestPath, weight AS
distance
85             ORDER BY distance ASC
86             LIMIT 1""" %(sensor["name"])
87         cursor= graph.run(query)

```

```

o= {}
89   for record in cursor: #solo un elemento
        path= Path(record["shortestPath"])
91         if(len(list(graph.match(start_node=path.nodes()[0],
end_node=path.nodes()[1]))) > 0):
            direction= "right"
93             if (len(list(graph.match(start_node=path.nodes()[1],
end_node=path.nodes()[0]))) > 0):
                direction= "left"
95                 o["sensor"]= sensor["name"]
                o["direction"]= direction
97                 o["distance"]= record["distance"]
                i= 0
99                 nodesTmp= path.nodes()
                    while i < len(nodesTmp)-1:
101                     o1= {"source":nodesTmp[i]["name"], "target":nodesTmp[i
+1]["name"], "color":color[c]}
                            if(i == len(nodesTmp)-2):
103                             o1["final"]= 1
                                result1.append(o1)
105                             i= i+1
                o["signal"]= list(graph.match(start_node=sensor, rel_type="
CONNECTED_TO"))[0].end_node()["name"]
107                 result.append(o)
                c= c+1
109
for r in result:
111     if("direction" in r):
        if(r["direction"] == "right"):
113             r["source"]= r["signal"]
                r["target"]= r["signal"] + "1"
115             else:
                r["source"]= r["signal"] + "1"
117             r["target"]= r["signal"]
        else:
119             r["source"]= r["signal"]
                r["target"]= r["signal"] + "1"
121
MYDIR = os.path.dirname(__file__)
123 with open(os.path.join(MYDIR, 'static/data.json')) as datafile
:
    data = json.load(datafile)
125    datafile.close()

with open(os.path.join(MYDIR, 'static/data1.json')) as
datafile1:
127    data1 = json.load(datafile1)
129    datafile1.close()

```

```
131     for s in sensors:
132         if(s["alerted"]):
133             for d in data["nodes"]:
134                 if(d["id"] == s["name"]):
135                     d["color"] = "red"
136             for d in data1["nodes"]:
137                 if(d["id"] == s["name"]):
138                     d["color"] = "red"
139
140     data["links"] = result
141
142     data1["links"] = result1
143
144     updateRes = {"graph1": json.dumps(data), "graph2": json.dumps(
145         data1)}
146
147     return updateRes
```

graph\_db.py

## Anexo C

Código fuente de la página web.

```
2 <!DOCTYPE html>
3 <meta charset="utf-8">
4 <link rel="stylesheet" href="https://www.w3schools.com/w3css/4/w3.
  css">
5 <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/
  libs/font-awesome/4.7.0/css/font-awesome.min.css">
6 <style>
7
8 svg{
9   display: block;
10  margin: auto;
11  background: url(static/images/tfm.png) center;
12  background-size: 100%;
13  background-repeat: no-repeat;
14 }
15
16 .links line {
17   stroke: #57D900;
18   z-index: 100;
19 }
20
21 marker{
22   fill: #57D900;
23 }
24
25 .flowline {
26   stroke-dasharray: 35, 2;
27   animation: flow 2s linear infinite;
28   -webkit-animation: flow 2s linear infinite;
29 }
30
31 @keyframes flow {
32   from {
33     stroke-dashoffset: 80;
34   }
35
36   to {
37     stroke-dashoffset: 0;
38   }
39 }
40
41 @-webkit-keyframes flow {
42   from {
43     stroke-dashoffset: 80;
44   }
45 }
```



```

44     to {
46         stroke-dashoffset: 0;
48     }
50 filter#shadow>:nth-child(2){
52     flood-color: navy;
54 }
54 .blink {
56     -webkit-animation: blink 2s linear infinite;
58     -moz-animation: blink 2s linear infinite;
60     -ms-animation: blink 2s linear infinite;
62     -o-animation: blink 2s linear infinite;
64     animation: blink 2s linear infinite;
66 }
68 @keyframes blink {
70     from {
72         flood-opacity: 1;
74     }
76     to {
78         flood-opacity: 0;
80     }
82 }
84 @-webkit-keyframes blink {
86     from {
88         flood-opacity: 1;
90     }
92     to {
94         flood-opacity: 0;
96     }
98 }
100 button{
102     padding: 15px 20px;
104     font-size: 17px;
106     color: white;
108     border: none;
110     cursor: pointer;
112     background: #1E90FF;
114     font-family: sans-serif;
116     -webkit-transition-duration: 0.2s; /* Safari */
118     transition-duration: 0.2s;
120 }

```

```

94 button:hover {
    background: #006EEE;
96    box-shadow: 1px 2px 5px 0 rgba(0,0,0,0.24), 1px 3px 7px 0 rgba
(0,0,0,0.19);
}
98
button:active {
100    background-color: #006EEE;
    box-shadow: inset 2px 2px 7px #777;
102    transform: translateY(4px);
}
104
</style>
106 <!-- Boton para reiniciar -->
<div style="width:100%; padding:10px;">
108     <form id="reset" method="POST" action="#">
        <button type="submit">Reset<i class="w3-margin-left fa fa-
refresh" </i></button>
110     </form>
</div>
112 <!-- Elementos donde se generan las rutas -->
<svg id="graph1" width="960" height="600"></svg>
114 <svg id="graph2" width="960" height="600"></svg>
116 <script type="text/javascript" src="//code.jquery.com/jquery
-1.4.2.min.js"></script>
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/
libs/socket.io/1.3.5/socket.io.min.js"></script>
118 <script src="https://d3js.org/d3.v4.min.js"></script>
<script>
120
$(document).ready(function() {
122     $.ajaxSetup({ cache: false });
    namespace = '/tfm';
124     var socket = io.connect(location.protocol + '//' + document.
domain + ':' + location.port + namespace);
126
    socket.on('connect', function() {
        //alert("connected");
128    });
130 //Actualizar las rutas
    socket.on('my_response', function(msg) {
132        //alert(msg.graph);
        d3.selectAll("svg#" + msg.graph + " > *").remove();
134        updateData(JSON.parse(msg.data), msg.graph);
    });
136

```

```

138 //Evento del boton para reiniciar
$( 'form#reset' ).submit(function(event) {
140     socket.emit('reset');
    return false;
142 });
$.getJSON("static/data.json", function( graph ){
144     updateData(graph, "graph1");
    updateData(graph, "graph2");
146 });
148 });
150 function updateData(graph, graphId){
    var svg = d3.select("svg#" + graphId),
152     width = +svg.attr("width"),
    height = +svg.attr("height");
154
    var color = d3.scaleOrdinal(d3.schemeCategory20);
156
    var simulation = d3.forceSimulation()
158     .force("link", d3.forceLink().id(function(d) { return d.id;
    }))
    .force("charge", d3.forceManyBody())
160     .force("center", d3.forceCenter(width / 2, height / 2));
162
    //Construir flecha
    svg.append("svg:defs").selectAll("marker")
164     .data(["end"])
    .enter().append("svg:marker")
166     .attr("id", "end")
    .attr("viewBox", "0 -5 10 10")
168     .attr("refX", 6)
    .attr("markerWidth", 3)
170     .attr("markerHeight", 3)
    .attr("orient", "auto")
172     .append("svg:path")
    .attr("d", "M0,-5L10,0L0,5");
174
    //Construir enlaces
176 var link = svg.append("g")
    .attr("class", "links")
178     .selectAll("line")
    .data(graph.links)
180     .enter().append("line")
    .attr("stroke-width", 7)
182     .each(function(d){
    if("direction" in d){
184         d3.select(this).attr("marker-end", "url(#end)");
    }
    });

```

```

186         .attr("class", "flowline");
187     }else if("final" in d){
188         d3.select(this).attr("marker-end", "url(#end)");
189     }else{
190         d3.select(this).style("stroke", "red");
191     }
192
193     if(d.color){
194         d3.select(this).style("stroke", "#ffae1a"/*d.color*/);
195         svg.selectAll("marker").style("fill", "#FF5733");
196     }
197 });
198
199 // Construir nodos
200 var node = svg.append("g")
201     .attr("class", "nodes")
202     .selectAll("circle")
203     .data(graph.nodes)
204     .enter().append("circle")
205     .attr("r", function(d) { return (d.width ? d.width : 1)
206 ;})
207     .attr("fill", function(d) { return (d.width ? d.color? d
208 .color : "blue" : "transparent");})
209     .each(function(d){
210         if(d.width){
211             d3.select(this).style("filter", "url(#shadow)");
212         }
213     });
214
215 var filter = svg.select("defs").append("svg:filter")
216     .attr("id", "shadow")
217     .attr("x", "0")
218     .attr("y", "0")
219     .attr('filterUnits', "userSpaceOnUse")
220     .attr('width', '300%')
221     .attr('height', '300%');
222
223 filter.append("svg:feOffset")
224     .attr("dx", 0)
225     .attr("dy", 0)
226     .attr("result", "offOut");
227
228 filter.append("svg:feFlood")
229     .attr("class", "blink");
230
231 filter.append("svg:feComposite")
232     .attr("operator", "in")
233     .attr("in2", "offOut");

```

```

232 filter.append("svg:feGaussianBlur")
      .attr("stdDeviation", "15")
234      .attr("result", "offsetblur");

236 var feMerge = filter.append("svg:feMerge");
feMerge.append("svg:feMergeNode");
238 feMerge.append("svg:feMergeNode")
      .attr("in", "SourceGraphic");

240
node.append("title")
242     .text(function(d) { return d.id; });

244 simulation
      .nodes(graph.nodes)
246      .on("tick", ticked);

248 simulation.force("link")
      .links(graph.links);

250
function ticked() {
252   link
      .attr("x1", function(d) { return d.source.x; })
254      .attr("y1", function(d) { return d.source.y; })
      .attr("x2", function(d) { return d.target.x; })
256      .attr("y2", function(d) { return d.target.y; });

258   node
      .attr("cx", function(d) { return d.x; })
260      .attr("cy", function(d) { return d.y; });
}
262
264 </script>

```

index.html

## Anexo D

Código del Dockerfile para construir la imagen del contenedor donde se ejecutará la aplicación (consumidor de Python y página web).

```
FROM python:2.7
2
RUN apt-get update -y
4
COPY TFM /app
6
RUN pip install flask-socketio
8
RUN pip install -r /app/requirements.txt
RUN pip install kafka-python
10
RUN pip install py2neo

12
WORKDIR /app

14
EXPOSE 5000

16
ENTRYPOINT ["python"]
CMD ["app.py"]
```

## Anexo E

Código del Dockerfile para construir la imagen del contenedor donde se ejecutará la base de datos Neo4j.

```
1 FROM neo4j:3.2
3 COPY plugin /plugins
5 WORKDIR /var/lib/neo4j
7 EXPOSE 7474 7473 7687
9 CMD [ "neo4j" ]
```