Simulación en la nube de nodos medidores de radiación solar desplegados en un contexto IoT
Cloud simulation of solar radiation measuring nodes deployed in an IoT context

Trabajo de Fin de Máster
Curso 2020–2021

Autor
Ignacio Regueiro Tuñón

Director
José Luis Risco Martín
Katzalin Olcoz Herrero

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Simulación en la nube de nodos medidores de radiación solar desplegados en un contexto IoT
Cloud simulation of solar radiation measuring nodes deployed in an IoT context

Trabajo de Fin de Máster en Ingeniería Informática
Departamento de Arquitectura de Computadores y Automática

Autor
Ignacio Regueiro Tuñón

Director
José Luis Risco Martín
Katzalin Olcoz Herrero

Convocatoria: *Septiembre 2021*
Calificación: *9*

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

7 de septiembre de 2021

Dedicatoria

A mis padres y hermana por su apoyo incondicional

Agradecimientos

En primer lugar agradecer a los directores, José Luis Risco y Katzalin Olcoz, por su ayuda durante todas las fases del trabajo para alcanzar el mejor resultado posible.

Por último, mencionar a Ernesto Valero, quien me ha ayudado indirectamente a abordar con otra mentalidad proyectos de gran envergadura y mejorar a nivel técnico en la parte de Ansible.

Resumen

Simulación en la nube de nodos medidores de radiación solar desplegados en un contexto IoT

El despliegue de sensores siguiendo una arquitectura IoT (del inglés, *Internet of Things*) permite procesar gran cantidad de datos de forma ordenada. Estas mediciones son generalmente analizadas en las capas *edge* y *fog* y almacenadas en la capa *cloud* para contar con un histórico que permita ver cómo han fluctuado en los últimos años los valores para estudios posteriores. Un ejemplo es el estudio de la radiación solar, donde se despliegan multitud de sensores para analizar la evolución de la radiación en las distintas estaciones del año. No obstante, no existen herramientas para analizar mediante métodos de simulación el despliegue de estos sensores en un contexto IoT.

En este TFM se propone realizar un sistema simulado dentro de un contexto IoT en el que existen elementos que generan mediciones de radiación, que son transmitidas a través de eventos a la capa *fog* de procesado. Esta se encarga de detectar valores atípicos, que son reemplazados por valores correctos utilizando distintos métodos de regresión. A su vez se integran técnicas de kriging para calcular la radiación en otros puntos donde no existen mediciones.

Para que la simulación sea escalable, se propone un despliegue de la simulación en la nube pública (Cloud). Toda la configuración necesaria para ejecutarla en Cloud se ha planteado utilizando la filosofía *DevOps* de infraestructura como código. Se introduce con esto la automatización del despliegue y puesta a punto de todos los pasos que conlleva la simulación. Se finaliza con un ejemplo de cómo el framework desarrollado permite realizar simulaciones de manera automática.

Palabras clave

Simulación, IoT, radiación solar, métodos de regresión, kriging, outliers, DEVS, Cloud, infraestructura como código (IaC), Terraform, Ansible

Abstract

Cloud simulation of solar radiation measuring nodes deployed in an IoT context

The deployment of sensors following an IoT (*Internet of Things*) architecture allows us to process large amounts of data in an orderly manner. These measurements are generally analyzed in the *edge* and *fog* layers and stored in the *cloud* layer to have a history of how the values have fluctuated in the past few years for subsequent studies. An example is the study of solar radiation, where a multitude of sensors are deployed to analyze the evolution of radiation in the different seasons of the year. However, there is no analysis of the deployment of these sensors in an IoT context.

It is proposed to implement a simulated system within an IoT context where there are elements that generate radiation measurements, which are transmitted through events to the processing layer. This is responsible for detecting outliers, which are replaced by correct values using different regression methods. At the same time, kriging techniques are integrated to calculate radiation from other points where no measurements exist.

To ensure that the simulation is distributed and scalable in terms of dimensions, it is taken to a public cloud provider. All the configuration required to run it in the Cloud has been approached using the *DevOps* philosophy of infrastructure as code. It is introduced with this the automation of the deployment and configuration of all the steps involved in the simulation.

Key words

IoT simulation, solar radiation, regression methods, kriging, outliers, DEVS, Cloud, infrastructure as code (IaC), Terraform, Ansible

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este capítulo expone en primer lugar la motivación que ha dado lugar a este trabajo de fin de máster, continuando con los objetivos que se pretenden cubrir y con el plan de trabajo a seguir para cumplir con las metas marcadas. Por último, se añade un breve resumen de los distintos capítulos que se desarrollan en esta memoria.

1.1. Motivación

Nos encontramos en un mundo globalizado donde la generación y el intercambio de información es extremadamente elevado y además ocurre en un breve instante de tiempo. Implementar algún mecanismo capaz de recoger todos los datos que se generan en tiempo real para procesarlos y explotarlos posteriormente es un gran desafío, debido al gran número de elementos que componen el sistema, lo que hace que sea muy complejo.

Internet de las Cosas propone conectar cualquier objeto a la red de Internet, siguiendo un patrón bien definido compuesto por tres niveles interconectados: edge, fog y cloud (Kumar et al., 2019). Estos objetos generan medidas que pueden ser usadas como métricas de interés. Hasta hace poco tiempo, era irrelevante conocerlas ya que no aportaban mucho conocimiento al dominio del problema. A día de hoy, de todos esos datos se puede extraer algún patrón o agrupación que se convierta en un caso de estudio que mejore nuestra vida o que monitorice el entorno natural. Esto está siendo posible con los desarrollos actuales que se realizan en los campos de *Big Data* e *Inteligencia Artificial* (Elgendy y Elragal, 2014), que requieren datos masivos.

Por ejemplo, contar con múltiples medidas de distintos puntos de la tierra tales como humedad, temperatura, presión atmosférica o radiación solar nos permite conocer el estado de todo el globo terráqueo y contar con un histórico en el que ver la evolución durante las distintas estaciones y años anteriores. Un caso de observación actual consiste en estudiar el cambio climático y cómo están variando todas estas medidas en los últimos años. Tomar medidas no solo permite realizar acciones reactivas, sino también elaborar modelos predictivos y tomar acciones más proactivas. Esto es bastante ambicioso debido a la gran cantidad de información y a las distintas áreas de conocimiento involucradas que hacen que sea extremadamente complejo elaborar dichos modelos.

Este trabajo se centra en la parte de despliegue, generación y procesado de información de valores de radiación solar. Existen distintos mecanismos para la toma de mediciones. El más extendido es mediante pequeños dispositivos que contienen sensores que generan valores que son enviados a medios de almacenamiento (Joint Research Centre, 2020).

Proponemos un método, basado en modelado y simulación, que nos permite analizar un despliegue de nodos sensores de radiación solar siguiendo el paradigma de Internet de las Cosas. Con ello diseñamos un entorno que nos permite simular un despliegue de sensores generadores de datos. Se tratan estos datos, teniendo en cuenta que los dispositivos eléctricos en ocasiones introducen errores que deben ser eliminados. También se proponen distintos mecanismos de interpolación y regresión que otorguen mayor versatilidad al sistema a la hora de calcular nuevas medidas para reemplazar valores erróneos u obtener medidas de localizaciones en las que no se cuenta con un sensor.

Adicionalmente, este entorno de simulación debe ser altamente escalable en función del número de elementos conectados y la información que circula en el sistema. Una solución para garantizar la escalabilidad pasa por la computación en la nube, ya que ofrece estas características.

Se plantea el desarrollo de un sistema de simulación guiado por eventos que está dentro de un contexto IoT para medir la radiación solar, el cual es fácilmente escalable y se despliega en Cloud.

1.2. Objetivos

Para llevar a cabo el trabajo se ha fijado el principal objetivo que es diseñar un entorno de simulación IoT que contenga las tres capas edge, fog y cloud, donde la capa edge genere mediciones de radiación solar, la capa fog procese y transforme los datos de la capa anterior y en cloud se almacene toda la información generada por las capas anteriores.

Junto con el objetivo principal del trabajo, se incluyen otras metas igual de necesarias ya que complementan el entorno de simulación. Estas son:

- Capacitar al sistema para identificar valores atípicos (*outliers*) que son introducidos por los generadores de mediciones de radiación.
- Implementar un mecanismo de reemplazo para valores atípicos.
- Demostrar la viabilidad de obtener valores de radiación de un punto en el espacio en el que no se tienen mediciones, a partir de métricas de otros puntos cercanos, con algoritmos de Kriging.
- Dotar de escalabilidad a la simulación para que sea rápido y sencillo cambiar entre distintos escenarios.
- Llevar la simulación a un entorno de computación en la nube, donde toda configuración siga la práctica emergente denominada “infraestructura como código”.

1.3. Plan de trabajo

Para la consecución de los objetivos descritos en el apartado anterior se ha definido un plan de trabajo, que ha constado de las siguientes fases:

- Estudio preliminar de los elementos que deben estar presentes en el escenario de simulación en cada una de las capas edge, fog y cloud. Creación del mismo empleando una implementación del formalismo DEVS.
- Análisis e implementación de algoritmos para detectar valores atípicos y generación de valores de reemplazo para estos con métodos de regresión.

- Incorporación de algún algoritmo de Kriging, para ver la viabilidad de interpolar distintos valores.
- Desarrollo del código necesario que lleve la simulación a Cloud así como para automatizar la creación de objetos en la nube, su configuración y destrucción cuando la simulación finalice.
- Redacción de la memoria en la que se plasme el trabajo realizado, con los resultados y conclusiones extraídas.

1.4. Estructura de la memoria

El presente documento consta de seis capítulos. Su contenido a grandes rasgos es el siguiente:

- El **capítulo 1** se corresponde con la introducción al tema del trabajo junto con los objetivos marcados y el plan de trabajo definido para abordarlo.
- El **capítulo 2** incluye el análisis del estado del arte junto con la mayor parte de las bibliografías consultadas. Se compararan arquitecturas IoT actuales con la definida en este trabajo.
- El **capítulo 3** incorpora el diseño del sistema a desarrollar dentro de un contexto IoT siguiendo el formalismo DEVS. Por último también se incluyen los cálculos matemáticos y fórmulas necesarias para implementar los distintos algoritmos.
- El **capítulo 4** contiene el grueso del trabajo, centrado en la implementación del sistema. Se detallan todos los elementos presentes para llevarlo a cabo y las relaciones entre ellos. Por último, se cierra con el desarrollo que se ha realizado para la construcción de la arquitectura Cloud y la gestión de la configuración como código.
- El **capítulo 5** muestra los resultados obtenidos en este trabajo. Se detallan los pasos para poner en marcha la simulación. Se analizan los resultados de la simulación, incluyendo la detección de valores atípicos, sustitución de estos, y los resultados de aplicar *kriging*.
- El **capítulo 6** hace una recapitulación del trabajo. En primer lugar, se exponen las conclusiones y se valora el cumplimiento de los objetivos previamente marcados. Para finalizar se proponen varias lineas de trabajo futuro.

Capítulo 2

Estado de la Cuestión

Este capítulo recoge la motivación originaria que ha dado lugar a este proyecto. Se realiza una revisión al estado actual en materia de radiación solar. Después se propone una solución alternativa a la actual, que resuelve los problemas detectados. Dichos problemas son fácilmente abordables con la proliferación de soluciones más novedosas como son la computación en la nube y el desarrollo de una automatización de todos los procesos.

2.1. Medida de radiación solar

La radiación solar, que ya ha sido estudiada en profundidad por la comunidad científica, varía en función de diversos factores (Joint Research Centre, 2020). Cuando atraviesa la atmósfera, la radiación sufre distintos procesos de absorción o dispersión debidos a componentes atmosféricos como pueden ser el ozono, dióxido de carbono, vapor de agua o partículas en suspensión. Por tanto, los valores de radiación que se pueden tomar a nivel atmosférico difieren notablemente de los tomados a nivel del suelo. Por ello, es interesante conocer la radiación cuando existen condiciones de cielo despegado y atmósfera limpia, ya que proporciona la cantidad máxima de radiación en cualquier sitio, suelo o atmósfera.

Respecto a la radiación medida en suelo, es conocida como *radiación global*. Está compuesta de tres componentes. El primero de ellos se conoce como *radiación directa*, que es una fracción de la radiación que llega a la tierra sin que sea atenuada por la atmósfera. El segundo componente, conocido como *difuso*, corresponde con la radiación que llega al suelo después de sufrir una dispersión o atenuación por la atmósfera. El tercer y último componente, en muchas ocasiones, no es tenido en cuenta. Se trata de la radiación reflejada desde la superficie terrestre. Obviamente, debido a factores meteorológicos no siempre se puede obtener la radiación directa, por ejemplo si el cielo se encuentra nublado. En cambio la parte difusa siempre puede ser obtenida a pesar de que las nubes dificulten la visibilidad del disco solar.

La mejor forma de estimar la intensidad de la radiación solar es utilizando sensores de alta calidad en el suelo, pero para que los valores obtenidos tengan una buena precisión, es recomendable cumplir con las siguientes condiciones:

- Emplear únicamente sensores que proporcionen una medición de alta calidad.
- Tomar mediciones al menos cada hora.
- Efectuar la calibración de los sensores con regularidad.

- Realizar un mantenimiento periódico de los sensores, limpiándolos frecuentemente.
- Recoger las mediciones para almacenarlas durante un tiempo prolongado de más de 10 años. A su vez las mediciones deben estar estar publicadas.

Existen muy pocas estaciones que cumplan todos estos requisitos, lo que implica que el número de mediciones en suelo es relativamente bajo. Por ello, es más común utilizar mediciones obtenidas mediante satélites solares y estimar la radiación que llega a la superficie terrestre. Las ventajas de utilizar estos satélites es que con unos pocos se cubre una gran cantidad de espacio terrestre y tienen una vida útil de más de 30 años. En cambio, la principal desventaja es que se requieren algoritmos matemáticos de alta complejidad que hacen uso de otras variables como ozono, vapor de agua atmosférico y aerosoles, para calcular una estimación con un margen de error aceptable.

Por tanto, al existir estas dos opciones, se debe valorar el coste económico de cada una de ellas. El coste es notablemente mayor al utilizar satélites solares, sumando además el coste computacional para obtener una estimación fiable que contiene un margen de error aceptable. Por estas razones es más barato y fácil trabajar con valores de sensores de estaciones terrestres.

2.2. Detección y estimación de valores anómalos o inexistentes

Estos factores han hecho que los datos utilizados en este trabajo provengan de una de las estaciones terrestres, que se compone por un grupo de catorce de sensores. Estos sensores no son eficaces al 100 %, ya que al ser un elemento electrónico y analógico en ocasiones, introducen valores anómalos o atípicos que no se corresponden con el resto de las mediciones. A estos valores se les conoce como *outliers* y es importante detectarlos y corregirlos, para que los análisis que se realicen posteriormente de estos datos solo contengan valores correctos.

Recientemente han surgido diversos estudios (Malik et al., 2014) ligados a la minería de datos, que proponen distintos métodos para localizar los valores *outliers*. Es importante localizar estos valores erróneos, ya que, por ejemplo, en minería de datos se trata de transformar los datos en información y para ello se debe realizar un procesamiento de datos masivo en busca de cualquier patrón oculto. Este proceso fallará o perderá mucha efectividad si no se lleva a cabo previamente una búsqueda de valores anómalos.

Existen dos formar de congregar *outliers* en función del origen de estos valores. Los valores atípicos agrupados, también conocidos como *clustered outliers* son una forma de agrupación de valores atípicos (Aggarwal, 2017). La desventaja de este método es que cuando se cuenta con una masiva cantidad de datos provoca que aparezcan muchos valores atípicos. Esto requiere hacer un agrupamiento en modelos escalables con el fin de desarrollar métodos que manejen gigantescas cantidades de datos para paliar la limitación de recursos, como podrían ser la memoria o la capacidad de cómputo. Este método de cálculo se utiliza principalmente para detectar patrones de comportamiento similares y que poseen una fuerte conexión entre ellos, por ejemplo, para brotes de enfermedades o búsqueda de actividad fraudulenta.

Existe otra forma de agrupación de *outliers* conocida como valores atípicos dispersos o *scattered outliers* (Xu et al., 2019) que es opuesta a la anterior. En este caso los valores atípicos tienen una relación entre ellos muy débil, incluso nula, y aparecen a lo largo de todo el espacio de datos. Estos valores corresponden con la excepción o con errores en el

funcionamiento de, por ejemplo, un dispositivo hardware o un sistema. Tanto los valores atípicos dispersos como los agrupados son suficientemente importantes como para que sea necesario detectarlos.

Además de la detección de fallos en sensores, es muy común aplicar métodos de regresión e interpolación sobre los datos. Numerosos estudios han realizado comparaciones de los distintos métodos de interpolación sobre datos de radiacción solar (Eberly et al., 2005). En general, se hace un estudio partiendo de un conjunto de datos de radiación solar directa (DSR) utilizado como referencia. Con este conjunto se quiere obtener una estimación aplicando distintas teorías de *krigring* (Ryu et al., 2002). En dicho estudio, para calcular la DSR en lugar de utilizar sensores implementan el módulo llamado GRASS-GIS que, a partir de una fórmula física, puede calcular la radiación solar directa, difusa o reflejada en el suelo en una latitud determinada. Aplican las técnicas de *kriging simple*, ordinario, simple con medios locales y *kriging con deriva externa*. Para aplicar estas técnicas usan un subconjunto de datos, para hacer más realista el estudio, ya que en función de la meteorología no siempre se cuenta con datos en las zonas en las que se quiere recoger medidas, por ejemplo porque el tiempo puede ser adverso en esa región. Los resultados que se obtuvieron respecto a la aplicación de las técnicas mostraron que el kriging simple y ordinario no proporcionan una representación satisfactoria de la variabilidad espacial de los datos. En cambio, se obtuvo una mejor concordancia con el modelo físico obtenido al proporcionar información secundaria para aplicar el kriging simple con medios locales y kriging con derivada externa. Estos datos externos que se han proporcionado corresponden con un modelado digital del terreno con valores de pendiente y materiales de los que está compuesto.

Otro estudio similar al anterior analiza la aplicación de distintas técnicas de *kriging* sobre valores de radiación solar obtenidos en el sur de España (Alsamamra et al., 2009). En este caso se cuenta con 166 estaciones meteorológicas repartidas a lo largo de la región de Andalucía. Las técnicas empleadas son el kriging ordinario y el kriging residual. En la primera técnica, en primer lugar se calcula el semivariograma y acto seguido se realiza una evaluación de los valores de radiación solar basados en este semivariograma. Para la segunda técnica se necesitan dos fases. En la primera de ellas, los valores de radiación observados se parametrizan mediante variables explicativas, generando un modelo determinista. En la segunda fase, se aplican técnicas de regresión sobre este modelo para obtener el componente estocástico del modelo y aplicar kriging ordinario sobre éste. Los resultados de esta comparación de técnicas son que el kriging residual tiene una mejor estimación principalmente en los meses de invierno. En cambio, el kriging ordinario tiende a sobrestimar los valores de estos meses. Sin embargo, para estimaciones realizadas sobre lugares topográficos montañosos, el kriging ordinario obtiene valores con errores más altos a los del kriging residual.

Todos estos estudios aplican distintas técnicas de interpolación en base a un grupo de datos que previamente ha sido recogido por unos sensores IoT. En este trabajo, se van a emplear distintos métodos de regresión e interpolación con los cuales se obtendrán valores de radiación en puntos en los que no hay sensores o se reemplazarán valores atípicos (*outliers*) por otros obtenidos con estas técnicas.

2.3. Arquitecturas IoT actuales

Actualmente y como se observa en los artículos bibliográficos anteriores, las granjas de sensores IoT tienen una arquitectura que sigue un modelo muy similar en todas ellas, ya que se encuentran dentro de un marco bastante estandarizado.

Diversos artículos explican estos modelos actuales basados en IoT. Por ejemplo, una arquitectura empleada para el estudio de la radiación solar (Kraemer et al., 2017) está compuesta por un número determinado de sensores con mejor o peor precisión que recogen tanto valores del propio sensor (batería, voltaje) como valores de radiación solar. Los nodos ejecutan periódicamente un proceso conocido como *sensing mode*, que los activa de un letargo programado de base para el ahorro de energía.

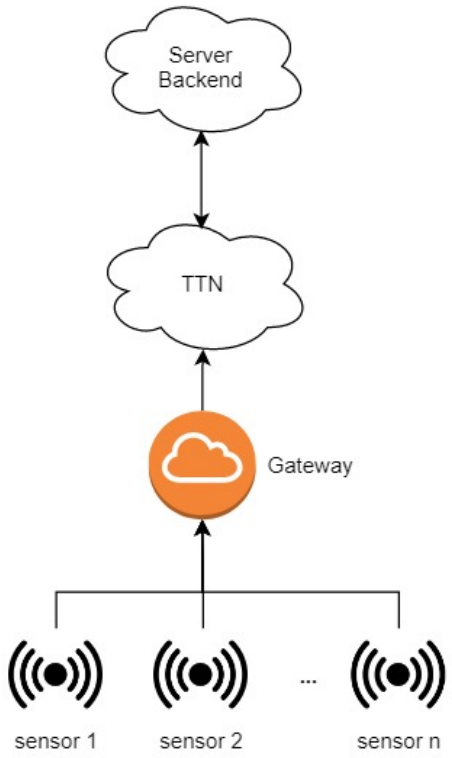

Figura 2.1: Esquema de un modelo IoT actual

El *sensing mode* permite cambiar el comportamiento deseado de los sensores para que estos estén activos un tiempo mayor o menor. En función de este comportamiento, se transmitirá un determinado número de muestras y la duración de la batería variará. A mayor cantidad de muestras, el gasto de energía será mayor; en contra posición, a menor número de muestras, la duración energética será mayor. Cuando los sensores trasmiten datos, se conectan a un *gateway*, que es el encargado de conectar cada uno de los sensores con la red TTN (*The Things Network*). Por otro lado, la red TTN, se conecta con los servidores de backend que son los encargados de recopilar la información de los sensores y generar unos ficheros en formato CSV. Este sistema se muestra en la figura 2.1.

2.4. Propuesta del trabajo

El sistema anterior es un ejemplo de cómo suelen ser todos los sistemas IoT actuales, los cuales se basan a grandes rasgos, en una capa que recoge la información y otra que la recibe para almacenar en disco, sin realizar un procesado previo de las mediciones, por ejemplo para eliminar valores anómalos o simular la generación de datos si un nodo se queda sin energía. Normalmente no se contempla ni posibilita la opción de conectar un sistema autónomo IoT a un sistema mayor que analice datos de varias granjas de sensores.

Por estos motivos, en este trabajo se quiere cubrir esta necesidad de contar con un marco IoT con mayor complejidad, que permita enlazar distintas granjas de sensores con una arquitectura por capas conectadas entre sí, de tal modo que las mediciones tomadas por las distintas granjas sean recogidas en un sistema común (esta capa se ha llamado *Data Center*). Antes de ser almacenadas en el sistema común, las mediciones tienen que ser analizadas para detectar *outliers* y/o aplicar técnicas de kriging para obtener datos de radiación de puntos intermedios a los sensores. Estos cálculos se realizan en la capa denominada *servidor fog*. Se ha empleado el kriging para calcular la radiación en otro punto en el cual no hay sensor. Ademas, se aplican distintas técnicas de regresión para reemplazar valores atípicos. Cada servidor fog corresponde con una granja de nodos IoT y el *data center* tiene tantas conexiones como *servidores fog* haya presente en la arquitectura.

Cada *servidor fog* se conecta a sus respectivos nodos IoT. Esta capa se conoce como *nodos virtuales*, que son los que se encargan de recoger las mediciones y enviarlas al *servidor fog*. Los nodos virtuales están pensados para que puedan recibir mediciones a través de tres posibles vías: mediante ficheros en formato csv generados con las mediciones, estableciendo conexión con un nodo físico mediante un socket o conectándose a una base de datos que contiene mediciones. Para este trabajo, se ha utilizado la primera vía, aunque el sistema está preparado para soportar cualquiera de las otras dos opciones.

Como estudio preliminar se ha desarrollado un marco de simulación que, en función del sistema a reproducir, requerirá una mayor o menor cantidad de recursos de cómputo. Esto obliga a que el sistema sea escalable. Por ello, se ha preparado el trabajo para que su despliegue utilice los recursos de proveedores Cloud.

Esta solución implica la virtualización de máquinas. Esto supone una ventaja, ya que el sistema no tiene que estar funcionando continuamente, sino que se ha diseñado para crearlo y destruirlo en función del uso que se desee hacer. Esto supone un ahorro de costes importante, ya que el proveedor Cloud solo facturará por el cómputo que se realice.

Estos proveedores facilitan a través de su interfaz web la creación de estos recursos. Pero esto limita la creación de máquinas virtuales a demanda de forma masiva, ya que se invertiría un tiempo en la creación y destrucción de recursos junto con la instalación de la configuración necesaria para cada máquina virtual. Por ello, se ha realizado una automatización a la hora de la creación de los recursos y su configuración utilizando la filosofía DevOps (Senapathi et al., 2019). El objetivo de esta filosofía es unificar el desarrollo del software con las operaciones del mismo. Con esto es posible integrar el desarrollo del software del simulador, con la parte de operaciones sobre un servicio de tipo IaaS (*infraestructure as a service*) como son las máquinas virtuales.

Capítulo 3

Diseño del sistema

Este capítulo se corresponde con la descripción detallada del diseño del sistema. Se comienza explicando de forma general el formalismo DEVS (del inglés, *Discrete Event System Specification*) utilizado como marco de simulación. Posteriormente se detallan las técnicas para la detección de *outliers*, interpolación con kriging y métodos de regresión. Por último se plantea el despliegue del sistema llevado a Cloud.

3.1. El formalismo DEVS

DEVS (Zeigler et al., 2000) es un formalismo genérico, universal y único que permite realizar análisis o simulaciones de sistemas que se conducen por eventos discretos. Un sistema que admita eventos como entrada a lo largo del tiempo y genere eventos de salida es un sistema DEVS. El modelo DEVS está formado por modelos atómicos (*atomic models*) que poseen un estado, procesan eventos de entrada, generan eventos de salidas y transicionan a otro estado. A su vez, los modelos atómicos se pueden enlazar entre sí y agruparse en un sistema mayor conocido como modelo acoplado (*coupled model*). Estos modelos son ejecutables en múltiples plataformas como pueden ser computadores de sobremesa o servidores de altas prestaciones. Formalmente, los componentes de un modelo atómico son los que se agrupan en la siguiente expresión:

$$A = \langle X, Y, S, \delta_e, \delta_i, \delta_c, \lambda, ta \rangle$$

- X es el conjunto de eventos de entrada.
- Y es el conjunto de eventos de salida.
- S es el conjunto de estados.
- δ_e es la función de transición externa que se ejecuta automáticamente cuando llega un evento a través de uno de los puertos de entrada. Puede cambiar el estado si es necesario.
- δ_i es la función de transición interna. Se ejecuta después de la función de salida (λ) y puede cambiar el estado $s \in S$.
- δ_c es la función de confluencia y se utiliza para decidir el siguiente estado S en caso de colisión entre las funciones δ_e y δ_i.

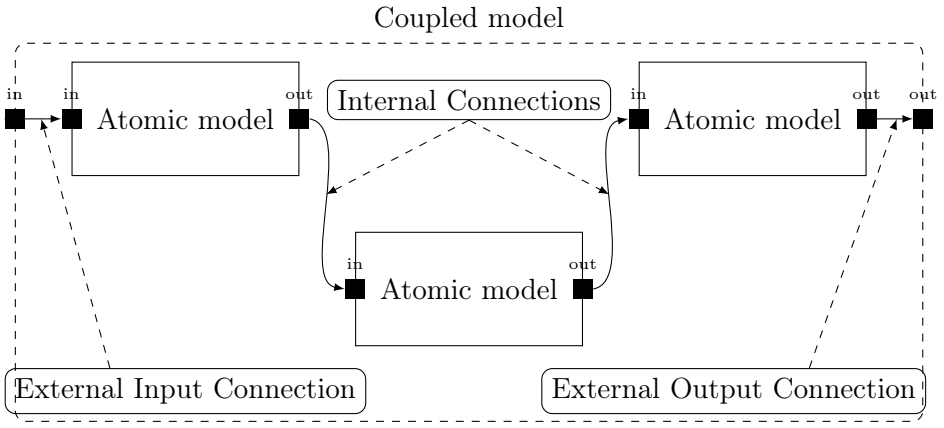

Figura 3.1: Esquema general de un modelo DEVS acoplado formado por varias unidades atómicas (Zeigler et al., 2000)

- λ es la función que genera la salida del sistema atómico.
- ta es la función que calcula el tiempo hasta la siguiente ejecución de λ.

Los componentes de un modelo acoplado son los que se agrupan en la siguiente expresión:

$$M = \langle X, Y, C, EIC, EOC, IC \rangle$$

- X es el conjunto de eventos de entrada.
- Y es el conjunto de eventos de salida.
- C es el conjunto de componentes DEVS que forman el modelo acoplado (pueden ser atómicos o acoplados).
- EIC es el conjunto de conexiones entrantes.
- EOC es el conjunto de conexiones salientes.
- IC es el conjunto de conexiones entre los componentes del modelo acoplado.

En la figura 3.1 se muestra un esquema de un modelo acoplado, el cual está compuesto por varias unidades de modelos atómicos.

Para este trabajo se ha utilizado un modelo similar a la imagen anterior, el cual está formado por varias unidades atómicas conectadas entre ellas para construir un modelo acoplado.

Para lanzar una simulación en DEVS se tiene que definir la arquitectura de simulación, que se compone de coordinadores y simuladores. Cada coordinador se asocia con un modelo acoplado y cada simulador con un modelo atómico. Esta estructura se puede observar en la figura 3.2

El coordinador raíz es el responsable de lanzar la simulación. Este se inicializa con el modelo acoplado raíz y con el tiempo que durará la simulación, que puede ser infinito, es decir, hasta que no haya más eventos que procesar.

El coordinador solicita a cada simulador un valor denominado tN que determina cuándo va a producirse el siguiente evento. Cuando este ocurre, se encarga de propagar los eventos

arrojados por λ en todos los simuladores. En este punto se produce una transición en todos los modelos. En función del instante de tiempo t y de los eventos de entrada X, se producen las siguientes transiciones en cada simulador:

- Si $t == tN$ y $X = \emptyset \Rightarrow \delta_i$
- Si $t \neq tN$ y $X \neq \emptyset \Rightarrow \delta_e$
- Si $t == tN$ y $X \neq \emptyset \Rightarrow \delta_c$

Por último, el coordinador tiene que realizar el cálculo para determinar el siguiente instante del próximo evento. Estos pasos se repiten constantemente hasta que el tiempo de ejecución de la simulación coincide con el valor de tiempo de simulación pasado como parámetro al crear el objeto del modelo acoplado.

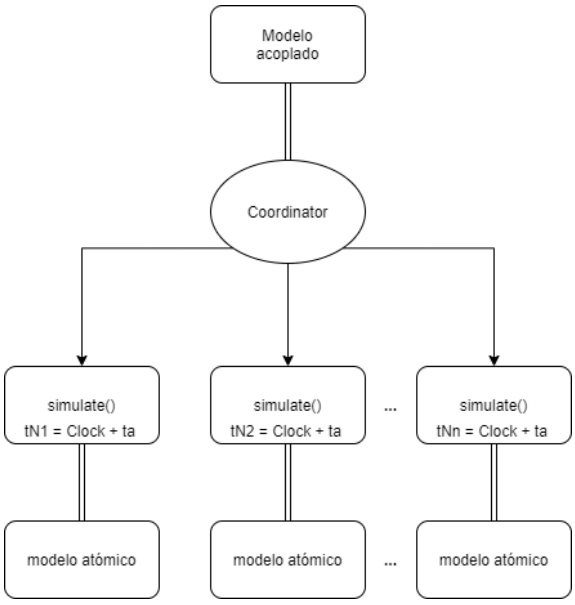

Figura 3.2: Estructura de simulación DEVS

El formalismo DEVS ha sido implementado de distintas formas y con distintos lenguajes de programación. Existen varias librerías que lo implementan, por ejemplo aDEVS (Nutaro, 2010), DEVSJAVA (ACIMS, 2009), PyDEVS (Sorge, 2015), etc. Para este trabajo se ha decidido utilizar la librería xDEVS (Risco, 2020) implementada en Java en el Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid. Este framework ofrece las clases necesarias para elaborar un sistema basado en eventos discretos.

En la figura 3.3 aparece reflejado el diagrama de clases del motor de modelado y simulación xDEVS, las cuales se invocan para crear el modelo de simulación deseado. Como se observa en la figura, la clase **coordinador** que orquesta la simulación, hereda de la clase abstracta **AbstractSimulator**. Esta última clase hace uso de **SimulatorClock** para poder crear el objeto. Por otro lado, se tienen los modelos atómicos y acoplados de DEVS, que se representan en las clases **Atomic** y **Coupled**, que ambas heredan de la clase **Component**. La clase padre contiene las variables comunes para ambas, como son los puertos de entrada y salida. A su vez, *Coupled* tiene los elementos IC, EIC, EOC para guardar las relaciones acopladas, como se ha visto anteriormente.

Por otro lado, *Atomic* cuenta con las distintas transiciones de estados solo presentes para esta clase.

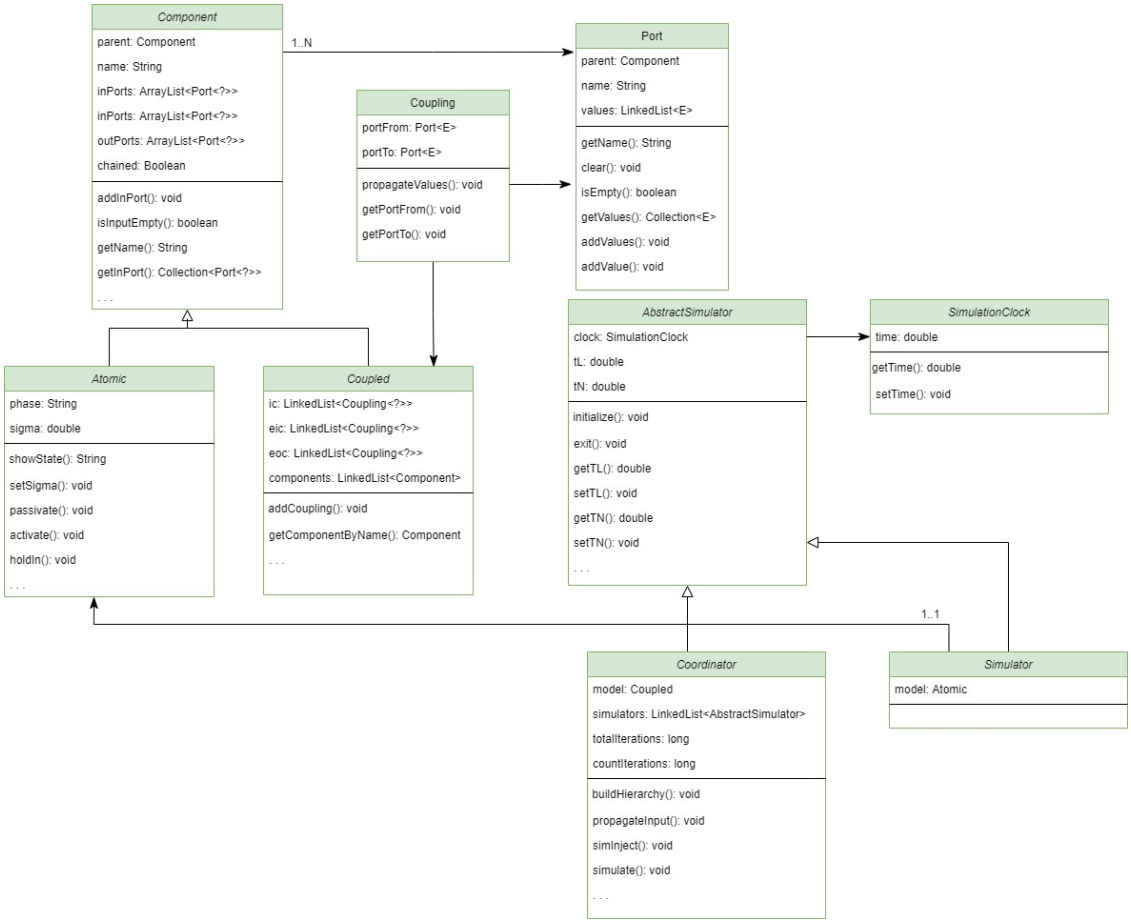

Figura 3.3: Diagrama de clases del *framework* de la librería xDEVS

Vistas las principales clases de la librería de modelado y simulación xDEVS, el desarrollo consiste en crear clases que heredan de **Atomic** para los elementos atómicos y **Coupled** para los modelos acoplados. El sistema acoplado, tiene que crear una instancia de un coordinador para que orqueste todos los elementos.

En el siguiente capítulo, se detalla cómo se ha implementado el modelo de simulación que mide la radiación solar dentro de un contexto IoT utilizando las clases de xDEVS. Este desarrollo ha implicado la implementación de un modelo acoplado que agrupa varios sistemas atómicos.

3.2. Modelo de radiación en un contexto IoT

Como se ha explicado anteriormente, para crear un modelo de simulación IoT que mide la radiación solar, se ha utilizado el formalismo DEVS. A la hora de plantear cómo abordar este punto, se ha tenido en cuenta que se quiere realizar una simulación integrando sensores de distintas *granjas*. Denominamos granja a un conjunto de sensores ubicado en una pequeña región geográfica en estudio. Teniendo en cuenta la situación actual vista en el estado del arte, las granjas existentes toman una gran cantidad de mediciones, que son guardadas y publicadas en ficheros en formato csv como se observa en la figura 3.4.

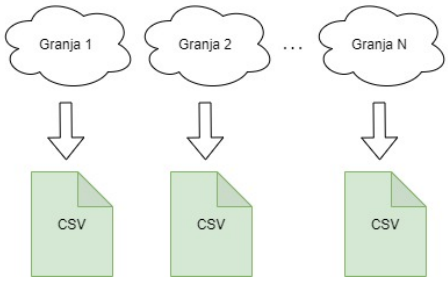

Figura 3.4: Generación de datos por los modelos actuales

En ningún momento se comunican entre ellas y no existe un intercambio de información que permita explotar estos datos a mayor escala. Por ello, la propuesta consiste en aunar un conjunto de granjas, de modo que en lugar de generar ficheros en formato csv, las mediciones sean enviadas en tiempo real a un sistema recolector, el cual manejará toda la información para posteriormente tratarla como se muestra en la figura 3.5 a alto nivel.

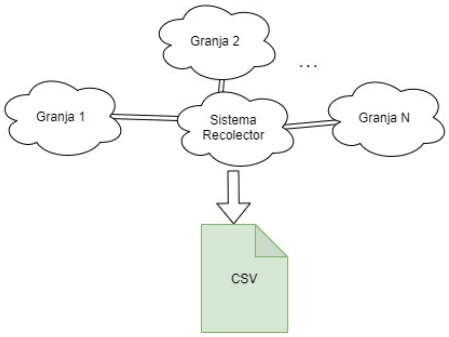

Figura 3.5: Diagrama a gran escala del sistema propuesto

A la hora de diseñar el sistema dentro de un contexto IoT se han implementado las tres capas que lo forman. En primer lugar, la capa *edge*, que corresponde con los generadores de datos de radiación. Se ha de tener en cuenta que no ha sido posible contar con sensores reales en este trabajo, por lo que se ha simulado el envío de mediciones de la misma forma que lo haría uno real.

Para realizar esto, se han utilizado datos recogidos previamente por sensores situados en Hawaii que están publicados en la web de *Measurement and Instrumentation Data Center* (MIDC, 2012).También se pueden obtener los datos de entrada para la simulación realizando consultas a una base de datos que los tenga almacenados, pero este caso ha quedado fuera del alcance del trabajo y se ha optado por la solución de los ficheros con mediciones tomadas previamente, aunque el resto de casuísticas se contemplan y se podrían añadir al sistema sin esfuerzo.

Otra capa que se encuentra presente es la capa *fog*, cuya principal función es aunar todos los datos de los sensores de una granja para procesarlos. Gracias al gran número de datos que el sistema recibe, se ha añadido distinta funcionalidad de procesado de información, que se detalla en el siguiente apartado.

Por último, se encuentra la capa *cloud*, que conecta todos los elementos de la capa *fog*. Es la responsable de recibir todos los datos del sistema una vez son procesados por la capa anterior y almacenar los datos para su posterior estudio o mantener un histórico de mediciones.

3.3. Funcionalidad al procesar datos

Como es bien conocido, los sensores IoT en ocasiones introducen valores que no son los esperados si se tiene como referencia el conjunto de las mediciones. Es importante detectar cuando esto sucede para eliminar estos datos generados erróneamente.

Por otro lado, al borrar los valores atípicos, se pierde una medición que puede ser realmente importante tener. Por tanto se ha planteado utilizar distintos métodos de regresión que recreen estos valores de manera aproximada y reemplazarlos de tal forma que se evita tener una pérdida total de información.

Otra funcionalidad adicional permite estimar con técnicas de kriging el valor de radiación solar que hay entre las localizaciones de los sensores de una granja. Esto es posible realizarlo debido a que se conocen las posiciones de los sensores y las medidas emitidas por estos en un instante de tiempo. En las siguientes secciones se ahonda en los algoritmos aplicados para cada una de estas casuísticas.

3.3.1. Cálculo de Outliers

A la hora de detectar valores atípicos, lo que se busca es encontrar en un grupo amplio de datos aquellos que no se asemejan con la mayoría de estos. Cuanto mayor sea el conjunto de valores a analizar, más fiable es el resultado.

Para plantear el algoritmo a utilizar en esta detección, se ha realizado una búsqueda mediante rangos intercuartiles (Mathwords, 2017), más conocida con el acrónimo *IQRs (interquartile ranges)*. Se calcula el primer cuartil (Q_1), que corresponde con una medida estadística de posición que equivale al percentil 25 % del grupo de valores. Se obtiene de igual manera el tercer cuartil(Q_3), que corresponde con un percentil 75 %.

Una vez que se obtienen los cuartiles se calcula el rango intercuartil, que es la resta del tercer cuartil menos el primero, $IQR = Q_3 - Q_1$. Por último, se determina el umbral de detección de valores atípicos. Este umbral es configurable en función de la severidad de detección que se desee, en este caso se ha fijado en un 50 % del rango intercuartil.

$$\begin{cases} x = Q_1 - 1{,}5 \cdot IQR \\ y = Q_3 + 1{,}5 \cdot IQR \end{cases}$$

Se calculan x e y que representan la franja de detección de *outliers*, por lo que todos los valores (v_i) del conjunto analizado que no estén comprendidos entre $x < v_i < y$, son considerados valores atípicos.

3.3.2. Métodos de regresión

Al realizar la búsqueda de valores atípicos, sucede que al encontrar alguno de estos es necesario sustituirlo por un valor más acorde a lo esperado. Por ello, para generar un valor más correcto, se ha decidido aplicar distintos métodos de regresión.

Se han empleado distintos métodos de generación, *Cubic Spline* (Ohiou, 2019), función de base radial (Fasshauser, 2017) y *Shepard* (Shepard, 1968) con el objetivo de obtener distintas sustituciones posibles.

El método *Cubic Spline* utiliza polinomios de bajo grado en cada intervalo con el objetivo de calcular las piezas polinomiales que encajen en la linealidad. La ventaja de Cubic es que introduce un error menor a una regresión lineal y el valor calculado es más suave.

Las funciones de base radial y *Shepard*, son muy utilizadas para puntos de datos que se distribuyen irregularmente en el espacio. Por lo que con este método no se obtendrá la mejor de las aproximaciones, ya que el aspecto de datos que se analizan están distribuidos de manera regular.

Al existir varias implementaciones de estas funciones ya realizadas, se ha empleado la librería *Smile* (Javadox, 2013) que implementa todas ellas en el lenguaje Java.

3.3.3. Interpolación con Kriging

El trabajo plantea calcular valores de radiación de puntos terrestres de los cuales se desconocen por no haber un sensor en esa posición. De las técnicas mencionadas en el apartado del estado del arte, se ha desarrollado el cálculo de Kriging ordinario.

Esta funcionalidad la implementa el servidor Fog. Actualmente, se permite el cálculo de un punto por cada servidor Fog presente en el sistema. Esta función devuelve un valor de radiación estimada, que se obtiene de los siguientes parámetros:

- Una lista de los valores de latitud del lugar donde se encuentran los nodos IoT de ese servidor Fog.
- Una lista de los valores de longitud del lugar donde se encuentran los nodos IoT de ese servidor Fog.
- El conjunto de las mediciones de radiación pertenecientes a cada uno de los nodos IoT de ese servidor Fog. Se utiliza la media de los últimos cien valores recibidos por cada sensor.
- La latitud del punto a interpolar.
- La longitud del punto a interpolar.

El algoritmo de kriging ordinario consiste en determinar el valor de una medida f_0 a partir de sus medidas vecinas $f_i, i \in 1 \dots n$

$$f_0 = \sum_{i=1}^{n} \omega_i f_i$$

Los valores ω_i se obtienen resolviendo el siguiente sistema de ecuaciones:

$$\begin{bmatrix} \gamma(x_1 - x_1) & \dots & \gamma(x_1 - x_n) & 1 \\ \vdots & & \vdots & \vdots \\ \gamma(x_n - x_1) & \dots & \gamma(x_n - x_n) & 1 \\ 1 & \dots & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_n \\ \lambda \end{bmatrix} = \begin{bmatrix} \gamma(x_1 - x_0) \\ \vdots \\ \gamma(x_n - x_0) \\ 1 \end{bmatrix}$$

Siendo x_0 el punto para el cual se quiere estimar la medida, $x_k, k \in 1 \dots n$ los puntos para los cuales se tienen las medidas f_k, y γ representa la función del variograma (Córdoba, 2015).

Por tanto, en primer lugar se tienen que tener todos los datos de entrada mencionados anteriormente. Con esto se calcula el variograma entre los datos conocidos. Los vectores tienen que tener el mismo tamaño, ya que se esperan tantas mediciones como sensores, en caso de que haya algún valor incorrecto o el tamaño no sea el esperado, se capturará la excepción y la simulación continuará sin poder realizar este cálculo.

Rigurosamente hablando, para realizar este cálculo, se tiene que incluir tolerancia y arreglos en función de los datos de entrada. Dado que esto dificulta notablemente el cálculo de kriging y que el objetivo del trabajo es aplicar un método de interpolación con la precisión que ofrece el algoritmo utilizado, no se han incluido estos arreglos, por lo que el cálculo obtenido es aproximado.

Tras conocer el variograma, se calcula la regresión esférica aplicando la fórmula:

$$v(h) = \begin{cases} c_0 + c \cdot \left(1{,}5 \cdot \frac{h}{a} - 0{,}5 \cdot \left(\frac{h}{a}\right)^3\right), h < a \\ c_0 + c, h \geq a \end{cases}$$

Por último, tras calcular la matriz ω_i se resuelve el sistema de ecuaciones para las coordenadas en cuestión.

Para llevar este método y sus sistemas de ecuaciones a código, se ha empleado la librería JAMA (Joe Hicklin, 2012), que es un paquete para Java que proporciona clases para construir y manipular con mayor facilidad matrices de gran tamaño. De todas las clases que ofrece, solamente se ha empleado la clase *Matrix*, con la cual se inicializa el objeto pasándole una lista doble como se muestra a continuación:

```
double [][] A = new double[size()][size()];
Matrix A_matrix = new Matrix(A);
```

Se hace uso de los métodos: *print* para las comprobaciones del contenido de la matriz; *inverse*, que calcula la inversa de una matriz; *times*, que realiza el cálculo de una multiplicación matricial.

Como ya es sabido, no es posible calcular la matriz inversa de todas las matrices, por tanto, cabe la posibilidad de que al invocar el kriging ordinario, no se obtenga un resultado. Cuando se produce esta situación, la función genera otro valor que corresponde con la media de las mediciones tomadas. Esto también se aplica cuando el resultado obtenido no se encuentra entre los esperados (se entiende como valor esperado a cualquiera que se sitúa entre la menor y mayor medición), esto puede ocurrir debido a la falta de arreglos durante el cálculo del variograma.

El valor obtenido, se envía a la siguiente capa IoT, indicando cuál es la naturaleza del dato calculado. Adicionalmente, también se adjunta la fecha que coincide con el tiempo del valor de la última muestra tomada para el cálculo.

3.4. Despliegue en nube pública (Cloud)

Actualmente existen varios proveedores de servicios en la nube, los más conocidos de los cuales son Amazon Web Services (AWS), Google Cloud Platform (GCP) y Microsoft Azure. A la hora de elegir el proveedor que mejor se adapta a este proyecto, valorando los recursos que es necesario crear, todas las opciones son igual de válidas, por lo que la elección se ha basado en aspectos secundarios como coste o destreza con alguna de estas nubes.

Como hito del proyecto, se marcó realizar todas las operaciones en algunos de los proveedores que ofrecen servicios de nube pública. En este caso, se ha realizado usando los servicios de *Google Cloud Platform* (GCP). Para ello, el primer paso fue crear el diseño de la infraestructura Cloud base sobre la que se realizará la simulación. Este esquema, se ha creado teniendo en cuenta la posibilidad de escalabilidad en futuras simulaciones y

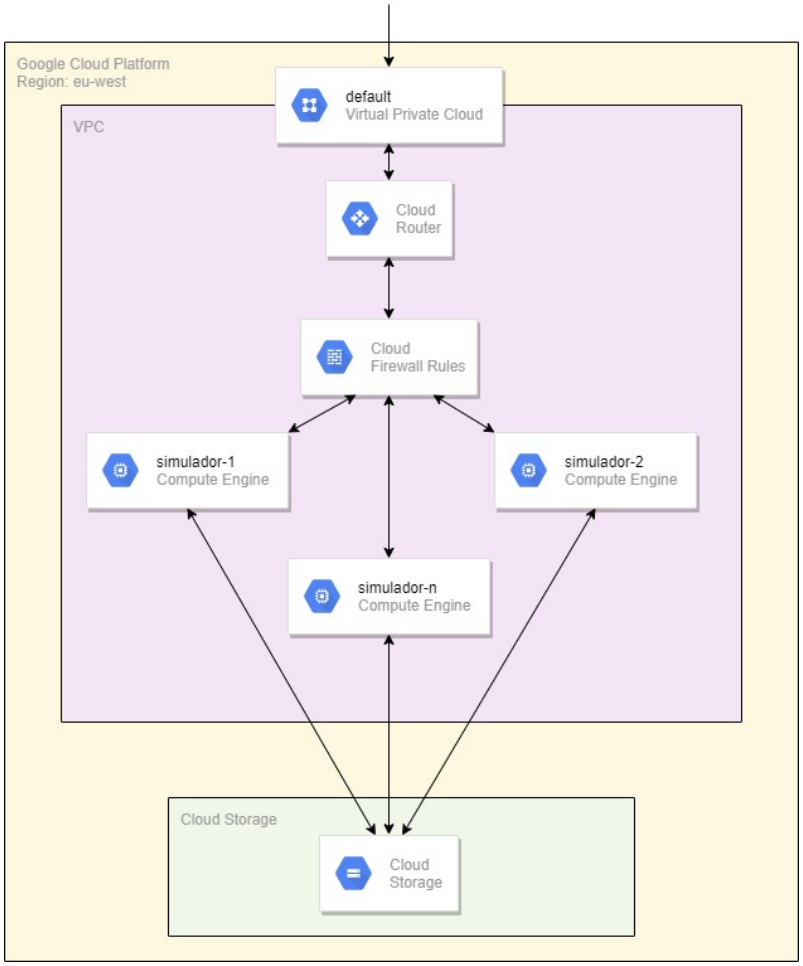

Figura 3.6: Esquema de la arquitectura Cloud base en GCP

durabilidad de la información en caso de que sea preciso almacenarla. A continuación se muestra el diseño final de todos los elementos de la nube que se han utilizado.

Como se indica en la imagen, todos los elementos creados para este trabajo se encuentran en la región de Google de Europa del Oeste (eu-west) que se conforma por varios centros de datos de esta empresa. Se ha seleccionado esta región por mantener los datos cercanos al lugar donde se quieren explotar. Para este trabajo, la simulación se realizó en los de Países Bajos y Bélgica. Dentro de esta región se encuentran dos elementos independientes: por un lado se tiene un *Cloud Storage* y por otro una *Virtual Private Cloud* (VPC). En Cloud Storage se almacenan archivos, y en este caso, se guardan todos los ficheros con la información que se ha extraído de los nodos IoT reales de Hawái. Adicionalmente, también se emplea para guardar el estado actual de la infraestructura y poder gestionarla como código. Esta información se encuentra estructurada en carpetas en función del nodo generador como se muestra en la figura 3.7.

Por otro lado, se encuentra la VPC, cuya finalidad es crear una red privada en la cual se despliegan los componentes necesarios. Los elementos que componen una VPC, son *Cloud Router*, en la cual se crean las subredes y tablas de rutas y *Firewall Rules* que permiten o deniegan el tráfico que se dirige o sale de la VPC en caso de ser necesario incluir medidas de seguridad en la plataforma. Para este trabajo, al ser una arquitectura cloud, a priori simple por los pocos elementos de red que se despliegan sobre ella, se usa la VPC por

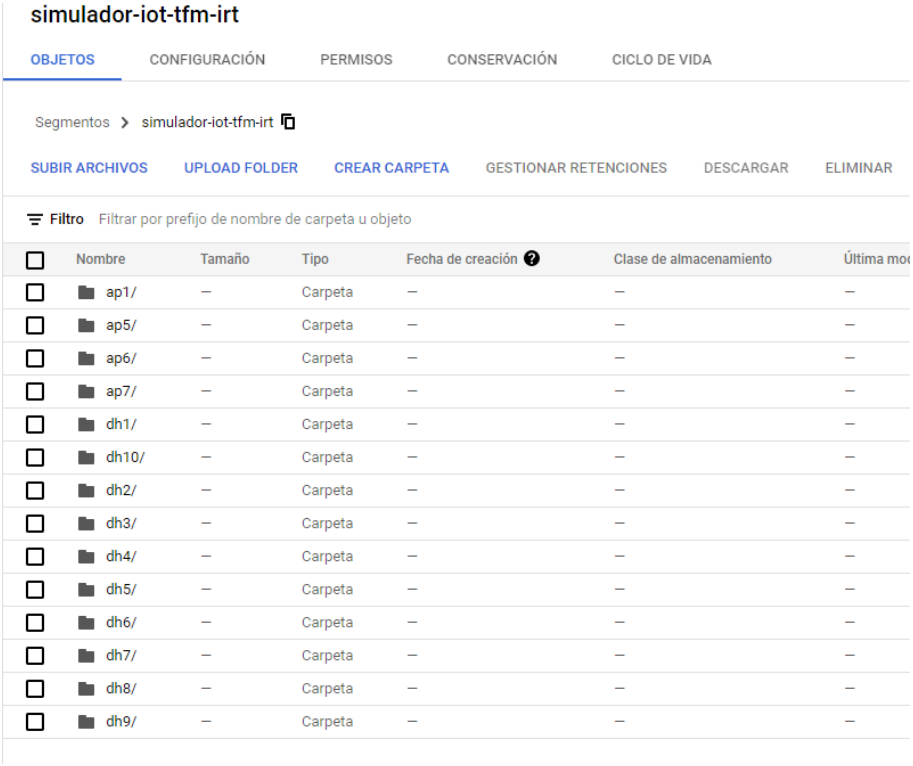

Figura 3.7: Estructura de directorios en Cloud Storage

defecto, junto con sus rutas y reglas de firewall.

Por último, se encuentran las instancias virtuales de Google Compute Engine sobre las cuales se configura el simulador de nodos, teniendo la posibilidad de poder crear tantas instancias como simuladores sean precisos. Las instancias utilizadas en el proyecto son del tipo *E2-medium* por lo que tienen 2 vCPUs con 4 GB de memoria RAM asociados. El sistema operativo base es un Debian 9 junto con un disco de 50 GB de memoria. El coste para este tipo de máquina es de 0.06701$ (Google, 2020b) por cada hora de computo.

Para hacer este diagrama arquitectónico de una forma que permita escalarlo con facilidad y replicarlo, se ha desarrollado toda la infraestructura como código. Esto significa que los elementos cloud están definidos en un fichero al que se invoca para crearlos, modificarlos o destruirlos. La implementación en GCP se detallará en el siguiente capítulo.

Capítulo 4

Implementación del sistema

Este capítulo corresponde con la implementación del sistema diseñado. Se comienza explicando cómo se ha empleado el formalismo DEVS como modelo de simulación junto con las clases desarrolladas. Igualmente se incluye un diagrama con la transitividad de los datos desde que son tomados por el sensor hasta que llegan al componente final. Por último, se incluye un apartado relativo al despliegue de la simulación en entornos de nube pública junto con la automatización realizada para la creación, destrucción y configuración del entorno en Cloud.

4.1. Elementos del sistema

En el capítulo dedicado al desarrollo del simulador, y partiendo del framework proporcionado por xDEVS, se realizó un análisis de los elementos necesarios para llevarlo a cabo. Para ello se plantean los distintos elementos que tiene que tener el sistema. Estos elementos al final se traducen en clases Java, las cuales son invocadas posteriormente para instanciar los objetos que contiene el simulador. En la figura 4.1 se muestran todas las clases que se han implementado. Aquellas que aparecen en verde, corresponden con las que forman parte de xDEVS y las de color naranja son las creadas para desempeñar la función de simulación del sistema. De esta forma se puede observar con mayor claridad la jerarquía y agregación del conjunto de clases.

La figura 4.2 se corresponde únicamente con las clases desarrolladas y cómo estas invocan o heredan directamente sobre otras que implementan el marco xDEVS. Como se observa, la clase *EntornoGlobal* hereda del modelo acoplado y el resto (ficheros, nodo virtual, servidor fog y datacenter), heredan de la clase del modelo atómico.

Por tanto, EntornoGlobal es el componente acoplado que contiene al resto de elementos atómicos. Por último se encuentra la clase *Input* que es la que encapsula la información que se envía entre los distintos componentes atómicos. Por tanto todo el intercambio de información que se realiza entre los elementos del sistema envía datos con el formato de la clase Input. En las siguientes secciones se detalla el funcionamiento de todas las clases mencionadas, en mayor profundidad.

Para entender los elementos que forman el sistema, y cómo se enlazan entre ellos, se tiene que tener en cuenta el funcionamiento del flujo de datos. Los generadores de datos son los componentes denominados *Ficheros*, que en el caso de este sistema generan mediciones reales. Esta información tiene que ser transformada a un objeto de tipo *Input*, que luego es enviado a través del puerto de salida del elemento atómico *Fichero*, al puerto de entrada del

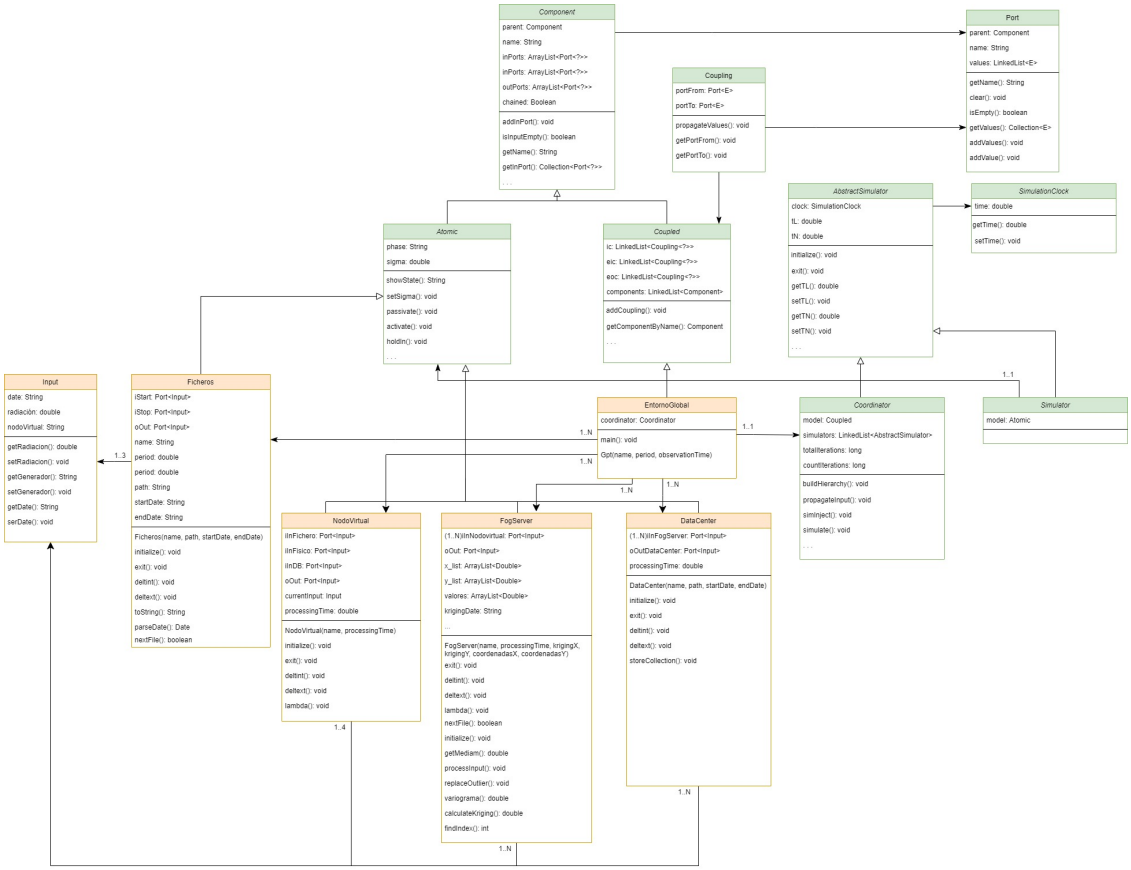

Figura 4.1: Diagrama a gran escala del sistema propuesto

componente atómico *NodoVirtual* al que esté destinado. Todas las conexiones son puerto a puerto, esto significa que cuando un puerto es utilizado en una conexión no se emplea para otra.

Una vez que la información en formato *Input* se encuentra en el puerto de entrada del *NodoVirtual*, se ejecuta la función externa (δ_e) que lee el dato recibido. Una vez procesado el dato, se envía a través del puerto de salida del *nodoVirtual* al puerto de entrada del *FogServer*, formando nuevamente otra conexión entre elementos.

Al igual que anteriormente, el dato es recogido una vez que se ejecuta la función δ_e del servidor fog. Por último, el servidor tras analizar y procesar el dato, enviará al *DataCenter* la medición a través de la conexión establecida entre el puerto de salida del servidor fog y el puerto de entrada del centro de datos.

Este flujo es el que aparece representado en la figura 4.3. Como se observa, este comportamiento se repite por cada granja existente en el sistema. Una granja está compuesta por al menos un *Fichero*, un *NodoVirtual* y un *FogServer*. El sistema puede contener tantas granjas como se deseen en la simulación. Cada granja está compuesta de todos los elementos de un modelo IoT: la capa edge contiene los *Ficheros* y *NodoVirtual*, la capa fog contiene los *FogServer* y por último la parte Cloud se corresponde con el *DataCenter*.

Por tanto, además de definir los objetos presentes, se tienen que especificar las conexiones entre ellos. En caso de que alguna conexión no se defina de manera correcta, la granja afectada no funcionará como se espera, aunque el resto del sistema no presentará problema alguno.

A continuación se detalla cada uno de los componentes, su funcionalidad y las conexio-

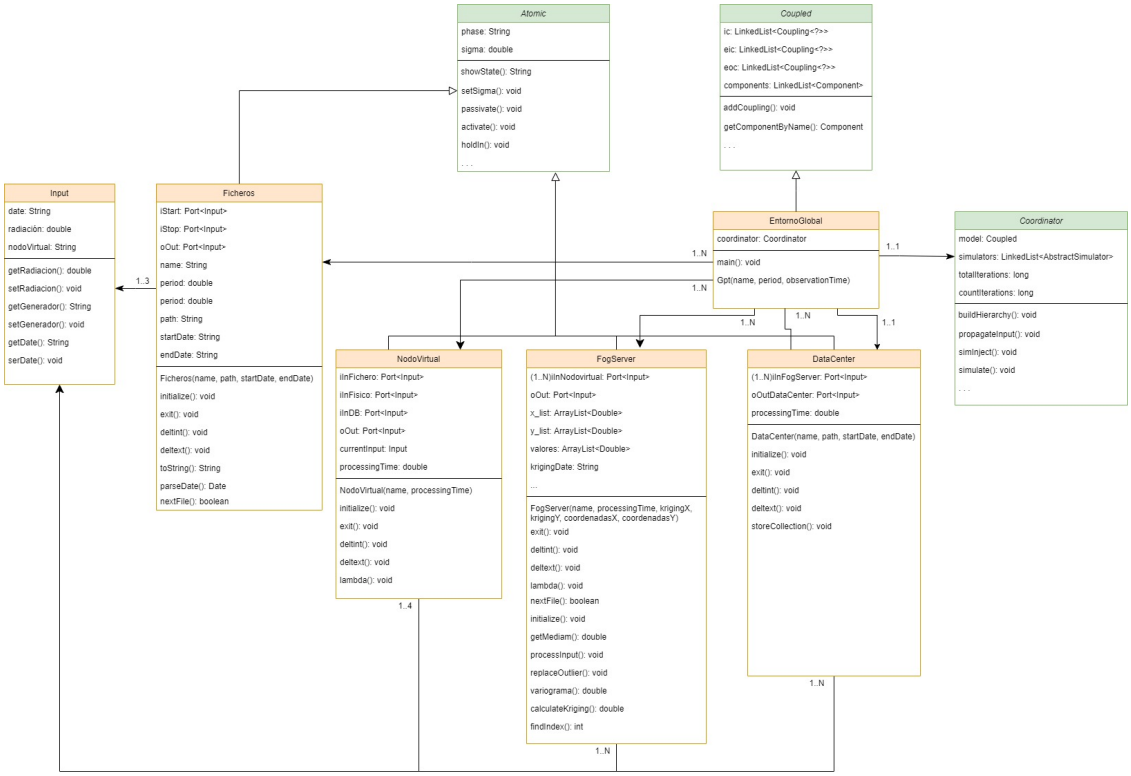

Figura 4.2: Diagrama de clases desarrolladas y su conexión con el framework xDEVS

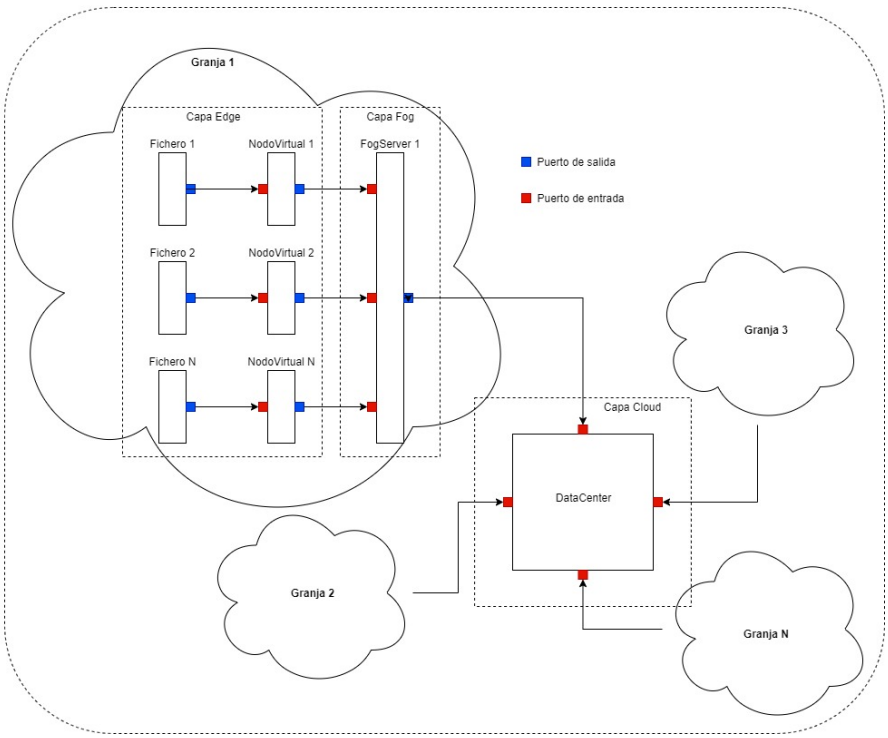

Figura 4.3: Flujo de datos entre componentes del sistema

nes que presenta.

4.1.1. Input

La clase Input se emplea para modelar el formato de la información que se envía entre los modelos atómicos del sistema. Está compuesta por la fecha en la cual se tomó la medición, el valor de radiación solar correspondiente y el identificador del generador del que procede. Es utilizada por todos los modelos atómicos del sistema.

4.1.2. Ficheros

La función de este componente atómico es la de generar mediciones de radiación e introducirlas en el sistema de simulación. Para ello, abre ficheros en formato csv previamente generados con datos reales de sensores de Hawaii y los envía al nodo virtual al que esté conectado.

Cuenta con la funcionalidad de que el tiempo de generación de datos es personalizable. Además, es posible especificar un rango de fechas entre las cuales se quieren generar datos, por lo que solo se transmitirán las mediciones de los archivos csv que estén comprendidas en ese rango.

Esta clase solo cuenta con un único puerto de salida, por lo que solo se puede conectar con un nodo virtual.

4.1.3. Nodo Virtual

El *NodoVirtual* es el elemento atómico que representa un sensor IoT simulado de la capa *edge* del sistema. Es el elemento intermedio entre los generadores de mediciones de radiación (*Ficheros*) y el servidor Fog.

Cada nodo está compuesto por tres puertos de entrada y uno de salida. Aunque solo se utiliza un puerto de entrada, están definidos dos adicionales para simular la generación de datos desde otra fuente que no sean los ficheros. El puerto *iInFichero* es el que se conecta con los objetos de tipo ficheros. El puerto *iInFisico* es el empleado para conectarse directamente a los sensores IoT reales y leer las mediciones directamente en tiempo real desde estos. El puerto *iInDB* está destinado a conectarse a una base de datos donde se almacenan las mediciones y realizar consultas para obtener las mediciones. Estos dos últimos son los que están presentes pero no son utilizados al no contar con nodos reales ni base de datos. El puerto *oOut* es el de salida y la conexión se establece con un puerto de entrada del servidor Fog.

4.1.4. Servidor Fog

El servidor fog es el componente que recoge las mediciones de todos los *NodosVirtuales* conectados a él. Esta agrupación equivale a lo que actualmente se conoce como una granja de sensores. Este servidor cuenta, por tanto, con un número de puertos de entrada que variará en función del número de los nodos virtuales asociados a él. La principal funcionalidad que implementa es la capacidad de recolección de un gran número de datos y su procesamiento.

En este procesado el sistema es capaz de detectar los valores atípicos para sustituirlos por otros correctos que sean similares. Se han empleado distintas técnicas de regresión para llevar a cabo el cálculo aproximado para remplazar los valores atípicos. Los métodos empleados se describen en el apartado de métodos de regresión.

Además, el servidor fog, cuenta con la funcionalidad para interpolar y hallar la radiación en un punto del espacio en el que no se cuenta con un sensor. Para realizar esto se han empleado técnicas de Kriging ordinario. Es necesario pasar las coordenadas en las cuales se quiere realizar el cálculo, junto con las localizaciones de los sensores asociados a ese servidor fog.

El servidor fog, a su vez, se enlaza con el *Data Center* al cual envía por su puerto de salida todos los valores no atípicos, junto con los resultados de los métodos de regresión e interpolación.

4.1.5. Data Center

El centro de datos es el nexo de unión entre todas las granjas de sensores IoT. Cuenta con tantos puertos de entrada como servidores fogs existan en el entorno de simulación. Su principal funcionalidad es leer de todos sus puertos los datos que se han enviado para adaptarlos a un formato csv y escribirlos en un archivo de salida. No cuenta con puertos de salida al ser el elemento que se encuentra el final de la simulación. Solo existirá un objeto Data Center por cada despliegue de simulación en Cloud. En caso de que haya más de uno, se estarían realizando escenarios en distintas máquinas virtuales.

4.1.6. EntornoGlobal

El *EntornoGlobal* es un modelo acoplado que contiene la función main() de la simulación. En él se crea la simulación con un *Coordinator*, como se ha mencionado anteriormente, para crear los componentes atómicos declarados en él y se invocan las funciones initialize() y simulate(). En el fragmento de código mostrado a continuación se muestra el inicio del programa.

```
public static void main(String args[]) {
    DevsLogger.setup(Level.INFO);
    EntornoGlobal eg = new EntornoGlobal("eg", 1, 30);
    Coordinator coordinator = new Coordinator(eg);
    coordinator.initialize();
    coordinator.simulate(Long.MAX_VALUE);
    coordinator.exit();
}
```

En su constructor se tienen que crear tantos modelos atómicos como presente la simulación junto con las conexiones entre ellos. Para esto se añaden al sistema acoplado. Poniendo un ejemplo básico, para crear una granja con un *Fichero*, un *NodoVirtual*, un *FogServer* y un *DataCenter*, siguiendo el flujo de datos mencionado anteriormente para las conexiones, el código que lo constituirá es el que se muestra seguidamente. Los componentes del modelo acoplado serían:

$$M = \langle X, Y, C, EIC, EOC, IC \rangle$$

Donde C contendría los *Ficheros*, nodos virtuales, fogservers y centro de datos e IC tendría la relación de estos componentes entre sí.

Como se observa, se invoca a los constructores de los componentes atómicos y posteriormente se realiza el enlace entre ellos con la función *addCoupling()* donde el primer

parámetro es el puerto origen y el segundo el puerto destino. En caso de plantear un escenario de simulación más grande el número de elementos declarados en esta clase se verá incrementado.

```
public EntornoGlobal(String name, double period, double observationTime) {
super(name);

    Ficheros ap1 = new Ficheros(...);
    super.addComponent(ap1);

    NodoVirtual nodoVirtual1 = new NodoVirtual(...);
    super.addComponent(nodoVirtual1);

     FogServer fogserver1 = new FogServer(...);
    super.addComponent(fogserver1);

    DataCenter dataCenter = new DataCenter(...);
    super.addComponent(dataCenter);

    super.addCoupling(ap1.oOut, nodoVirtual1.iInFichero);
    super.addCoupling(nodoVirtual1.oOut, fogserver1.iInNodoVirtual1);
    super.addCoupling(fogserver1.oOut, dataCenter.iInDataCenter);
}
```

En resumen, el principal cometido de esta clase es constituir el modelo acoplado y lanzar la simulación. Por tanto para ejecutar el programa, se llama a la función *main()* de este componente acoplado.

4.2. Despliegue de la simulación en Cloud

A la hora de implementar el diseño Cloud, se ha empleado la infraestructura como código para esta parte del desarrollo. Esta es una práctica muy extendida en la filosofía *DevOps*, que elimina la necesidad de realizar una gestión manual y actualización de componentes individualmente. Esto nos permite realizar cambios masivos en la arquitectura o desplegar varios entornos en paralelo con extremada rapidez, facilidad y sencillez. A su vez, permite destruir entornos de la misma manera. Por tanto contamos con el beneficio de poder crear y destruir entornos en pocos minutos, lo que supone un ahorro de costes importante, ya que solo se tiene la infraestructura desplegada cuando se quieran realizar operaciones de cómputo.

4.2.1. Infraestructura como código

Existen multitud de posibles herramientas para llevar a cabo la implementación. En caso de hacer el desarrollo en la nube de Amazon Web Services (AWS), se puede optar por utilizar la herramienta nativa de la nube, conocida como AWS Cloudformation (Amazon, 2020a), que a partir de ficheros construidos en JSON o YAML, despliega los objetos que se indiquen. Aunque, de igual modo, si no se conoce la sintaxis de este modelo de creación de recursos en Cloud, se puede utilizar AWS CDK (Amazon, 2020b), que es una librería de código abierto que permite a partir de código python desplegar recursos. La librería se encarga de transformar este código en el formato que entiende AWS Cloudformation,

JSON o YAML, por lo que facilita a los desarrolladores conocer cómo es la sintaxis de objetos bien formados en AWS Cloudformation. Para el caso de este proyecto, el cual se ha realizado en Google Cloud Platform (GCP), se puede optar por crear la infraestructura con Ansible o Terraform. Aunque Ansible cuenta con módulos orientados a creación de recursos Cloud, se ha decidido utilizar Terraform, desarrollado por *HashiCorp*, y reservar Ansible exclusivamente para gestión de la configuración interna de los recursos, junto con la puesta en marcha automatizada de la aplicación.

Terraform (HashiCorp, 2018) nos permite construir, cambiar y versionar infraestructura en la nube de una manera segura y eficiente. Actualmente es capaz de gestionar los proveedores Cloud más conocidos en el mercado como Amazon Web Services, Microsoft Azure o Google Cloud Platform junto con soluciones para servicios On-premise (se refiere al software instalado en los equipos locales de una organización). Terraform funciona a través de las APIs que los proveedores exponen, por lo que cuando estas interfaces son modificadas añadiendo nueva funcionalidad, se deberá actualizar la versión del proveedor de Terraform para poder utilizarlas.

4.3. Gestión de la configuración Cloud

En primer lugar es necesario descargar Terraform en el equipo que se encargue de lanzar la ejecución. Para ello, solo se tienen que ejecutar los siguientes comandos en un equipo linux:

```
$ git clone https://github.com/hashicorp/terraform.git
$ cd terraform
$ go install
```

Con esto se tendrá la última versión de Terraform instalada y lista para utilizar. El código Terraform es bastante común que utilice una versión previa a la *latest*, debido a la gran cantidad de versiones que se publican en poco tiempo. Para evitar tener que reinstalar distintas versiones según se requieran, existe una herramienta creada por la comunidad que facilita el cambio de versiones Terraform. Para este proyecto, se ha utilizado *tfenv* (Peachey, 2020) que se puede instalar como se indica en el README.md, con el comando:

```
$ brew install tfenv
```

Para ejecutar una sentencia *brew* se necesita tener *Homebrew* (Homebrew, 2021) ya sea para sistema operativo linux o mac. Una vez instalado, se tienen que instalar con él (si no se ha hecho ya previamente) las versiones que se desean tener. Para este proyecto, se utiliza la versión 0.13.5, por lo que ejecutando los siguientes comandos se tendrá configurada e instalada esta versión:

```
$ tfenv install 0.13.5
$ tfenv use 0.13.5
```

Tras esto, al ejecutar el siguiente comando se muestra que se utiliza dicha versión tal como se ve en la figura 4.4.

```
$ terraform version
```

Los recursos que se necesitan crear para este proyecto tienen que almacenarse en ficheros terraform con extensión tf. Todos ellos se encuentran bajo el directorio terraform y el

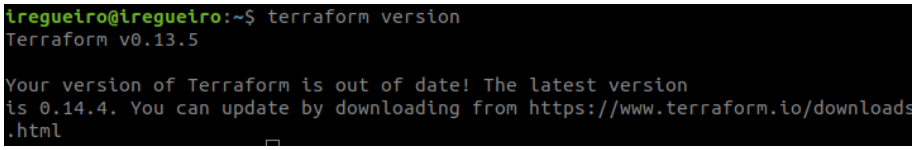

Figura 4.4: Versión de Terraform

servicio al que pertenecen, en este caso GCE (Google Compute Engine). Existen tres ficheros, `terraform_config.tf`, `remote_state.tf` y `main.tf`.

El primero de ellos, `terraform_config.tf`, contiene la configuración del proveedor de Terraform que se utiliza junto con el identificador del proyecto en GCP. En este proyecto, el proveedor empleado es el de Google, desarrollado por *HashiCorp* con la versión 3.47.0. Adicionalmente, junto con el *project id*, que es un identificador único de un proyecto de Google Cloud, se indica la región a utilizar, esto determina en qué centro de procesamiento de datos se van a crear los recursos por defecto (existe la posibilidad de indicar en cada recurso la región deseada, por lo que en caso de no indicarlo, se usará esta por defecto). La configuración empleada es la siguiente:

```
terraform {
  required_providers {
    google = {
      source  = "hashicorp/google"
      version = "3.47.0"
    }
  }
}

provider "google" {
  project = "skillful-hope-295119"
  region  = "europe-west4"
  zone    = "europe-west4-a"
}
```

El segundo de ellos, `remote_state.tf`, contiene la configuración del *state* de Terraform. En este fichero se indica en qué lugar se va a almacenar el fichero terraform.state que contiene la configuración que está aplicada en Cloud. Este fichero es importante debido a que se comparan con él las futuras ejecuciones y el estado actual, por lo que gracias a este es posible mostrar al usuario los cambios que se realizarán en caso de aplicar una nueva configuración en la nube. Para este proyecto, el *state* es almacenado en un *bucket* del servicio Storage de GCP, de modo que queda persistido en Cloud, y múltiples personas pueden trabajar simultáneamente. En la configuración, que aparece a continuación, se indica el nombre identificador del bucket y el prefijo, que corresponde con la ruta de directorios en la cual se va a almacenar.

```
terraform {
  backend "gcs" {
    bucket = "tf-simulador-iot-tfm"
    prefix = "gce/terraform.tfstate"
  }
}
```

El último de ellos es el fichero `main.tf` que contiene la configuración de los recursos que se van a crear. Para este proyecto es necesario crear una máquina virtual por cada entorno de simulación. Por tanto en caso de querer tener varios entornos simultáneos, el código es fácilmente escalable, creando tantas máquinas virtuales como entornos. El código base es el siguiente:

```
resource "google_compute_instance" "simulator" {
  name         = "compute-simulator"
  machine_type = "e2-medium"
  zone         = "europe-west4-a"

  tags = ["project", "tmf"]

  boot_disk {
    auto_delete = true

    initialize_params {
      image = "debian-cloud/debian-9"
      size  = 50
    }
  }

  network_interface {
    network = "default"

    access_config {
      // Ephemeral IP
    }
  }
}
```

En este caso, se crea un objeto del tipo `google_compute_instance` con el identificador único *simulator* que tiene los atributos *name*, el nombre con el que aparece en la consola de Google Cloud; *machine-type*, el tipo de maquina que tiene asociados unos recursos en vCPUs y memoria RAM; *zone*, el centro de datos de Google en el cual se desplegará el recurso; *tags*, etiquetas que asignan los desarrolladores para facilitar la identificación de recursos; *boot_ disk*, que contiene un subconjunto de atributos con las características del disco que se va a crear junto con la máquina virtual (se utiliza una imagen de Debian 9 y tiene un tamaño de 50 GB). Con el atributo *auto_ delete* se indica que cuando el recurso sea eliminado, este disco también se elimine; *network_ interface* contiene la configuración de red virtual en la que se colocará el recurso. La red *default* corresponde con la *Virtual Private Network* que se crea por defecto en cada proyecto. Al indicar una configuración de acceso, *access_ config* indica que se asigne una dirección IP pública efímera al recurso.

Antes de aplicar los cambios, se debe crear en la consola de Google Cloud una cuenta de servicio que tenga permisos de edición sobre los recursos que se vayan a crear. Para realizar esto, hay que dirigirse a la sección de IAM (Cloud Identity and Access Management) de la consola. En el apartado de *service accounts* se ha de crear uno nuevo con permisos de *Editor*. El role *Editor* es nativo de GCP y permite modificar todos los recursos de la cuenta. Una vez hecho esto, se ha de crear una clave asociada a esta cuenta de servicio en formato JSON. Al generar la clave, se descarga localmente el fichero que la contiene. Se ha de tener

especial cuidado de guardar cuidadosamente esta clave y no compartirla con terceros.

Con estos tres ficheros, la configuración de Terraform y la clave de la cuenta de servicio generada, es posible aplicar los cambios y crear los recursos. Para ello, se tiene que hacer en primer lugar un cambio de directorio hasta situarse en el que contiene los ficheros con extensión tf. Se tiene que cargar la clave de la cuenta de servicio como variable de entorno ejecutando el comando:

```
$ export GOOGLE_APPLICATION_CREDENTIALS="/path/to/key.json"
```

En caso de no cargar esta variable, se obtiene el error:

```
Error: storage.NewClient() failed: dialing: google: could not find default
credentials. See
https://developers.google.com/accounts/docs/application-default-credentials
for more information.
```

Acto seguido, al ejecutar el comando `$ terraform plan` se visualizan los cambios que se van a producir en el proveedor Cloud pero sin que dichos cambios se apliquen. En cambio, si se está seguro de que los cambios son correctos, se usa el comando `$ terraform apply` para aplicarlos. Al ejecutar este último comando, se obtiene una salida por el terminal con el resultado del mismo. En el apéndice B se muestra la salida al provisionar los recursos en GCP.

En caso de que la simulación requiera más recursos de los fijados en el código base, es posible, modificando los atributos explicados anteriormente en el fichero `main.tf`, ajustarlos acorde a las necesidades de cada ejecución. Al ejecutar el comando `$ terraform apply`, se produce la modificación de estos recursos o la creación si no existían previamente.

Al acceder a la consola web de Google Cloud, en la sección de Compute Engine, una vez se haya hecho `$ terraform apply` y haya finalizado satisfactoriamente, aparecen los recursos creados con las características especificadas en la interfaz web. De esta forma, se puede verificar que la configuración es correcta. Como se observa en la siguiente imagen, se ha creado una instancia con el nombre `compute-simulator`.

En la sección de monitorización de la instancia, se ha hecho una evaluación de los recursos necesarios para llevar a cabo la simulación. Cuando los recursos asignados eran menores de los necesarios, se ha implementado un escalado vertical del recurso, hasta obtener las características mostradas previamente. Este escalado ha consistido en ejecutar la aplicación repetidamente, hasta que las métricas de monitorización no mostraban valores anómalos.

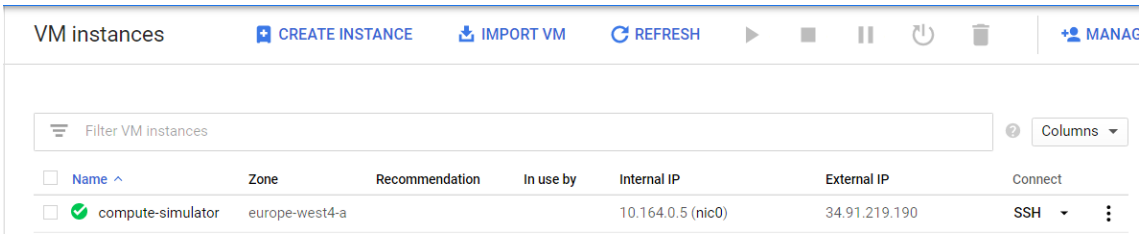

Figura 4.5: Verificación de las instancias virtuales creadas

Una vez que se haya finalizado la simulación, se borrarán los recursos Cloud al ejecutar el comando `$ terraform destroy`. Estos desaparecerá de la consola web y no será posible volverlos a recuperar, aunque al tener la configuración en código, se puede crear nuevamente.

4.4. Gestión de la configuración de la Máquina virtual con Ansible

Al tener ya la máquina de simulación creada, esta solamente cuenta con el sistema operativo instalado. Como se indicó en el apartado anterior, en este caso se tiene una instancia con Debian 9. Por lo que es requerido instalar y configurar múltiples paquetes para poder ejecutar la aplicación.

Como bien es sabido, se puede utilizar cualquier gestor de paquetes para su instalación, por ejemplo se puede usar *apt, npm o pip* entre otros. Pero si hacemos esto de una manera tradicional, en la cual se escriben los comandos por la terminal, tenemos que repetir constantemente estos pasos cada vez que se lance una instancia nueva. A esto se le añade un posible error humano en la instalación lo que implica un gasto adicional de tiempo.

Por estos motivos, se ha optado por utilizar una herramienta que permita persistir esta configuración y sea fácilmente reutilizable. Sin olvidar que, en caso de levantar varios simuladores independientes, esto se pueda realizar con el menor esfuerzo posible.

Entre las posibles opciones que existen para llevar a cabo esta implementación las más conocidas son Puppet (Kanies, 2020) y Ansible (RedHat, 2020a). Ambas permiten trabajar contra un grupo de máquinas para configurarlas y administrarlas de forma masiva. Se ha optado por utilizar Ansible sobre Puppet, por experiencia previa con esta herramienta.

Ansible gestiona los distintos nodos mediante conexión SSH y no requiere ningún software adicional instalado en ellos, salvo tener Python con la versión 2.4 o posterior. A través de esta conexión es posible ejecutar acciones ad-hoc o administrar configuraciones. Por ello surge el concepto de máquina controladora, que corresponde a la máquina desde la cual se ejecutan las acciones descritas en los ficheros de tipo YAML de Ansible, contra el inventario de equipos sobre los que actuar.

En el proyecto se tiene un inventario variable en función de la cantidad de entornos de simulación. Por cada uno de ellos se debe introducir su dirección IP pública en un grupo. Esta dirección se obtiene una vez que se ha creado la maquina virtual con Terraform, como se ha mostrado anteriormente. Los inventarios que se han usado son estáticos para facilitar el desarrollo del proyecto, aunque existen los inventarios dinámicos, que se actualizan en función de las nuevas máquinas que se vayan creando. Por tanto, se tiene que actualizar manualmente este inventario, que tiene el siguiente formato:

```
[local]
localhost
[remote]
xx.xx.xx.xx
yy.yy.yy.yy
```

El inventario se compone de dos grupos, local y remote. El primero sirve para hacer pruebas en la máquina local y comprobar previamente que el código Ansible funciona como se espera. El segundo grupo, remote, se compone de todas las instancias virtuales con entornos de simulación. Cada entrada corresponde con la dirección IP que Google Cloud asigna.

Además de los inventarios, existe el fichero `ansible.cfg`, cuya finalidad es fijar distintos parámetros de configuración de Ansible. Normalmente no es necesario modificar este fichero, ya que con las opciones por defecto se ejecuta correctamente. Sin embargo, en este caso se ha tenido que cambiar la variable `interpreter_python` que indica que interprete de python va a emplearse. Por tanto la configuración final de este fichero contiene la siguiente información:

```
[defaults]
interpreter_python=/usr/bin/python3
```

Debido a los módulos que se utilizan, que se comentan más adelante, se ha tenido que cambiar el interprete por defecto, que se encuentra en `/usr/bin/python` apuntando a la versión 2.7, por el intérprete de la versión 3.

Tras tener inventario y configuración, se ha creado un *playbook*. Este es el nombre empleado para los ficheros que se ejecutan en Ansible y están compuestos por uno o más *plays*. Un *play* es cada una de las tareas que se van a realizar. El inicio de la ejecución en Ansible empieza desde un *playbook* que puede estar compuesto por *plays* y *roles*. El *playbook* desde el que se ejecuta el despliegue tiene las siguientes instrucciones.

```
---
- hosts: remote
  become: no
  gather_facts: yes

  roles:
    - configure_host
    - download_application
    - xdevs_configuration
    - start_app
...
```

Al inicio del *playbook* se le indica sobre qué grupo del inventario debe ejecutarse, que en este caso es sobre el grupo *remote*, que contiene todas las máquinas virtuales que se han creado en Google Cloud. En el campo *become* se le indica al playbook si se le concede escalado de privilegios a *root*. Con el campo *gather_facts* (RedHat, 2020e) se le indica a ansible que recoja variables (hostname, dirección IP, etc.) de la máquina remota que son de utilidad para utilizarlas en caso de necesidad.

A continuación, se tiene la sección *roles* (RedHat, 2020j). Un role contiene una serie de tareas que pueden estar condensadas en uno o más *playbooks*, que se ejecutan en función de unas variables de entrada. La ventaja que presentan los roles es que pueden ser reutilizados en cualquier playbook y utilizarse aunque los haya desarrollado otra persona. Para este proyecto, ha sido necesario desarrollar los roles `configure_host`, `download_application`, `xdevs_configuration` y `start_app`.

Para la creación de los *roles*, se ha utilizado `ansible-galaxy`, que es una herramienta que se incluye en la instalación de ansible. Ansible-Galaxy es un repositorio que contiene todos los roles públicos que ha desarrollado la comunidad. Aunque en el proyecto no se utiliza ningún *role* de terceros, se ha empleado este comando ya que crea la estructura de directorios y ficheros. La estructura de directorios que se crea para cada *role* es la siguiente:

```
roles/
    role_name/
    README.md
        tasks/
            main.yml
        handlers/
            main.yml
        vars/
```

```
        main.yml
    defaults/
        main.yml
    meta/
        main.yml
```

El comando empleado para crear estos *roles* ha sido el siguiente:

```
$ ansible-galaxy init nombre-del-role
```

En el siguiente apartado se describen todas las acciones que realiza cada *role* desarrollado en profundidad, junto con los módulos de ansible empleados.

4.4.1. configure_host

Este es el primer *role* que se ejecuta en ansible y es el encargado de realizar la configuración inicial necesaria sobre la máquina virtual. Este *role* se compone de un *playbook* que contiene cuatro *plays*.

En el primero se utiliza el módulo `apt` (RedHat, 2020b) con el cual se indica que se instale `maven` (gestor de paquetes que se emplea en el proyecto), `git` (permite poder descargarse el código de la aplicación desde GitHub), `python-pip3` (gestor de paquetes empleado en python3), `python3-setuptools` (paquete requerido para poder ejecutar módulos de ansible posteriores) y `openjdk-11-jre-headless` (conjunto de utilidades necesarias para poder ejecutar programas en Java).

El siguiente *play* hace uso del módulo `pip` (RedHat, 2020h) para instalar en el gestor de paquetes de python la librería `google-auth` con la versión 1.23.0. Este paquete es requerido por los módulos de ansible que se utilizan en el despliegue y de no incluirse aparece en la ejecución una traza de error, indicando la ausencia de éste. Es importante resaltar que, como en el *play* anterior se ha instalado *pip*, es posible realizar este, una vez el primero finaliza correctamente.

Los dos últimos plays hacen uso del módulo `shell` (RedHat, 2020k) que ejecuta un comando bash por el terminal de la máquina remota. Estos comandos fijan las variables de entorno java que se requieren para poder usar el JRE de Java 11 descargado en el *play* anterior de este *role*. Los comandos ejecutados son los siguientes:

```
export JAVA_HOME="/usr/lib/jvm/java-11-openjdk-amd64/"
export PATH=$JAVA_HOME/bin:$PATH
```

Esta configuración es requerida debido a que el sistema operativo base de la máquina virtual es un Debian 9 y por defecto cuenta con Java 8 instalado. Esta versión de Java no es soportada por la aplicación.

4.4.2. download_application

El siguiente *role* desarrollado es el encargado de descargar el código necesario para que la aplicación pueda ejecutarse junto con otros ficheros adicionales. Está compuesto de cuatro *plays*, el primero de ellos hace uso del módulo `file` (RedHat, 2020d). Este módulo tiene la funcionalidad de realizar cualquier tipo de cambio sobre ficheros o directorios en Unix.

El primer *play*, realiza un borrado del directorio en el cual se descarga la aplicación. El principal objetivo de este es agilizar la fase de pruebas, pudiendo descargar distintas

versiones de la aplicación, sin necesidad de lanzar una nueva máquina virtual. En una ejecución real, al intentar borrar un directorio que aún no ha sido creado, el módulo se percata de este hecho y no realiza ninguna acción.

El segundo *play* hace uso del módulo `git` (RedHat, 2020f). Este módulo permite realizar las acciones de esta herramienta mediante ansible. En este caso se realiza un *clone* del repositorio de la aplicación. En este *play* se realiza un registro de variable con el parámetro *register* (RedHat, 2020i), variable que se utiliza en el siguiente *play* para verificar que la operación git se ha realizado correctamente.

El último play de este *role* hace uso del módulo `command` (RedHat, 2020c). Este módulo es similar al `shell` ya que ejecuta un comando en la terminal de la máquina remota. El objetivo de esta tarea, es descargar localmente un listado de ficheros que se alojan en Google Cloud Storage (Google, 2020a). El código de este *play* es el siguiente:

```
- name: Get data files
  command: gsutil -m cp -r gs://simulador-iot-tfm-irt/{{ item }} {{ dest }}
  loop: "{{ ficheros }}"
```

Se ejecuta el comando `gsutil` (Google, 2021), que es una aplicación en Python que permite acceder al servicio de almacenamiento de Google. Al haber creado una máquina virtual en la nube de Google, esta herramienta viene instalada por defecto, por lo que no es necesario indicárselo a Ansible para que lo haga.

Con la opción `-m` se indica que el comando que sigue se va a ejecutar múltiples veces de manera simultánea. Esto se hace para reducir el tiempo de este *play*, que es más elevado a mayor número de ficheros a descargar. El comando `cp` indica que se realiza una copia de archivos de manera recursiva sobre el *bucket* con el identificador `simulador-iot-tfm-irt`. Debido a que `gsutil` no permite descargar el *bucket* entero, se le tiene que indicar las carpetas o archivos que se desean descargar. Por tanto, con la variable `loop: "{{ ficheros }}"` e `{{ item }}` de Ansible, se le indica cuáles tiene que copiar. Por último, con la variable `{{ dest }}` se informa del directorio destino, que albergará una copia de los ficheros.

Las variables mencionadas anteriormente se almacenan en el fichero destinado a este propósito. Para este role, se cuenta con el siguiente fichero de variables:

```
---
clone_dir: ~/simuladorIoT
dest: ~/simuladorIoT/data/
ficheros:
  - ap1
  - ap5
```

Aunque estas variables se pueden pasar al role de múltiples formas (como `extra-vars`, en el fichero `host_vars`, en el propio *playbook*, etc.), se ha elegido utilizar los propios ficheros del *role*, por mera sencillez de código.

4.4.3. xdevs_configuration

Este *role* realiza la configuración a nivel del código de la aplicación. Al estar en un simulador que se compone de varios elementos, existe la posibilidad de querer adaptarlo a unas necesidades necesidades especificas, en el que se requieran un cierto número de nodos, una conexión entre elementos personalizada, etc.

Está compuesto por varios *plays* que modifican ficheros Java de la aplicación. Todos hacen uso del módulo `lineinfile` (RedHat, 2020g) que edita ficheros en linux. Se modifican dos ficheros, el primero de ellos es el fichero *main* del programa, en el que se realizan varias acciones.

En primer lugar lugar se crean objetos de la clase *ficheros*. El módulo busca un patrón en *regex* en el código, donde se le indica el lugar en que que se debe insertar las líneas de java que crean estos objetos. La cantidad de objetos a insertar depende de una lista en el fichero de variables, por lo que está parametrizado el *play*. El código resultante se muestra a continuación

```
- name: configure ficheros
  lineinfile:
    path: "{{ entornoGlobal_path }}"
    insertafter: '^// ANSIBLE FICHEROS$'
    backup: no
    state: present
    line: |
    Ficheros {{ item }} = new Ficheros("{{ item }}", period, userDirectory);
    super.addComponent({{ item }});
  loop: "{{ ficheros|reverse|list }}"
```

En este *play* se insertan las líneas que aparecen en el bloque `line` después de la línea *// ANSIBLE FICHEROS* que se indica con el parámetro `insertafter`. Con la opción `path` se indica la ruta del fichero a modificar. Con `backup` se tiene la opción que genere un fichero adicional antes de la modificación, por si esta resulta errónea, en cuyo caso no es necesaria. Por último esta acción se repite tantas veces como entradas en la lista de la variable `ficheros` haya.

El resto de *plays* de este role son similares, variando el bloque de líneas, `path`, `insertafter` y otra lista de variables en función del objeto a crear. En función de la arquitectura del sistema DEVs que se quiera simular, el fichero de variables de este *role* variará. Un ejemplo de este fichero sería el siguiente:

```
entornoGlobal_path: ~/simuladorIoT/.../EntornoGlobal.java
fog_server_path: ~/simuladorIoT/.../FogServer.java
ficheros:
  - ap1
  - ap5
nodo_virtual:
  - nodoVirtual1
  - nodoVirtual2
fog_server:
  - fogserver1
data_center:
  - dataCenter
conectores:
  - out_type: ficheros
    out: ap1
    in: nodoVirtual1
  - out_type: ficheros
    out: ap5
```

```
      in: nodoVirtual2
    - out_type: nodoVirtual
      out: nodoVirtual1
      in: fogserver1
    - out_type: nodoVirtual
      out: nodoVirtual2
      in: fogserver1
    - out_type: fogServer
      out: fogserver1
      in: dataCenter
deltext:
  - nodoVirtual1
  - nodoVirtual2
```

Debido a que este fichero de configuración define la configuración del sistema que se quiera simular, en la figura 4.6 se muestra como queda el sistema con la configuración de variables mostrada anteriormente. Igualmente, a continuación se describe el significado de cada una de estas variables

- entornoGlobal_path: Directorio de la máquina virtual en la cual se encuentra el fichero main del programa.
- fog_server_path: Directorio de la máquina virtual en la cual se encuentra el fichero `fogServer.java`.
- ficheros: Contiene una lista con los identificadores de los objetos ficheros. Se crearán tantos objetos de tipo ficheros como entradas existan en la lista.
- nodo_virtual: Contiene una lista con los identificadores de los nodos virtuales. Se crearán tantos objetos de tipo nodo_virtual como entradas existan en la lista.
- fog_server: Contiene una lista con los identificadores de los *fogServer*. Se crearán tantos objetos de tipo *fogServer* como entradas existan en la lista.
- data_center: Contiene una lista con los identificadores de los *dataCenter*. Se crearán tantos objetos de tipo *dataCenter* como entradas existan en la lista.

- conectores: Contiene una lista con las conexiones que se establecen dentro del entorno DEVS. La lista está compuesta por objetos con tres elementos:
  - out_type: Se indica el tipo de conector, ya que no es lo mismo enlazar un fichero con un nodo virtual, que un servidor Fog con el *dataCenter*. Los valores permitidos son:
    - ficheros: Para enlazar la salida del objeto ficheros con la entrada de los nodos virtuales.
    - nodoVirtual: Permite conectar la salida del nodo virtual con la entrada del servidor fog.
    - fogServer: Conecta la salida del servidor fog con la entrada al centro de datos.
  - out: Identificador de la salida del objeto a conectar.
  - in: Identificador de la entrada del objeto a enlazar.

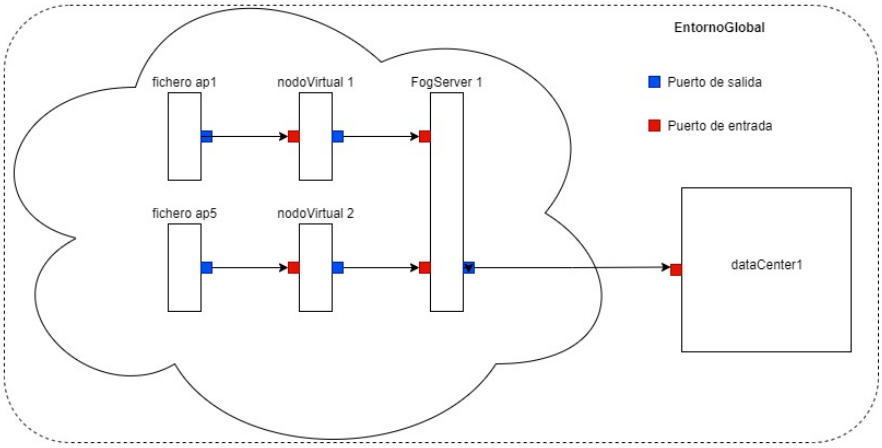

Figura 4.6: Disposición de elementos DEVS según las variables

- deltext: Identificador de los nodos virtuales que se quieran procesar en el fogServer. Por defecto y como buena practica deberían estar incluidos todos los elementos de la lista `nodo_virtual`.

Durante la ejecución del *role*, varios *plays* tienen condiciones *when* que evitan crear objetos de más a los especificados en las variables. Con estas condiciones, se agiliza la ejecución, ya que ansible al no cumplirse la condición, realiza omite esa ejecución y pasa a la siguiente.

4.4.4. start_app

Este es el último *role* desarrollado necesario para ejecutar la aplicación. Su función principal es ejecutar en la máquina virtual el programa. Se compone de dos *plays*.

El primero de ellos, se encarga de generar el archivo con extensión `jar` listo para ejecutarse. Se hace uso del módulo `command` (RedHat, 2020c) y se ejecuta el siguiente comando:

```
chdir=~/simuladorIoT mvn package
```

El segundo y último *play* hace uso del módulo `shell` (RedHat, 2020k) para invocar la función `main` de la aplicación. El comando que ejecuta ansible es el siguiente:

```
chdir=~/simuladorIoT java -cp target/xdevs-*.jar \
xdevs.core.examples.EntornoGlobal
```

Con la opción `-cp` se le indica el *classpath* en el que se encuentra el ejecutable. Al hacer el empaquetado del programa, este se guarda en la carpeta `target`. Seguido a este, se le indica el inicio del programa. Tras ejecutar esto, el proceso que ejecuta ansible finaliza.

4.5. Despliegue automatizado

Como se ha mencionado en los puntos anteriores, la implementación con *Terraform* y *Ansible* han sido empleadas para poder llevar a cabo una puesta en marcha de la aplicación de manera desatendida.

Una vez que se tiene todo el desarrollo, se tienen que enlazar los elementos entre sí. En la figura 4.7 se muestran los pasos durante la provisión y configuración de la aplicación.

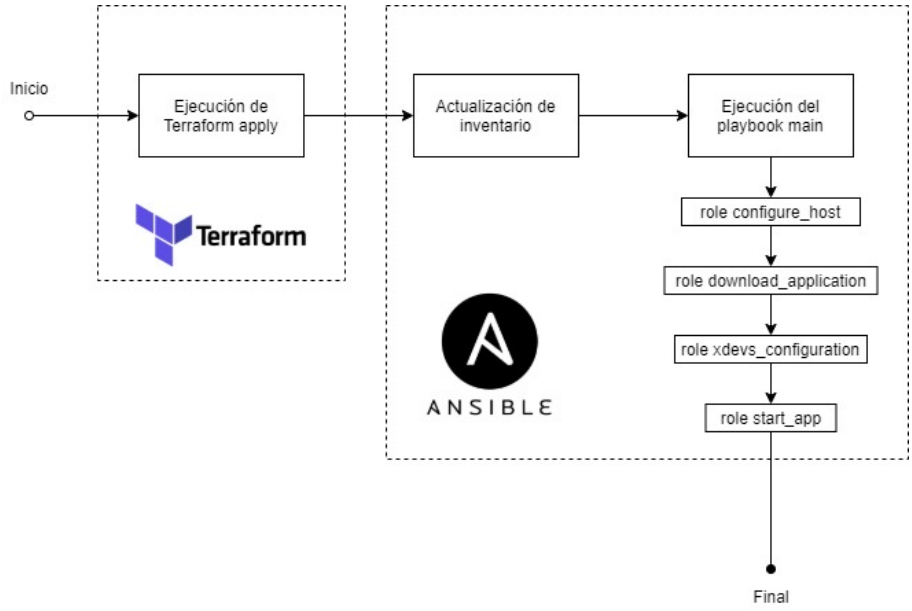

Figura 4.7: Diagrama con los pasos de ejecución durante el despliegue

En primer lugar se comienza ejecutando el código *Terraform* para provisionar en cloud las n instancias virtuales deseadas, en función de la cantidad de escenarios de simulación. Cada máquina virtual corresponderá con una simulación distinta. Una vez que finaliza la provisión en la nube, se tiene que actualizar el inventario de ansible con las n instancias. Tras esto se lanza el *playbook* de configuración que invoca los cuatro roles implementados. El primero realiza la configuración básica de paquetería necesaria para ejecutar la aplicación. El segundo descarga el código del repositorio *Git* en el que se encuentra. El tercero configura la aplicación en función de los ficheros de variables y el último role lanza la ejecución de la aplicación.

Con esto se consigue poner en marcha la aplicación en un número n de máquinas virtuales con configuraciones diferentes entre todas ellas, de manera simultánea y desatendida. Esto permite escalar el sistema tanto como se quiera y el tiempo para realizar esta acción es siempre el mismo.

Capítulo 5

Resultados

Este capítulo contiene un caso de uso del cual se han sacado los resultados del trabajo elaborado. En primer lugar se muestra el despliegue y configuración Cloud para ejecutar la aplicación. En segundo lugar se incluye un análisis de resultados con distintas comparativas y gráficas, tras aplicar los métodos de detección de outliers, métodos de regresión e interpolación.

Se ha creado un entorno de simulación sobre el cuál se han extraído los resultados que se detallan a continuación. El escenario tiene dos granjas de nodos IoTs con distintos elementos, pero la misma disposición. Se tienen cuatro generadores por granja conectados a cuatro nodos virtuales para la capa edge, siguiendo la nomenclatura del capítulo 4. Estos se conectan con un servidor fog que hace de puente con la capa cloud formada por un centro de datos único para ambas granjas de nodos.

Se ha utilizado un grupo de datos del año 2010, que contienen mediciones de radiación solar recogidas por sensores localizados en el aeropuerto de Kalaeloa en la isla de Oáhu en el archipiélago de Hawaii. Estos valores están accesibles en la web de Measurement and Instrumentation Data Center (MIDC, 2012).

5.1. Despliegue Cloud

Para poder obtener los resultados del trabajo, en primer lugar se tiene que desplegar la aplicación en Cloud y tener la escalabilidad necesaria en función del escenario. Para simulaciones pequeñas o pruebas iniciales, se puede omitir este paso y ejecutarlo localmente. Para este ejemplo se ha utilizado *Google Cloud Platform* como proveedor. Por tanto hay que crearse una cuenta de Google. Una vez que se tenga la cuenta creada, se accede a la consola de Google en `https://console.cloud.google.com/`. Si es la primera vez que se accede, nos pedirá cierta información y la aceptación de las condiciones y términos para concluir la creación de una cuenta de prueba en GCP. En caso de que ya se tenga una cuenta consolidada se abrirá la consola web.

Por defecto la cuenta incorpora un proyecto creado que se puede utilizar para este trabajo. En caso de que se desee, es posible crear un nuevo proyecto en GCP para lanzar la simulación en él. Para comenzar, es necesario crear una cuenta de servicio en la sección IAM (Identity and Access Management). Se le indica un nombre y los permisos. Es importante asignar el rol de Editor para evitar tener problemas durante el despliegue. En la figura 5.1 se muestra la cuenta de servicio utilizada llamada *simuladoriot* con la función de Editor.

Después de esto, se ha de generar una clave de esta cuenta de servicio que se usará

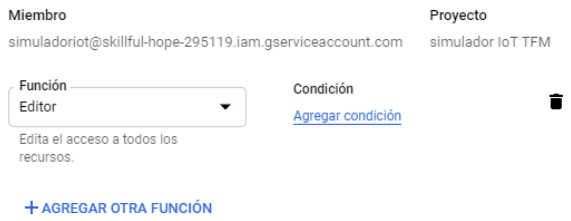

Figura 5.1: Cuenta de servicio de GCP

para hacer llamadas a Google Cloud mediante la API. Para ello, se accede a la sección de cuentas de servicio de IAM, se selecciona la cuenta en cuestión y se accede a la pestaña de claves. Al hacer clic en *Agregar clave* preguntará por el formato de la misma. Se seleccionará JSON y se descargará en nuestro ordenador un fichero con las credenciales. En la figura 5.2 se muestran las claves generadas para una cuenta de servicio. Es importante guardar este archivo y no compartirlo, ya que tendrá acceso para gestionar todos los recursos del proyecto en GCP.

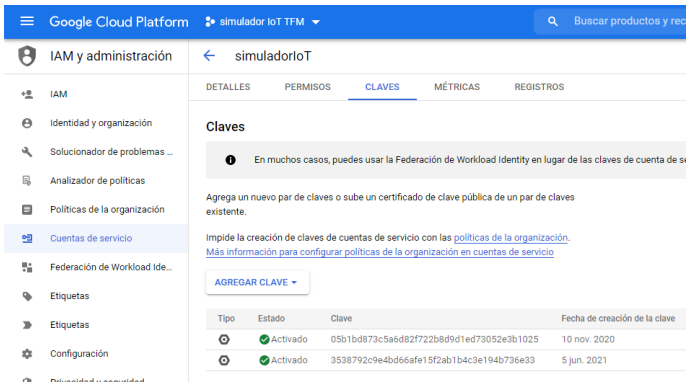

Figura 5.2: Generación de claves de una cuenta de servicio

Tras esta configuración inicial en la cuenta de GCP, se pasa a configurar el equipo desde el que se lanzarán todos los procesos de despliegue. Para ello es necesario instalar Terraform y Ansible. En este caso se ha utilizado una distribución Ubuntu de GNU/Linux, por lo que la paquetería inicial se descarga con los siguientes comandos:

```
$ sudo apt update
$ sudo apt install software-properties-common
$ sudo add-apt-repository --yes --update ppa:ansible/ansible
$ sudo apt install ansible

$ echo "eval \$($(brew --prefix)/bin/brew shellenv)" >>~/.profile
$ brew install tfenv
$ tfenv install 0.13.5
$ tfenv use 0.13.5
```

Con esto se tendrá configurado tanto Terraform como Ansible. Se puede comprobar que los binarios están instalados invocándolos con el flag `--version`. Tras comprobar que la instalación es correcta, se procede a descargar el repositorio de despliegue. Esto se hace ejecutando los siguientes comandos en una terminal.

```
$ terraform --version
$ ansible-playbook --version
$ git clone https://github.com/iregue/simuladorIoT-deploy.git
```

Este repositorio contiene dos carpetas, una para la parte de terraform y otra para la de ansible. En primer lugar se hace un cambio de directorio hasta llegar a la carpeta ./simuladorIoT-deploy/terraform/gce donde se encuentra el fichero main.tf. Antes de ejecutarlo, en primer lugar se tiene que configurar el fichero terraform_config.tf indicando el identificador del proyecto y la región que se prefiera. En segundo lugar se ha de eliminar el fichero remote_state.tf si se quiere guardar el estado de terraform localmente o de otro modo, se ha de crear un Storage de Google. Para hacer esto último se accede a Cloud Storage y se crea este recurso con un nombre. En este caso el storage creado tiene el nombre de tf−simulador−iot−tfm, que es el mismo que aparece en el segundo fichero a modificar. En la figura 5.3 aparecen todos los recursos storage creados en el proyecto de GCP. Tras esto se estaría en condiciones de ejecutar el código Terraform.

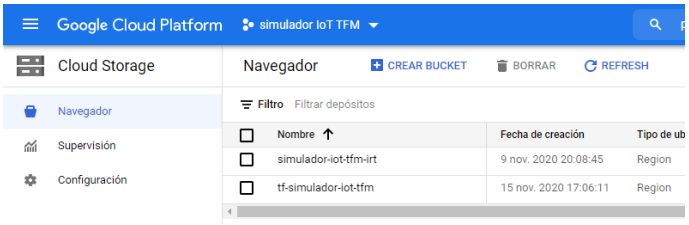

Figura 5.3: Recurso Storage destinado a guardar el estado de terraform

Para ejecutar el código Terraform, en primer lugar es necesario definir la variable de entorno GOOGLE_APPLICATION_CREDENTIALS indicando el *path* a la clave de la cuenta de servicio descargada previamente.

Acto seguido se inicializa terraform. Al hacer esto, se descarga la versión del proveedor definida en el fichero terraform_config.tf, se crea un estado de terraform vacío, entre otras operaciones. En la figura 5.4 se muestra la traza de salida al realizar la inicialización.

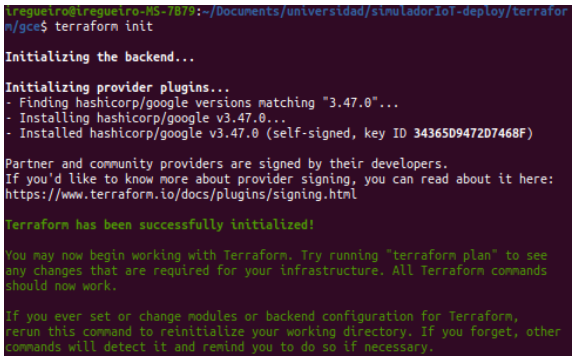

Figura 5.4: Inicialización de Terraform

Tras esto se puede ejecutar el código del fichero main.tf. Con el comando `terraform apply` se crean todos los recursos y se obtendrá una salida similar a la que figura en el Anexo B. Esto generará la creación de los recursos en Google Cloud especificados en main.tf.

En la figura 5.5 aparece cómo se está creando la máquina virtual en la consola de Google Cloud en el apartado de Google Compute Engine. Esto también causa que el estado se actualice y se listen todos los recursos creados como se observa en el Anexo B.

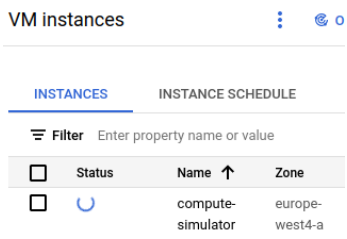

Figura 5.5: Creación de recursos con Terraform

Al realizar esta operación, es posible que devuelva algún error. Las cuentas de Google Cloud cuando se crean tienen todas las APIs deshabilitadas por defecto, por lo que si el output de terraform devuelve un error de API, incluirá un enlace que lleva a la consola para su habilitación. Por ejemplo, en la figura 5.6 se muestra la página para habilitar una de estas APIs y solo es necesario hacer clic en el botón de activación y esperar unos pocos segundos.

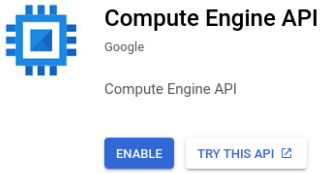

Figura 5.6: Habilitar API de Google Compute Engine

Tras tener las máquinas virtuales creadas, para poder configurarlas se necesita acceso mediante ssh para que ansible pueda conectarse. Se requiere por tanto contar con una pareja de claves pública/privada. En caso de no tenerlas previamente, se generan ejecutando el comando ssh−keygen −t rsa en el terminal. En la carpeta /.ssh se encontrarán estas claves.

Para poder conectarnos por ssh a las máquinas virtuales, tenemos que compartir con Google Cloud nuestra clave pública. Como se observa en la figura 5.7 en la sección de metadatos de Google Compute Engine en posible publicar nuestra clave, lo que nos otorgará acceso a cualquier máquina virtual de ese proyecto.

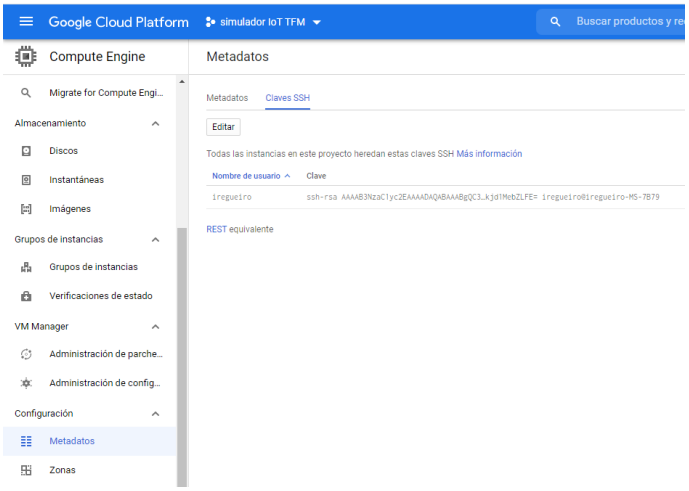

Figura 5.7: Acceso ssh a máquinas virtuales de Google Cloud

Tras esto habrá terminado todo el despliegue relativo a terraform y faltará la parte de configuración del simulador. Para ello se tienen que definir los componentes del escenario de simulación IoT que se deseen. Para este trabajo, se ha creado un escenario que está compuesto por ocho sensores de la capa edge, dos objetos de la capa fog y un centro de datos, según la nomenclatura del capítulo 4. Este escenario cuenta con dos granjas, cada una de ellas con cuatro objetos de tipo *Ficheros*, cuatro objetos de tipo *NodoVirtual* y un objeto *FogServer*. Ambas granjas se conectan con el mismo *DataCenter*. Esta estructura es la que se observa en la figura 5.8. Para que se aplique esta configuración, es necesario elaborar el fichero de variables. Este se puede encontrar en el anexo B.

Este fichero se puede importar de múltiples formas, la que se ha empleado es fijando las variables del rol. Para ello solo se tiene que sobreescribir el fichero que se encuentra en ./simuladorIoT−deploy/playbooks/roles/xdevs_configuration/vars/main.yml.

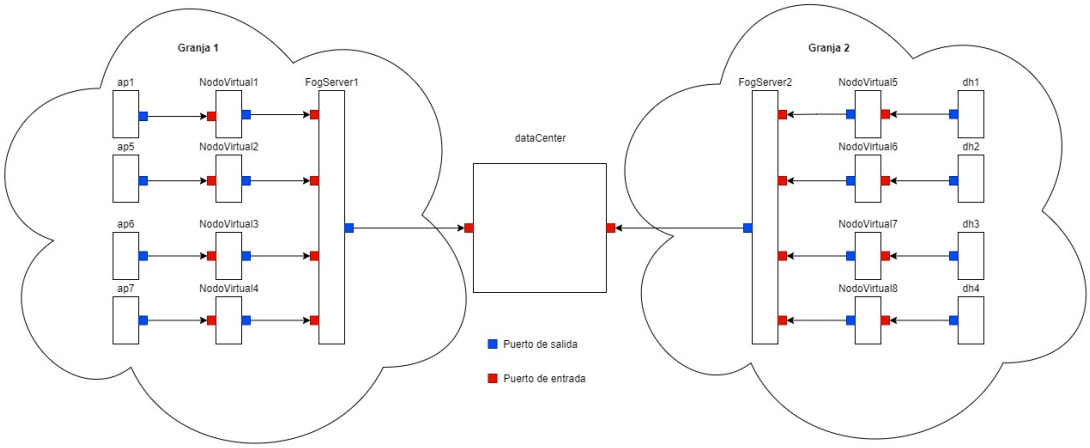

Figura 5.8: Escenario empleado para los resultados

Una vez que ya se tiene el fichero con las variables ansible definidas, falta actualizar el inventario. Se tiene que copiar la IP pública de la máquina. Esta se puede consultar por la consola de Google Cloud y ponerla en el fichero inventory de la carpeta playbooks. En este caso, se ha asignado la IP 34.141.183.148. Como se muestra a continuación se añade dentro del grupo remote.

```
[local]
localhost
[remote]
34.141.183.148
```

Tras esto, ya será posible ejecutar el código ansible contra la máquina remota, sin errores de conexión. En caso de que aparezca algún error de conexión, estará causado por los firewall de GCP. Para solucionarlo, se accede a la sección de reglas de firewall dentro de VPC y se añade o modifica la regla que afecta al puerto 22, para dar acceso a la IP de salida desde la cual se encuentra el equipo que ejecuta el despliegue. En su defecto se puede, aunque no es nada recomendable por seguridad abrirlo a Internet con el filtro 0.0.0.0/0. Para este caso se ha empleado este último filtro, ya que las máquinas se crean y se destruyen cuando finaliza la simulación.

La figura 5.9 se corresponde con la regla empleada para permitir el acceso al puerto 22

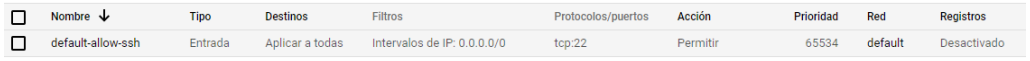

Figura 5.9: Escenario empleado para los resultados

mediante ssh. Como se observa se aplica a la red *default* por lo que se aplica sobre todas las máquinas que se creen sobre esa red.

Para ejecutar el código ansible, en primer lugar se tiene que hacer un cambio de directorio hasta el *playbook* que inicia el proceso, que se encuentra en el mismo repositorio de despliegue en el path ./simuladorIoT−deploy/playbooks/configure_app.yml. Con el comando siguiente, se lanza todo el proceso.

```
$ ansible-playbook -i inventory configure_app.yml
```

Al invocar la ejecución del *playbook* devuelve por la salida estándar el log del proceso. En el anexo B se incluye la traza de salida para el escenario planteado anteriormente.

Si se accede por ssh a la instancia de google, a la carpeta donde se ha configurado el simulador, se habrá generado el fichero *collectios_ output.txt* con los valores obtenidos tras la ejecución y el fichero *logger.log* con los logs que genera el programa. El listado de los ficheros de la carpeta raíz, entre los cuales están los mencionados anteriormente, se muestran en la figura 5.10.

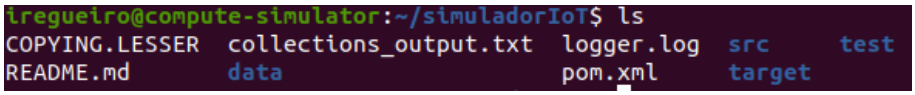

Figura 5.10: Ficheros generados tras la ejecución

Por último, revisando la monitorización propia de GCP, se puede comprobar si los recursos de la máquina para la simulación han sido suficientes o no, ya que es posible que en caso de tener un escenario amplio y complejo, no haya suficiente cómputo o espacio en disco y sea necesario cambiar los recursos en el código terraform.

Figura 5.11: Gráfica de monitorización por el uso de cpu

En este caso el uso de CPU para esta ejecución fue del 60 % y los accesos a disco no han llegado a 20MiB/s ni se han llenado los 50Gib de memoria.

Por tanto, en caso de querer ahorrar costes para la siguiente simulación, es posible reducir el tier (distintos niveles fijados por Google Cloud para los recursos de computación en la nube) de la máquina virtual al inferior y reducir el disco hasta un espacio más acorde al utilizado. Con esta acción, se optimizan los recursos y estamos iterando por cada ejecución para ajustar el coste al consumo Cloud que se está haciendo.

Los costes para esta simulación se pueden consultar a través de la consola de Google Cloud y se muestran en la tabla 5.1 separado el gasto por cada uno de los servicios utilizados.

Figura 5.12: Gráfica de monitorización de procesamiento de disco

Nombre del servicio	Coste total
Standard Storage	0.08€
E2 instance Core	0.02€
E2 instance RAM	0.02€

Tabla 5.1: Coste total de la simulación en Cloud

5.2. Análisis de los datos de salida

Al abrir el fichero de salida, se encuentran los datos en formato csv, separados por punto y coma, donde la primera columna corresponde con el valor de radiación obtenido, la segunda columna identifica la fuente desde la cual procede esa radiación y la tercera contiene un valor de fecha para la medición. Esto se puede observar en el siguiente dato de ejemplo:

```
159.488;ap1;2010-03-20 07:30:18-10:00
```

A la hora de analizar los datos obtenidos, el estudio se va a separar en varios apartados que cubren las distintas implementaciones realizadas. Estas son: la detección de *outliers*, la sustitución de valores atípicos, y la interpolación aplicando el algoritmo de kriging desarrollado.

Hay que tener en cuenta que todos estos análisis se realizan en tiempo de simulación, que imite el comportamiento del sistema y las soluciones implementadas en tiempo real.

5.2.1. Detección de Outliers

Para analizar la detección de *outliers* se han recogido todos los valores de los sensores de la granja 1 durante un día (20/03/2021) antes de ser procesados por el servidor fog. Para determinar si un valor es un *outlier*, se analizan periodos de tiempo más cortos, donde la variación de estos sea reducida en el tiempo. Por tanto se ha seleccionado un total de 60 datos comprendidos dentro de una franja de 10 minutos. Al ser una fracción pequeña de tiempo, se analiza el conjunto de valores sin tener en cuenta el momento temporal.

En primer lugar se calculan los cuartiles uno y tres para estos datos. Los valores obtenidos son 843,78 para $q1$ y 909,08 para $q3$. Con estos valores se calcula el rango intercuartil de restar $q1$ a $q3$ cuyo resultado es 65,31. Con esto ya se tienen todos los valores de la ecuación $x = Q_1 - 1{,}5 \cdot IQR$ e $y = Q_3 + 1{,}5 \cdot IQR$ para obtener x e y. En este caso se obtiene que $x = 745,81$ e $y = 1007,05$. Por tanto los valores que no estén comprendidos entre x e y son considerados valores atípicos.

Esto se puede observar visualmente en la figura 5.13, donde el eje Y representa el valor de radiación, se muestra este conjunto de datos mediante los puntos de color azul y las líneas que indican cuando un valor es considerado atípico están representadas por los

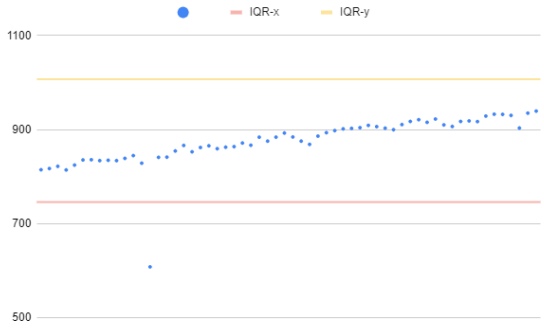

Figura 5.13: Detección de outliers para la granja 1

colores amarillo y rosa. Por tanto, en este caso, el programa identifica uno de los valores como *outliers* y procederá a remplazarlo.

Este proceso se repite con todos los datos, analizando periodos cortos de tiempo sobre cada una de las granjas de forma independientemente. Tras la detección de los outliers, viene el proceso de reemplazo de estos, cuyos resultados se explican en el siguiente apartado.

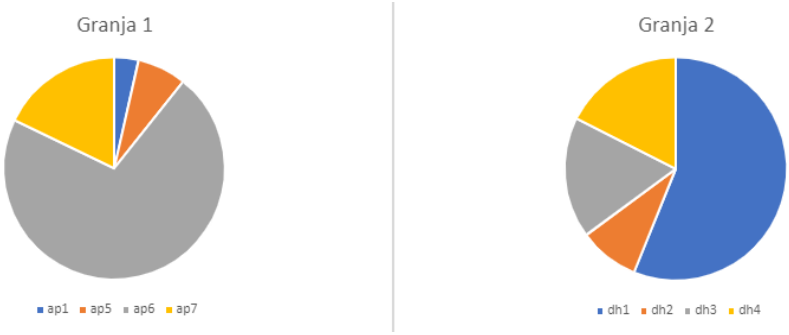

Figura 5.14: Generación relativa de outliers por nodo

Analizando los datos globalmente generados por ambas granjas durante el mismo periodo de 10 minutos, teniendo en cuenta que cada sensor genera un dato por segundo, se puede llegar a analizar la cantidad de outliers generados por cada sensor para poder determinar cuál de ellos produce más fallos.

Nombre del nodo	Número de Outliers	Porcentaje de generación
ap1	1	0,16 %
ap5	2	0,33 %
ap6	20	3,32 %
ap7	5	0,83 %

Tabla 5.2: Total de outliers detectados para la granja 1

En las tablas 5.2 y 5.3 se muestra el número total de outliers generados durante el periodo establecido de muestra junto con el porcentaje de generación de un valor atípico.

Se puede observar que la granja 2 genera prácticamente el doble de outliers en comparación con la granja 1, ambas en porcentajes bajos, aunque algún porcentaje destaca notablemente en varios dispositivos. En el ejemplo considerado, los sensores *dh1* y *ap6* son los más propensos a introducir fallos en el sistema. Se tendría que estudiar a qué es debido esto, ya que puede haber múltiples factores como la localización del sensor, la altitud,

Nombre del nodo	Número de Outliers	Porcentaje de generación
dh1	32	5,35 %
dh2	5	0,83 %
dh3	10	1,67 %
dh4	10	1,67 %

Tabla 5.3: Total de outliers detectados para la granja 2

humedad o la degradación del dispositivo con el paso del tiempo. Estos datos mostrados en las tablas anteriores, también se reflejan de manera visual en la figura 5.14 donde solo un nodo por granja generan más de la mitad de los valores atípicos.

5.2.2. Sustitución de valores atípicos

Para el proceso de remplazo de valores atípicos se generan datos con los distintos métodos de regresión explicados anteriormente en el capítulo 3. Para este análisis se han tomado como referencia los datos del apartado anterior. A simple vista, únicamente mirando estos valores es posible detectar algún valor atípico.

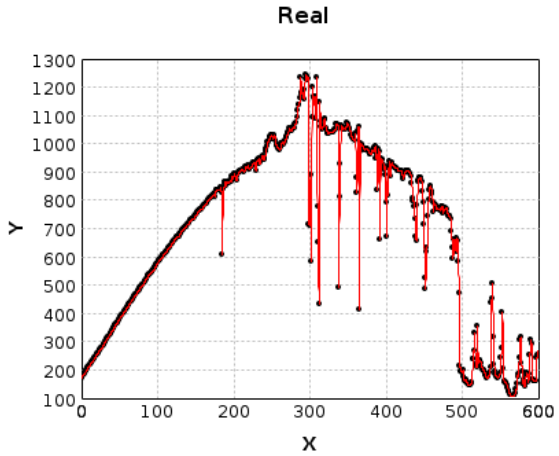

Figura 5.15: Valores generados por ap1 durante un día

El eje Y de la figura 5.15 representa el valor de radiación, mientras que el eje X el número de valores representados a lo largo del tiempo se puede apreciar que aparecen líneas verticales que son indicativas de outliers. Estos valores son remplazados por los generados por los métodos *CubicSpline*, *Shepard* o *RBF*. El cálculo se realiza pasando los valores reales más próximos al valor atípico, para obtener una mayor precisión. Para la figura 5.15 se han remplazado con los resultados de *CubicSpline*.

Por tanto, teniendo en cuanta los valores del sensor de la figura 5.15, se generan gráficas de estos métodos de regresión, que contienen los valores de sustitución para los outliers tomando como referencia varias mediciones. Esto se muestra en la figura 5.16.

Para este caso, la generación se ha realizado pasando diez datos, y se generan tantos valores intermedios como se indiquen en la implementación de los métodos. Como en la gráfica original se cuenta con seiscientos valores, el resto de gráficas generan ese mismo número de valores intermedios. La sustitución se realiza cambiando la posición que ocupa el valor atípico dentro de la lista de mediciones, por el resultado que ocupa la misma posición del listado generado de aplicar *CubicSpline*.

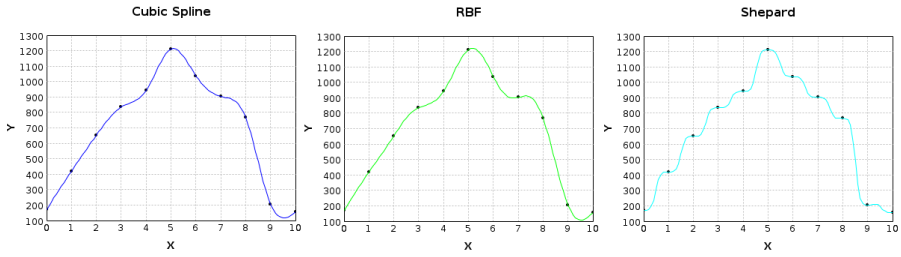

Figura 5.16: Gráficas generadas con los metodos de regresión a partir de la Figura 5.15

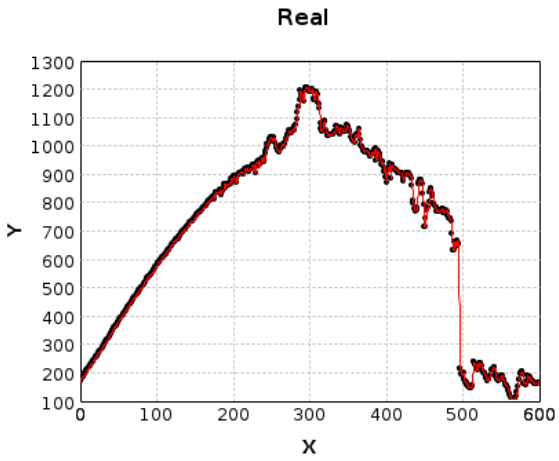

Figura 5.17: Resultado de sustituir outliers con los métodos de regresión

La gráfica resultante es la que se muestra en la figura 5.17. Como se observa, dejan de aparecer los puntos esporádicos que no tienen relación con los adyacentes a ellos.

Se procede a mostrar los resultados de los tres métodos de regresión para obtener una comparación entre ellos. Este estudio se ha realizado con un grupo de valores más reducido. Los resultado de aplicar uno u otro resultado apenas difiere en unas décimas, debido a que cuando se realiza un cálculo se toman las mediciones previas y posteriores al valor atípico.

La figura 5.18 muestra los datos correspondientes a los valores adyacentes del sensor *dh1*, donde el eje Y representa el valor de radiación. Se ha de tener en cuenta que la variación es muy pequeña ya que la gráfica está acotada en un rango de 0,8 (unidades). La línea de color azul representa las mediciones del sensor, en donde aparecen dos outliers que quedan fuera de la gráficas, representados con las líneas verticales. Se calculan por tanto resultados exclusivamente para estos valores anómalos. Tomando como referencia la leyenda que aparece en la parte superior, cuando se aplica *CubicSpline* se obtiene un valor acotado entre los resultado de *RBF* y *Shepard*. La tendencia general muestra que los valores de *RBF* son ligeramente mejores comparándolo con el resto de métodos de regresión, mientras que ocurre lo contrario con los de *Shepard*. Debido a esta pequeña variación es por lo que no se observa cambio significativo entre gráficas con gran densidad de datos como ocurre con la figura 5.17.

5.2.3. Resultados al aplicar kriging

A la hora de aplicar el algoritmo de kriging, se dispone de un conjunto de sensores $1..n$ ubicados en sus respectivas coordenadas $(x_1, y_1)..(x_n, y_n)$ y se quiere conocer el valor de radiación en un punto (x_0, y_0) sin sensor. Para este ejemplo, la localización de estos se

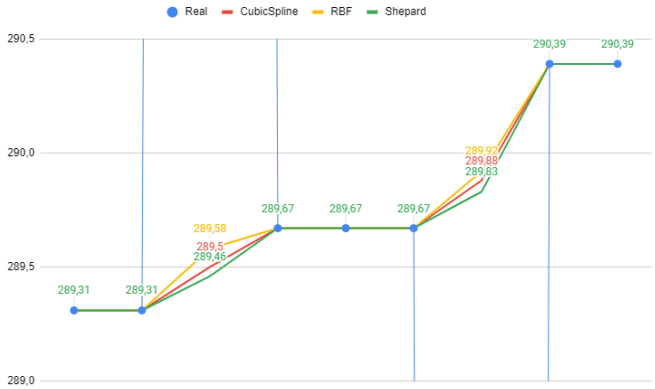

Figura 5.18: Comparativa al aplicar distintos métodos de regresión

encuentra en la figura 5.19 y los valores de radiación están tomados en el mismo instante de tiempo para todos ellos.

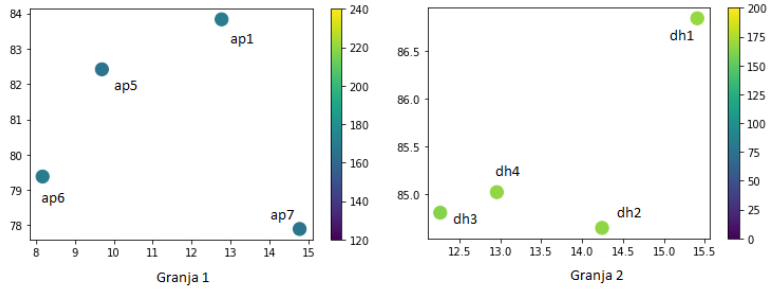

Figura 5.19: Posición de los sensores en el espacio

La figura 5.19 muestra la posición de los sensores de dos granjas y el valor de la radiación mediante un color cuyo espectro queda definido en el margen derecho. Si a la imagen anterior añadimos los puntos obtenidos de aplicar el algoritmo de kriging ordinario detallado en el capítulo 3, aparece un punto nuevo. Esto se muestra en la figura 5.20.

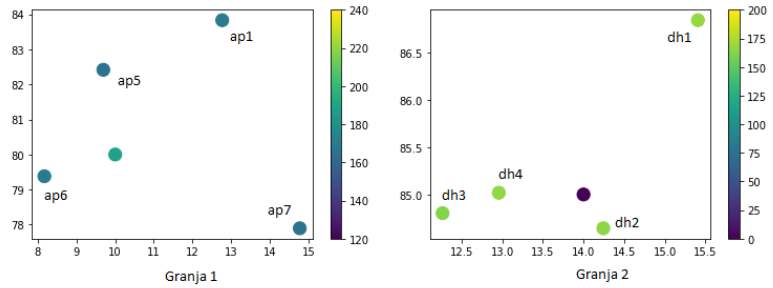

Figura 5.20: Punto calculado al aplicar el algoritmo de kriging

Con esto se demuestra que es posible integrar un algoritmo de kriging en el entorno implementado en este trabajo de fin de máster. Queda a elección del usuario la precisión del algoritmo a usar. El objetivo de este trabajo es demostrar la viabilidad de esta incorporación en el sistema. Entrando a valorar los resultados del algoritmo de kriging, para la granja 1 el punto incorporado tiene un valor similar al del resto de los sensores, por lo que se ha conseguido una medición adicional coherente en ese punto. Para el caso de la granja

Nombre del sensor	Valor de radiación
ap1	169.29
ap5	165.666
ap6	170.064
ap7	165.595
Kriging granja 1	189.065
dh1	168.741
dh2	166.141
dh3	162.881
dh4	166.57
Kriging granja 2	-0.8503

Tabla 5.4: Valores de radiación para la Figura 5.20

2, la medición obtenida no concuerda según lo marcado por el resto de sensores. Este resultado indica que el algoritmo no es perfecto y que la falta de arreglos que implementan este tipo de algoritmos causa la aparición de algún resultado erróneo. Los valores de los sensores de las figuras 5.19 y 5.20 son los que aparecen en la tabla 5.4.

Como desarrollo adicional, se ha implementado en el simulador un arreglo para los valores de kriging del algoritmo desarrollado que transforme los resultados que no son esperados a otros que entran dentro de unos parámetros más razonables. Esta corrección entra en juego cuando el resultado obtenido no está comprendido entre el valor medio del sensor con menor radiación y el valor medio del sensor con mayor radiación.

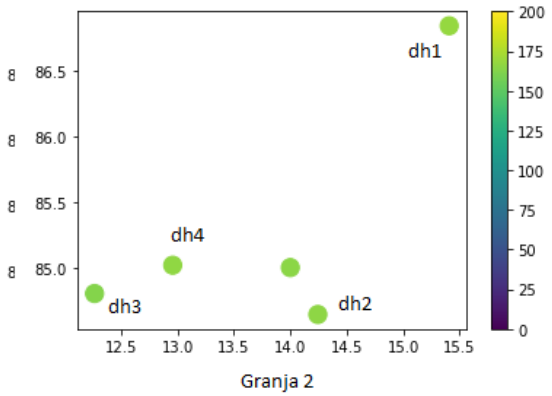

Figura 5.21: Punto de kriging corregido para la granja 2

En la figura 5.21 se muestra el resultado después de aplicar la corrección al algoritmo de kriging en la granja 2. Como se puede observar, todos los sensores y el punto adicional, representan ahora valores más coherentes.

Capítulo 6

Conclusiones y Trabajo Futuro

Este apartado incorpora las principales conclusiones obtenidas tras la elaboración de este trabajo junto con varias líneas de trabajo futuro.

6.1. Conclusiones

En este trabajo se ha desarrollado un entorno de simulación de nodos de radiación solar dentro de un contexto IoT donde se diferencian las tres capas típicas de su arquitectura: edge, fog y cloud, que se conectan entre sí. Para crear esta simulación, se ha utilizado el formalismo DEVS, que permite discernir entre los elementos de las distintas capas y establecer un intercambio de información guiado por eventos discretos.

Como se ha explicado durante la memoria, la capa edge está compuesta por nodos virtuales y ficheros, que se encargan de generar los valores de radiación y transmitirlos a las siguientes capas. La capa fog procesa estos valores. Finalmente, la capa fog envía los datos a la capa cloud. Al realizar este procesado se cubren varios de los objetivos inicialmente marcados, como son:

- La búsqueda de valores atípicos (*outliers*), se ha aplicado mediante el análisis de rangos intercuartiles, en función de un grupo de datos de entrada.
- La sustitución de los *outliers* se ha implementado aplicando distintos métodos de regresión como *Cubic Spline*, la función de base radial y *Shepard*. Esto permite reemplazar los datos generados erróneamente, en lugar de eliminarlos, lo que provocaría una pérdida de información.
- La incorporación de un algoritmo de *kriging* ordinario que permite calcular los valores de radiación de un punto cercano al resto de sensores en donde no haya mediciones. Se ha probado que es posible realizar este cálculo y que el resultado obtenido será más o menos preciso en función del algoritmo introducido en la simulación.

En cuanto a los objetivos para dotar al entorno de simulación de una parte de escalabilidad y facilidad a la hora de crear distintos escenarios, esta ha sido cubierta con la configuración de ficheros de variables de tipo *yaml*, que son empleados por *Ansible* en los distintos roles implementados para aplicar la correspondiente distribución de elementos en las capas del entorno de simulación IoT.

Por último, se ha llevado la simulación a la computación en la nube, en este caso a *Google Cloud* aunque es posible exportarlo a otro proveedor Cloud, ya que se ha seguido la

practica de "infraestructura como código" en donde la configuración Cloud y del entorno simulado está en los distintos archivos de *terraform* y *ansible*. Esto agiliza la puesta a punto de cualquier escenario simulado, ya que la automatización permite crear los recursos en la nube necesarios y configurarlos de forma desatendida.

6.2. Trabajo futuro

En este trabajo se han cubierto los objetivos inicialmente fijados, pero es posible continuar mejorando este entorno de simulación con otras metas que no han sido posible incluir en este, detallados a continuación:

Se han modelado mediante DEVS todos los elementos atómicos presentes en este trabajo a un nivel lógico en el que cada uno realiza su principal funcionalidad. Una línea de trabajo futuro pasaría por evolucionar el modelado del generador de datos de tal modo que sea más realista a nivel físico, por ejemplo con la implementación de gemelos digitales. Con esta mejora, los generadores serían más fieles a los propios sensores de radiación.

El entorno creado durante este trabajo incluye únicamente nodos simulados. Dotar al sistema de algún nodo físico aportaría más realismo y permitiría poder contrastar los valores transmitidos por unos u otros. Esto otorgaría la posibilidad de realizar estudios adicionales sobre ellos, por ejemplo para determinar la desviación de valores generados entre los distintos tipos de nodos.

Como se ha demostrado, es posible obtener con el algoritmo de *kriging* ordinario valores de otros puntos con falta de medición. Algunos de los cálculos obtenidos no son del todo acorde a lo esperado debido a la falta de arreglos en esta implementación, considerados fuera del ámbito de este trabajo final de máster. En caso de requerir una mayor precisión en los resultados obtenidos, se propone reemplazar el algoritmo desarrollado en la capa fog por otro que obtenga valores con mayor exactitud.

Chapter 7

Introduction

This chapter explains in first place the motivation that has led to this master's thesis, followed by the objetives to be covered and the work plan to follow in order to achieve the goals defined. Finally, a breif summary of the different chapters developed in this paper.

7.1. Motivation

In a globalised world where the generation and exchange of information is extremely high and it happens in a short period of time. Implementing a mechanism able to collect all the data generated in real time, in order to be precessed and exploited later, is a great challenge, mainly due to the complexity of these systems.

The Internet of Things proposes to connect any object to the internet network, following a well-defined pattern composed of three interconnected levels: edge, fog and cloud (Kumar et al., 2019). These objects generate measurements that can be used as metrics of interest. Until recently, it was irrelevant to know them as they did not provide much knowledge or insight into the problem. Nowadays, from all these data it is possible to extract some pattern or association that can be turned into a case of study to improve our lives or to monitor the natural environment. This has become possible due to current developments in fields such as *big data* or artificial intelligence (Elgendy y Elragal, 2014), which require massive data.

For instance, having multiple measurements of different points of the earth such as humidity, temperature, atmospheric pressure or solar radiation allows us to know the state of the entire globe and to have a history in which it can be seen the evolution during the different seasons or the previous years. A case of current observation is to study climate change and how all these measures have been changing in recent years. A case of current observation is the study of Climate Change and how all these measures have been changing over the last few years. Taking measurements allows not only to perform reactive actions, but also to build predictive models and to take more proactive decisions. This is quite ambitious, due to the large amount of information and the different areas of knowledge involved, which makes modelling a extremely complex task.

This project, in particular, focuses on the part of deployment, generation and processing of information on solar radiation values. There are different mechanisms for taking measurements. The most widespread is through small devices containing sensors that generate values that are sent to storage media (Joint Research Centre, 2020). It is proposed a method, based on modelling and simulation, that lets us to analyse a deployment of solar

radiation sensor nodes following the Internet of Things paradigm. Thus, it is designed an environment that helps us to simulate a deployment of data-generating sensors. These inputs are processed taking into account that electrical devices sometimes introduce errors that must be eliminated. It is also proposed different interpolation and regression methods that give the system greater versatility when calculating new measurements to replace erroneous values or to obtain measurements from locations where there is no sensor.

In addition, this simulation environment must be highly scalable depending on the number of connected elements and information circulating in the system. One solution to guarantee scalability is cloud computing, as it offers these characteristics.

7.2. Objectives

In order to accomplish this task, a series of objectives have been set. The main one is to design an IoT simulation environment containing the three layers: edge, fog and cloud. The edge layer generates solar radiation measurements, the fog layer processes and transforms the data from the previous layer; and the cloud stores all the information generated by the previous layers.

Along with the main objective of the project, other equally important goals are included to complement the simulation environment.

- Enable the system to identify atypical values (*outliers*) that are introduced by the generators of radiation measurements.
- Implement a replacement mechanism for outliers.
- Demonstrate the feasibility of obtaining radiation values of a point in space where no measurements are available from metrics of other nearby points with Kriging algorithms.
- Provide scalability to the simulation where it would be quick and easy to switch between different scenarios.
- Bringing the simulation to a cloud computing environment, where all configuration follows the emerging practice called 'infrastructure-as-code'.

7.3. Work plan

In order to achieve the objectives described in the previous section, a work plan has been defined which consists of the following phases:

- Preliminary study of the elements that must be present in the simulation scenario in each of the edge, fog and cloud layers. Carry out the creation of the same using an implementation of the DEVS formalism.
- Analysis and implementation of an algorithm for outlier detection and generation of replacement values for outliers using regression methods.
- Incorporate some Kriging algorithm, to see the feasibility of interpolating different values.

- Development of the necessary code that takes the simulation to the Cloud as code that automates the creation of objects in the cloud, their configuration and destruction when the simulation is finished.
- Compiling a report on the work carried out, with the results and conclusions drawn.

7.4. Memory structure

This document consists of six chapters. The broad outline of the content is as follows:

- **Chapter 1** corresponds to the introduction to the subject of the paper together with the objectives set and the work plan defined to address it.
- **Chapter 2** includes the analysis of the state of the art together with most of the bibliographies consulted. Current IoT architectures will be compared with the one defined in this work.
- **Chapter 3** incorporates the design of a system to be developed following the DEVS formalism based on an IoT context together with the deployment in the Cloud. Finally, it is also included the mathematical calculations and formulas necessary to implement the different algorithms.
- **Chapter 4** contains the main part of the work focused on the implementation of the system. It details all the elements present to carry it out and the relationships between them. Finally, it closes with the development that has been carried out for the construction of the Cloud architecture and configuration management as code.
- **Chapter 5** shows the results obtained from this work. It is detailed the steps to run the simulation and it is also analysed the results of the simulation with the detection of outliers, their substitution and the results of applying kriging.
- **Chapter 6** summarises the work with the conclusions drawn from it, assessing the objectives previously set. To conclude, several lines of work are proposed.

Chapter 8

Conclusions and Future Work

This section contains the main conclusions obtained after the elaboration of this paper and for future research.

8.1. Conclusions

In this approach, a simulation environment of solar radiation nodes has been developed within an IoT context where the three typical layers of its architecture are differentiated: edge, fog and cloud, which are connected to each other. In order to carry out this simulation, the DEVS formalism was used, which makes it possible to differentiate between the elements in the different layers and establishes an exchange of information guided by discrete events.

As explained during this work, the edge layer is composed of virtual nodes and files, which are responsible of generating radiation values and transmitting them to the following layers. The fog layer processes these values during which several of the initially set objectives are covered, such as:

- The search for outliers has been applied through the analysis of interquartile ranges, based on a group of input data.
- The substitution of outliers has been implemented by applying different regression methods such as *Cubic Spline*, the radial basis function and *Shepard*. This allows erroneously generated data to be replaced, rather than deleted.
- The development of an ordinary *kriging* algorithm that allows the calculation of radiation values from a point close to the rest of the sensors. It has been proven that this calculation is possible and that the result obtained will be more or less accurate depending on the algorithm introduced in the simulation.

The fog layer sends the data to the cloud layer. Regarding the provided objectives in the simulation environment with part of scalability and ease when creating different scenarios, it has been covered with the configuration of variable files of type *yaml* that are used by *Ansible* in the different roles implemented to apply the corresponding distribution of elements in the layers of the IoT simulation environment.

Finally, the simulation has been taken to cloud computing, in this case to *Google Cloud*, although it is equally viable to export it to another cloud provider, as the practice of

"infrastructure as code" has been followed, where the cloud configuration and the simulated environment is made up of different *terraform* and *ansible* files. This speeds up the setup of any simulated scenario, as automation allows the necessary cloud resources to be created and configured remotely.

8.2. Future Work

This work has covered the objectives initially set, but it is possible to continue improving this simulation environment with other challenges that have not been possible to include in this study, as detailed below:

All the atomic elements present in this work have been modelled by DEVS at a logical level where each one performs its main functionality. One future direction of work would be to evolve the modelling of the data generator in such a way that it is more physically realistic, for example by implementing digital twins. With this improvement, the generators would be more faithful to the radiation sensors themselves.

The environment created during this essay includes only simulated nodes. Providing the system with a physical node would provide more realism and would allow the values transmitted by one or the other to be contrasted. This would make it possible to carry out additional studies on them, for example to determine the deviation of values generated between the different types of nodes.

As demonstrated, it is possible to obtain from other points with missing measurement through the ordinary *kriging* algorithm. Some of the results obtained are not quite in line with expectations due to the lack of arrays in this implementation, considered outside the scope of this master's thesis. In the case of requiring greater precision in the results obtained, it is proposed to replace the algorithm developed in the fog layer with another one that obtains values with greater accuracy.

Bibliografía

ACIMS. DEVSJAVA. 2009. Disponible en `https://acims.asu.edu/software/devsjava/` (último acceso, Junio, 2021).

Aggarwal, C. C. *Outlier analysis*. Springer, 2017.

Alsamamra, H., Ruiz-Arias, J. A., Pozo-Vázquez, D. y Tovar-Pescador, J. *A comparative study of ordinary and residual kriging techniques for mapping global solar radiation over southern Spain.*. Agricultural and Forest Meteorology, 149(8), 1343–1357, 2009. Disponible en `https://doi.org/10.1016/j.agrformet.2009.03.005` (último acceso, Enero, 2021).

Amazon. *AWS CloudFormation*. Versión electrónica, 2020a. Disponible en `https://aws.amazon.com/es/cloudformation/` (último acceso, Enero, 2021).

Amazon. *Kit de desarrollo de la nube de AWS*. Versión electrónica, 2020b. Disponible en `https://aws.amazon.com/es/cdk/` (último acceso, Enero, 2021).

Córdoba, M. *Geoestadística*. 2015. ISBN 978-987-591-646-3.

Eberly, S., Swall, J., Holland, D., Cox, B. y Baldridge, E. *A comparison of some kriging interpolation methods for the production of solar radiation maps.*. Scientific and Technical Research Series, 4(October), 169, 2005. Disponible en `https://www.mendeley.com/catalogue/7cb03183-ff06-3cb8-93d2-2b4bbc4af818/` (último acceso, Enero, 2021).

Elgendy, N. y Elragal, A. Big data analytics: A literature review paper. vol. 8557, páginas 214–227. 2014. ISBN 978-3-319-08975-1.

Fasshauser, G. E. Meshfree approximation methods with Matlab. 2017.

Google. *Cloud Storage*. Versión electrónica, 2020a. Disponible en `https://cloud.google.com/storage?hl=es-419` (último acceso, Enero, 2021).

Google. *Precio de instancias de VM*. Versión electrónica, 2020b. Disponible en `https://cloud.google.com/compute/vm-instance-pricing#e2_predefined` (último acceso, Diciembre, 2020).

Google. *Cloud Storage*. Versión electrónica, 2021. Disponible en `https://cloud.google.com/storage/docs/gsutil` (último acceso, Enero, 2021).

HashiCorp. *Terraform by HashiCorp*. Versión electrónica, 2018. Disponible en https://www.terraform.io/ (último acceso, Diciembre, 2020).

Homebrew. Homebrew. 2021. Disponible en https://docs.brew.sh/Homebrew-on-Linux (último acceso, Enero, 2021).

Javadox. Package smile.interpolation. 2013. Disponible en http://javadox.com/com.github.haifengl/smile-interpolation/1.5.1/smile/interpolation/package-summary.html (último acceso, Noviembre, 2020).

Joe Hicklin, P. W., Cleve Moler. JAMA: A java matrix package. 2012. Disponible en https://math.nist.gov/javanumerics/jama/ (último acceso, Febrero, 2021).

Joint Research Centre. Radiación solar, fuente de datos y métodos de cálculo. 2020. Disponible en https://ec.europa.eu/jrc/en/PVGIS/docs/methods (último acceso, Enero, 2021).

Kanies, L. *Puppet*. Versión electrónica, 2020. Disponible en https://puppet.com/ (último acceso, Enero, 2021).

Kraemer, F. A., Ammar, D., Braten, A. E., Tamkittikhun, N. y Palma, D. *Solar Energy Prediction for Constrained IoT Nodes Based on Public Weather Forecasts*. Department of Information Security and Communication Technology NTNU, Norwegian University of Science and Technology Trondheim, Norway, 2017. Disponible en https://www.mendeley.com/catalogue/30c86b3c-69a2-3f93-89b8-e9cba2224c1b/ (último acceso, Febrero, 2021).

Kumar, S., Tiwari, P. y Zymbler, M. *Internet of Things is a revolutionary approach for future technology enhancement: a review*. Versión electrónica, 2019. Disponible en https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0268-2 (último acceso, Febrero, 2021).

Malik, K., H.Sadawarti, IEEE, M. y G.S., K. *Comparative Analysis of Outlier Detection Techniques*. Versión electrónica, 2014. Disponible en https://www.researchgate.net/publication/269802647_Comparative_Analysis_of_Outlier_Detection_Techniques (último acceso, Enero, 2021).

Mathwords. Outlier. 2017. Disponible en http://www.mathwords.com/o/outlier.htm (último acceso, Noviembre, 2020).

MIDC. Oahu solar measurement grid, oahu ghi data files. 2012. Disponible en https://midcdmz.nrel.gov/apps/rawdata.pl?site=oahugrid;data=Oahu_GHI;type=zipdata (último acceso, Noviembre, 2020).

Nutaro, J. aDEVS. 2010. Disponible en https://web.ornl.gov/~nutarojj/adevs/ (último acceso, Junio, 2021).

Ohiou. Polynomial and spline interpolation. 2019. Disponible en http://www.math.ohiou.edu/courses/math3600/lecture19.pdf (último acceso, Noviembre, 2020).

Peachey, M. *tfenv*. Versión electrónica, 2020. Disponible en https://github.com/tfutils/tfenv (último acceso, Enero, 2021).

RedHat. *Ansible*. Versión electrónica, 2020a. Disponible en https://www.ansible.com/ (último acceso, Enero, 2021).

RedHat. *apt – Manages apt-packages*. Versión electrónica, 2020b. Disponible en `https://docs.ansible.com/ansible/2.9/modules/apt_module.html` (último acceso, Enero, 2021).

RedHat. *command – Execute commands on targets*. Versión electrónica, 2020c. Disponible en `https://docs.ansible.com/ansible/2.9/modules/command_module.html` (último acceso, Enero, 2021).

RedHat. *file – Manage files and file properties*. Versión electrónica, 2020d. Disponible en `https://docs.ansible.com/ansible/2.9/modules/file_module.html` (último acceso, Enero, 2021).

RedHat. *Gather_facts Ansible*. Versión electrónica, 2020e. Disponible en `https://docs.ansible.com/ansible/2.9/modules/gather_facts_module.html` (último acceso, Enero, 2021).

RedHat. *git – Deploy software (or files) from git checkouts*. Versión electrónica, 2020f. Disponible en `https://docs.ansible.com/ansible/2.9/modules/git_module.html` (último acceso, Enero, 2021).

RedHat. *lineinfile – Manage lines in text files*. Versión electrónica, 2020g. Disponible en `https://docs.ansible.com/ansible/2.9/modules/lineinfile_module.html` (último acceso, Enero, 2021).

RedHat. *pip – Manages Python library dependencies*. Versión electrónica, 2020h. Disponible en `https://docs.ansible.com/ansible/2.9/modules/pip_module.html` (último acceso, Enero, 2021).

RedHat. *Registering variables*. Versión electrónica, 2020i. Disponible en `https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#registering-variables` (último acceso, Enero, 2021).

RedHat. *roles*. Versión electrónica, 2020j. Disponible en `https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html` (último acceso, Enero, 2021).

RedHat. *shell – Execute shell commands on targets*. Versión electrónica, 2020k. Disponible en `https://docs.ansible.com/ansible/2.9/modules/shell_module.html` (último acceso, Enero, 2021).

Risco, J. xDEVS. 2020. Disponible en `https://github.com/iscar-ucm/xdevs/tree/master` (último acceso, Mayo, 2021).

Ryu, J.-S., Kim, M., Cha, K., Lee, T. y Choi, D.-H. Kriging interpolation methods in geostatistics and dace model. *KSME International Journal*, vol. 16, páginas 619–632, 2002.

Senapathi, M., Buchan, J. y Osman, H. Devops capabilities, practices, and challenges: Insights from a case study. 2019. Disponible en `https://arxiv.org/ftp/arxiv/papers/1907/1907.10201.pdf` (último acceso, Junio, 2021).

Shepard, D. A two-dimensional interpolation function for irregularly-spaced data. 1968. Disponible en `https://dl.acm.org/doi/10.1145/800186.810616` (último acceso, Noviembre, 2020).

Sorge, A. Pydevs. 2015. Disponible en `https://pydevs.readthedocs.io/en/latest/` (último acceso, Junio, 2021).

Xu, H., Wang, Y., Wang, Y. y Wu, Z. *MIX: A Joint Learning Framework for Detecting Both Clustered and Scattered Outliers in Mixed-Type Data*. Versión electrónica, 2019. Disponible en `https://www.researchgate.net/publication/338947470_MIX_A_Joint_Learning_Framework_for_Detecting_Both_Clustered_and_Scattered_Outliers_in_Mixed-Type_Data` (último acceso, Enero, 2021).

Zeigler, B., Prähofer, H. y Kim, T. G. Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems. vol. 2, 2000.

Apéndice A

Documentación técnica de las clases Java desarrolladas

A.1. Input

El constructor de esta clase es el siguiente:

```
Input(String date, double radiacion, String generador)
```

Está compuesto por una fecha que indica el momento en el que se tomó la medición de tipo *String*, la medida de radiación solar de tipo *double* y el identificador del sensor que ha tomado la medición. Las funciones implementadas son las siguientes:

- double getRadiacion(): Función que devuelve el valor de la radiación.
- void setRadiacion(double radiacion): Función a la que se ha de pasar un valor de tipo *double* y modifica la radiación del objeto.
- String getGenerador(): Función que devuelve el identificador del sensor que ha producido la medición.
- void setGenerador(String generador): Función que recibe un identificador de sensor y lo modifica por este.
- String getDate(): Función que devuelve la fecha cuando fue tomada la medición.
- void setDate(String date): Función que recibe una fecha y la cambia por la que se pasa como parámetro.
- String toString(): Función que transforma el objeto a *String*.
- String toCSV(): Función que transforma el objeto a formato CSV.

A.2. Ficheros

Para la creación de estos objetos se tiene que invocar a su constructor que es el siguiente:

```
Ficheros(String name, double period, String path,
        String startDate, String endDate)
```

Los parámetros que figuran en el constructor son los siguientes:

- name: Identificador para el objeto *Fichero* creado.
- period: Valor que indica el periodo de tiempo para ejecutar otra iteración de la simulación.
- path: Directorio en el cual se encuentran los ficheros en formato csv que se introducirán en el sistema. Los archivos son procesados de menos a mayor, por tanto los que en el nombre tengan una fecha o número menor, serán procesados primero.
- startDate: Fecha inicial sobre la cual se quiere introducir datos en el sistema. Todas las mediciones con *timestamp* menor a este valor, serán saltados. El formato debe ser el mismo que aparece en los ficheros csv (YYYY-MM-DD HH:MM:SS-10:00).
- endDate: Fecha final hasta la cual se introducen datos en el sistema. Todas las mediciones que estén fechadas con un valor posterior a este parámetro, serán saltadas. El formato debe ser el mismo que se menciona anteriormente

Esta clase al heredar de *Atomic*, cuenta con todas las funciones padres. En la tabla A.1 se muestra el listado de todas ellas y su funcionalidad ya que han sido sobrescritas con la etiqueta *@Override*.

Por último, se han implementado dos funciones adicionales. La primera de ellas, *parseDate(String sDate)*, recibe una fecha de tipo *String* y la transforma a tipo *Date* siguiendo el formato *yyyy-MM-dd HH:mm:ss*. Se utiliza tanto la función *split()* como la clase *SimpleDateFormat* de la librería *java.text*. Esta función es invocada por el constructor, *deltint()* y *initialize()*.

Nombre de la función	***Funcionalidad***
initialize()	Se ejecuta al inicial por primera vez en objeto. Se encarga de abrir un BufferReader para leer los ficheros csv pasados en el *path* del constructor. Lee el primer valor del fichero y lo envía al puerto de salida tras comprobar los rangos de fecha (si la comprobación no aplica su envió, termina la ejecución de la función).
deltint()	Lee la siguiente línea del fichero. En caso de que no exista, se abre el siguiente fichero y se lee la primera línea. En caso de no haber más ficheros ni lineas se llama a la función *passivate()*. Cuando se lee una linea válida, se comprueba que que puede ser enviado en función de las fechas y del periodo establecido.
deltext(double e)	No implementa funcionalidad adicional.
lambda()	Guarda en el puerto de salida el objeto de tipo *Input* a enviar al *NodoVirtual*.
toString()	Transforma el objetos Ficheros a tipo String.
exit()	Cierra los buffer de lectura de los ficheros.

Tabla A.1: Implementación de las funciones heredadas por la clase Atomic en objetos Ficheros

La segunda función implementada es *nextFile()* que devuelve un booleano indicando si existen más ficheros para leer o no. Se devuelve verdadero en caso de existan y falso en

el otro caso. Este método es llamado por *deltint()* para determinar si la simulación puede continuar o ha finalizado la generación de mediciones.

A.3. NodoVirtual

Para la creación de estos objetos se tiene que invocar el siguiente constructor, junto con los parámetros que se explican a continuación:

```
NodoVirtual(String name, double processingTime)
```

- name: Identificador para el objeto *NodoVirtual*.
- processingTime: Valor que indica el periodo de tiempo para ejecutar otra iteración de la simulación.

Al ser una clase que hereda de *Atomic*, se tienen las funciones de esta. En la tabla A.2 se especifican la funcionalidad que implementan.

Nombre de la función	***Funcionalidad***
initialize()	No implementa funcionalidad adicional.
deltint()	No implementa funcionalidad adicional.
deltext(double e)	Lee el dato *Input* enviado desde el objeto *Fichero*.
lambda()	Envía por el puerto de salida la medición leída.
exit()	No implementa funcionalidad adicional.

Tabla A.2: Implementación de las funciones heredadas por la clase Atomic en objetos NodoVirtual

A.4. FogServer

Para invocar la creación de objetos *FogServer* se emplea el siguiente constructor:

```
FogServer(String name, double processingTime, double krigingX,
double krigingY, List<Double> coordenadasX, List<Double> coordenadasY)
```

- name: Identificador para el objeto *FogServer*.
- processingTime: Valor que indica el periodo de tiempo para ejecutar otra iteración de la simulación.
- krigingX: Coordenada del punto X donde se quiere interpolar la radiación.
- krigingY: Coordenada del punto Y donde se quiere interpolar la radiación.
- coordenadasX: Lista de coordenadas del punto X donde se encuentran los sensores.
- coordenadasY: Lista de coordenadas del punto X donde se encuentran los sensores.

Al ser una clase que hereda de *Atomic*, se tienen las funciones de esta. En la tabla A.3 se especifican la funcionalidad que implementan.

Nombre de la función	***Funcionalidad***
initialize()	No implementa funcionalidad adicional.
deltint()	Explora los datos recibidos para localizar valores atípicos y sustituirlos con las técnicas de regresión.
deltext(double e)	Lee el dato *Input* enviado desde el objeto *NodoVirtual* y realiza el calculo de kriging cuando se tienen suficientes datos.
lambda()	Envía por el puerto de salida todos los datos recogidos y generados.
exit()	No implementa funcionalidad adicional.

Tabla A.3: Implementación de las funciones heredadas por la clase Atomic en objetos FogServer

A.5. DataCenter

Para invocar la creación de objetos *DataCenter* se emplea el siguiente constructor:

```
DataCenter(String name, double processingTime)
```

- name: Identificador para el objeto *FogServer*.
- processingTime: Valor que indica el periodo de tiempo para ejecutar otra iteración de la simulación.

Al ser una clase que hereda de *Atomic*, se tienen las funciones de esta. En la tabla A.4 se especifican la funcionalidad que implementan.

Nombre de la función	***Funcionalidad***
initialize()	No implementa funcionalidad adicional.
deltint()	No implementa funcionalidad adicional.
deltext(double e)	Lee los datos *Input* enviados desde los *FogServer* y genera o actualiza el fichero de salida.
lambda()	No implementa funcionalidad adicional.
exit()	No implementa funcionalidad adicional.

Tabla A.4: Implementación de las funciones heredadas por la clase Atomic en objetos DataCenter

Apéndice B

Output de la ejecución

B.1. Terraform apply

A continuación se muestra la traza devuelta por terraform al crear un entorno de simulación. Como se ve en la traza, se va a crear un recurso del tipo `google_compute_instance` con identificador `simulator`. El resultado de la ejecución es satisfactorio y tarda 14 segundos en completar la creación de este recurso.

```
terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # google_compute_instance.simulator will be created
  + resource "google_compute_instance" "simulator" {
      + can_ip_forward       = false
      + cpu_platform         = (known after apply)
      + current_status       = (known after apply)
      + deletion_protection  = false
      + guest_accelerator    = (known after apply)
      + id                   = (known after apply)
      + instance_id          = (known after apply)
      + label_fingerprint    = (known after apply)
      + machine_type         = "e2-medium"
      + metadata_fingerprint = (known after apply)
      + min_cpu_platform     = (known after apply)
      + name                 = "compute-simulator"
      + project              = (known after apply)
      + self_link            = (known after apply)
      + tags                 = [
          + "project",
          + "tmf-irt",
        ]
```

```
      + tags_fingerprint     = (known after apply)
      + zone                 = "europe-west4-a"

      + boot_disk {
          + auto_delete                = true
          + device_name                = (known after apply)
          + disk_encryption_key_sha256 = (known after apply)
          + kms_key_self_link          = (known after apply)
          + mode                       = "READ_WRITE"
          + source                     = (known after apply)

          + initialize_params {
              + image  = "debian-cloud/debian-9"
              + labels = (known after apply)
              + size   = 50
              + type   = (known after apply)
            }
        }

      + network_interface {
          + name               = (known after apply)
          + network            = "default"
          + network_ip         = (known after apply)
          + subnetwork         = (known after apply)
          + subnetwork_project = (known after apply)

          + access_config {
              + nat_ip       = (known after apply)
              + network_tier = (known after apply)
            }
        }

      + scheduling {
          + automatic_restart   = (known after apply)
          + on_host_maintenance = (known after apply)
          + preemptible         = (known after apply)

          + node_affinities {
              + key      = (known after apply)
              + operator = (known after apply)
              + values   = (known after apply)
            }
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
```

```
  Only 'yes' will be accepted to approve.

  Enter a value: yes

google_compute_instance.simulator: Creating...
google_compute_instance.simulator: Still creating... [10s elapsed]
google_compute_instance.simulator: Creation complete after 14s [id=...]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

B.2. Terraform state

Este código corresponde con el estado que almacena Terraform para conocer el estado de los objetos Cloud creados en la nube y poder conocer que cambios hacer sobre ellos en caso de una modificación. El estado que se muestra a continuación corresponde con el escenario de simulación del apartado de resultados, una vez que se han creado todos los recursos.

```
"version": 4,
"terraform_version": "0.13.5",
"serial": 18,
"lineage": "ce58bf28-dec1-6f0f-cb23-26df3ae4bff6",
"outputs": {},
"resources": [
  {
    "mode": "managed",
    "type": "google_compute_instance",
    "name": "simulator",
    "provider": "provider[\"registry.terraform.io/hashicorp/google\"]",
    "instances": [
      {
        "schema_version": 6,
        "attributes": {
          "allow_stopping_for_update": null,
          "attached_disk": [],
          "boot_disk": [
            {
              "auto_delete": true,
              "device_name": "persistent-disk-0",
              "disk_encryption_key_raw": "",
              "disk_encryption_key_sha256": "",
              "initialize_params": [
                {
                  "image": "https://.../debian-9-stretch-v20210512",
                  "labels": {},
                  "size": 50,
```

```
        "type": "pd-standard"
      }
    ],
    "kms_key_self_link": "",
    "mode": "READ_WRITE",
    "source": "https://www.googleapis.com/compute/v1/..."
  }
],
"can_ip_forward": false,
"cpu_platform": "Intel Skylake",
"current_status": "RUNNING",
"deletion_protection": false,
"description": "",
"desired_status": null,
"enable_display": false,
"guest_accelerator": [],
"hostname": "",
"id": "projects/skillful-hope-295119/.../compute-simulator",
"instance_id": "219468004471524576",
"label_fingerprint": "42WmSpB8rSM=",
"labels": null,
"machine_type": "e2-medium",
"metadata": null,
"metadata_fingerprint": "6oOeW0lcpM8=",
"metadata_startup_script": "",
"min_cpu_platform": "",
"name": "compute-simulator",
"network_interface": [
  {
    "access_config": [
      {
        "nat_ip": "34.141.183.148",
        "network_tier": "PREMIUM",
        "public_ptr_domain_name": ""
      }
    ],
    "alias_ip_range": [],
    "name": "nic0",
    "network": "https://www.googleapis.com/compute/v1/...",
    "network_ip": "10.164.0.2",
    "subnetwork": "https://www.googleapis.com/compute/v1/...",
    "subnetwork_project": "skillful-hope-295119"
  }
],
"project": "skillful-hope-295119",
"resource_policies": null,
"scheduling": [
  {
    "automatic_restart": true,
```

```
                "node_affinities": [],
                "on_host_maintenance": "MIGRATE",
                "preemptible": false
              }
            ],
            "scratch_disk": [],
            "self_link": "https://www.googleapis.com/compute/v1/...",
            "service_account": [],
            "shielded_instance_config": [],
            "tags": [
              "project",
              "tmf-irt"
            ],
            "tags_fingerprint": "RczWDx3ODYI=",
            "timeouts": null,
            "zone": "europe-west4-a"
          },
          "private": "..."
        }
      ]
    }
  ]
}
```


B.3. Fichero de variables del escenario de resultados

En esta sección se encuentran definidas las variables para el escenario de ejemplo utilizado para la obtención de los resultados.

```
template_path: ~/simuladorIoT/src/xdevs/core/efp
gpt_path: ~/simuladorIoT/src/xdevs/core/efp/Gpt.java
fog_server_path: ~/simuladorIoT/src/xdevs/core/efp/FogServer.java
data_center_path: ~/simuladorIoT/src/xdevs/core/efp/DataCenter.java
ficheros:
  - name: ap1
    startDate: "null"
    endDate: "null"
  - name: ap5
    startDate: "null"
    endDate: "null"
  - name: ap6
    startDate: "null"
    endDate: "null"
  - name: ap7
    startDate: "null"
    endDate: "null"
  - name: dh1
    startDate: "null"
    endDate: "null"
```

```
  - name: dh2
    startDate: "null"
    endDate: "null"
  - name: dh3
    startDate: "null"
    endDate: "null"
  - name: dh4
    startDate: "null"
    endDate: "null"
nodo_virtual:
  - nodoVirtual1
  - nodoVirtual2
  - nodoVirtual3
  - nodoVirtual4
  - nodoVirtual5
  - nodoVirtual6
  - nodoVirtual7
  - nodoVirtual8
fog_server:
  - name: fogserver1
    krigingX: 10.0
    krigingY: 80.0
    coordenadasNodosX: 12.771, 9.692, 8.165, 14.779
    coordenadasNodosY: 83.829, 82.415, 79.381, 77.897
  - name: fogserver2
    krigingX: 14.0
    krigingY: 85.0
    coordenadasNodosX: 15.409, 14.244, 12.264, 12.957
    coordenadasNodosY: 86.845, 84.642, 84.802, 85.018
data_center:
  - dataCenter
conectores:
  - out_type: ficheros
    out: ap1
    in: nodoVirtual1
  - out_type: ficheros
    out: ap5
    in: nodoVirtual2
  - out_type: ficheros
    out: ap6
    in: nodoVirtual3
  - out_type: ficheros
    out: ap7
    in: nodoVirtual4
  - out_type: ficheros
    out: dh1
    in: nodoVirtual5
  - out_type: ficheros
    out: dh2
```

```
    in: nodoVirtual6
  - out_type: ficheros
    out: dh3
    in: nodoVirtual7
  - out_type: ficheros
    out: dh4
    in: nodoVirtual8
  - out_type: nodoVirtual
    out: nodoVirtual1
    in: fogserver1
  - out_type: nodoVirtual
    out: nodoVirtual2
    in: fogserver1
  - out_type: nodoVirtual
    out: nodoVirtual3
    in: fogserver1
  - out_type: nodoVirtual
    out: nodoVirtual4
    in: fogserver1
  - out_type: nodoVirtual
    out: nodoVirtual5
    in: fogserver2
  - out_type: nodoVirtual
    out: nodoVirtual6
    in: fogserver2
  - out_type: nodoVirtual
    out: nodoVirtual7
    in: fogserver2
  - out_type: nodoVirtual
    out: nodoVirtual8
    in: fogserver2
  - out_type: fogServer
    out: fogserver1
    in: dataCenter
  - out_type: fogServer
    out: fogserver2
    in: dataCenter
deltext:
  - nodoVirtual1
  - nodoVirtual2
  - nodoVirtual3
  - nodoVirtual4
  - nodoVirtual5
  - nodoVirtual6
  - nodoVirtual7
  - nodoVirtual8
```

B.4. Ansible output

Se muestra la salida de log de ejecutar *ansibleplaybook i inventory configure_ app.yml* para el ejemplo de simulación de la sección de resultados. Como se observa se ejecuta sobre el equipo con ip 34.141.183.148 y según las acciones que el módulo ansible detecte, aplicará el cambio, no hará nada o saltará la tarea. En el log, respectivamente equivale a *changed*, *ok* o *skipping*.

```
$ ansible-playbook -i inventory configure_app.yml

PLAY [remote] ************************************************************

TASK [configure_host : install maven (and other packages if needed)] ******
changed: [34.141.183.148] => (item=maven)
changed: [34.141.183.148] => (item=git)
changed: [34.141.183.148] => (item=python3-pip)
ok: [34.141.183.148] => (item=python3-setuptools)
changed: [34.141.183.148] => (item=openjdk-11-jre-headless)

TASK [configure_host : Install pip dependencies] **************************
changed: [34.141.183.148] => (item=google-auth==1.23.0)

TASK [configure_host : Set JAVA_HOME path] ********************************
changed: [34.141.183.148]

TASK [configure_host : Set PATH] ******************************************
changed: [34.141.183.148]

TASK [download_application : Delete previous app version] *****************
ok: [34.141.183.148]

TASK [download_application : Clone a application repository] **************
changed: [34.141.183.148]

TASK [download_application : Debug git] ***********************************
ok: [34.141.183.148] => {
    "git_download": {
        "after": "ce7e641b555edda44c8ffa1dc9ee09c7077f913b",
        "before": null,
        "changed": true,
        "failed": false
    }
}

TASK [xdevs_configuration : configure ficheros] ***************************
changed: [34.141.183.148] => (item={'name': 'dh4', 'startDate':...})
changed: [34.141.183.148] => (item={'name': 'dh3', 'startDate':...})
changed: [34.141.183.148] => (item={'name': 'dh2', 'startDate':...})
changed: [34.141.183.148] => (item={'name': 'dh1', 'startDate':...})
changed: [34.141.183.148] => (item={'name': 'ap7', 'startDate':...})
```

```
changed: [34.141.183.148] => (item={'name': 'ap6', 'startDate':...})
changed: [34.141.183.148] => (item={'name': 'ap5', 'startDate':...})
changed: [34.141.183.148] => (item={'name': 'ap1', 'startDate':...})

TASK [xdevs_configuration : configure nodos virtuales] ********************
changed: [34.141.183.148] => (item=nodoVirtual8)
changed: [34.141.183.148] => (item=nodoVirtual7)
changed: [34.141.183.148] => (item=nodoVirtual6)
changed: [34.141.183.148] => (item=nodoVirtual5)
changed: [34.141.183.148] => (item=nodoVirtual4)
changed: [34.141.183.148] => (item=nodoVirtual3)
changed: [34.141.183.148] => (item=nodoVirtual2)
changed: [34.141.183.148] => (item=nodoVirtual1)

TASK [xdevs_configuration : configure fog servers] ************************
changed: [34.141.183.148] => (item={'name': 'fogserver2', 'krigingX':...})
changed: [34.141.183.148] => (item={'name': 'fogserver1', 'krigingX':...})

TASK [xdevs_configuration : configure data center] ************************
changed: [34.141.183.148] => (item=dataCenter)

TASK [xdevs_configuration : configure conectores fogServer] ***************
changed: [34.141.183.148] => (item={'out_type': 'fogServer', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'fogServer', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})

TASK [xdevs_configuration : configure conectores NodoVirtual] *************
skipping: [34.141.183.148] => (item={'out_type': 'fogServer', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'fogServer', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
```

```
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})

TASK [xdevs_configuration : configure conectores ficheros] ****************
skipping: [34.141.183.148] => (item={'out_type': 'fogServer', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'fogServer', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})

TASK [xdevs_configuration : configure fogserver.java Port initialize] *****
skipping: [34.141.183.148] => (item={'out_type': 'fogServer', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'fogServer', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
```

```
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})

TASK [xdevs_configuration : configure fogserver.java kriging initialize] **
changed: [34.141.183.148] => (item=nodoVirtual8)
changed: [34.141.183.148] => (item=nodoVirtual7)
changed: [34.141.183.148] => (item=nodoVirtual6)
changed: [34.141.183.148] => (item=nodoVirtual5)
changed: [34.141.183.148] => (item=nodoVirtual4)
changed: [34.141.183.148] => (item=nodoVirtual3)
changed: [34.141.183.148] => (item=nodoVirtual2)
changed: [34.141.183.148] => (item=nodoVirtual1)

TASK [xdevs_configuration : configure fogserver.java constructor] *********
skipping: [34.141.183.148] => (item={'out_type': 'fogServer', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'fogServer', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
changed: [34.141.183.148] => (item={'out_type': 'nodoVirtual', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})
skipping: [34.141.183.148] => (item={'out_type': 'ficheros', 'out':...})

TASK [xdevs_configuration : configure fogserver.java deltext] *************
changed: [34.141.183.148] => (item=nodoVirtual8)
changed: [34.141.183.148] => (item=nodoVirtual7)
changed: [34.141.183.148] => (item=nodoVirtual6)
changed: [34.141.183.148] => (item=nodoVirtual5)
changed: [34.141.183.148] => (item=nodoVirtual4)
changed: [34.141.183.148] => (item=nodoVirtual3)
changed: [34.141.183.148] => (item=nodoVirtual2)
changed: [34.141.183.148] => (item=nodoVirtual1)

TASK [xdevs_configuration : configure fogserver.java deltext] *************
changed: [34.141.183.148] => (item=nodoVirtual8)
changed: [34.141.183.148] => (item=nodoVirtual7)
changed: [34.141.183.148] => (item=nodoVirtual6)
changed: [34.141.183.148] => (item=nodoVirtual5)
changed: [34.141.183.148] => (item=nodoVirtual4)
```

```
changed: [34.141.183.148] => (item=nodoVirtual3)
changed: [34.141.183.148] => (item=nodoVirtual2)
changed: [34.141.183.148] => (item=nodoVirtual1)

TASK [xdevs_configuration : configure dataCenter.java Port initialize] ****
changed: [34.141.183.148] => (item={'name': 'fogserver2', 'krigingX':...})
changed: [34.141.183.148] => (item={'name': 'fogserver1', 'krigingX':...})

TASK [xdevs_configuration : configure dataCenter.java constructor] ********
changed: [34.141.183.148] => (item={'name': 'fogserver2', 'krigingX':...})
changed: [34.141.183.148] => (item={'name': 'fogserver1', 'krigingX':...})

TASK [xdevs_configuration : configure dataCenter.java deltext] ************
changed: [34.141.183.148] => (item={'name': 'fogserver2', 'krigingX':...})
changed: [34.141.183.148] => (item={'name': 'fogserver1', 'krigingX':...})

TASK [start_app : Create .jar] ********************************************
changed: [34.141.183.148]

TASK [start_app : Run main class] *****************************************
changed: [34.141.183.148]

PLAY RECAP ****************************************************************
34.141.183.148             : ok=23   changed=22
unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```