Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

DESARROLLO DE LA PLATAFORMA DE SEGMENTO EN TIERRA DEL SATELITE UPMSAT-2 BASADA EN COREOS

TRABAJO FIN DE MÁSTER

Adrián Correa Guerrero

2018

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

Máster Universitario en
Ingeniería de Redes y Servicios Telemáticos

TRABAJO FIN DE MÁSTER

DESARROLLO DE LA PLATAFORMA DE SEGMENTO EN TIERRA DEL SATELITE UPMSAT-2 BASADA EN COREOS

Autor
Adrián Correa Guerrero

Director
Alejandro Alonso Muñoz

Departamento de Ingeniería de Sistemas Telemáticos

2018

Resumen

Este proyecto tiene como objetivo la investigación, diseño e implementación de una plataforma funcional para almacenar y desplegar la información que será obtenida por el satélite UPMSat-2.

El satélite UPMSat-2 es un microsatélite que sirve de plataforma de demostración y validación de diversos dispositivos y experimentos. Toda la información proporcionada por el satélite debe ser almacenada en la plataforma terrestre con la finalidad de ser administrados y controlados por el usuario mediante una interfaz web.

Desde un ámbito general, se da a conocer la arquitectura y los componentes del satélite UPMSat-2. La comunicación entre el sistema que se encuentra a bordo del satélite y la base terrestre, es generada por radio-enlace utilizando como medio de programación a Ada. En este proyecto se propone el uso de Python para realizar el enlace entre los diferentes componentes que se encuentran en la plataforma de Tierra y el satélite; para garantizar la escalabilidad, flexibilidad y alta disponibilidad en el almacenamiento y control de la información del satélite UPMSat-2.

Desde un ámbito específico, se diseñó e implementó un prototipo que simula la vinculación entre el satélite y un clúster de almacenamiento de información que cuenta con una interfaz web para mostrar los datos entregados por el satélite.
Para alcanzar los objetivos planteados en este proyecto, se realizó un estudio sobre los diferentes sistemas operativos orientados a la alta disponibilidad y redundancia conocidos como (PaaS), dentro de los cuales se encuentra CoreOS, que ha sido elegido para la implementación de este proyecto porque es una distribución mínima de Linux desarrollada para ejecutarse como un host de contenedores, y que permite la coordinación de múltiples nodos en el clúster. Adicionalmente, se implementó una base de datos dentro del clúster, que cumple con los parámetros necesarios para el almacenamiento y gestión de la información que es entregada por el satélite. De la investigación realizada, se determinó que Apache Cassandra satisface las necesidades de almacenamiento que requiere el proyecto; sus características de alta disponibilidad y escalabilidad, crean una infraestructura altamente tolerante a fallas y ampliamente replicable. Su instalación se realizó sobre contenedores Docker debido a la ligereza, portabilidad y autosuficiencia en sus procesos. Para visualizar los datos almacenados en Apache Cassandra se utilizó Python Django porque permite la comunicación de la plataforma con la base de datos y muestra la información a los usuarios a través de una interfaz web.

Abstract

This project's goals are to research, design and implement a functional platform to store and display information obtained by UPMSat-2 satellite.

UPMSat-2 is a micro-satellite which is used as a platform to demonstrate and validate different devices and experiments. All information given by the satellite must be stored in the terrestrial platform in order to be managed and controlled by the user through a web interface.

From a general perspective, this works let us know about the architecture and the UPMSat-2 satellite components. Communication between the system on board the satellite and the terrestrial base is performed through radio link by using Ada programming environment. In this project, we propose the use of Python to link the two components located in the terrestrial platform and the satellite in order to guarantee scalability, flexibility and high availability in the storage and control of the UPMSat-2 satellite information.

From a specific perspective, we designed and implemented a prototype that simulated the bond between the satellite and an information storage cluster which has a web interface to show the data provided by the satellite.

To reach the stated goals in this project, a study was carried out about the different high-availability-and-redundancy operating systems known as PaaS. CoreOs is one of such operating systems and it was chosen for the implementation of this project since it is part of any Linux minimum distribution. It was developed to execute as a hosts container and it allows coordination between multiple nodes within the cluster. Additionally, a database was implemented within the cluster. This database fulfils the needed parameters to store and manage the information provided by the satellite.

From this research, it was concluded that Apache Cassandra satisfies the storage requirements in this project. Its high availability and scalability traits offer a high failure tolerant infrastructure and it is possible to replicate it. Its installation was performed on Docker container due to its lightness, portability and auto proficiency in its processes. To visualize the data stored in Apache Cassandra, a Python Django web server was used because it allows the communication between the platform and the database and it shows the information to users through a web interface.

Índice general

Índice de figuras

Índice de tablas

Siglas

API	Application Programming Interface
CAP	Consistency, Available, Portability
CLI	Command Line Interface
CQL	Cassandra Query Language
GTK	Graphical Tkinter
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
KVM	Kernel-Based Virtual Machine
LAN	Local Area Network
QT	Query Transfer
RKT	Rocket CoreOS
SSL	Secure Socket Layer
SQL	Structured Query Language
TC	Telecomand UPMSat-2
TK	Tkinter Interface
TM	Telemetric UPMSat-2
TTC	Telemetric Telecomand Communications
UID	Unique Indentifier Digit
URL	Uniform Resource Locator
VCS	Version Control System

VM	Virtual Machine
VXLAN	Local Area Network Virtual Extension
YAML	Human-Readable Data Serialization Language

1 Introducción

La evolución de las nuevas tecnologías nos ha trasladado a un mundo constantemente conectado y consumidor, de servicios y aplicaciones, la línea que se sigue el día de hoy en la mayoría de las universidades a nivel nacional e internacional, es la creación y diseño de satélites pequeños conocidos como nano o micro satélites; el objetivo de estos satélites es desarrollar experimentos científicos.

El primer microsatélite universitario español fue diseñado y puesto en órbita por la Universidad Politécnica de Madrid con el nombre de UPM-Sat1. En la actualidad se está desarrollando un microsatélite con el nombre UPMSat-2, pensado como una plataforma para la investigación y desarrollo.

El microsatélite UPMSat-2 tiene la función de desarrollar experimentos de carga útil, además de otras funciones como el control de comunicaciones con Tierra, el control de actitud (posición del satélite en la órbita), almacenamiento de los datos comprendidos entre los experimentos y los resultados, etc. La información generada por el satélite deberá ser enviada hacia la plataforma en Tierra. El objetivo de este trabajo de fin de Master comprende en la implementación de un clúster de almacenamiento de datos del microsatélite UPMSat-2.

La plataforma de segmento en Tierra que se encargará de almacenar la información enviada por el satélite debe garantizar la integridad, confidencialidad y disponibilidad de los datos; lo cual implica la implementación de un sistema distribuido que permita modelar la información, sea redundante y mantenga la calidad del servicio. El mantenimiento de una plataforma con estas características, supone un costo considerablemente elevado y se requiere una estructura compleja que difícilmente o casi siempre es complicado mantener. Ante estos inconvenientes se encontró que existen diferentes maneras de conservar, desarrollar y actualizar estos servicios. CoreOS es una solución viable, diseñada para proporcionar la infraestructura necesaria para despliegues en clúster, combina un sistema operativo mínimo con las herramientas necesarias para ejecutar servicios y aplicaciones. Todo esto empaquetado y listo para funcionar con un equipo físico o virtual en plataformas como KVM, Google Cloud, Digital Ocean, Amazon Elastic Cloud entre otras. CoreOS no proporciona un gestor de paquetes por lo que requiere ejecutar todas sus aplicaciones dentro de contenedores Docker o contenedores Linux. Estos contenedores se basan en un núcleo de sistema operativo común, son mucho más ligeros y eficientes que los hipervisores, donde toda la pila de computación desde el procesador a la memoria para el almacenamiento, se

virtualiza proporcionando a las aplicaciones la utilización de menores recursos en el sistema.

Para el desarrollo de la plataforma del segmento en Tierra se instaló y configuró un clúster basado en CoreOS que garantiza alta disponibilidad. Además, se instaló y configuró una base de datos llamada Apache Cassandra, la cual tiene como característica principal alta tolerancia a fallos; garantizando que la información recibida por el segmento de Tierra se pueda modelar y visualizar mediante una interfaz web.

1.1 Objetivos

Con este proyecto de Fin de Master se pretende contribuir al desarrollo de la plataforma de segmento en tierra del Satélite UPMSat-2, incluyendo el diseño, configuración, implementación, pruebas e integración de un clúster de servicios, que permita almacenar y visualizar los datos de Telemetría (TM) proporcionados por el satélite.

1.1.1 Objetivos Específicos

- Diseñar e implementar un clúster basado en CoreOS que soporte un servicio de contenedores, para la implementación de una base de datos que permita almacenar la información entregada por el sistema del satélite UPMSat-2.
- Integración de los sistemas entre la plataforma de Tierra y el sistema de satélite UPMSat-2.
- Analizar la alta disponibilidad y tolerancia a fallos en el clúster de CoreOS al recibir la información proporcionada por el sistema del satélite UPMSat-2.
- Analizar los resultados obtenidos y determinar la fiabilidad del clúster de CoreOS para este entorno con datos reales recibidos por parte del satélite.
- Desarrollo de un prototipo de la plataforma terrena de UPMSat-2.

1.2 Alcance

El presente trabajo engloba tres etapas para su entrega final, con los recursos proporcionados por parte de la Universidad Politécnica de Madrid se implementó las siguientes funcionalidades:

1.2.1 Descripción de los componentes de la plataforma terrena

- Descripción de la arquitectura software de la plataforma
- Descripción de las funciones de los subsistemas de TC y TM
- Desarrollo de los componentes del prototipo

1.2.2 Levantamiento de Clúster CoreOS de la plataforma

- Sistema operativo CoreOS instalado en tres equipos servidores con conexión entre ellos para simular un clúster funcional.

- Clúster funcional con direccionamiento IP fijo dentro de la LAN de la UPM y con acceso a Internet.
- Servicios configurados y operativos dentro del clúster para habilitar las características de redundancia: ETCD, Fleet y Flannel.
- Configuración de servicios de Apache Cassandra con Docker para levantar DB de alta redundancia.

1.2.3 Desarrollo de un prototipo de la plataforma terrena de UPMSat-2

- Conexión de Apache Cassandra utilizando Docker, sobre el clúster de Coreos para el almacenamiento de la información.
- Enlace de comunicación entre Apache Cassandra y satélite UPMSat-2 para que leer datos de un archivo plano proporcionado por el sistema.
- Comunicación de un entorno Web utilizando Python para la visualización de los datos e información almacenados en Apache Cassandra.
- Análisis de resultados obtenidos tras realizar pruebas de carga y conexión.

1.3 Estructura del Documento

A continuación, se realiza una descripción del contenido de cada uno de los capítulos que componen el presente Trabajo de Fin de Master.

En el Capítulo 1 se realiza una introducción al desarrollo del trabajo de fin de master colocándolo en contexto, indicando sus objetivos y el alcance obtenido.

En el Capítulo 2 se realiza un estudio del estado del arte y una descripción de las características del proyecto. Concretamente se detalla el sistema operativo que se va a utilizar para el levantamiento del clúster, así como también, los componentes utilizados para tener el clúster en funcionamiento.

En el Capítulo 3 se detallará a profundidad el sistema operativo Coreos que será utilizado para el diseño y configuración de la plataforma de segmento en tierra. Así como también el uso de contenedores Docker para el levantamiento de servicios como Apache Cassandra que será la base de datos que almacenará la información entregada por el satélite y Python que mostrara los datos utilizando una interfaz web.

En el Capítulo 4 se detalla el diseño e implementación de la plataforma de segmento en Tierra del satélite UPMSat-2. Se aborda de forma detallada el desarrollo del clúster, sus servicios y del sistema que interpretara la información del satélite y que almacenara en la base de datos Apache Cassandra.

En el Capítulo 5 se realizaran pruebas de funcionamiento del sistema, se determinara la alta disponibilidad, la redundancia de los datos y la tolerancia a fallos que son almacenados en la base de datos Apache Cassandra y expuestos por el servidor Web.

En el Capítulo 6 contiene las conclusiones obtenidas luego de la finalización del trabajo fin de master y posibles líneas futuras.

2 Estudio del Estado del Arte

La creciente necesidad de sistemas de información que permitan el almacenamiento y modelamiento de datos ha llevado al desarrollo de diferentes tipos de arquitecturas que garantizan la seguridad y disponibilidad de la información. Para cualquier empresa, una interrupción de sus sistemas de información supone un serio problema, por lo que en los últimos años se han utilizado infraestructuras de tipo clúster que proveen enormes capacidades de procesamiento y almacenamiento de la información; así como también disponibilidad y fiabilidad en los sistemas para mantenerse operativos a pesar de cualquier fallo [1]. La alta redundancia es otro pilar fundamental que buscan las organizaciones a la hora de implementar sus sistemas de información, ya que previene transacciones fallidas que pueden marcar la diferencia entre el éxito y el fracaso en la toma y muestra de datos e información.

Las pérdidas de fiabilidad de datos son cada vez más comunes conforme las empresas procesan rápidamente mayores volúmenes de datos y de mayor variedad para cumplir las expectativas de los clientes [2]. Con la aparición de tecnologías como el Cloud Computing, Big Data, el internet de las cosas (IoT), se requieren aplicaciones que ofrezcan alto rendimiento, disponibilidad, flexibilidad y escalabilidad. Sin embargo, el aumento masivo de los datos e información crea nuevos obstáculos que dificultan que las aplicaciones satisfagan estas demandas.

El uso de clústeres dentro de la infraestructura de los sistemas de información, ha sido visto como una solución para mantener los servicios y aplicaciones disponibles. Esta solución debe contar con la capacidad para manejar conmutación de errores y alta disponibilidad de los servidores, en el caso de que uno pueda quedar inoperativo, los demás servidores que conforman el clúster deben tener la capacidad de suplir las funciones del servidor desconectado [3].

2.1 Clústeres

En contexto, el uso de clústeres aumenta la disponibilidad, mejora el rendimiento, brinda escalabilidad, es tolerante a fallos y permite la recuperación ante posibles fallos en un tiempo aceptable causando el mínimo impacto; además, reduce costes, y consolida servidores y almacenamiento.

En la figura 1, se muestra un ejemplo de clústeres de base de datos cuya estructura es inflexible ya que los servicios en funcionamiento dentro del clúster pueden tener necesidades variables. Con esta estructura no se tiene la posibilidad de redirigir los recursos no utilizados a un servicio. Es por ello que en la actualidad se tiende adoptar un sistema que permita disponer de todos los nodos que conforman el clúster para cualquier tarea requerida, en otras palabras tener un gestor de clúster y que este decida que nodo ejecuta que tarea y con qué tipo de sistema.

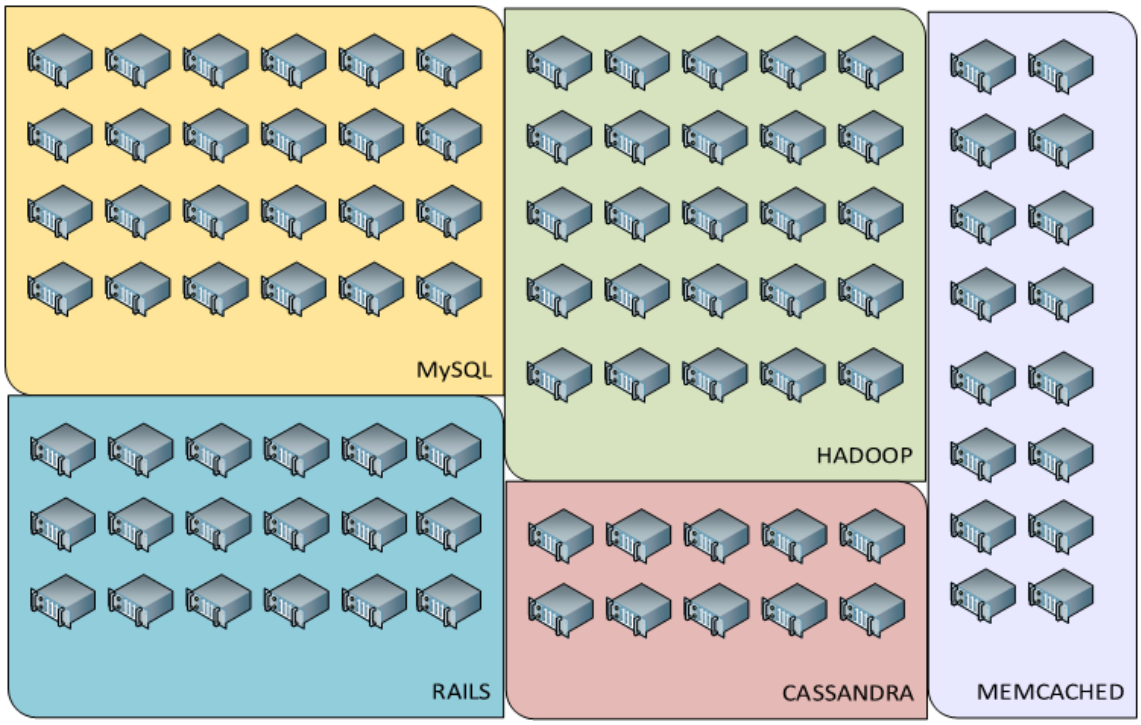


Figura 1. Estructura de Clúster Tradicional

El gestor del clúster (clúster manager), es un elemento del sistema que se encarga de asignar los procesos, controlar los nodos y gestionar los fallos. Se encuentra un gran abanico de opciones tanto en la manera de implementar la gestión del clúster como de las herramientas que son utilizadas.

En la figura 2, se muestra un administrador de clúster que permite administrar y configurar servicios; a diferencia de lo mostrado en la figura 1, donde únicamente se tiene una estructura de clúster para cada tipo de servicio.

Existe dos tipos de estructuras del clúster manager, la primera se basa en proporcionar recursos y asignarlos en función de las necesidades y capacidades de cada

nodo, y la segunda opción se encarga de que las tareas se realicen constantemente, priorizando la estabilidad del sistema.

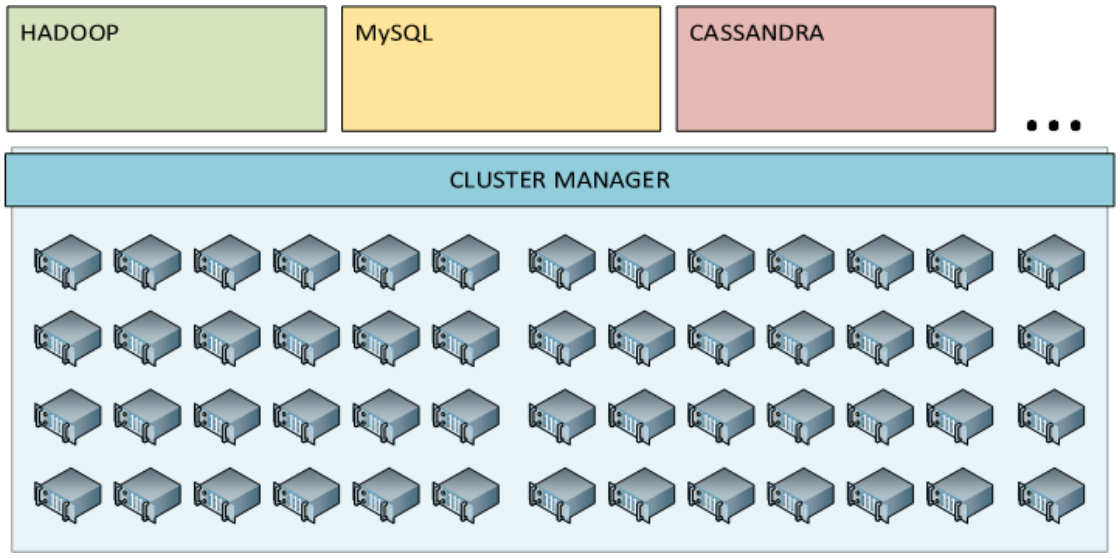


Figura 2. Estructura de Clúster Flexible

La primera estructura nos da eficiencia y disponibilidad, el problema que presenta es que se debe contabilizar de manera periódica los recursos de cada máquina y su disponibilidad en función de la carga que se esté ejecutando. La segunda estructura tiene como diferencia la estabilidad, seguridad y rapidez; pero por otra parte, se puede dar el caso de no utilizar los recursos disponibles de manera eficiente.

A continuación se detallan diferentes tipos de sistema que hacen uso de estas estructuras.

1. **Apache MesOS:** Es un sistema diseñado por la Universidad de California en Berkley 2010. MesOS es un ejemplo perfecto de un servicio PaaS y funciona sobre CentOS o Ubuntu Server. Cumple con las condiciones del primer tipo de estructura ya que su sistema se basa en contabilizar los recursos que tiene, el número y el tipo de nodos, para entonces poder distribuir las tareas o trabajos que llegan directamente al clúster [4].

   Su arquitectura se basa en un nodo maestro y una serie de nodos esclavos que realizan el trabajo asignado por parte del nodo maestro. La fiabilidad y tolerancia a fallos es controlada por el nodo maestro, el que dispone de un par de nodos "backup" o como MesOS los nombra, StandBy Master. Estos nodos

pueden reconstruir el estado del nodo master en cualquier momento gracias a un sistema de mensajes periódicos que mantiene.

El sistema de distribución de las tareas se basan en un sistema de oferta y demanda, el nodo maestro pregunta el tipo de recurso que tiene cada nodo y en función de la memoria disponible se asigna un tipo de tarea. Para gestionar la fiabilidad y la tolerancia a fallos sobre los nodos se utiliza una herramienta llamada Zookeeper la que se beneficia de la virtualización para obtener un aislamiento sobre el framework con el que se quiere trabajar [5].

2. **CoreOS:** Es un sistema con características similares a la arquitectura anterior pero completamente diferente en otros aspectos. A diferencia de MesOS, CoreOS no posee un master que gestiona directamente la carga de cada nodo en función a las tareas a realizar, sino todo lo contrario, el conjunto de nodos utiliza un sistema de levantamiento para cada tarea y se pone de acuerdo entre los nodos para asignarlo entre ellos. CoreOS trabaja sobre la máquina y está diseñado específicamente para funcionar a través de contenedores lo cual hace que sea extremadamente ligero, escalable y que el consumo de recursos sea ilimitado.

   Además, CoreOS utiliza una instancia llamada ETCD [6] para que exista una comunicación entre todos los miembros del clúster; y otra instancia llamada Fleet [7] que se encarga de gestionar el clúster y cualquier fallo que existiera; al tener levantados estos servicios se puede aprovechar la tecnología de los contenedores para desplegar servicios y aplicaciones. Otra de las ventajas que muestra CoreOS es su modelo de actualización, que trabaja con un módulo de partición manteniendo una partición activa mientras se actualiza y la otra certificando la estabilidad y posibilidad de realizar un rollback en cualquier momento [8].

En base a esto, se concluye que se utilizará CoreOS como sistema operativo para la implementación de la plataforma de segmento en tierra, debido a las prestaciones que brinda y a la simplicidad para su despliegue.

2.2 Contenedores

Tradicionalmente sería necesario implementar un servidor físico o virtual para desplegar servicios o aplicaciones; gracias a la tecnología de contenedores, no es necesario el uso de muchos recursos para obtener estas funcionalidades. Los

contenedores ofrecen una opción más liviana y fácil de administrar para entregar aplicaciones listas para su funcionamiento, independiente del entorno. Es decir, un contenedor es un conjunto de elementos que permiten ejecutar correctamente una aplicación determinada en cualquier sistema operativo y sin fallos a pesar de que cambie el entorno donde esta se ejecute.

Docker y rkt son plataformas de contenedores cada una con características importantes, que presentan diferencias casi imperceptibles considerando que los contenedores ofrecen una consistencia y portabilidad sin precedentes.

En primer lugar, antes de tener una definición clara de que entorno se debe utilizar para la implementación del proyecto, es importante mencionar que CoreOS admiten entornos Docker y rkt. A continuación se detallarán las características principales de cada una de estas tecnologías.

1. **Rkt:** En el 2014, CoreOS lanzó al mercado la aplicación rkt como una alternativa más segura, interoperable y de libre acceso, que ofrece más seguridad al usuario, en comparación con versiones anteriores a Docker. Se diferenica de Docker debido a su formato de contenedor abierto conocido como (appc).

   Rkt fue diseñado para tener una separación de privilegios más robusta y fácilmente integrable con los sistemas init de Linux. No posee un demonio como su competidor pero si depende del sistema de inicio que usa para administrar el control de proceso de un contenedor [9].

2. **Docker:** Es un proyecto open source creado en el 2013, que utiliza el aislamiento de recursos del kernel para permitir que varios contenedores puedan funcionar en una misma instancia de sistema. Lo que hace es virtualizar contenedores a un nivel muy bajo, por tal motivo está desarrollado encima de libvirt y lxc y escrito en Go [10]. Al igual que rkt, crea un contenedor con aplicaciones ejecutables en cualquier sistema independientemente del sistema operativo. Docker utiliza una plataforma web conocida como Docker Hub en la que se puede encontrar una gran variedad de servicios configurados y listos para ser utilizados a manera de repositorios. La ventaja principal de Docker es que permite a los usuarios crear sus propias imágenes de una manera simple utilizando un Docker File.

Teniendo en cuenta el significado de estas dos plataformas a continuación en la tabla 1 se muestra las diferencias de características entre los contenedores Rkt y Docker.

Característica	*Docker*	*Rkt*
Conjunto de Capacidades	Organización empresarial de contenedores, administración de aplicaciones simples.	Operativo, liviano de código abierto enfocado en la seguridad.
Facilidad de Uso	Docker ofrece Kitematic como una solución basada en GUI para administración [11].	Creo la plataforma Tectonic para administración visual de contenedores.
Apoyo de la Comunidad	La comunidad de Docker utiliza su plataforma Hub para foros y discusiones.	CoreOS posee un centro activo de recursos con la comunidad en el que opta por incluir temas de actualidad con respecto a Rkt [9].
Tasa de Liberación	Docker posee una oferta de lanzamientos y actualizaciones más óptima, su versión (17.12.1).	Rkt de CoreOS ha optado por actualizaciones prolongadas, su versión actual (1.28).
Precio y Soporte	Gratuito para CE (Community Edition) para empresas las cuales requieren otro tipo de servicios son bajo pago EE (Enterprise Edition).	Gratuito para CE (Community Edition), las opciones de pago de CoreOS están completamente ligadas a soporte.
API y Extensibilidad	Docker utiliza un API REST y SDK permite a desarrolladores controlar la pila de contenedores desde aplicaciones personalizadas [11].	Rkt de CoreOS utiliza gRPC un marco de alto rendimiento y código abierto, ofrece en sus ofertas una API RESTful [9].

Integración con terceros	El Hub de Docker se diferencia exponencialmente de Rkt y todos proyectos alojados en sus plataformas, ofrecen más de 100.000 aplicaciones gratuitas.	Rkt, no tiene una alta integración pero posee una gran cantidad de proyectos en GitHub, lo que puede representar un pequeño alcance a su competencia.

Tabla 1. Comparativa Docker vs Rkt

En resumen, se utilizara Docker en la implementación por presentar ventajas únicas, al ser open source no es necesario adquirir una licencia para su funcionamiento; es versátil y ligero, nos permitirá la instalación de la base de datos a través de una imagen con una aplicación determinada, facilita la compartición de las aplicaciones a través de los contenedores desplegado y brinda un entorno seguro sin variaciones.

2.3 Bases de datos NoSQL

En los últimos años, una gran variedad de bases de datos NoSQL han salido a la luz, creadas por compañías principalmente para cubrir sus propias necesidades. Temas como escalabilidad, rendimiento, mantenimiento, etc. que no encontraban en ninguna solución que existía en el mercado.

Debido a la variedad de enfoques que existe entre requisitos y funcionalidades que debe cumplir una base de datos NoSQL, es bastante difícil mantener una visión general de la situación actual de las bases de datos no relacionales.

Se puede decir que las bases de datos NoSQL son una categoría independiente dentro del conjunto de bases de datos. Se realizará una clasificación general de las bases de datos NoSQL más importantes.

Base de Datos NoSQL Clave Valor, como si de un diccionario se tratara, se utiliza una pablara clave para identificar el significado, en este caso, es un tipo de datos que contienen tuplas de clave valor. Los clientes añaden y solicitan valores a partir de una clave asociada que se conoce de antemano. Los sistemas modernos de almacenamiento clave valor se caracterizan por tener una elevada escalabilidad y un rendimiento muy bueno para volúmenes de datos muy grandes [12].

Base de Datos NoSQL Documentales, o bases de datos orientadas a documento, son otro tipo de base de datos NoSQL con un grado de complejidad y flexibilidad superior

a las bases de datos clave valor. En las bases de datos documentales el concepto principal es el de "documento". Un documento es la unidad principal de almacenamiento de este tipo de base de datos, y toda la información que aquí se almacena, se hace en formato de documento.

Base de Datos NoSQL Columna, este tipo de base son otro caso particular de la enorme familia de las bases de datos NoSQL. En este tipo de almacenes, en contraposición con el modelo relacional, la información se estructura en columnas en lugar de en filas. Algunas de las bases de datos NoSQL más importantes y con una mayor aceptación pertenecen a este grupo. Bigtable, la solución NoSQL de Google, HBase, la base de datos de Hadoop, Cassandra, impulsada por Facebook y ahora perteneciente a Apache Software Foundation, son solo algunos ejemplos de este tipo de bases de datos [12].

Base de Datos NoSQL Grafos, orientadas a grafos tienen la particularidad de representar la información como si de un grafo se tratara. La información viene representada por los nodos, y las relaciones entre los datos por las aristas [12]. De este modo, se puede emplear la teoría de grafos para recorrer la base de datos y así gestionar y procesar la información. Una base de datos orientada a grafos, de forma generalizada, es cualquier sistema de información, donde cada elemento tiene un puntero directo hacia sus elementos adyacentes es decir, no sería necesario realizar consultas mediante índices.

A continuación se detallan dos diferentes tipos de bases de datos que hacen uso de la clasificación antes mencionada.

1. **MongoDB:** Almacena datos en documentos flexibles, similares a JSON, lo que significa que los campos pueden varias de un documento a otro y la estructura de datos se puede cambiar con el tiempo. MongoDB es una base de datos distribuida en su núcleo, por lo que la escalabilidad y distribución geográfica están integradas y son fáciles de usar.

   MongoDB es gratuito y de código abierto y la principal característica de esta base de datos es que utiliza el modelo de documento que se correlaciona con los objetos en el código de las aplicaciones, facilitando el trabajo de los datos [13].

2. **Apache Cassandra:** Es una base datos NoSQL que presenta como característica principal su alto desempeño en la escalabilidad y alta disponibilidad sin comprometer el rendimiento. La escalabilidad lineal y su probada tolerancia a fallos ya sea en hardware básico o infraestructura en la nube la convierte en la plataforma perfecta para datos de "misiones críticas".

> [14] Utiliza un modelo columna que permite una búsqueda eficiente dentro de los datos.
>
> El respaldo de Cassandra para la replicación en múltiples centros de datos, una latencia baja para sus usuarios y la fiabilidad de poder mantener los datos convierte a Cassandra en una solución ampliamente eficaz.

Al exponer las características de este tipo de base de datos NoSQL se determina que el uso de Apache Cassandra es la mejor opción para el proyecto, por su alta disponibilidad que admite un modelo "maestro múltiple" que permite tener varios nodos maestros y la pérdida de un solo nodo no afecta la capacidad del clúster para tomar escrituras; la escalabilidad permite escrituras en cualquier servidor, básicamente depende de la cantidad de servidores que se puede tener en el clúster para una mejor escalabilidad.

3 Arquitectura de la Plataforma

En este capítulo se describe la arquitectura de la plataforma utilizada en este trabajo para el desarrollo de la plataforma de segmento en tierra del satélite UPMSat-2, el cual comprenderá el uso de las siguientes plataformas para el almacenamiento de la información que será proporcionada por el satélite UPMSat-2.

3.1 CoreOS

CoreOS es una distribución basada en Linux, parecido o similar al sistema operativo de Chrome de Google, está basado en el sistema operativo de Gentoo, el que cumple con el objetivo de estar diseñado para tener escalabilidad y ser distribuido, pensado específicamente para trabajar en clúster, entre sus herramientas podemos encontrar tecnología muy diversa como son (Fleet, Rkt, ETCD, Flannel, Docker) que cumplen la función de mantener un clúster operativo y consistente. [15]

Es un sistema con muy pocas funcionalidades iniciales ya que parte de la base de ser muy ligero, es un sistema con licencia Apache, originalmente es un actualización de ChromeOS y el primer lanzamiento se lo hizo en el 2013, como se puede observar es un sistema bastante nuevo con un tiempo considerablemente escaso en el mercado y que aún se encuentra en intensas fases de desarrollo, al ser un sistema tan básico y liviano no posee de una interfaz gráfica, para la utilización de aplicaciones lo que se requiere es una virtualización a nivel de kernel, el hecho de que sea un servicio tan sencillo hace que el consumo de recursos sea escaso [16].

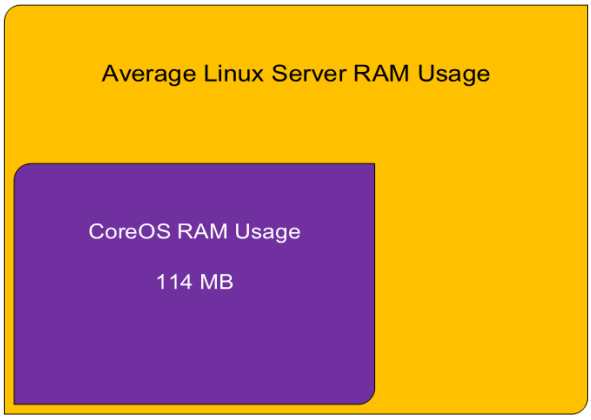


Figura 3. Comparación de Memoria CoreOS vs Linux

Como se observa en la figura 3 la utilización de la memoria RAM para el sistema operativo CoreOS tiene un porcentaje menor al 40% del promedio de los servidores Linux que se utilizan comúnmente; esto es una de las ventajas de este sistema operativo, el uso de recursos por parte del hardware hacen que CoreOS sea un sistema ligero y apto para cualquier entorno sea físico o virtual. Como se mencionó al inicio de este proyecto y una de las varias ventajas que tiene es que dispone de un sistema de doble partición para las actualizaciones del sistema operativo, esto permite que se pueda realizar cambios sin dar previo aviso pero de una manera totalmente segura ya que al hacerlo en dos pasos siempre se puede regresar al estado anterior.

Si el sistema llegara a fallar o si la actualización no se completara de la manera adecuada como se puede observar en la figura 4 el proceso de actualización de servicios o aplicaciones.

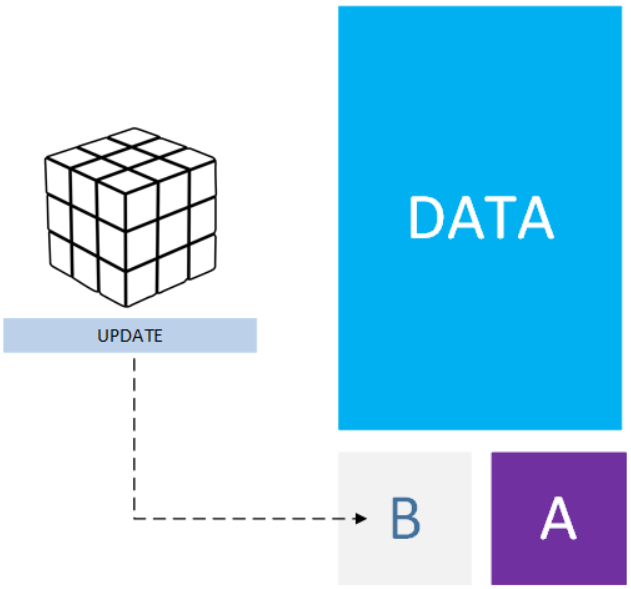


Figura 4. Estructura de Actualización

CoreOS posee varios sistemas claves para su funcionamiento los cuales en conjunto proporcionan las características antes mencionadas del sistema operativo que son:

- Etcd que actúa como el estado de configuración persistente del clúster.
- Fleet actúa como el programador de tiempo de ejecución distribuido del clúster.
- Systemd son el mecanismo por el cual fleet realiza sus ejecuciones.

Como se puede observar en la figura 5 se detalla de forma gráfica los componentes y se puede tener una perspectiva adecuada de cómo se encuentra formado el clúster, lo único esencial que faltaría detallar en la figura es el cloud-config, que es utilizado para para establecer el estado de configuración inicial.

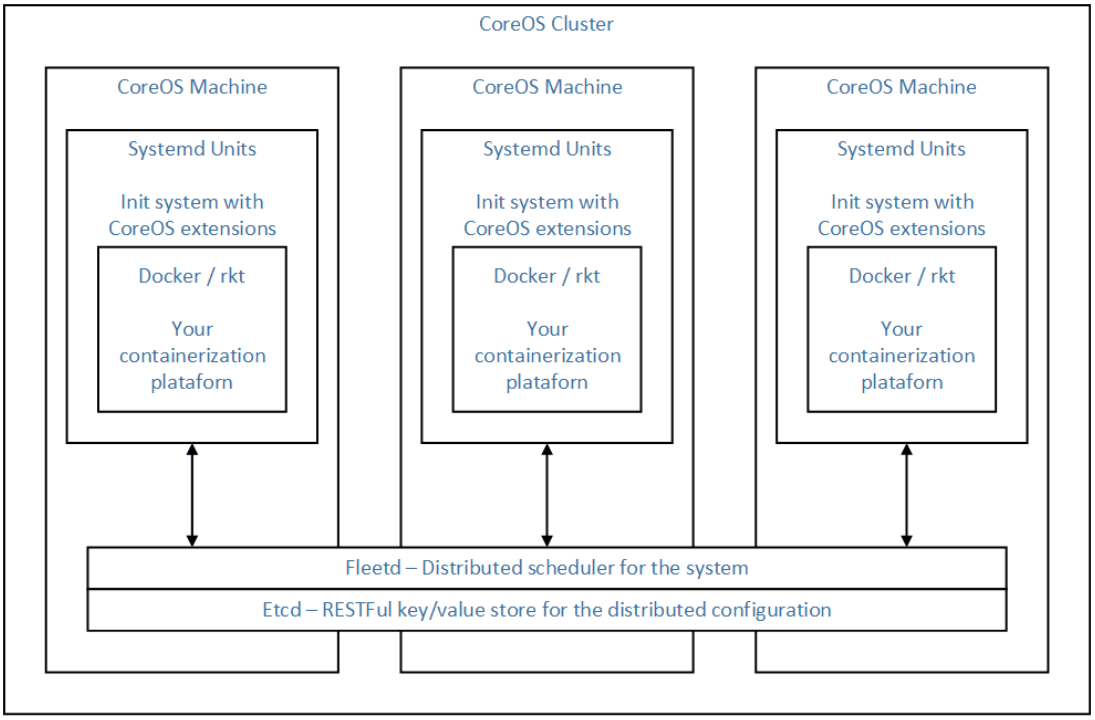


Figura 5. Diseño de CoreOS (Clúster) [15]

3.1.1 ETCD y el Estado de Configuración Distribuida

ETCD es un servicio distribuido de clave/valor y de alta confiabilidad, específicamente el sistema lo utiliza para compartir la configuración y servicios de descubrimiento, se puede acceder a través de herramientas de líneas de comandos personalizadas totalmente RESTful y basado en JSON [15]. CoreOS utiliza ETCD ya que dispone de herramientas que funcionan sobre este servicio como por ejemplo Fleet; ETCD está escrito en Go y bajo una licencia Apache 2.0 como su nombre lo indica ETCD está diseñado para distribuir su sistema y configuración de servicios.

ETCD se basa en las siguientes características:

- Seguro: Permite la utilización de certificados SSL
- Simple: Se utiliza a través de URL con HTTP + JSON
- Fiable: Utiliza algoritmos Raft consensuados para tratar con los sistemas distribuidos

CoreOS utiliza en concreto ETCDCTL, el cual es una versión para línea de comandos en este sistema, utilizamos esta tecnología básicamente para mantener parámetros de configuración entre diversos nodos del clúster. Esta herramienta nos permite escribir claves con valores donde se podrá obtenerlas, modificarlas y borrarlas de forma totalmente distribuida y manteniendo su fiabilidad. Para lograr que esto funcione de la forma adecuada se utiliza el algoritmo de Raft consensuado para gestionar el sistema de clave y valor, este algoritmo detalla que la información de cada nodo debe estar completamente replicada a todos los demás nodos y para realizar algún cambio se lo hará mediante un sistema de consenso.

Consenso de Raft

Este algoritmo está diseñado para que su utilización sea fácil y dinámica, las características de este algoritmo es que descompone en subproblemas relativamente independientes y trata limpiamente todas las piezas principales necesarias para los sistemas prácticos [17]. Lo que significa que cada nodo tiene una máquina de estados y una serie de logs. Lo que se quiere lograr es que la maquina sea totalmente tolerante a fallos y los logs muestren los cambios que se deben realizar; Si se requiere realizar algún cambio del log se debe poner de acuerdo mediante el algoritmo de consenso que se realizara el cambio. Si se llegara aprobar el cambio todos los nodos realizaran esta acción y el algoritmo se encargara de vigilar que todas las maquinas lo apliquen y por lo tanto concluirá con el mismo estado.

Este procedimiento asegurara que todas las maquinas en conjunto mantienen el mismo resultado y sus estados son iguales, se debe entender que al tratarse de un algoritmo de consenso para realizar un cambio se necesita una mayoría de nodos para mantener la consistencia, por ejemplo si un sistema tiene 5 nodos y llegara a perder comunicación con 3 de los 5 el sistema ya no podría funcionar de la manera adecuada, un ejemplo práctico seria en la replicación de registros, se tiene un conjunto 5 nodos sin líder, todos los nodos son posibles candidatos y proponen a los otros nodos que voten por ellos; A medida que el tiempo pasa se realizan votaciones si se llega un consenso se llega a elegir al nodo líder y el resto toma su posición como seguidor, al no llegar a un consenso se realizan nuevamente votaciones hasta elegir un líder, para no entrar en un "deadlock" cada nodo tiene un reloj interno que con un intervalo aleatorio vuelve a proponer su candidatura al resto de los nodos [18].

Una vez elegido el nodo principal o líder, este nodo puede proponer a el resto de nodos un cambio de valor, los nodos seguidores responde si están de acuerdo con la nueva asignación y finalmente se hace efectivo el cambio, si existiera el caso de que alguno de los nodos no esté de acuerdo con el cambio no se realizara ningún cambio y

retomara el estado normal como se muestra en la figura 6 el proceso de asignación de valores entre el nodo líder y sus seguidores.

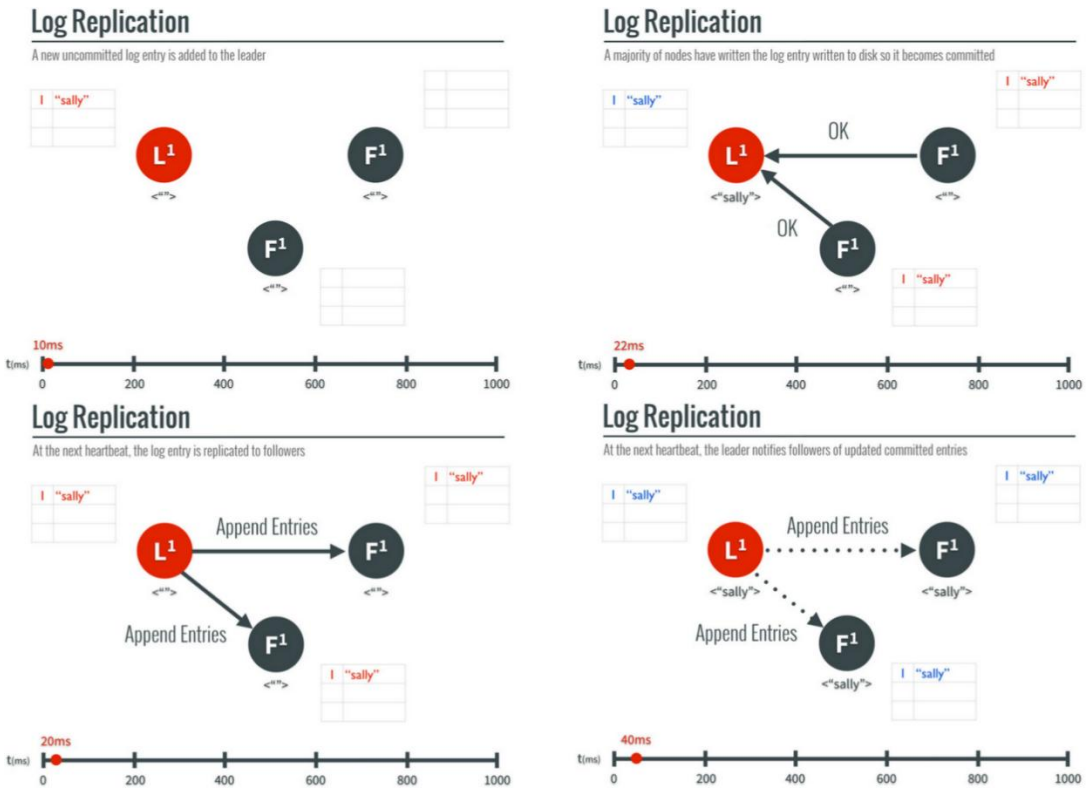


Figura 6. Asignación de Valores (Raft Consenso) [19]

3.1.2 Fleet y el Servicio de los Estados Distribuido

Fleet está desarrollado para estar al corriente del estado de una máquina, poner en marcha un servicio u observar el estado de todo el clúster, construido sobre systemd y ETCD, Fleet corre en segundo plano (demonio) en cada nodo del clúster [16]. Fleet está formado por dos partes que son el motor (engine) y un agente, toda la comunicación que hace los agentes con los motores se los hace a través de ETCD.

1. **Motor:** Es el responsable de organizar y gestionar las decisiones en el clúster, para gestionar utiliza un bucle que periódicamente se activa a través de alguno eventos de ETCD [15]. El sistema Fleet se encarga de controlar que no funcione más de un motor a la vez dentro del mismo clúster. El motor se encuentra conformado de la siguiente manera:
    - El motor realiza una instantánea del estado general del clúster tanto unidades como agentes.

- Tomando en cuenta el estado actual del clúster se propone el motor para ser elegido
- Si la elección se realiza de la forma correcta este motor toma parte de la reconciliación, de otra manera el motor entra en un estado de "standby" hasta que dé inicio a la siguiente elección.

2. **Agente:** Es el responsable de ejecutar las unidades en el sistema, este agente utiliza D-Bus para la comunicación con la instancia local de systemd [20], como el motor también utiliza un bucle periódicamente para determinar que se debe hacer, generando un snapshot del ETCD y ejecutando los cambios necesarios para alcanzar el estado que se refleja, aparte es el encargado de transmitir el estado de sus unidades a ETCD.

Fleet también está desarrollado para utilizar ETCD y guardar todos los datos tan persistentes puedan ser, este servicio también utiliza dos objetos necesarios que son:

1. **Units:** representa un conjunto de propiedades de configuración de los servicios que se requieren ejecutar. Una vez ingresados en el clúster no se pueden modificar, una unidad también puede especificar una serie de requerimientos que debe cumplir el nodo para poderlo ejecutar, las unidades son tratadas como un servicio mas no como un demonio, si llegara existir una caída de alguna unidad, Fleet tiene el objetivo de reubicarla en otra máquina del clúster.

2. **States:** Tanto la maquina como los ficheros tienen un estado dinámico que es publicado por el clúster.

3.1.3 Systemd Sistema de inicio de CoreOS

Systemd es un sistema Init relativamente nuevo diseñado para abordar significativamente más funciones que en el sistema sysvinit tradicional, este sistema ha ganado un impulso considerable, lo suficiente como para que la mayoría de las distribuciones de Linux hayan optado por cambiar parcialmente o completamente a systemd [15]. Este es el caso de CoreOS que usa extensivamente este sistema, para la gestión de nodos o ver el estado desde una conexión SSH se puede utilizar. Cabe mencionar que en gestión de procesos no es recomendable utilizarlo ya que como antes de menciono Fleet nos permite visualizar mejor el estado de los nodos sin la necesidad de estar en el mismo nodo.

3.2 Docker

Docker es una plataforma de servicio (PaaS), de código abierto creada en el año 2013 por dotCloud, convirtiéndose en un entorno que permite a desarrolladores la creación de aplicaciones y servicios que funcionan a través de Internet, alojados en la nube, inicialmente Docker era una extensión de una tecnología de la empresa que lo había desarrollado para ejecutar su negocio en la nube en miles de servidores. Como se mencionó al inicio esta plataforma está escrita en Go, es un lenguaje de programación desarrollado por Google con su sintaxis basa en C [21].

Docker permite desarrollar, enviar y ejecutar aplicaciones, proporciona la facilidad de separar las aplicaciones de la infraestructura para así poder entregar software de una forma más rápida. Aprovechando las metodologías de Docker para enviar, probar e implementar código rápidamente, se puede reducir significativamente el retraso entre código y ejecutarlo en un entorno de producción. Docker provee la capacidad de empaquetar y ejecutar una aplicación en un entorno aislado llamado contenedor. La seguridad y el aislamiento permiten ejecutar muchos contenedores simultáneamente en un host determinado.

Por lo general los contenedores son ligeros y podrían no necesitar la carga adicional de un hipervisor como es el caso de las máquinas virtuales, en este caso se ejecuta directamente dentro del núcleo de la maquina host. Lo que significa que se puede ejecutar más contenedores en una combinación de hardware dada. Incluso se puede ejecutar contendores Docker dentro de máquinas host que en realidad son máquinas virtuales [22].

3.2.1 Características de Docker

Como se mencionó en el capítulo 2 existen características que diferencian a esta plataforma de la competencia, a continuación se explicara a fundo las características más importantes de Docker:

- **Ligereza:** Los procesos de Docker son más ligeros, se debe a que no emula o virtualiza una máquina y su sistema operativo, como se realiza en el caso de la virtualización. No es necesario el sistema de archivos completo, es por eso que se puede aprovechar de mejor manera el hardware ya que aumenta la cantidad de servicios que se pueden tener en una maquina utilizando una fracción del espacio de almacenamiento.

- **Portabilidad:** Docker es manejado por imágenes que pueden ser desplegadas en cualquier entorno o sistema que soporte esta tecnología, de esta manera no es necesario volver a instalar y configurar todas las aplicaciones que se utilizan.

- **Seguridad:** El aislamiento es fundamental en esta plataforma ya que las aplicaciones entre si no tiene contacto, de igual sucede con las infraestructuras subyacente. Docker proporciona aislamiento por defecto lo que significa una ventaja cuando se presenta un problema en algún aplicación esto proporciona que se limite a un único contenedor en lugar de toda la máquina.

- **Autosuficiencia:** Los contenedores Docker son autosuficientes ya que solo se necesita de la imagen del contenedor para poder desplegar los servicios que este contiene.

3.2.2 Arquitectura de Docker

La arquitectura que utiliza Docker es (cliente–servidor), que consiste de un Docker Engine que es una ligera y potente tecnología de contenedorización de código abierto combinada con un flujo de trabajo para construir y contener aplicaciones.

Docker Engine es el motor principal de Docker que contiene un servidor que es un tipo de programa de larga ejecución llamado "daemon" (demonio). Aparte contiene un API REST que especifica las interfaces que los programas pueden usar para hablar con el daemon e instruir que acciones realizar y por ultimo contiene una CLI que se utiliza como una interfaz de línea de comando.

En la figura 7 se muestra los elementos anteriormente mencionados, donde se puede observar que la CLI utiliza un API REST de Docker para controlar y gestionar con el daemon de Docker a través de scripts o comandos directos de la CLI, además se puede observar que el daemon crea y administra objetos de Docker como son contenedores, la red, volumen de información, imágenes [22].

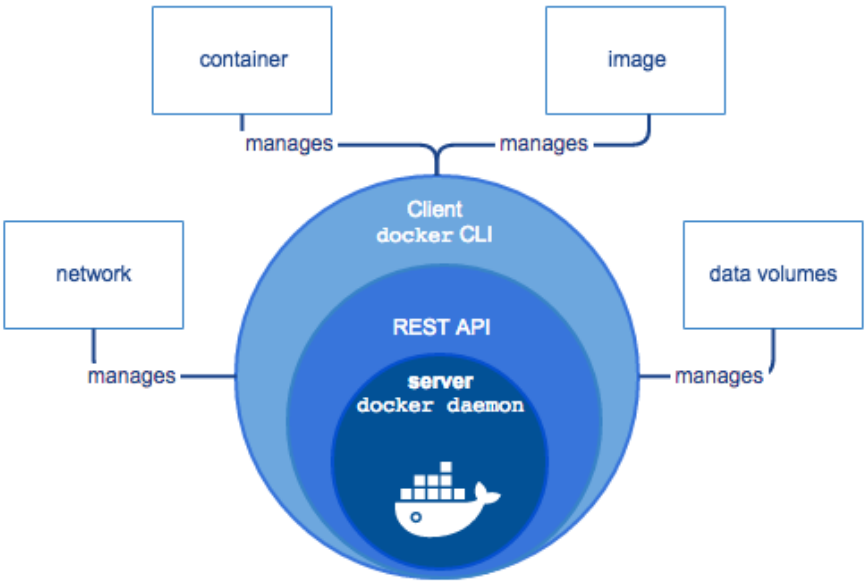


Figura 7. Docker Engine [22]

Al hablar de la arquitectura de Docker se debe considerar que el daemon de Docker escucha solicitudes de API y administra objetos de Docker como imágenes, volúmenes, redes y contenedores, un daemon puede comunicarse con otro para administrar servicios. Como se puede apreciar en la figura 8 el cliente Docker es la principal forma en que los usuarios de Docker interactúan.

Docker Hub y Docker Cloud son registros públicos que cualquier persona puede usar, Docker está configurado para buscar imágenes en Docker Hub de forma predeterminada.

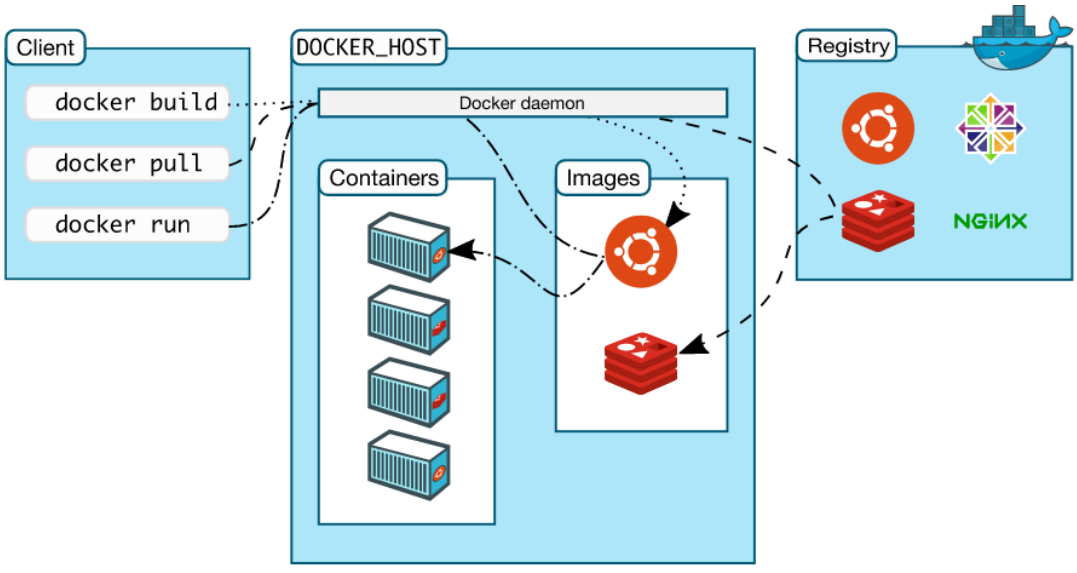


Figura 8. Arquitectura de Docker [22]

Los contenedores y las máquinas virtuales tienen beneficios similares de aislamiento y asignación de recursos, pero funcionan de manera diferente, en el caso de los contenedores virtualizan el sistema operativo en lugar del hardware. En la figura 9 se puede observar la distribución que manejan las máquinas virtuales y Docker.

Las máquinas virtuales (VM) son una abstracción del hardware que convierte a un servidor en muchos servidores, el hipervisor cumple la función de ejecutar múltiples máquinas virtuales en una sola máquina. Cada máquina virtual incluye una copia completa de un sistema operativo, lo que produce un excesivo desperdicio de gigabytes (GB) y representa un coste considerablemente alto al momento de poner en marcha este tipo de hipervisores [23].

Los contenedores son una abstracción en la capa de la aplicación que agrupa el código y las dependencias, múltiples contenedores se pueden ejecutar en la misma máquina y compartir el núcleo del sistema operativo con otros contenedores, los contenedores ocupan menos espacio que las máquinas virtuales, por lo general las imágenes suelen tener decenas de megabytes (MB) y su ejecución es casi al instante.

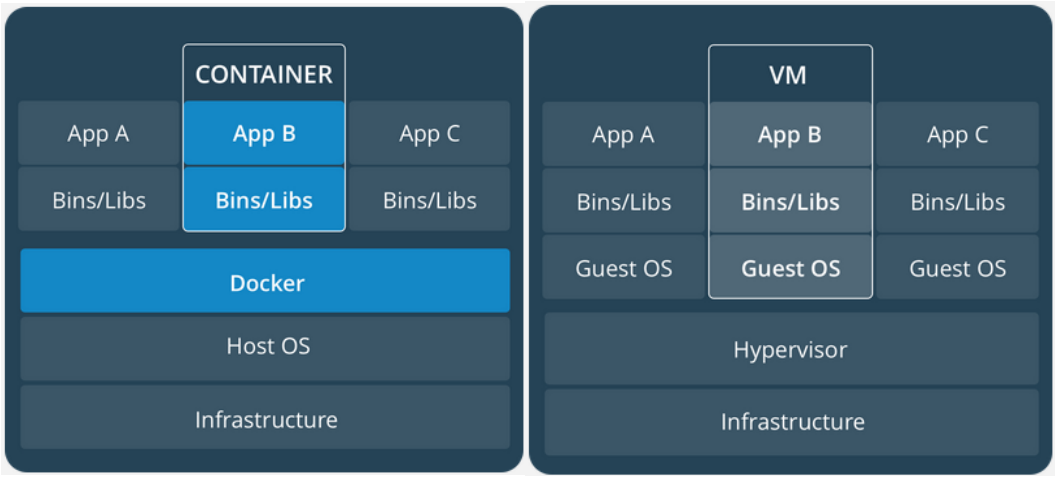


Figura 9. Docker (Container) vs Máquinas Virtuales (VM) [23]

3.2.3 Componentes de Docker

Docker posee 4 tipos de componente que son:

- **Imágenes:** Una imagen Docker es un archivo binario que incluye todos los requisitos para ejecutar un único contenedor Docker, así como metadatos que describen sus necesidades y capacidades. Los contenedores Docker solo

tienen acceso a los recursos definidos en la imagen, a menos que otorgue al contenedor acceso adicional al crearlo.

Si se desea modificar el contenido de una imagen, la única opción que Docker permite es añadir otra capa con los nuevos cambios. La arquitectura de Docker aprovecha de una manera efectiva este concepto de capas para añadir perfectamente capacidades adicionales a las imágenes existentes para satisfacer las diferentes necesidades del negocio y aumentar la reutilización de imágenes [24].

Lo que significa es que se pueden agregar capacidades a las imágenes existentes agregando capas adicionales encima de la imagen y derivando una nueva imagen. Docker maneja la relación padre e hijo en la figura 10 como se puede observar se llama a una imagen base que no tiene ningún padre.

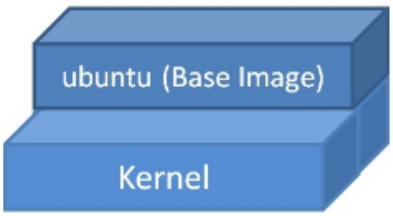


Figura 10. Imagen base de Docker [24]

Al tener en cuenta que siempre se empieza con una imagen base y cada adición que se realiza a la imagen base original se almacena en una capa separada como una imagen que hace referencia a otra imagen en este caso sería un padre. En la figura 11 se puede observar las nuevas imágenes creadas a partir de una base Docker.

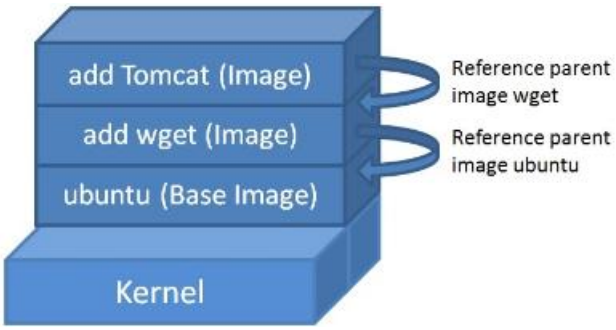


Figura 11. Nueva imagen a partir de la base imagen de Docker [24]

- **Contenedor:** Un contenedor es una instancia ejecutable de una imagen, se puede crear, ejecutar, detener o eliminar un contenedor mediante un API o mediante CLI, además se pueden conectar un contenedor a una o más redes internas o externas, agregar almacenamiento o incluso crear una nueva imagen basada en su estado actual.

- **Registro:** Las imágenes son almacenadas de forma pública o privada para que se puedan acceder a ellas y se puedan utilizar por parte de los desarrolladores de software.

- **Docker Hub:** El repositorio donde las imágenes a nivel mundial son guardadas, la comunidad de Docker se encarga de administrar y velar por la integridad de cada imagen, Docker Hub promueve y asegura que todas sus imágenes son seguras.

3.3 Apache Cassandra

Un proyecto Apache de alto nivel, nacido y creado por Facebook y Dynamo de Amazon con participación de Big Table de Google, es una base de datos distribuida para administrar grandes cantidades de datos estructurados en una gran cantidad de servidores, al tiempo que proporciona un servicio altamente disponible, rendimiento de escala lineal, simplicidad operativa y fácil distribución de datos en múltiples centros de datos o en zonas de disponibilidad en la nube [25].

La arquitectura de Apache Cassandra es responsable de su estructura escalar, en lugar de utilizar un maestro esclavo heredado o una arquitectura heredada manual que es complicada de mantener, Cassandra tiene una arquitectura tipo "anillo" y esto hace que sea fácil de mantener. Esta arquitectura tiene el objetivo de proporcionar un papel idéntico para todos los nodos, como se mencionó no existe el concepto de maestro esclavo, todos los nodos mantiene comunicación continua entre si y por igual.

Muchas empresas han implementado con éxito Apache Cassandra, convirtiendo a esta tecnología en líder en el área de entornos de alta disponibilidad y tolerancia a fallos.

3.3.1 Características de Apache Cassandra

Apache Cassandra muestra diferentes características, una serie de beneficios que optimizan el potencial de esta herramienta, entre los que cabe destacar:

- **Tolerancia a Fallos:** La información es automáticamente replicada en múltiples nodos, admite la replicación en varios centro de datos lo que permite que si un nodo falla se pueda reemplazar sin tiempos de inactividad.

- **Escalable:** Por su diseño, en el que todos los nodos son tratados por igual, ofrece simplicidad operativa y fácil escalabilidad horizontal, esto proporciona un diseño activo de principio a fin ya que en todos los nodos se puede escribir y leer.

- **Durable:** Convierte a Apache Cassandra en una sistema adecuado para aplicaciones que no pueden permitirse la perdida de datos.

- **Protección de Datos Sólida:** Un diseño de registros de confirmación evita la perdida de datos y construye copias de seguridad para facilitar la restauración a la vez que se mantiene los datos protegidos y seguros lo que produce consistencia de los datos sintonizables produciendo un clúster altamente distribuido.

- **Compresión de Datos:** garantiza que los datos se compriman hasta en un 80% sin que ello suponga un gasto de recurso en el clúster [25].

- **Lenguaje de Consulta Cassandra (CQL):** Para el acceso a la información el sistema usa un lenguaje propio llamado CQL, que permite una sintaxis similar al conocido SQL, aunque el número de posibilidades es mucho menor, Cassandra no permite realizar joins y en lugar de ello, para aumentar el rendimiento de las aplicaciones y facilitar el acceso a la información, se recomiendo optar por la desnormalización de los datos.

3.3.2 Arquitectura o Modelado de Apache Cassandra

Al hablar de base de datos RDBMS por definición se toma en cuenta que se utilizara tablas, claves primarias, claves foráneas, relaciones, etc. Cuando se trata de pasar de un modelo entidad relación a un modelo relacional, se debe contemplar la normalización para evitar la duplicidad de datos. No todas las aplicaciones tienen las mismas necesidades, como lo menciona el teorema de CAP no se puede tener todo (Consistencia, Disponibilidad y Tolerancia a Fallos) se debe elegir dos de las características mencionadas.

Para este proyecto se utilizó como característica principal la Disponibilidad y Tolerancia a Fallos haciendo referencia al teorema de CAP, la información almacenada dentro de la base de datos debe cumplir con esos parámetros. Para este caso se crea un nuevo esquema de datos, siempre se debe tener en cuenta que consultas se va a realizar a la base de datos para crear un esquema el permita conseguir el máximo de rendimiento en los accesos a esa información y si es necesario duplicar los datos para conseguir un mejor rendimiento.

Apache Cassandra define una familia de columnas para asociar datos similares. Por ejemplo, podemos tener una familia de columnas llamada Usuario, otra llamada Hotel, otra llamada Libro de direcciones y muchas más. De este modo, una familia de columnas es análoga a una tabla en el modelo relacional. En Cassandra existen dos tipos de estructuras básicas, que son las columnas, las cuales son pares nombre/valor, y la familia de columnas, la cual es un contenedor de registros que contienen columnas similares.

En las bases de datos relacionales, los nombres de columnas eran los que podían almacenarse como cadenas únicamente, en Cassandra ya no existe esta limitación, tanto llaves de registro como columnas pueden ser cadenas, o enteros, o de algún otro tipo. No necesitamos almacenar un valor para cada columna al momento de crear un nuevo registro porque quizás no conozcamos los valores cada columna. Por ejemplo, algunas personas tienen un segundo número de teléfono y otras no. En vez de colocar NULL para los valores que no conocemos, lo cual gasta espacio, simplemente no se tiene en cuenta la columna para ese registro. Con esto tenemos una estructura de arreglos multidimensional como se muestra en la figura 12.

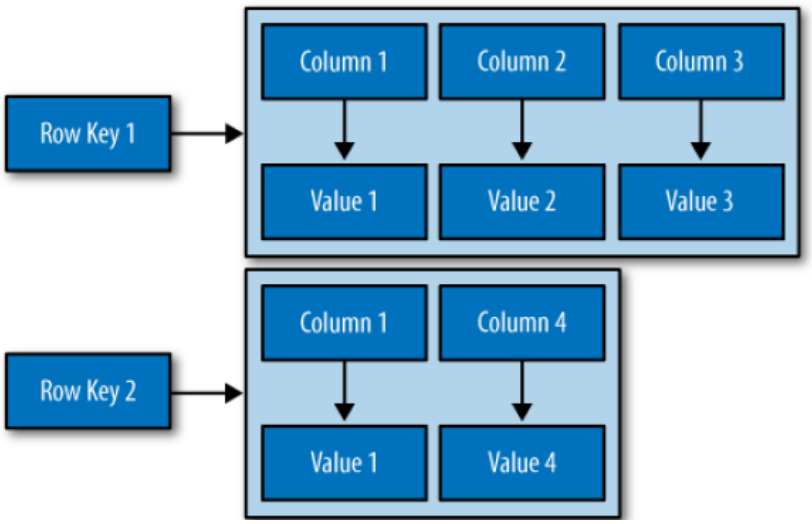


Figura 12. Estructura multidimensional de Apache Cassandra [26]

Cada columna en Cassandra posee una marca horaria, la cual graba la última fecha en que la columna fue actualizada, la marca es de las columnas y no de los registros. Este dato no puede consultarse, se usa únicamente para resolución de conflictos en el servidor.

Cassandra al proporcionar una estructura multidimensional posee características únicas que forman parte de la arquitectura de desarrollo entre ellas se describen las siguientes:

1. **Clúster:** Cassandra está diseñado para distribuirse en múltiples máquinas operando juntas y presentándose como una única entidad al usuario final. Así, la estructura más externa de Cassandra es el clúster.

2. **Espacio Clave:** Un clúster es un contendor para espacios claves. Un espacio clave es el contenedor de datos más externo en Cassandra, similar a un contenedor de tablas en el modelo relacional. Cada espacio contiene un nombre y propiedades. Se puede crear tanto espacios clave como la aplicación lo necesite, aunque se considera como una práctica aceptable crear un solo espacio clave por aplicación [26].

3. **Familia de Columnas:** La familia de columnas es un contenedor para una ordenada colección de registros, los cuales cada uno es una ordenada colección de columnas. En el área de las bases de datos relacionales, cuando crea una base de datos a partir de un modelo, se especifica el nombre de la base de datos (un espacio clave en Cassandra), los nombres de las tablas (remotamente similares a una familia de columnas) y entonces se define los nombres de las columnas que irán en cada tabla.

   Existen tres diferencias por las cuales las familias de columnas no son iguales a las tablas. La primera, aunque las familias de columnas son definidas, las columnas no, se pueden agregar columnas en cualquier momento. La segunda, una familia de columnas tiene dos atributos: nombre y comparador, este último es la forma en cómo se presentarán los datos en una consulta [26].

   Y la tercera diferencia es que en el modelo relacional, las tablas contienen únicamente columnas y registros, mientras que en una familia de columnas puede haber columnas relacionadas conocidas como súper columnas comunes [26] como se observa en la figura 13.

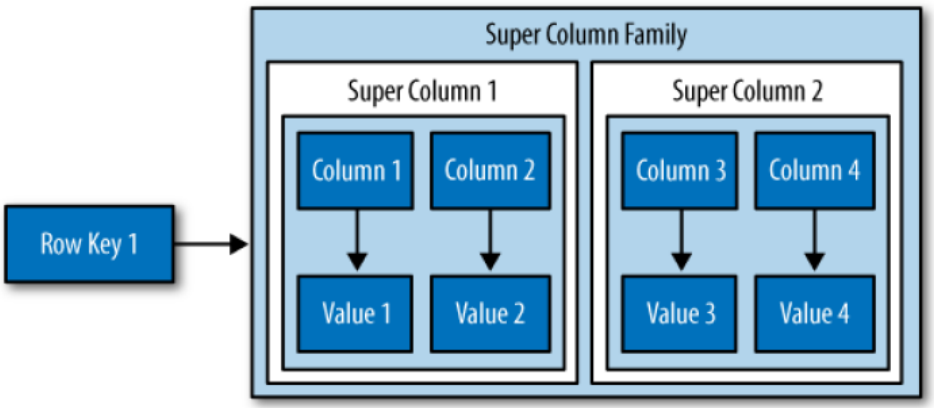


Figura 13. Estructura de Familias Súper Columnas Apache Cassandra [26]

Cuando escribes datos a una familia de columnas, especificas valores para una o más columnas, esa colección de valores juntos con una única llave primaria se conoce como registro, de este modo, un registro puede entenderse como un contenedor para columnas.

3.3.3 Componentes de Apache Cassandra

En Apache Cassandra el componente principal de su funcionamiento es la estructura de los nodos donde se almacena los datos por tal motivo se clasifica en los siguientes:

1. **Commit Log:** Fichero donde se almacena la información sobre los cambios en los datos. La función principal es recuperar los datos en caso de que ocurriera un fallo en el sistema [27].

2. **MemTable:** Estructura de almacenamiento en memoria, contiene los datos que aún no han sido escritos en un SSTable.

3. **SSTable:** Fichero de almacenamiento de datos escritos en disco, cada fichero SSTable es inmutable una vez haya sido creado.

4. **About Internode Communications (gossip):** Protocolo de comunicación peer-to-peer para descubrir y compartir información sobre la localización y el estado de los nodos en una clúster Apache Cassandra.

5. **Partitioner:** Determina como se distribuyen los datos entre los nodos (primera copia).

6. **Replica Placement Strategy:** Define la estrategia a seguir para almacenar copias de los mismo datos en diferentes nodos, de forma que se asegure la accesibilidad y la tolerancia a fallos, se puede definir diferentes estrategias (No hay copia principal ni copias secundarias de los datos, todas las copias incluida la primera, se replican) [27].

7. **Snitch:** Define la topología que utiliza las estrategias de replicación para colocar las réplicas y dirigir las consultas de forma eficiente.

3.4 Python

Es un lenguaje de programación potente y fácil de aprender, cuenta con estructuras de datos eficientes de alto nivel y un enfoque simple pero efectivo para la programación orientada a objetos. Es un lenguaje ideal para el desarrollo de scripting y aplicaciones rápidas en muchas áreas sobre la mayoría de plataformas ya que posee una elegante sintaxis y tipado dinámico [28].

Se puede utilizar esta herramienta como un lenguaje de scripting para ejecutar códigos, como un lenguaje procedimental para la organización de un programa en una colección de funciones que se pueden llamar entre sí, o como un lenguaje orientado a objetos que usa clases, herencias y módulos para crear una jerarquía. Cuando se está desarrollado y se utiliza lenguajes tradicionales como C o C#, se requiere compilar y vincular el código antes de volver a ejecutarlo, en el caso de Python el código modificado es visible en ese instante por lo que en la práctica utilizar Python tiene sus ventajas en el área de servicios web.

3.4.1 Características de Python

Python permite dividir el programa en módulos reutilizables desde otros programas Python. Posee una gran colección de módulos estándar que se pueden utilizar como base de los programas o como ejemplos al momento de a aprender a utilizar Python. También hay módulos incluidos que proporcionan E/S de ficheros, llamadas al sistema, Sockets y hasta interfaces a GUI como Tk, GTK, Qt entre otros.

Python se utiliza como lenguaje de programación interpretado, lo que ahorra un tiempo considerable en el desarrollo del programa, pues no es necesario compilar ni enlazar. El intérprete se puede utilizar de modo interactivo, lo que facilita experimentar

con características del lenguaje, escribir programas desechables o probar funciones durante el desarrollo del programa.

La sintaxis breve, concisa y expresiva de Python hace que sea fácil realizar operaciones complejas con sólo unas pocas líneas de código. Además existen formas estándar para llamar código C/C++ desde Python y viceversa, se puede encontrar librerías para hacer casi cualquier cosa en Python, una característica importante y muy utilizada es que se puede acoplar a cualquier módulo de desarrollo ayudando de forma exponencial la combinación de componentes de software [29].

4 Desarrollo de la plataforma terrena del UPMSat-2

Se debe tomar en consideración antes de mencionar el diseño e implementación de la plataforma de segmento en tierra del Satélite UPMSat-2 cuál es el objetivo del satélite y lo importante que es el Satélite UPMSat-2 para este trabajo fin de master.

El UPMSat-2 es un proyecto de microsatélite utilizable como plataforma de demostración tecnológica en órbita. El proyecto fue llevado a cabo en la Universidad Politécnica de Madrid (UPM), junto a otros grupos y empresas del sector espacial, dentro de la universidad ya se había abordado un proyecto similar, el UPM-Sat 1, un satélite universitario puesto en órbita el 7 de julio de 1995, con fines científicos y tecnológicos, con este proyecto se pretendía crear una base para la educación, la investigación y el desarrollo en el ámbito de la ingeniería espacial.

El satélite UPM-Sat 1 tiene una continuidad con el proyecto UPMSat-2, que comparte los mismo objetivos educativos, científicos y tecnológicos, pero con una complejidad técnica mayor. Entre las características del satélite se encuentra que gira alrededor de la Tierra a 600 km de altitud, las características del satélite físicamente proporcionan información que debe ser almacenada y administrada esto significaría que el satélite pasara por el mismo punto de la Tierra dos veces al día, siempre a las mismas horas, y durante aproximadamente 10 minutos en cada pasada. Este intervalo de tiempo hábil servirá para la comunicación, durante en el que puede haber visibilidad entre la antena del satélite y la antena situada en la Tierra, el satélite UPMSat-2 proporcionara información de aspectos técnicos y experimentos que son proporcionados por varios grupos de investigación y empresas [30]. Estos experimentos servirán para probar equipos y tecnología en entorno espacial.

Entre el satélite y tierra existe un sistema de comunicación que conecta el satélite con la plataforma en Tierra, permitiendo recibir señales desde la Tierra (telemando) y trasmitir información desde el satélite hacia la Tierra (telemetría). Las funciones de la estación de Tierra (o estación base) son mantener la comunicación con el satélite y determinar la posición del satélite, los parámetros orbitales y los tiempos de paso. Todas las operaciones que son ejecutadas por el ordenador del satélite como son: el Control de Actitud, Telecomunicaciones, Gestión de Datos, etc. Se controlan por el ordenador del satélite, esa información es la que la plataforma en Tierra almacenara y administrara.

Este proyecto ha desarrollado un prototipo de la plataforma del segmento en tierra la que receptará la información enviada por el satélite utilizando diferentes

componentes, a continuación se detallara la configuración e implementación de los componentes que fueron mencionados en el capítulo 3.

4.1 Descripción de Componentes Prototipo de la plataforma

En el equipo de investigación se ha desarrollado un prototipo de la plataforma usando una interfaz con GTK y GTKAda para definir los TC (Telemando) y TM (Telemetría) del satélite UPMSat-2, mediante los estudios ya realizados y la preparación de este proyecto, no se ha podido generar un componente para acceder a la información de la base de datos, esto se llevó a cabo antes de ejecutar este trabajo fin de master donde además se utilizó SQLite, como base de datos de la información proporcionada por el satélite.

El prototipo inicial ha proporcionado las funciones fundamentales de la plataforma. Los plazos iniciales para lanzar el satélite se han retrasados, lo que ha dado tiempo para experimentar y mejorar algunas características como son las garantías de escalabilidad, disponibilidad y flexibilidad. El uso de plataformas orientadas a sistemas en la nube, base de datos NoSQL, como Apache Cassandra y servidores web, para la interacción con el sistema de vuelo.

Los problemas que presentaba el prototipo antes de la realización del trabajo fin de master, era que mantenerlo requería de costos relativamente altos, aparte de lo que significaría una gran inversión en recurso humano. El objetivo es desarrollar una plataforma que maneje comunicación con varias tecnologías y no necesite de una gran cantidad de recursos para funcionar al cien por ciento.

Este proyecto cumple con el objetivo de ayudar a docencia y a investigadores a tener una visión más clara del funcionamiento y experimentos realizados en órbita.

Como se muestra en la figura 14 se puede observar el diseño del prototipo final del proyecto que como objetivo tiene la comunicación entre la estación terrena y el satélite UPMSat-2. El componente que se ha desarrollado en Ada se debe mantener ya que permite conectar con el radio-enlace para interaccionar con el satélite, las características del lenguaje de programación aconseja su utilización.

Actualmente, no se disponen bibliotecas para interaccionar directamente con Cassandra, por este motivo, se ha incluido un componente de intermediación con la base de datos usando Python. El objetivo fundamental de este proyecto ha sido desarrollar un prototipo básico que permita validar la tecnología propuesta. En la figura 14, se muestra un sistema similar al que sería en el sistema final.

Para la validación del desarrollo del sistema, se ha hecho una versión que permite conectar al satélite con el segmento de tierra mediante *sockets*. Se han sustitutos (*stubs*) de los componentes de software reales de la entrada/salida. De forma, que los telemandos y telemetría se mandan mediante *sockets*, para probar el software desarrollado.

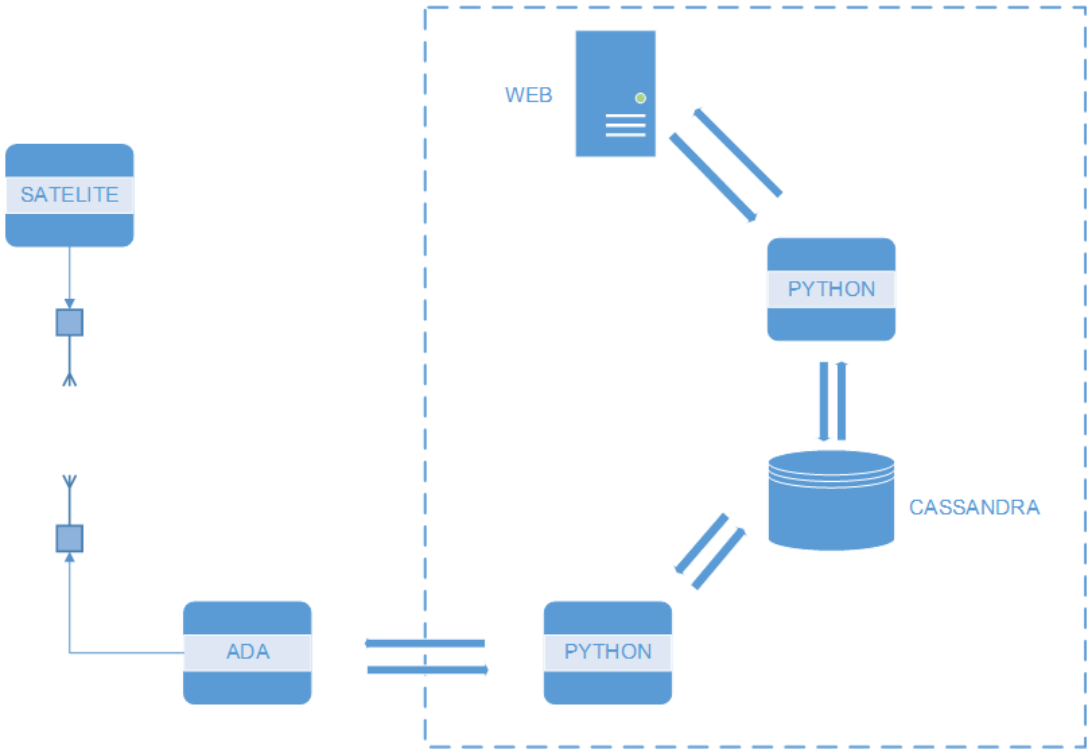


Figura 14. Componentes Prototipo de satélite UPMSat-2

Mediante este enfoque se ha decidido desarrollar el prototipo incremental para validar esta propuesta. El objetivo fundamental en este proyecto ha sido completar un prototipo de la estructura básica que se muestra en la figura 14. Con esta idea, se han utilizado ficheros con datos reales del sistema de satélite, en lugar del componente en Ada para radio-enlace.

Por otro lado, se ha desarrollado un componente para interactuar entre el sistema en Ada terrenal con una biblioteca con Python. Si el objetivo inicial se cumpla con éxito, se podría completar con poco esfuerzo el sistema mostrado en la figura 14.

Teniendo en cuenta que para el funcionamiento completo del prototipo del satélite es necesario mencionar varias características importantes de los componentes que se muestran en la figura 14, Ada y la comunicación entre satélite y base terrena es importante definirlas conforme a los componentes ya generados en estudios posteriores.

4.1.1 Ada

Ada conforme a las especificaciones técnicas del satélite es un lenguaje de programación de alto nivel de propósito general, especialmente indicado en aplicaciones de gran tamaño donde se requiere un alto grado de fiabilidad y eficiencia, particularmente en sistemas de tiempo real embebidos.

El software del satélite ha sido elaborado utilizando Ada en un estándar del 2005, las características de Ada que ofrece al programador son: estructura de control, capacidades de composición de datos y creación de tipos, estructuras para modularización del código y programación orientada a objetos, manejo de excepciones, programación concurrente, etc. Todo lo antes mencionado con una sintaxis simple y limpia que permita mayor facilidad a la hora de programar.

Ada proporciona recursos suficientes para programación de bajo nivel, y programación concurrente, estas características son útiles para sistemas embebidos y de tiempo real, en los que el programa tiene que interactuar con dispositivos físicos en un tiempo determinado.

4.1.2 Mensajes de Telemetría (TM) y Telemandos (TC)

En este apartado se describe el subsistema (TTC), de acuerdo con los requisitos funcionales extraídos de [30].

El satélite UPMSat-2 tiene comunicación con la estación de Tierra a través de mensajes de telemetría y telemandos intercambiados sobre un radio enlace, los mensajes de telemetría (TM), se envían desde el satélite hacia la Tierra, conteniendo información sobre el estado de funcionamiento del satélite.

Los telemandos (TC), son enviados de la Tierra al satélite, son órdenes remotas que se han de ejecutar en el sistema del satélite para cambiar su operación.

Embarcado en el satélite se encuentra un equipo de radio operación trasmisión/receptor de la empresa Emxys que proporciona el modem y antena de comunicación.

- **Tipos de Mensajes de Datos**

El equipo de comunicación puede procesar cuatro tipos de estructura de datos, dos de las cuales contienen datos para la configuración del propio equipo y se envían desde

el software abordo o desde Tierra, como por ejemplo se pude utilizar banda de frecuencia de transmisión o cambiar de tasa de bits, otro tipo de datos son los mensajes de telemetría y telemando intercambiados entre la estación de Tierra y el satélite.

El subsistema de TTC maneja solo mensajes de telemetría y telemando, en la tabla 2 se describe los tipos de formato de los mensajes para telemetría (TM).

El mensaje (*Hello*) contiene información sobre la operación del sistema, la marca de tiempo en el momento que se envía el mensaje y el último valor leído por los sensores de la plataforma, en el mensaje de error de eventos (*ErrorEvent*) se envía el registro de los errores y eventos detectados durante la ejecución del software. Un evento es un cambio de modo de operación en el satélite.

El mensaje *(Housekeeping)* corresponde a medidas anteriores sobre el estado de la plataforma, que se han almacenado en memoria para enviar a Tierra como por ejemplo: 8voltajes, temperatura, nivel de batería, medida de magnetómetros, etc.

Categoría	***Mensaje***	***Contenido***
General	Hello	Tabla del estado del Sistema
	Error Event	Error y Eventos registrados durante la operación del sistema
	Housekeeping	Registro de datos del estado de la plataforma

Tabla 2. Tipos de Mensajes de Telemetría

Mientras por parte de los telemandos recibidos por el satélite permite a la estación terrena realizar varias funciones que son: cambiar el modo de operación del sistema, cambiar los parámetros de configuración de los subsistemas, cambiar la configuración de los sensores y actuadores, cambiar los parámetros de control del algoritmo de control de actitud y empezar y parar la ejecución de un experimento si fuera necesario hacerlo.

En la tabla 3, se detalla los telemandos definidos por el documento de requerimientos funcionales del satélite UPMSat-2 [30].

Categoría	*Mensaje*	*Contenido*
TTC	OpenLink	Visibilidad de Intervalos
	SetLostCommTimmer	Intervalo de Tiempos Nuevo valor de temporizador
Manager	Change Mode	Nuevo modo de operación del sistema
Platform	Set Sampling Period	Grupos, Segundos
	Enable Digital Signal	Señal
	Disable Digital Signal	Señal

Tabla 3. Tipos de Mensajes de Telemando

4.1.3 Funciones de Subsistema de Telemetría (TM) y Telemandos (TC)

Existe un intervalo de visibilidad en el que existe una fracción de tiempo durarte la cual el satélite se encuentra bajo la zona de cobertura de la antena de la estación base y puede haber comunicación.

La duración de este intervalo viene determinado por la órbita polar heliosíncrona que describe el satélite, en cada vuelta completa que realiza a la Tierra, el satélite permanece visible desde un mismo punto de la Tierra durante aproximadamente 10 minutos, el intervalo de visibilidad se detecta por el software del satélite, al recibir un telemando de tipo *OpenLink* y termina cuando ocurre estos sucesos: Cuando no se ha recibido ningún telemando en los últimos 60 segundos y cuando desde el inicio del intervalo ha transcurrido la duración máxima de comunicación que en este caso es de 10 minutos.

- **Intercambio de Mensajes entre Satélite UPMSat-2 y Base Tierra**

Los mensajes de telemetría (TM) se envían a la estación base de Tierra tanto si existe comunicación o no exista, los tipos de mensaje que se envían son diferentes en cada caso. En la tabla 4 se muestra los tipos intercambio de mansajes que pueden ser visibles o no visibles [30].

Estado	*Tipo de Mensaje*	*Modo de Envió*	*Banda de Frecuencia*
No Visible	Hello	Periódico	Radioaficionados
Visible	Hello, ErrorEvent, Housekeeping	Aperiódico	Investigación espacial

Tabla 4. Intercambio de Telemetría en estados Visibles / No Visibles

Cuando no existe visibilidad el satélite no debe enviar información sensible sobre el funcionamiento del satélite, se envía mensajes simples del tipo (*Hello)*, cuando existe visibilidad, se trasmite toda la información almacenada para enviar a Tierra.

Los tipos de mensajes enviados son los que se muestran en la tabla 2, hasta realizar las pruebas de funcionamiento correcto de los datos enviados por el satélite, se está proporcionado la misma información en todos los casos.

El satélite está trasmitiendo un mensaje de tipo (*Hello)*, la estación en Tierra detecta que el satélite está en su zona de cobertura cuando recibe uno de estos mensajes, y trasmite entonces un telemando *OpenLink* para indicar al satélite que empieza el intervalo de visibilidad. Al recibir este telemando, se cambia el transmisor del satélite a la banda de frecuencia adecuada y se trasmite un mensaje telemetría con la información del estado del sistema.

El satélite queda a la espera de recibir telemandos por parte de la estación en Tierra, esta secuencia es posible generarla con la implementación de la base de datos y su comunicación, se pueden cargar los comandos necesarios en Apache Cassandra y al momento de tener comunicación (intervalo de visibilidad) enviarlos al satélite mediante funciones generadas por Python.

Para la recepción de los telemandos que son enviados desde la estación de Tierra durante el intervalo de visibilidad, permite un enlace ascendente entre la antena del satélite y la antena de la base en Tierra, significaría que en cualquier modo de operación del sistema se puede recibir un telemando. Las ordenes enviadas de los telemandos se pueden ejecutar de dos maneras: Mediante una ejecución programada (en un tiempo futuro especifico) y mediante una ejecución inmediata (tan pronto sea posible) desde la recepción. En la figura 15 se observa el esquema de envió.

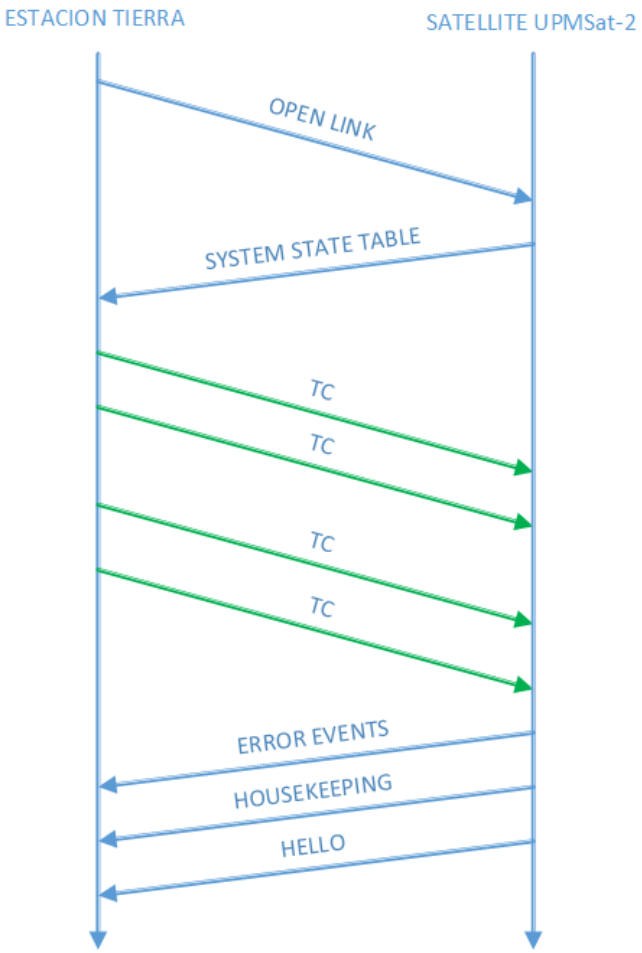


Figura 15. Esquema Envió de Mensajes de TC y TM

4.2 Implementación de Plataforma de Segmento en Tierra del Satélite

Se describe el diseño de la plataforma utilizada en el desarrollo de este trabajo para realizar el segmento en Tierra, que almacenará y administrará la información proporcionada por el satélite UPMSat-2, comprende en el uso de componentes que permite el levantamiento del clúster con los parámetros necesarios para la optimización del almacenamiento de los datos, la alta disponibilidad y redundancia de los mismos.

En la figura 16 se detalla los componentes principales que formarán parte del diseño de la plataforma utilizada en este proyecto definido en tres principales áreas, las cuales se detallaran más adelante.

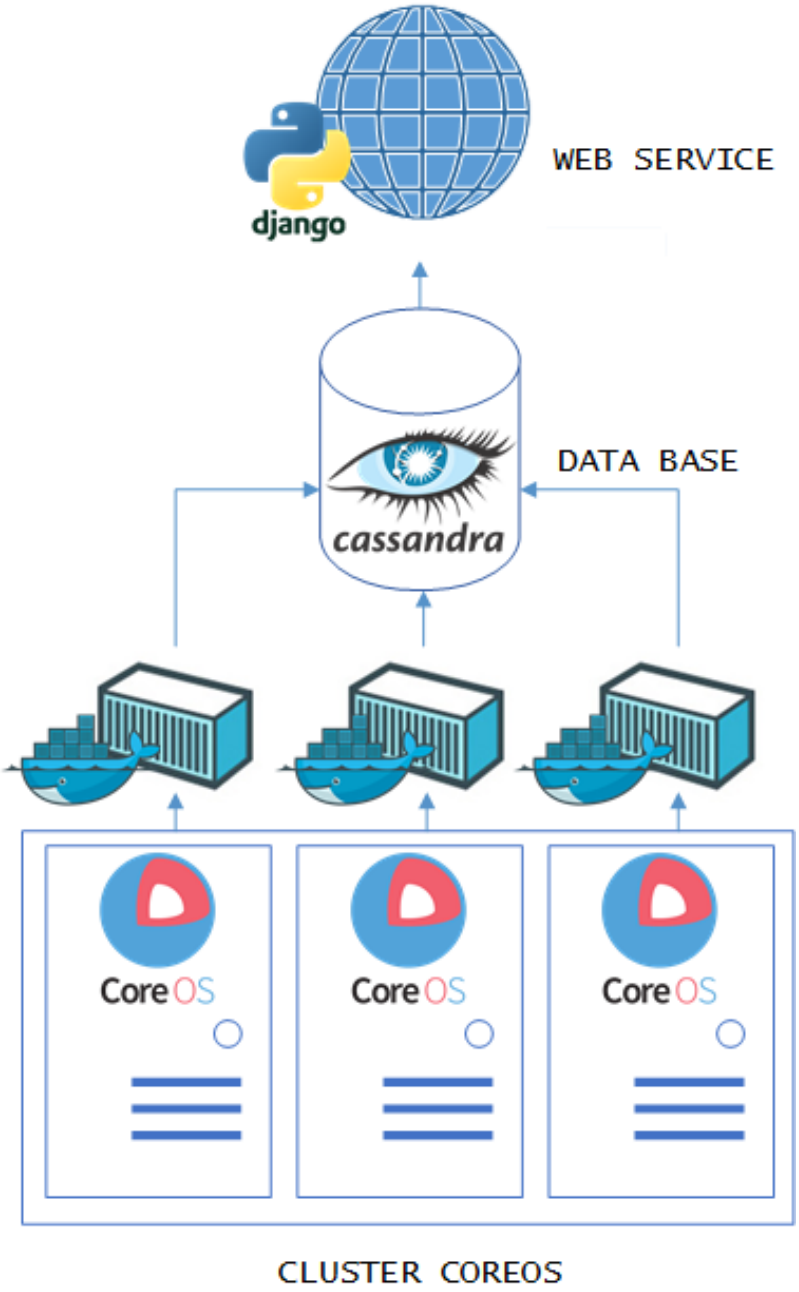


Figura 16. Diseño de la plataforma en Tierra

4.2.1 Características de los Componentes de Hardware de la Plataforma

Como parte de la implementación se requiere determinara las características de los componentes físicos, como se muestra en la tabla 5 se detalla nombres de la maquinas, sistemas operativos, memoria RAM, procesador y almacenamiento utilizado por el segmento en Tierra.

Componentes Hardware de la Plataforma				
Nombre de Maquina	***Sistema Operativo***	***Memoria RAM (GB)***	***Procesador***	***Almacenamiento (GB)***
CoreOS01	Coreos Ver.(1576.5.0)	16	Intel Core i5-7300 CPU 3.60GHz	1024
CoreOS02	Coreos Ver.(1576.5.0)	16	Intel Core i5-7300 CPU 3.60GHz	1024
CoreOS03	Coreos Ver.(1576.5.0)	16	Intel Core i5-7300 CPU 3.60GHz	1024
Admin-Coreos	Linux Ubuntu 16.04 LTS	16	Intel Core i7-7700 CPU 3.60GHz x 8	1024

Tabla 5. Componentes Físico de la Plataforma

4.2.2 Características de los Componentes de Software de la Plataforma

En los componentes de hardware se instalaron varias herramientas que fueron configuradas para el correcto desempeño de la plataforma de segmento en Tierra del satélite, las cuales se describen a continuación.

4.2.3 Implementación y Diseño de CoreOS

Como se observa en la figura 16 en los puntos anteriores, se implementó un clúster de CoreOS que tiene una arquitectura interna definida, la que servirá para determinar la cantidad de nodos que serán utilizados.

En este caso se utilizó una estructura de 3 nodos, está estructura es ideal para entornos de desarrollo la cual este formada por nodos de actividades parecidas. Al tratarse de un clúster de arquitectura simple cada nodo tiene una instancia de ETCD local, que sirve de comunicación entre todos los nodos. En la figura 17 se puede observar el diagrama de entorno de un clúster en CoreOS con las características principales para el funcionamiento óptimo del clúster.

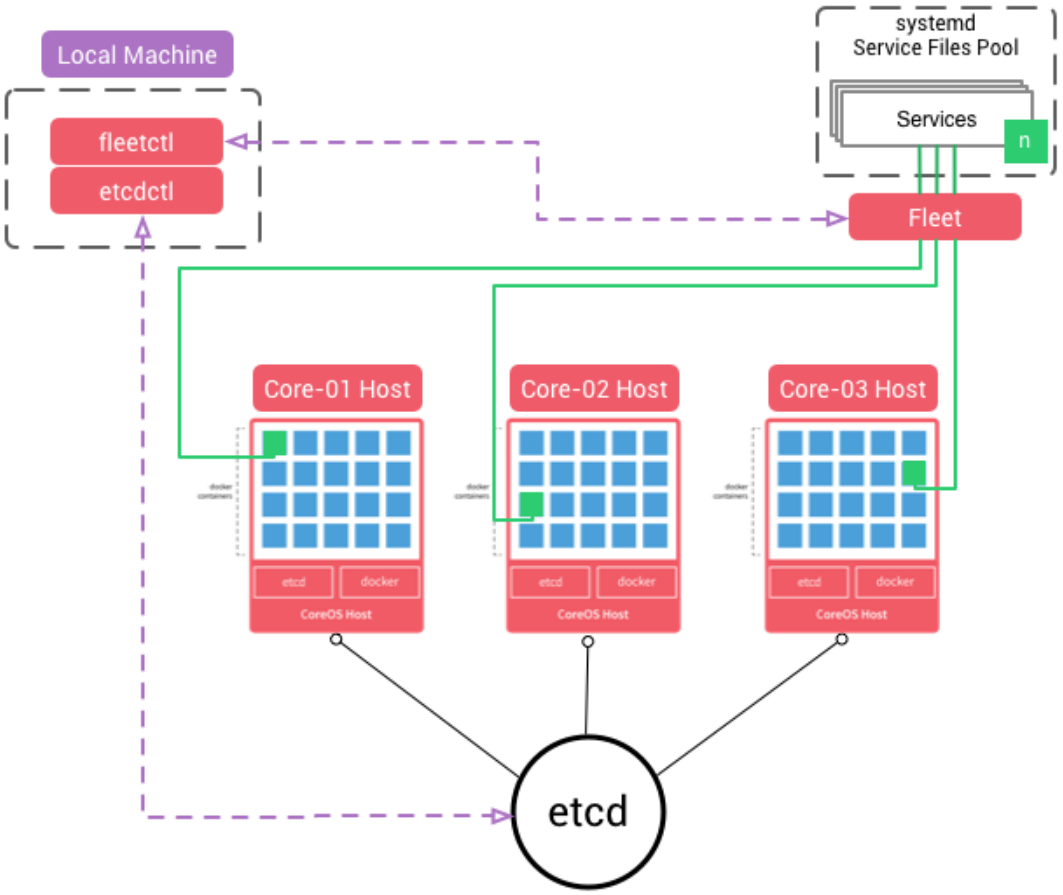


Figura 17. Diagrama de Entorno de Clúster CoreOS

En nuestro caso para la evaluación y comprobación del sistema y como requisito principal en la elaboración del trabajo fin de master si implemento el clúster en equipos físicos (*Bare Metal*), en el apartado de Apéndice 1, se podrá observar a profundidad los pasos que se deben seguir para el levantamiento del sistema.

Se describirán los archivos de configuración necesarios, en el caso del escenario expuesto se toma en consideración la utilización de un archivo propio de CoreOS llamado *Cloud-Config*.

Cloud-Config es un archivo escrito en YAML que es interpretado por el sistema de ficheros (*cloud-init*), permitiendo personalizar declarativamente varios elementos a nivel del sistema operativo, como por ejemplo la configuración de red, las cuentas de usuario, las unidades del sistema, etc. Este archivo es utilizado para la inicialización del sistema, como se mencionó anteriormente guarda toda la información relevante de todos los servicios que se encuentran levantados por parte del clúster.

Debido a que el sistema de fichero (*cloud-init*) incluye herramientas que no son parte de CoreOS, solo el subconjunto relevante de sus elementos de configuración se

implementara en nuestro archivo de configuración, además de esto el archivo de configuración posee varios elementos específicos de CoreOS, como la configuración de ETCD, la definición de servicios y las unidades de almacenamiento del sistema. Como se observa en la figura 18 se puede ver un ejemplo de un archivo *cloud-config* con varios parámetros necesarios para la configuración del clúster.

```
#cloud-config

coreos:
  etcd:
    name: "node001"
    # generate a new token for each unique cluster from https://discovery.etcd.io/new
    discovery: "https://discovery.etcd.io/<token>"
    # multi-region and multi-cloud deployments need to use $public_ipv4
    addr: "$public_ipv4:4001"
    peer-addr: "$private_ipv4:7001"
```

Figura 18. Ejemplo de Archivo Cloud-Config CoreOS

El archivo *cloud-config* se ejecutará cada vez que un nodo arranque, si existiera alguna falla o daño del archivo, automáticamente será descartado y no se ejecutara hasta que sea corregido, CoreOS tomará este error y lo depositara en una bitácora (log) con las especificaciones del error.

A continuación se detallará a profundidad los parámetros y características del archivo *cloud-config* que se utilizó para el levantamiento del clúster de la plataforma de segmento en tierra del satélite UPMSat-2.

Al ser un archivo YAML se deben definir una serie de parámetros utilizando indentación. Como se mencionó anteriormente, el archivo se encuentra dividido en tres bloques que son: Servicio ETCD, Unidades y Usuarios.

1. **Servicio ETCD:** En este bloque determina la comunicación y configuración de los servidores que formaran parte del clúster, se determinó que para el proyecto se utilizara la versión de ETCD2, como parte de la configuración y estándar se define las listas de URL pares del miembro anunciarse (initial_advertise_peer_urls) en el resto del clúster, Estas direcciones se utilizan para comunicar datos ETCD en todo el clúster.

   Al menos una de las direcciones debe ser enrutable a todos los miembros del clúster y estas URL pueden contener nombres de dominio. En el caso de clúster para la plataforma se utilizó direcciones de IP fijas que fueron proporcionadas.

Aparte de configurar las URL en el servicio ETCD2 se configura un punto importante en el desarrollo del clúster, es aquí donde se describe la conexión entre los servidores para conformar el clúster, (*initial_cluster*) menciona los nombres de cada servidor y las IP, utilizando el formato de URL como comunicación. Es necesario determinar el estado del clúster ("Nuevo o Existente") en este caso se determinó el estado de un clúster (nuevo) que estará presente durante el arranque inicial de cada nodo, si esta opción estuviera como (existente), ETCD intentara unirse a un clúster existente.

Si se establece un valor incorrecto, ETCD intentara iniciarse pero automáticamente fallara de forma segura.

2. **Unidades:** En este parámetro se definen los servicios que serán utilizados en el clúster, en el caso de nuestro proyecto se utilizaran tres servicios importantes.

   Fleet pertenece a CoreOS y proporciona una unión entre systemd y ETCD en un sistema distribuido simple. Está diseñado como base para una orquestación de órdenes superiores, es una elaboración de un clúster en unidades de systemd, totalmente diferente a un administrador de contenedores o un sistema de orquestación.

   Systemd–Network proporciona la activación y determinación de la IP que se va a utilizar en los nodos, como punto principal se debe determinar el puerto establecido por el sistema operativo CoreOS en nuestro caso la interfaz de red tiene el nombre de "enp0s3", las IP que se utilizó para los nodos fue 138.4.11.224, 138.4.11.225 y 138.4.11.226 la puerta de enlace *(gateway)* configurado es 138.4.11.131 con acceso a la internet.

   Docker-Service es una plataforma embebida en el sistema operativo que debe ser activada al momento de instalar, como parte de este trabajo es necesario la activación de Docker para la instalación del servicio que manejara Cassandra y la comunicación que tendrá entre el clúster y la base de datos.

   Flannel es una forma simple y fácil de configurar una capa de red interna, Flannel ejecuta un pequeño agente binario único llamado "Flanneld" en cada host, y es responsable de asignar una concesión de subred a cada host fuera de un espacio de direcciones preconfiguradas [31].

Flannel utiliza la API de Kubernetes, directamente para almacenar la configuración de red, las subredes asignadas y cualquier información auxiliar (como la IP publica del host). Los paquetes se reenvían usando uno o varios mecanismos de back-end que incluyen redes LAN extensibles (VXLAN) y varias integraciones en la nube.

Las plataformas que utilizan este tipo de redes se supone que en cada contenedor tiene una única IP enrutable dentro el cluster, la ventaja de estos modelos es que se elimina la difícil tarea del mapear los puertos que provienen de compartir una sola IP host.

Flannel es responsable de proporcionar una red IPv4 de capa 3 entre varios nodos en un clúster. Flannel no controla como se conecta en red los contenedores al host, solo como se transporta el tráfico entre los hosts.

3. **Usuario:** Parte del sistema de inicio de CoreOS es la configuración de un usuario Root que tenga los privilegios de súper usuario para la instalación y administración de paquetes, y nuevos contenedores Docker.

Como se muestra en la figura 19 la arquitectura física del clúster contaría con los siguientes servidores y conexiones entre equipos, esto hace referencia al diseño interno de CoreOS como se pudo observar en la figura 5.

En el punto anterior se ha puesto en marcha el funcionamiento del clúster que dispone de tres nodos, a continuación se configurar los servicios que estarán alojados en el clúster de CoreOS.

El proceso de configuración e instalación del archivo *cloud-Config* se encuentra en el Apéndice 1.

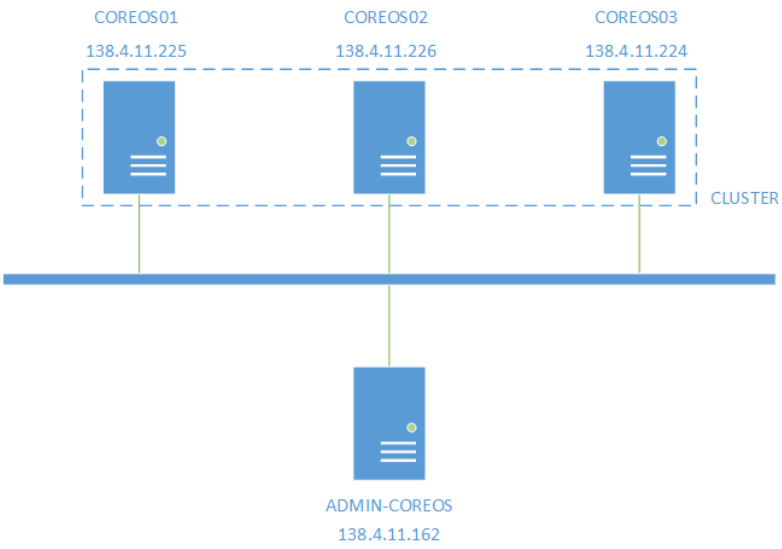


Figura 19. Arquitectura Física del Clúster

4.3 Instalación y configuración de componentes: Docker, Apache Cassandra y Servidor Web

Al tener en cuenta el diseño de la plataforma, es impórtate detallar mediante un esquema practico la disposición de cada uno de los componente que conforman la plataforma terrestre como se detalla en la figura 20 se puede ver la comunicación entre todos los componentes para generar un desarrollo concreto del proyecto.

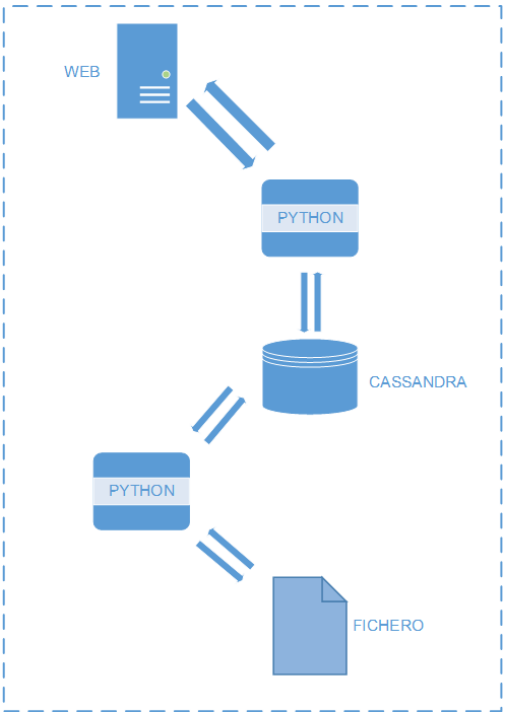


Figura 20. Componentes de Software de la Plataforma

La estructura genérica que se sigue a la hora de crear un servicio distribuido a través del clúster se basa en tener dos archivos, un fichero de servicio y un archivo de presentación o descubrimiento. Antes de introducirnos la configuración de los archivos y ficheros es importante conocer qué lenguaje o estructura utilizaremos para estos servicios.

Como ya hemos visto anteriormente CoreOS utiliza *systemd* para gestionar el conjunto de procesos, si queremos por tanto crear un servicio de este tipo y ponerlo en marcha también con el servicio *Fleet* necesitaremos conocer la estructura de estos ficheros.

Como se muestra en la figura 21 no es necesario que se disponga de todos los apartados que se muestran en el ejemplo de estructura de servicio, muchos de los apartados en esta estructura no serán relevantes en esta investigación por tal motivo no se profundizo en la funcionalidad de cada uno de ellos. La parte de Unit se encarga de describir qué tipo de servicio es y especifica también si está ligado a otro servicio o no, también puede especificar si va antes o después.

El apartado de Service se permite especificar una serie de comandos que se ejecutarán antes (*ExecStartPre*), en el momento de ejecución del servicio (*ExecStart*) y una vez este se detenga (*ExcStop*). En el último apartado nos permite especificar el objetivo de este servicio, esta es la estructura básica, si queremos hacer funcionar un servicio en local en uno de los nodos.

Para las aplicaciones distribuidas las cuales se van ejecutar dentro del clúster de CoreOS, necesitaremos utilizar la estructura de *fleet*, la cual es exactamente igual en todos los apartados menos en el último.

```
[Unit]
Description=MyApp
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill busybox1
ExecStartPre=-/usr/bin/docker rm busybox1
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name busybox1 busybox /bin/sh -c "while true;
do echo Hello World; sleep 1; done"

[Install]
WantedBy=multi-user.target
```

Figura 21. Ejemplo de Estructura de Servicios en CoreOS

El archivo o fichero de presentación o descubrimiento (Service Discovery) básicamente registrar una clave *etcdctl* con un tiempo de vida determinado, También realiza un bucle en que va comprobando si el servicio está activo y en buen estado para mantener la clave o descartarla. Como se puede observar en la figura 22 muestra un ejemplo de un fichero de descubrimiento.

```
[Unit]
Description=Apache web server on port %i etcd registration

# Requirements
Requires=etcd.service
Requires=apache@%i.service

# Dependency ordering and binding
After=etcd.service
After=apache@%i.service
BindsTo=apache@%i.service

[Service]

# Get CoreOS environmental variables
EnvironmentFile=/etc/environment

# Start
## Test whether service is accessible and then register useful information
ExecStart=/bin/bash -c '\
  while true; do \
    curl -f ${COREOS_PRIVATE_IPV4}:%i; \
    if [ $? -eq 0 ]; then \
      etcdctl set /services/apache/${COREOS_PRIVATE_IPV4} \
        '${COREOS_PRIVATE_IPV4}:%i\' --ttl 30; \
    else \
      etcdctl rm /services/apache/${COREOS_PRIVATE_IPV4}; \
    fi; \
    sleep 20; \
  done'

# Stop
ExecStop=/usr/bin/etcdctl rm /services/apache/${COREOS_PRIVATE_IPV4}

[X-Fleet]
# Schedule on the same machine as the associated Apache service
X-ConditionMachineOf=apache@%i.service
```

Figura 22. Ejemplo de Fichero de Descubrimiento en CoreOS

Es necesario que en el apartado de describir el servicio (Unit), se definió que se necesita del servicio en sí para funcionar. También es preferible que se describa en el apartado de fleet (*X-fleet*) que el servicio debe ejecutarse en la misma máquina.

Después tener haber visto el proceso que se encarga de publicar el servicio al ETCD, se debe configurar otro proceso que se encarga de hacer funcionar el servicio. En otras palabras lo que se hace es crear un contenedor Docker a partir de una imagen y ejecutar lo que contenga dentro. En nuestro caso para la plataforma del segmento en Tierra del satélite UPMSat-2 alojara una base de datos Apache Cassandra.

Se puede observar en la figura 23, lo que hace un servicio cuando se inicia es mirar si hay algún otro servicio como él para eliminarlo y destruirlo. Una vez hecho esto realiza

un pull de la imagen de Docker que posteriormente ejecutará. Después inicia el servicio de Docker con un nombre determinado, en una ubicación determinada y con una imagen específica.

Para poder ejecutar más de una instancia de un servicio en una misma máquina (o en una diferente) podemos asignar varios puertos sin necesidad de modificar el archivo original, simplemente el nombre del archivo será (servicionuevo@.service). Así cuando creamos la instancia dentro del fichero podemos recuperar el valor con el comando %i.

```
[Unit]
Description=Apache web server service on port %i

# Requirements
Requires=etcd.service
Requires=docker.service
Requires=apache-discovery@%i.service

# Dependency ordering
After=etcd.service
After=docker.service
Before=apache-discovery@%i.service

[Service]
# Let processes take awhile to start up (for first run Docker containers)
TimeoutStartSec=0

# Change killmode from "control-group" to "none" to let Docker remove
# work correctly.
KillMode=none

# Get CoreOS environmental variables
EnvironmentFile=/etc/environment

# Pre-start and Start
## Directives with "=-" are allowed to fail without consequence
ExecStartPre=-/usr/bin/docker kill apache.%i
ExecStartPre=-/usr/bin/docker rm apache.%i
ExecStartPre=/usr/bin/docker pull hirocat/apache
ExecStart=/usr/bin/docker run --name apache.%i -p ${COREOS_PRIVATE_IPV4}:%i:80 \
hirocat/apache /usr/sbin/apache2ctl -D FOREGROUND

# Stop
ExecStop=/usr/bin/docker stop apache.%i

[X-Fleet]
# Don't schedule on the same machine as other Apache instances
X-Conflicts=apache@*.service
```

Figura 23. Ejemplo de Fichero de Servicio en CoreOS

En el apartado (*X-fleet*) del final del fichero podemos especificar si queremos que funcione en una máquina específica o si queremos permitir que haya múltiples servicios iguales corriendo en una misma máquina.

El proceso de configuración e instalación del servicio que alojara la base de datos Apache Cassandra se encuentra descrita en el Apéndice 1.

4.3.1 Comunicación de componentes con Servidor Web

Python es un lenguaje que se usa en varias áreas de la tecnología, como por ejemplo: en el área web, redes, procesamiento de datos, inteligencia artificial, etc. Ha sido desarrollado para programadores que están incursionando en esta área, es muy sencillo de familiarizarse con la sintaxis del lenguaje. El uso de expresiones comunes hace que Python requiera menos líneas de código para realizar tareas básicas. Según [32] el promedio del código escrito en Python es de tres a cinco veces más corto que de Java y cinco a diez veces más corto que C++.

Adicionalmente, Python tiene una librería estándar que permite ejecutar otras funciones y tareas más complejas con mayor facilidad que otros lenguajes y la comunicación con otras aplicaciones resultan más eficiente que otras herramientas de programación, Python es compatible con sistemas operativos Linux, OSX, Windows.

Por este motivo, para el desarrollo de la plataforma de segmento en Tierra del satélite UPMSat-2, se decidió implementar como medio de comunicación entre la información proporcionada por el satélite (file) y la base de datos (Ver Figura 14), una aplicación desarrollada en Python (.py) que generara un puente entre Apache Cassandra y Python.

Se utilizó la versión 2.7 de Python y la versión 3.11.2 de Apache Cassandra, debido a que la compatibilidad entre estas dos aplicaciones es óptima.

Como entorno de desarrollo integrado (IDE), se escogió el sistema PyCharm, que proporciona análisis de código, un depurador de gráficos, un probador de unidades integrado, integración con el sistema de versionalidades (VCS) y admite el desarrollo de entornos web con la utilización de django.

Entre las características que presenta Pycharm se puede encontrar que da asistencia y análisis de codificación, navegación de proyecto y código utilizando vistas de estructura de archivos, soporte para *frameworks* web, depurador integrado de Python, entre otras. La versión estable utilizada para el desarrollo del programa es la 181.5087.37 publicada en mayo del 2018. A continuación se detallara los scripts utilizados para la comunicación entre Apache Cassandra y la integración de la información proporcionada por el satélite UPMSat-2 entregada en formato de archivo plano.

- **Enlace de comunicación entre Apache Cassandra y Servidor Web**

En la figura 24 se visualiza la implementación de cassandra_engine, el nombre de la base de datos y las IP que forman parte del clúster de CoreOS, como adicional es aquí

donde se determina el factor de replicación en nuestro caso será de 3 por tener configurado un clúster de 3 nodos, previamente se debe tener configurado e instalado cassandra.cqlengine y django cassandra engine.

```
# Database
# https://docs.djangoproject.com/en/1.11/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django_cassandra_engine',
        'NAME': 'dbsatellite',
        'HOST': '138.4.11.225 138.4.11.226 138.4.11.224',
        'OPTIONS': {
            'replication': {
                'strategy_class': 'SimpleStrategy',
                'replication_factor': 3
            }
        }
    }
}
```

Figura 24. Código de Conexión Apache Cassandra y Servidor Web

- **Creación de tablas en la Base de Datos Apache Cassandra**

Se creó una clase llamada Modelos que realiza la creación de las tablas en la base de datos, utilizando los objetivos del proyecto y según la teoría expuesta en apartados anteriores, se crean tres tablas con el nombre de EventErroComm, HelloComm y HouseKeepComm. La información que va a ser alojada dentro de estas tablas se encuentra totalmente desnormalizada, es por eso que se crea de una clave primaria compuesta y una clave de clusterizacion, la cual comparten todas las tablas para poder realizar consultas a futuro como por ejemplo los intervalos de tiempo entre cada acción que se ejecute en el satélite o filtrar por la telemetría que es proporcionada desde el satélite hacia la estación base.

Es importante recalcar que para este proyecto fin de master y para el objetivo de esta investigación, se mencionó anteriormente que se utilizara como clave primaria compuesta (año y el instante de subida del registro a la base de datos), información proporcionada por el satélite en Apache Cassandra, y para obtener un orden en los datos de cada tabla se crea una clave de clusterizacion para especificar qué campo se va a ordenar por nodo. Al ser una base de datos distribuida los datos son distribuidos alrededor de todos los nodos por ese motivo se utiliza esta forma de determinar las claves primarias.

Los demás registros de las tablas fueron creados según el tipo de dato establecido por parte del archivo plano como se muestra en la figura 25.

```
class EventErrComm(DjangoCassandraModel):
    #uid_ev = columns.TimeUUID(primary_key=True, default=uuid.uuid1)
    year = columns.Integer(primary_key=True, partition_key=True)
    post_time = columns.DateTime(primary_key=True, clustering_order="DESC")
    eve_name = columns.Text()
    mis_clock = columns.BigInt()
    seq_number = columns.Integer()

    class Meta:
        get_pk_field = 'post_time'

    def __str__(self):
        return "Error Event {} with clock {} and sequence {}.".format(self.eve_name, self.mis_clock, self.seq_number)


class HelloComm(DjangoCassandraModel):
    #uid_he = columns.TimeUUID(primary_key=True, default=uuid.uuid1)
    year = columns.Integer(primary_key=True, partition_key=True)
    post_time = columns.DateTime(primary_key=True, clustering_order="DESC")
    battery = columns.Text()
    boom1_vbus = columns.Text()
    boom2_vbus = columns.Text()
    mis_clock = columns.BigInt()
    mts_vbus = columns.Text()
    n15v_tm = columns.BigInt()
    ope_mode = columns.Text()
    p15v_tm = columns.BigInt()
    p3v3_tm = columns.BigInt()
    p5v_tm = columns.BigInt()
    rw_p5v = columns.Text()
    seq_number = columns.Integer()
    ttc_stat = columns.Text()

    class Meta:
        get_pk_field = 'post_time'

    def __str__(self):
        return "Hello Event, operating mode {} with clock {} and sequence {}.".format(self.ope_mode, self.mis_clock,
                                                                                       self.seq_number)
class HouseKeepComm(DjangoCassandraModel):
    #uid_hk = columns.TimeUUID(primary_key=True, default=uuid.uuid1)
    year = columns.Integer(primary_key=True, partition_key=True)
    post_time = columns.DateTime(primary_key=True, clustering_order="DESC")
    battery = columns.Text()
    boom1_vbus = columns.Text()
    boom2_vbus = columns.Text()
    mis_clock = columns.BigInt()
    mts_vbus = columns.Text()
    n15v_tm = columns.BigInt()
    ope_mode = columns.Text()
    p15v_tm = columns.BigInt()
    p3v3_tm = columns.BigInt()
    p5v_tm = columns.BigInt()
    rw_p5v = columns.Text()
    seq_number = columns.Integer()
    ttc_stat = columns.Text()

    class Meta:
        get_pk_field = 'post_time'

    def __str__(self):
        return "Housekeeping Event, operating mode {} with clock {} and sequence {}.".format(self.ope_mode,
                                                                                              self.mis_clock,
                                                                                              self.seq_number)


class Checksum(DjangoCassandraModel):
checksum = columns.Text(primary_key=True)
```

Figura 25.Código de Creación de Tablas en Cassandra mediante conexión con Servidor Web

- **Carga de archivo texto plano proporcionado por el Satélite**

En este paso se toma en consideración la implementación de una clase la cual leerá línea a línea el archivo subido en Python, esto simulara una conexión entre el satélite y la base de datos, analizando que todos los datos entregados por parte del archivo plano son originales e iguales a los que realmente se generan por el sistema del satélite.

Como punto importante recalcar que teniendo información real se puede determinar que el sistema es idóneo para almacenar datos de ese tipo, además de que el archivo es subido de manera adecuada, posee un control de versión, significa que el archivo tiene un código asignado para evitar que vuelva a ser introducido dentro de la base de datos y así mantener integridad dentro de la base de datos. En la figura 26 se detalla las líneas de código utilizado.

```
from django.core.exceptions import ObjectDoesNotExist
from django.http import HttpResponse
from django.utils import timezone
from enum import Enum
import re
import satellite.models as models
import hashlib


HELLO_COMMAND = 'HELLO'
ERROR_COMMAND = 'EVENTERROR'
HOUSE_COMMAND = 'HOUSEKEEPING_TM'


def md5(file_name):
    hash_md5 = hashlib.md5()
    with open(file_name, "rb") as f:
        for chunk in iter(lambda: f.read(4096), b""):
            hash_md5.update(chunk)
    return hash_md5.hexdigest()


def already_uploaded(file_name):
    checksum = md5(file_name)
    try:
        models.Checksum.objects.get(pk=checksum)
    except models.Checksum.DoesNotExist:
        return False, checksum
    return True, checksum


def register_upload(token):
    chk = models.Checksum(checksum=token)
    chk.save()


class Event(Enum):
    Error = 1
    Hello = 2
    Housekeeping = 3


class TmFileManager:
    def __init__(self):
        self.current_type = 0
        self.event = None
```

```
48      def process_line(self, line, line_number):
49          # If line contains "Received TM command" string, a new event is created.
50          m = re.search('Received TM command\s*:?\s*(\w+)', line)
51          if m is not None:
52              # But first, save the previous event if it exists.
53              if self.event is not None:
54                  self.event.save()
55                  self.event = None
56              # Which event type?
57              type_ = m.group(1)
58              # Capture sequence number which is present in all types.
59              m = re.search('Sequence Number\s*:?\s*(\d+)', line)
60              if m is None:
61                  raise ValueError('Sequence Number was expected after "Received TM command" in line {}.'.format(line_number))
62              seq_number = int(m.group(1))
63              post_time = timezone.now()
64              # Create corresponding event object based on type. Moreover, set attributes captured in this line.
65              # IMPORTANT: Set the current type which is needed for coming lines.
66              if type_ == ERROR_COMMAND:
67                  self.current_type = Event.Error
68                  self.event = models.EventErrComm(year=post_time.year, post_time=post_time)
69                  # Capture and set event name which is present in ERROR events only.
70                  m = re.search('Event Name\s*:?\s*(\w+)', line)
71                  if m is None:
72                      raise ValueError('Event Name was expected after "Sequence Number" in line {}.'.format(line_number))
73                  self.event.eve_name = m.group(1)
74              elif type_ == HELLO_COMMAND or type_ == HOUSE_COMMAND:
75                  if type_ == HELLO_COMMAND:
76                      self.current_type = Event.Hello
77                      self.event = models.HelloComm(year=post_time.year, post_time=post_time)
78                  else:
79                      self.current_type = Event.Housekeeping
80                      self.event = models.HouseKeepComm(year=post_time.year, post_time=post_time)
81                  # Capture operating mode which is present in two types, i.e., HELLO and HOUSEKEEPING.
82                  m = re.search('Operating Mode\s*:?\s*(\w+)', line)
83                  if m is None:
84                      raise ValueError('Operating Mode was expected after "Sequence Number" in line {}.'.format(line_number))
85                  self.event.ope_mode = m.group(1)
86              else:
87                  # raise NotImplementedError('Received TM command has not been implemented.')
88                  return
89              # Set sequence number which is present in all types
90              self.event.seq_number = seq_number
91          else:
92              if self.event is None:
93                  raise ValueError('Received TM command was expected at the beginning.')
```

```
93                  raise ValueError('Received TM command was expected at the beginning.')
94              # If mission clock is captured is set regardless the type.
95              m = re.search('Mission Clock\s*:?\s*(\d+)', line)
96              if m is not None:
97                  self.event.mis_clock = int(m.group(1))
98              # When any other attribute is captured, the type is checked, i.e., the attribute is not set for ERROR type.
99              m = re.search('p3V3_TM\s*:?\s*(\d+)', line)
100             if m is not None:
101                 if self.current_type == Event.Hello or self.current_type == Event.Housekeeping:
102                     self.event.p3v3_tm = int(m.group(1))
103             m = re.search('p5V_TM\s*:?\s*(\d+)', line)
104             if m is not None:
105                 if self.current_type == Event.Hello or self.current_type == Event.Housekeeping:
106                     self.event.p5v_tm = int(m.group(1))
107             m = re.search('p15V_TM\s*:?\s*(\d+)', line)
108             if m is not None:
109                 if self.current_type == Event.Hello or self.current_type == Event.Housekeeping:
110                     self.event.p15v_tm = int(m.group(1))
111             m = re.search('n15V_TM\s*:?\s*(\d+)', line)
112             if m is not None:
113                 if self.current_type == Event.Hello or self.current_type == Event.Housekeeping:
114                     self.event.n15v_tm = int(m.group(1))
115             m = re.search('RW_P5V\s*:?\s*(FALSE|TRUE)', line)
116             if m is not None:
117                 if self.current_type == Event.Hello or self.current_type == Event.Housekeeping:
118                     self.event.rw_p5v = m.group(1)
119             m = re.search('MTS_VBUS\s*:?\s*(FALSE|TRUE)', line)
120             if m is not None:
121                 if self.current_type == Event.Hello or self.current_type == Event.Housekeeping:
122                     self.event.mts_vbus = m.group(1)
123             m = re.search('BOOM1_VBUS\s*:?\s*(FALSE|TRUE)', line)
124             if m is not None:
125                 if self.current_type == Event.Hello or self.current_type == Event.Housekeeping:
126                     self.event.boom1_vbus = m.group(1)
127             m = re.search('BOOM2_VBUS\s*:?\s*(FALSE|TRUE)', line)
128             if m is not None:
129                 if self.current_type == Event.Hello or self.current_type == Event.Housekeeping:
130                     self.event.boom2_vbus = m.group(1)
131             m = re.search('TTC_STAT\s*:?\s*(FALSE|TRUE)', line)
132             if m is not None:
133                 if self.current_type == Event.Hello or self.current_type == Event.Housekeeping:
134                     self.event.ttc_stat = m.group(1)
135             m = re.search('Battery\s*:?\s*(\w+)', line)
136             if m is not None:
137                 if self.current_type == Event.Hello or self.current_type == Event.Housekeeping:
138                     self.event.battery = m.group(1)
```

```
140     def process(self, file_name):
141         uploaded, token = already_uploaded(file_name)
142         if not uploaded:
143             f = open(file_name)
144             for line_number, line in enumerate(f):
145                 self.process_line(line, line_number)
146             register_upload(token)
147             return 1
148     return -1
```

Figura 26. Carga de archivo texto plano proporcionado por el Satélite

- **Interfaz HTML para muestra de datos en Servidor Web**

Como objetivo primordial del proyecto aparte de almacenar la información se solicitaba la información de las tablas EventErroComm, HelloComm y HouseKeepComm, puedan ser mostradas atreves de una página web.

En anteriores investigaciones se logró mostrar la información del satélite que era almacenada en una base de datos SQLite, aunque con algunas restricciones. Por tal motivo el proyecto debe cumplir con el objetivo de mostrar datos en tiempo real hacia los usuarios. Para la creación de tablas se ha utilizado como lenguaje de programación HTML y se exponen de manera sencilla los datos.

A continuación se muestra en la figura 27, 28 y 29 la creación del archivo HTML para la demostración de los datos que se encuentran en cada tabla respetivamente.

```
1   {% extends 'satellite/info.html' %}
2
3   {% block grid %}
4   <table>
5   ⟶<thead>
6   ⟶<tr>
7   ⟶⟶<th>Year</th>
8   ⟶⟶<th>Time</th>
9   ⟶⟶<th>Event Name</th>
10  ⟶⟶<th>Mission Clock</th>
11  ⟶⟶<th>Sequence Number</th>
12  ⟶</tr>
13  ⟶</thead>
14  ⟶<tbody>
15  ⟶{% for error in info %}
16  ⟶<tr>
17  ⟶⟶<td>{{ error.year }}</td>
18  ⟶⟶<td>{{ error.post_time|date:"d-m" }} {{ error.post_time|time:"H:i:s.u" }}</td>
19  ⟶⟶<td>{{ error.eve_name }}</td>
20  ⟶⟶<td>{{ error.mis_clock }}</td>
21  ⟶⟶<td>{{ error.seq_number }}</td>
22  ⟶</tr>
23  ⟶{% endfor %}
24  ⟶</tbody>
25  </table>
26  {% endblock %}
```

Figura 27. Interfaz HTML para muestra de datos de la Tabla EventError

```
{% extends 'satellite/info.html' %}

{% block grid %}
<table>
    <thead>
    <tr>
        <th>Year</th>
        <th>Time</th>
        <th>Battery</th>
        <th>BOOM1 VBUS</th>
        <th>BOOM2 VBUS</th>
        <th>Mission Clock</th>
        <th>MTS VBUS</th>
        <th>n15V TM</th>
        <th>Operating Mode</th>
        <th>p15V TM</th>
        <th>p3v3 TM</th>
        <th>p5v TM</th>
        <th>RW P5V</th>
        <th>Sequence Number</th>
        <th>TTC STAT</th>
    </tr>
    </thead>
    <tbody>
    {% for hello in info %}
    <tr>
        <td>{ hello.year }}</td>
        <td>{ hello.post_time|date:"d-m" }} { hello.post_time|time:"H:i:s.u" }}</td>
        <td>{ hello.battery }}</td>
        <td>{ hello.boom1_vbus}}</td>
        <td>{ hello.boom2_vbus }}</td>
        <td>{ hello.mis_clock }}</td>
        <td>{ hello.mts_vbus }}</td>
        <td>{ hello.n15v_tm }}</td>
        <td>{ hello.ope_mode }}</td>
        <td>{ hello.p15v_tm }}</td>
        <td>{ hello.p3v3_tm }}</td>
        <td>{ hello.p5v_tm }}</td>
        <td>{ hello.rw_p5v }}</td>
        <td>{ hello.seq_number }}</td>
        <td>{ hello.ttc_stat }}</td>
    </tr>
    {% endfor %}
    </tbody>
</table>
{% endblock %}
```

Figura 28. Interfaz HTML para muestra de datos de la Tabla Hello

```
{% extends 'satellite/info.html' %}

{% block grid %}
<table>
    <thead>
    <tr>
        <th>Year</th>
        <th>Time</th>
        <th>Battery</th>
        <th>BOOM1 VBUS</th>
        <th>BOOM2 VBUS</th>
        <th>Mission Clock</th>
        <th>MTS VBUS</th>
        <th>n15V TM</th>
        <th>Operating Mode</th>
        <th>p15V TM</th>
        <th>p3v3 TM</th>
        <th>p5v TM</th>
        <th>RW P5V</th>
        <th>Sequence Number</th>
        <th>TTC STAT</th>
    </tr>
    </thead>
    <tbody>
    {% for housek in info %}
    <tr>
        <td>{{ housek.year }}</td>
        <td>{{ housek.post_time|date:"d-m" }} {{ housek.post_time|time:"H:i:s.u" }}</td>
        <td>{{ housek.battery }}</td>
        <td>{{ housek.boom1_vbus}}</td>
        <td>{{ housek.boom2_vbus }}</td>
        <td>{{ housek.mis_clock }}</td>
        <td>{{ housek.mts_vbus }}</td>
        <td>{{ housek.n15v_tm }}</td>
        <td>{{ housek.ope_mode }}</td>
        <td>{{ housek.p15v_tm }}</td>
        <td>{{ housek.p3v3_tm }}</td>
        <td>{{ housek.p5v_tm }}</td>
        <td>{{ housek.rw_p5v }}</td>
        <td>{{ housek.seq_number }}</td>
        <td>{{ housek.ttc_stat }}</td>
    </tr>
    {% endfor %}
    </tbody>
</table>
{% endblock %}
```

Figura 29. Interfaz HTML para muestra de datos de la Tabla Housekeep

- **Filtrado de información de la Base de Datos en Servidor Web**

Se requiere filtrar los datos obtenidos por parte del satélite y ser mostrados a través del servidor web, el cual tiene como objetivo mostrar los datos de los intervalos de tiempo en el que el satélite aloja su información en la base de datos, es importante mencionar que se puede realizar el filtrado de la información por año y por intervalos de tiempo como se muestra en la figura 30.

```
{% extends 'satellite/info.html' %}

{% block grid %}
<div style="margin-bottom: 20px;">
    <form action="{% url 'satellite:errors' %}" method="get">
        {% csrf_token %}
        <table>
            <tr>
                <td>Year =</td>
                <td><input type="text" name="year"></td>
            </tr>
            <tr>
                <td>Datetime from:</td>
                <td><input type="text" name="post_time_from"> (yyyy-mm-dd hh:mm:ss.ms)</td>
            </tr>
            <tr>
                <td>Datetime to:</td>
                <td><input type="text" name="post_time_to"> (yyyy-mm-dd hh:mm:ss.ms)</td>
            </tr>
        </table>
        <button type="submit">Filter</button>
        <button type="reset">Reset</button>
    </form>
</div>
<div style="margin-bottom: 20px;">
```

Figura 30. Filtro de Datos en HTML de Servidor Web

A continuación se muestra el desarrollo final de la página web donde se muestran los datos obtenidos mediante la comunicación entre el satélite UPMSat-2 y la base de datos Apache Cassandra y como medio de comunicación un servidor web Python Django. Se observa en la figura 31.

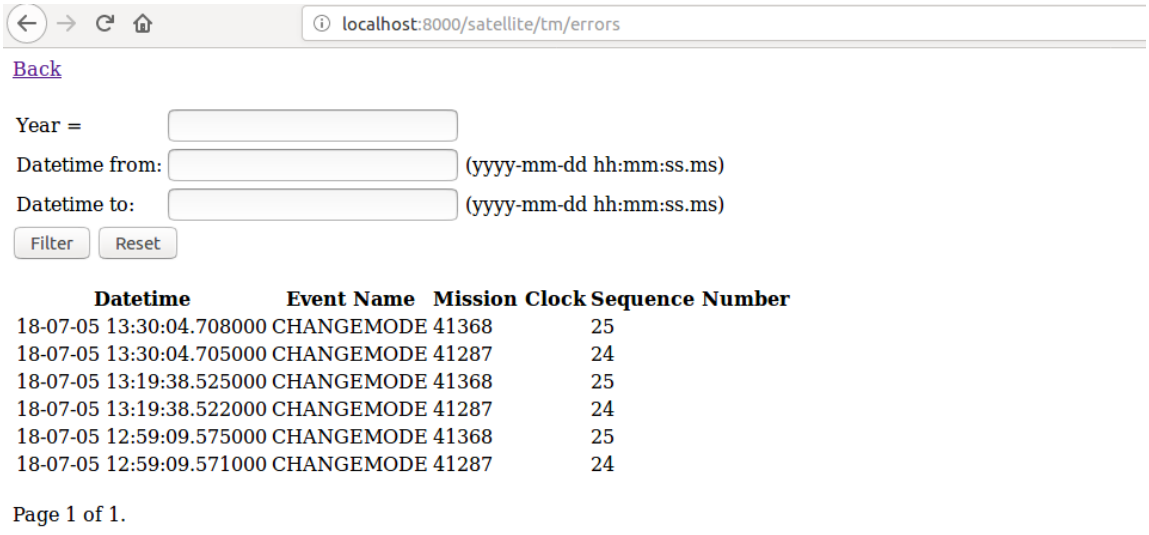


Figura 31. Página Web de Datos del Satélite UPMSat-2

Al tener la comunicación entre todos los componente de la plataforma el usuario puede obtener los datos del satélite que están alojados en la base de datos Cassandra, se puede filtrar por año y por intervalos de subida de información y muestra los datos de las tres tablas establecidas en los objetivos del proyecto.

5 Pruebas

En esta sección se detalla las pruebas realizadas con todo el sistema integrado, donde se utiliza el clúster de CoreOS para alojar una base de datos Apache Cassandra, y almacenar mediante un programa realizado en Python, que recuperar los datos generados por el satélite UPMSat-2, mediante un archivo de texto plano. Los datos serán expuestos mediante un servidor web alojado en la máquina de administración.

5.1 Pruebas en Clúster de CoreOS (Alta Disponibilidad)

Para realizar pruebas dentro del sistema se debe tener en claro los requisitos no funcionales del sistema, para el proyecto la alta disponibilidad es un punto importante, es por eso que para este requisito se requirió la instalación y configuración de un clúster de CoreOS, como se puede ver en la figura 32, el clúster se encuentra completamente saludable. Como se mencionó anteriormente ETCD se comprende de un algoritmo de consenso el que maneja la disposición de que nodo es o será líder (Leader) y quiénes serán sus seguidores (Followers).

Cada nodo tiene un identificador único para su ubicación, las IP de cada nodo son asignadas en el fichero de configuración cloud-config.

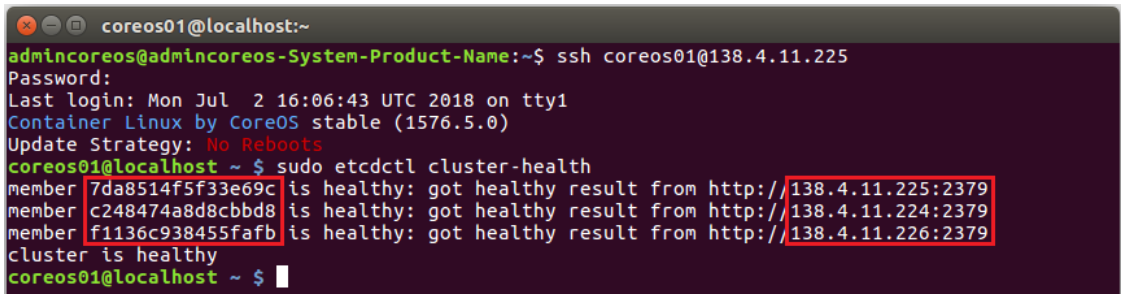


Figura 32. Estado de Salud del Clúster

Para determinar si el clúster se encuentra configurado de la manera correcta y cumple con los parámetros del proyecto uno de los nodos debe tener asignado un tipo, como se muestra en la figura 33, 34 y 35, los tres nodos tienen por defecto una asignación de tipo

(Leader o Follower), en nuestro caso el líder es el nodo con el identificador "f1136c938455fafb" con dirección IP "138.4.11.226" y con el nombre de host "Coreos02".

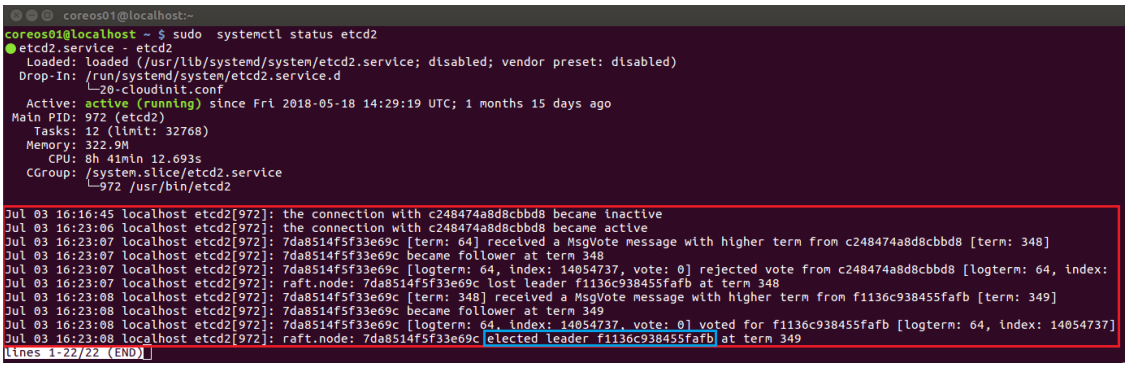


Figura 33. Primera Asignación de Nodo de Tipo (Follower)

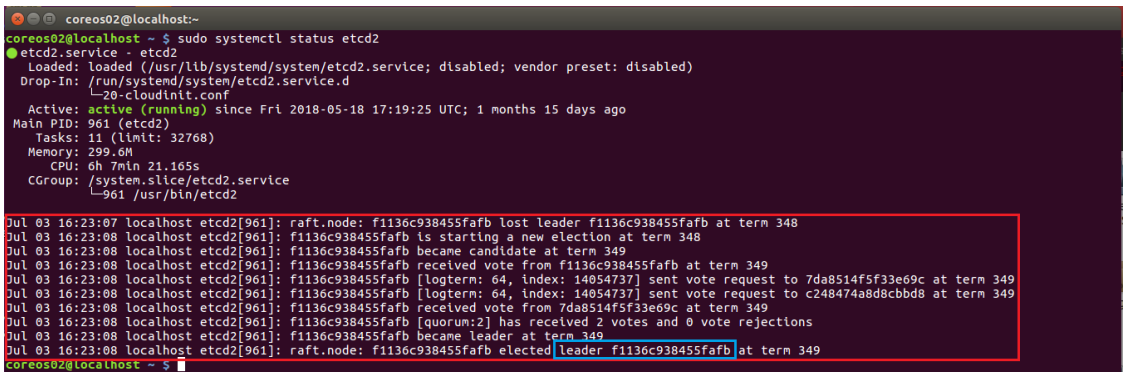


Figura 34. Primera Asignación de Nodo de Tipo (Leader)

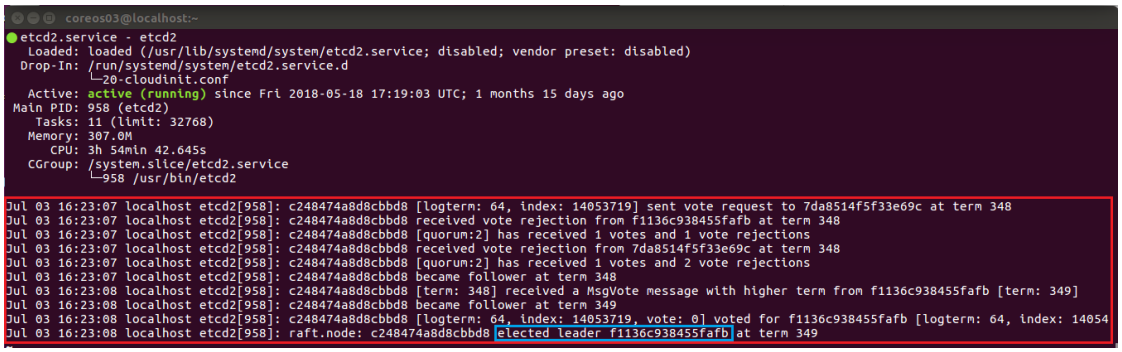


Figura 35. Segunda Asignación de Nodo de Tipo (Follower)

Para determinar que existe alta disponibilidad en el clúster es fundamental dar de baja a uno de los nodos, en nuestro caso daremos de baja al nodo líder y de esa manera

se tenga que elegir nuevamente el líder de clúster. De esta manera se determinara si se está cumpliendo con el consenso de Raft que es proporcionado por ETCD y proporciona alta disponibilidad en todos los nodos.

Es necesario mencionar que CoreOS alerta de que un nodo esta dado de baja al momento de preguntar nuevamente el estado de salud del clúster. Esto quiere decir que si existiera alguna caída de un nodo serán informados los demás. En las figuras 36 y 37 se puede observar que los nodos son informados de que un nodo, en este caso el nodo líder fue dado de baja, en ese momento los nodos restante deciden elegir a un nuevo líder y mantener la disponibilidad del clúster.

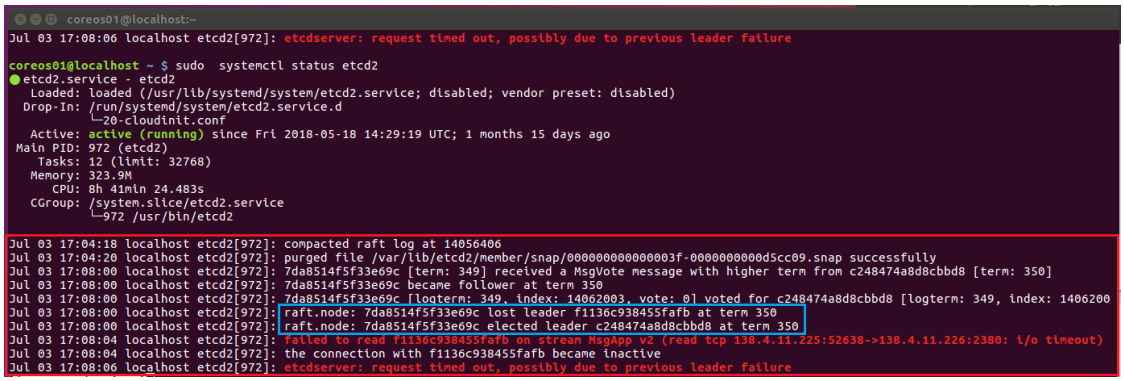


Figura 36. Asignación de nuevo líder de Nodo

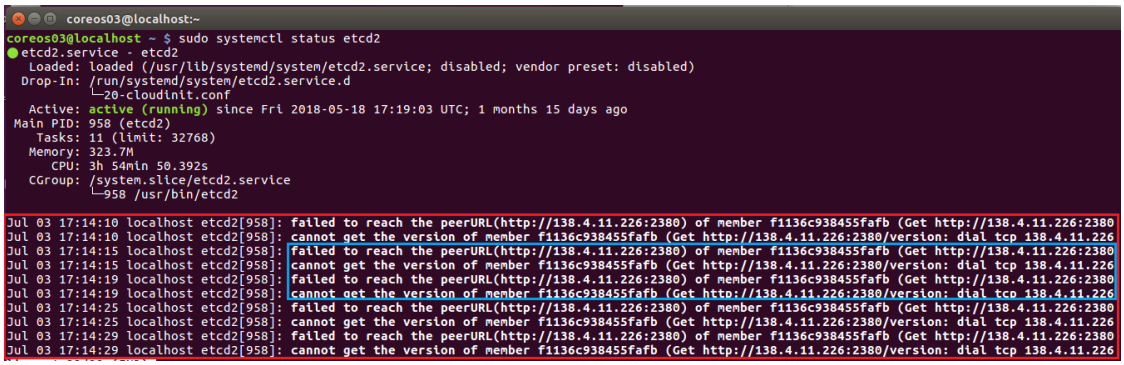


Figura 37. Información de Nodo Inactivo

Si el nodo que fue dado de baja restablece su comunicación, este desempeñara la función de seguidor y enviara un mensaje a los demás nodos de que esta nuevamente activo. Con esto se asegura de que si existieron cambios dentro de la base de datos Apache Cassandra puedan ser replicados a todos los nodos nuevamente.

5.2 Pruebas en Servicios Docker y Apache Cassandra (Alta Redundancia)

Las pruebas realizadas en los servicios Docker y Apache Cassandra determinaran la alta redundancia de los datos que son proporcionados por el satélite UPMSat-2, es necesario recordar que la información entregada por el satélite es de alta prioridad, por ese motivo y como objetivos de este trabajo fin de master, es importante que esa información este completamente resguardada y disponible.

El diseñó y configuración de Docker con Apache Cassandra permite la administración y almacenamiento de los datos, como se muestra en la figura 38, 39 y 40, se realiza una simple consulta a la base de datos sobre la tabla "event_err_comm", con esto obtenemos la información alojada y podemos comprobar que se encuentra replicada en todos los servidores.

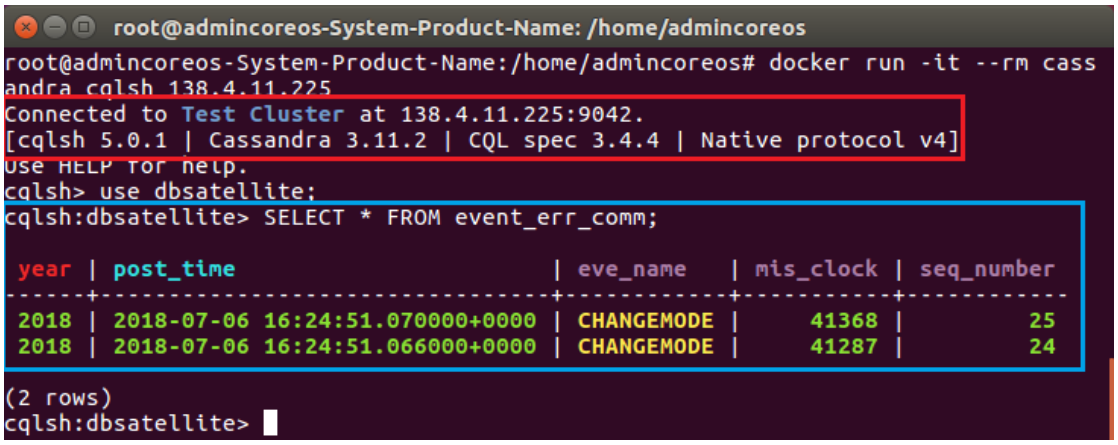


Figura 38. Prueba de Alta Redundancia de Datos en Nodo 1

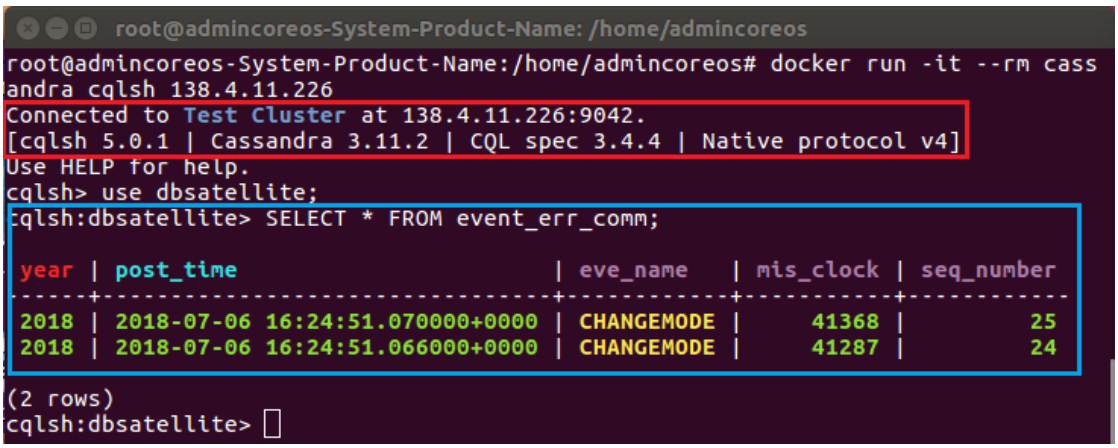


Figura 39. Prueba de Alta Redundancia de Datos en Nodo 2

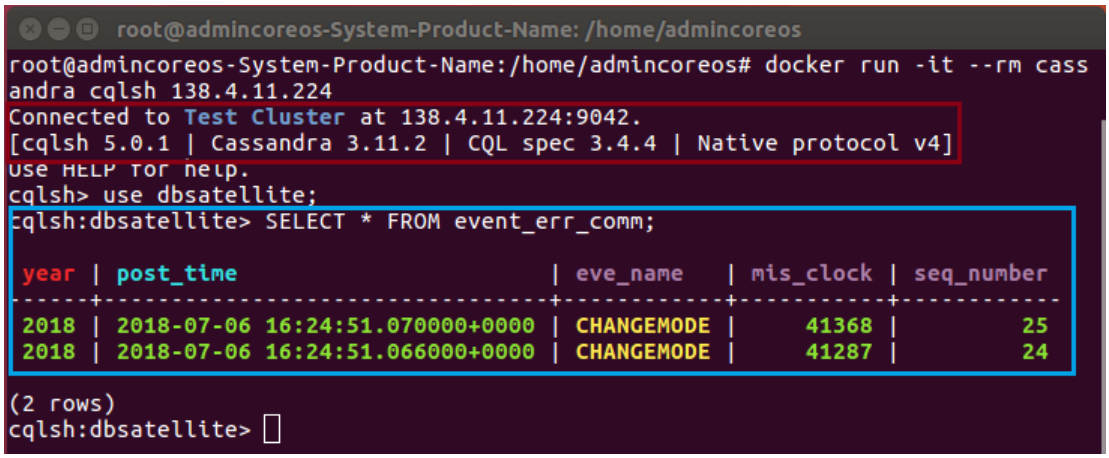


Figura 40. Prueba de Alta Redundancia de Datos en Nodo 3

Como se observar, los datos que son entregados por el satélite se están almacenando en la base de datos, para realizar esta prueba lo que se hizo fue dar de baja a cualquier nodo e ingresar nuevamente información dentro de la tabla.

Al ser parte de CoreOS y tener configurado los parámetros de alta redundancia por parte de Cassandra, permitirá que al volver activar el nodo, la información se replicara automáticamente.

En la figura 41 el nodo con la dirección IP 138.4.11.224 no posee la información recientemente agregada.

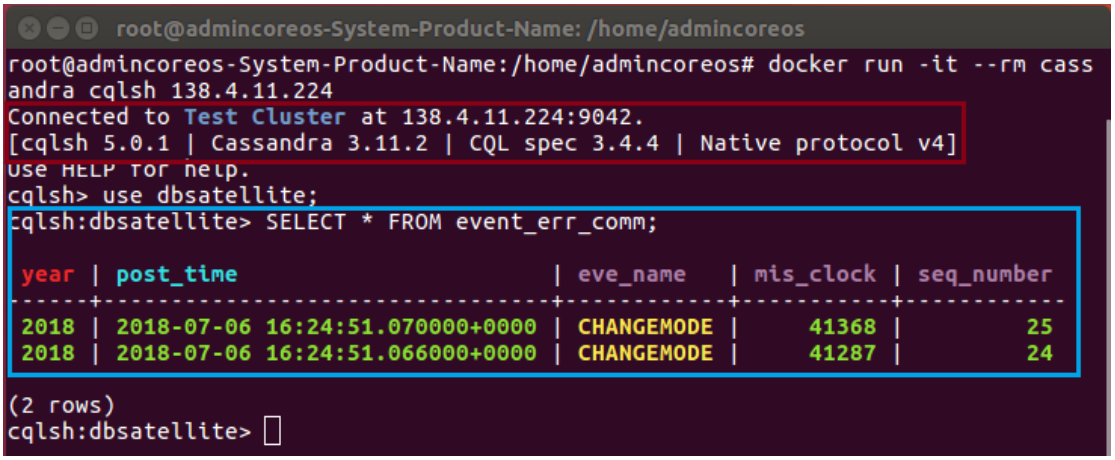


Figura 41. Nodo 3 sin Redundancia de Datos

Como se observa en la figura 42 y 43, la información nueva fue ingresada en los nodos 1 y 2 y se realizó una consulta simple a la tabla.

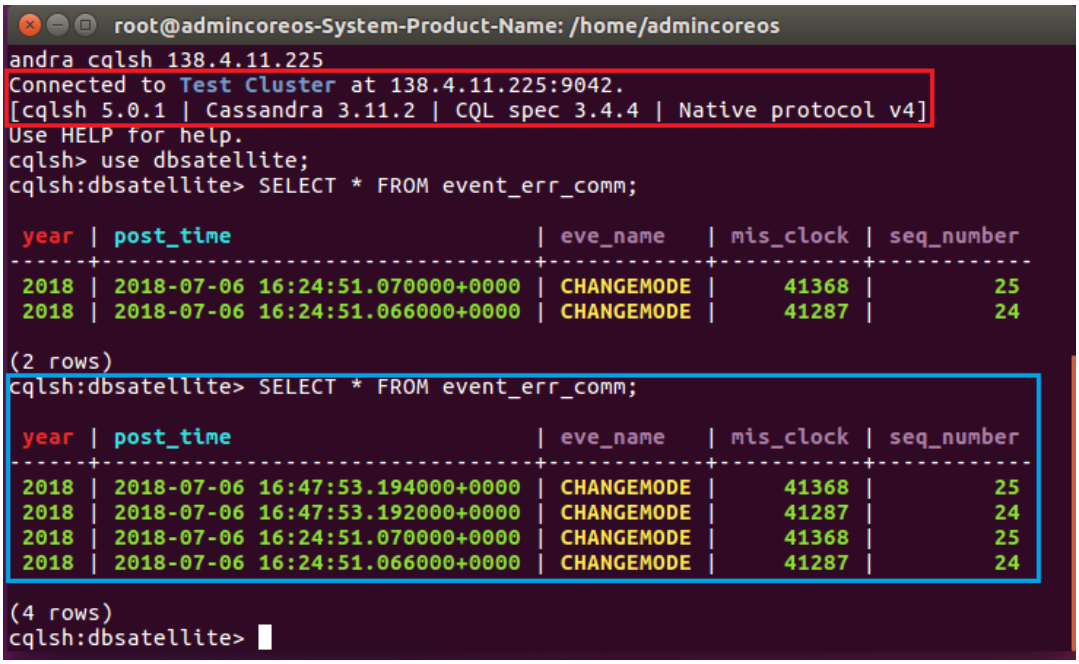


Figura 42. Ingreso de nueva Información en el Nodo 1

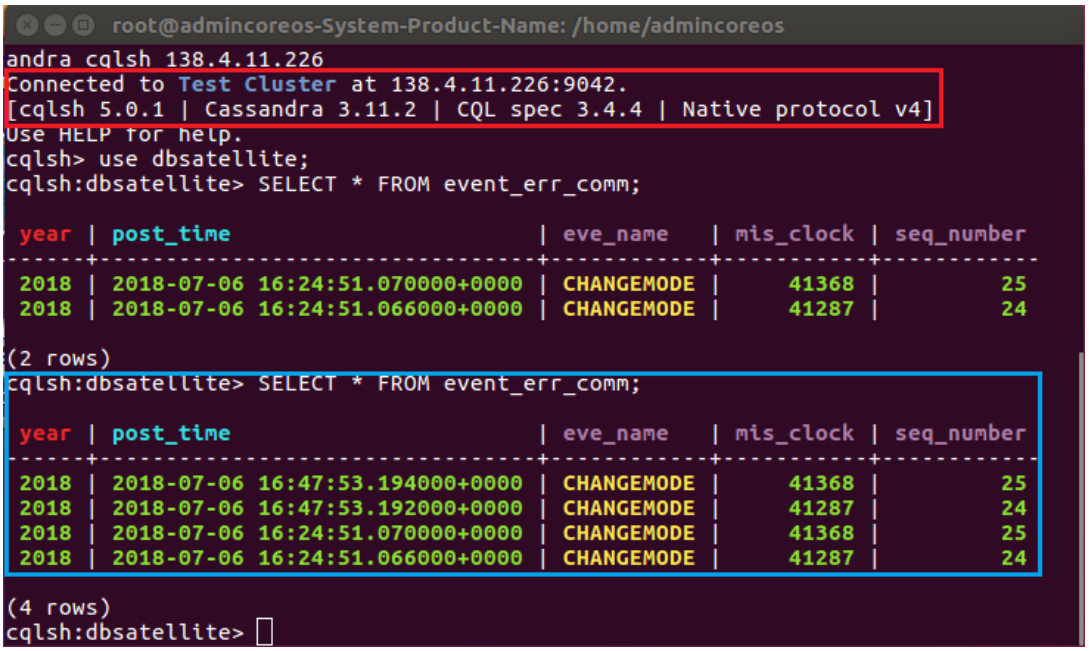


Figura 43. Ingreso de nueva Información en el Nodo 2

Como se mencionó anteriormente para determinar si existe alta redundancia en los datos, se vuelve a dar de alta el servidor y se realiza nuevamente la consulta en la base de datos en el nodo afectado, en la figura 44, los datos han sido replicados y cumple con la prueba realizada.

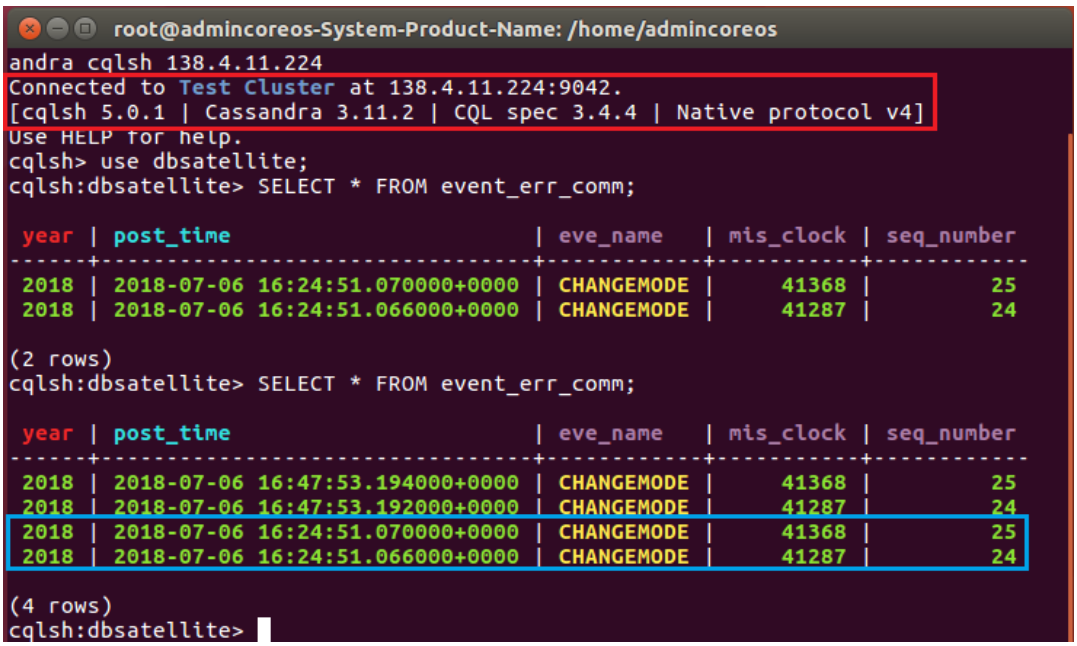
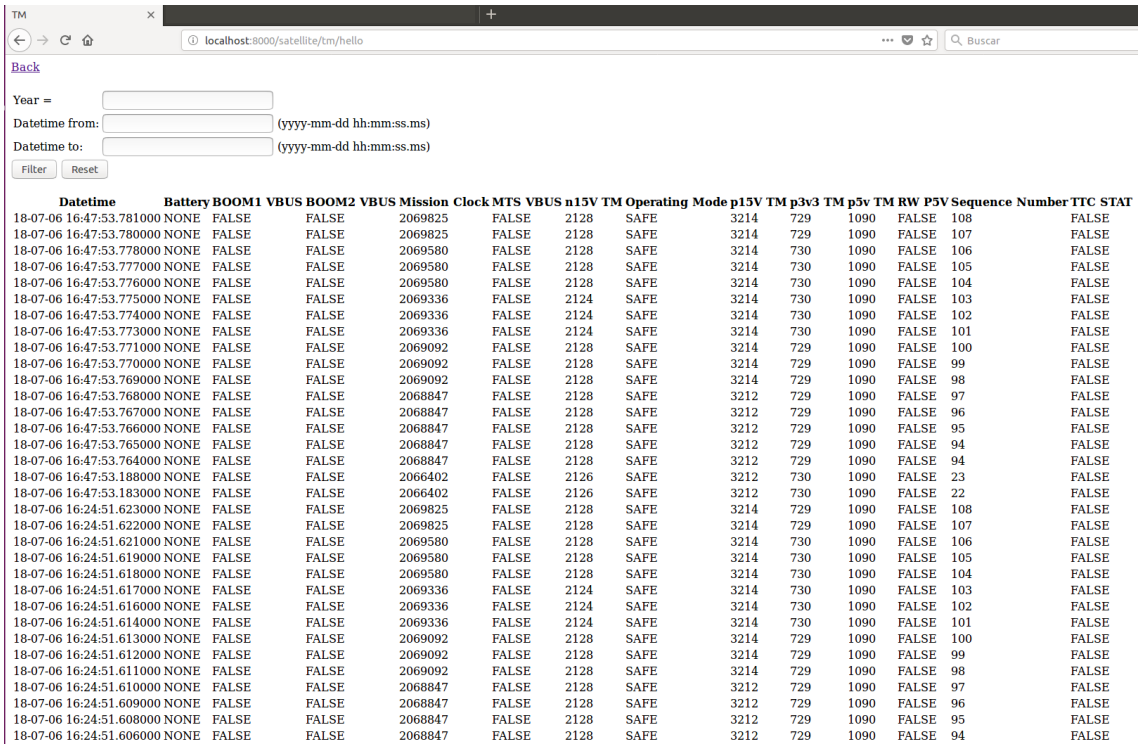


Figura 44. Prueba de Alta Redundancia Actualización de Datos en Nodo 3

5.3 Pruebas en Python y Servidor Web Django (Tolerancia a Fallos)

Para la comprobación de que el Servidor Web, Apache Cassandra y CoreOS cumplen con los parámetros señalados al inicio del proyector, es necesario detallar que la comunicación entre estos componentes se la realiza mediante Python que es nuestro intérprete de comunicación.

Al ser un sistema tolerante a fallos es necesario que para este prueba se tenga que dar de baja a uno de los servidores que se encuentran dentro del clúster, lo que se quiere demostrar es que si existiera una caída de servicio por cualquier nodo, la información que ingresa y que se muestra por parte del servidor web siempre estará disponible, como recomendación para este proyecto es necesario que el equipo Administrador de CoreOS tenga un total aislamiento en relación al del clúster, esto para evitar que si existiera una caída de energía o algún problema técnico donde se encuentran el clúster de CoreOS no tengan contacto alguno.



Datetime	Battery	BOOM1 VBUS	BOOM2 VBUS	Mission Clock	MTS VBUS	n15V TM	Operating Mode	p15V TM	p3v3 TM	p5v TM	RW P5V	Sequence Number	TTC STAT
18-07-06 16:47:53.781000	NONE	FALSE	FALSE	2069825	FALSE	2128	SAFE	3214	729	1090	FALSE	108	FALSE
18-07-06 16:47:53.780000	NONE	FALSE	FALSE	2069825	FALSE	2128	SAFE	3214	729	1090	FALSE	107	FALSE
18-07-06 16:47:53.778000	NONE	FALSE	FALSE	2069580	FALSE	2128	SAFE	3214	730	1090	FALSE	106	FALSE
18-07-06 16:47:53.777000	NONE	FALSE	FALSE	2069580	FALSE	2128	SAFE	3214	730	1090	FALSE	105	FALSE
18-07-06 16:47:53.776000	NONE	FALSE	FALSE	2069580	FALSE	2128	SAFE	3214	730	1090	FALSE	104	FALSE
18-07-06 16:47:53.775000	NONE	FALSE	FALSE	2069336	FALSE	2124	SAFE	3214	730	1090	FALSE	103	FALSE
18-07-06 16:47:53.774000	NONE	FALSE	FALSE	2069336	FALSE	2124	SAFE	3214	730	1090	FALSE	102	FALSE
18-07-06 16:47:53.773000	NONE	FALSE	FALSE	2069336	FALSE	2124	SAFE	3214	730	1090	FALSE	101	FALSE
18-07-06 16:47:53.771000	NONE	FALSE	FALSE	2069092	FALSE	2128	SAFE	3214	729	1090	FALSE	100	FALSE
18-07-06 16:47:53.770000	NONE	FALSE	FALSE	2069092	FALSE	2128	SAFE	3214	729	1090	FALSE	99	FALSE
18-07-06 16:47:53.769000	NONE	FALSE	FALSE	2069092	FALSE	2128	SAFE	3214	729	1090	FALSE	98	FALSE
18-07-06 16:47:53.768000	NONE	FALSE	FALSE	2068847	FALSE	2128	SAFE	3212	729	1090	FALSE	97	FALSE
18-07-06 16:47:53.767000	NONE	FALSE	FALSE	2068847	FALSE	2128	SAFE	3212	729	1090	FALSE	96	FALSE
18-07-06 16:47:53.766000	NONE	FALSE	FALSE	2068847	FALSE	2128	SAFE	3212	729	1090	FALSE	95	FALSE
18-07-06 16:47:53.765000	NONE	FALSE	FALSE	2068847	FALSE	2128	SAFE	3212	729	1090	FALSE	94	FALSE
18-07-06 16:47:53.764000	NONE	FALSE	FALSE	2068847	FALSE	2128	SAFE	3212	729	1090	FALSE	94	FALSE
18-07-06 16:47:53.188000	NONE	FALSE	FALSE	2066402	FALSE	2126	SAFE	3212	730	1090	FALSE	23	FALSE
18-07-06 16:47:53.183000	NONE	FALSE	FALSE	2066402	FALSE	2126	SAFE	3212	730	1090	FALSE	22	FALSE
18-07-06 16:24:51.623000	NONE	FALSE	FALSE	2069825	FALSE	2128	SAFE	3214	729	1090	FALSE	108	FALSE
18-07-06 16:24:51.622000	NONE	FALSE	FALSE	2069825	FALSE	2128	SAFE	3214	729	1090	FALSE	107	FALSE
18-07-06 16:24:51.621000	NONE	FALSE	FALSE	2069580	FALSE	2128	SAFE	3214	730	1090	FALSE	106	FALSE
18-07-06 16:24:51.619000	NONE	FALSE	FALSE	2069580	FALSE	2128	SAFE	3214	730	1090	FALSE	105	FALSE
18-07-06 16:24:51.618000	NONE	FALSE	FALSE	2069580	FALSE	2128	SAFE	3214	730	1090	FALSE	104	FALSE
18-07-06 16:24:51.617000	NONE	FALSE	FALSE	2069336	FALSE	2124	SAFE	3214	730	1090	FALSE	103	FALSE
18-07-06 16:24:51.616000	NONE	FALSE	FALSE	2069336	FALSE	2124	SAFE	3214	730	1090	FALSE	102	FALSE
18-07-06 16:24:51.614000	NONE	FALSE	FALSE	2069336	FALSE	2124	SAFE	3214	730	1090	FALSE	101	FALSE
18-07-06 16:24:51.613000	NONE	FALSE	FALSE	2069092	FALSE	2128	SAFE	3214	729	1090	FALSE	100	FALSE
18-07-06 16:24:51.612000	NONE	FALSE	FALSE	2069092	FALSE	2128	SAFE	3214	729	1090	FALSE	99	FALSE
18-07-06 16:24:51.611000	NONE	FALSE	FALSE	2069092	FALSE	2128	SAFE	3214	729	1090	FALSE	98	FALSE
18-07-06 16:24:51.610000	NONE	FALSE	FALSE	2068847	FALSE	2128	SAFE	3212	729	1090	FALSE	97	FALSE
18-07-06 16:24:51.609000	NONE	FALSE	FALSE	2068847	FALSE	2128	SAFE	3212	729	1090	FALSE	96	FALSE
18-07-06 16:24:51.608000	NONE	FALSE	FALSE	2068847	FALSE	2128	SAFE	3212	729	1090	FALSE	95	FALSE
18-07-06 16:24:51.606000	NONE	FALSE	FALSE	2068847	FALSE	2128	SAFE	3212	729	1090	FALSE	94	FALSE

Figura 45. Tabla Hello en Servidor Web

En la figura 45, muestra información de la tabla Hello, estos datos se pueden observar en orden descendentemente. Todos los datos son ingresados de manera correcta pero al dar de baja un nodo y consultar información en el servidor web, Django toma en control de administrar la conexión con la base de datos, como se muestra en la figura 24, el administrado web solicita que se ingrese una o más direcciones IP para realizar la conexión si existiría una configuración de clustering.

Como se muestra en la figura 46, se encuentra activo el servidor web de Django y proporciona una IP de "localhost" para este proyecto.

```
(venv) admincoreos@admincoreos-System-Product-Name:~/PycharmProjects/CassCore2-7/satellite$ python manage.py syncdb
Creating keyspace dbsatellite..
Syncing satellite.models.Checksum
Syncing satellite.models.EventErrComm
Syncing satellite.models.HelloComm
Syncing satellite.models.HouseKeepComm
(venv) admincoreos@admincoreos-System-Product-Name:~/PycharmProjects/CassCore2-7/satellite$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

Not checking migrations as it is not possible to access/create the django_migrations table.
July 06, 2018 - 18:24:36
Django version 1.11.13, using settings 'casscore.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Figura 46. Levantamiento de Servidor Web y Sincronización de BD

Al tener el servicio operativo simulamos un fallo de red en uno de los servidores y se observa que el servidor web alojado en la máquina de administración de CoreOS, reinicia sus operaciones inmediatamente al saber que un nodo ha sido dado de baja.

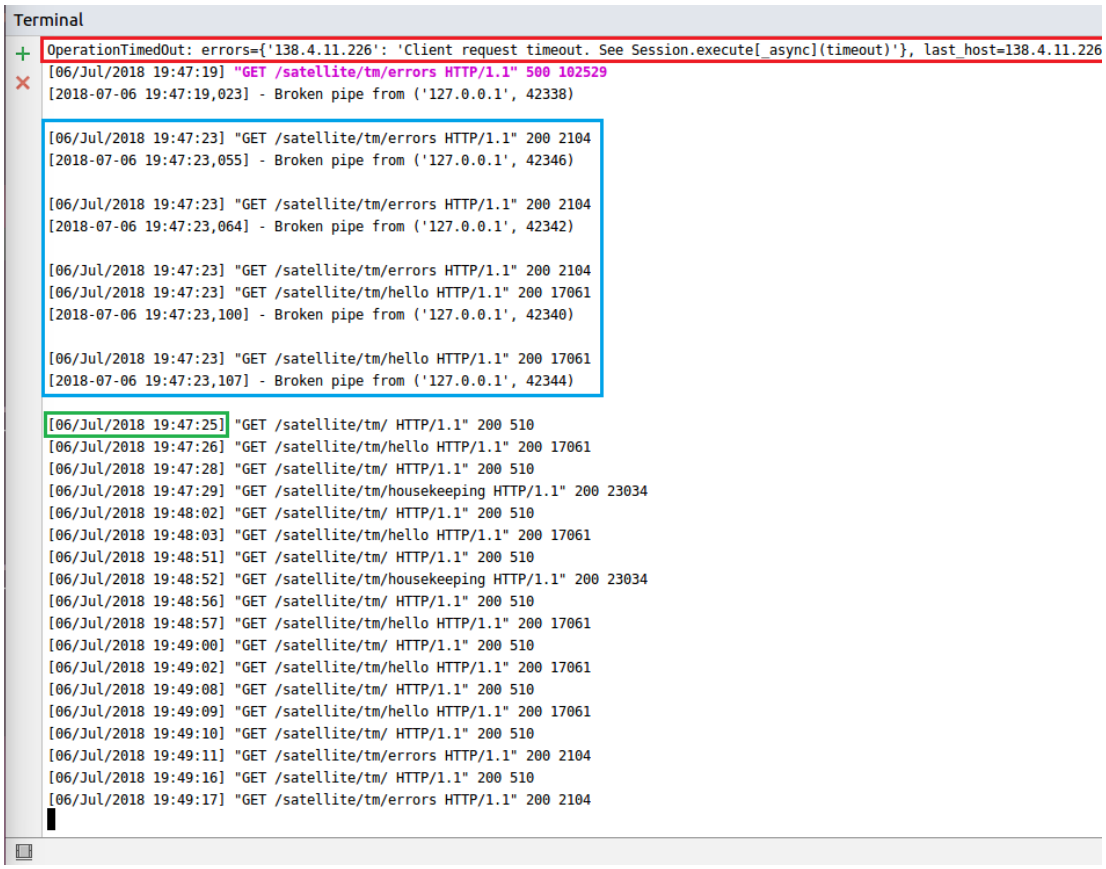


Figura 47. Servicio Web Activo Tolerante a Fallos

Como se observa en la imagen anterior Python informa mediante un advertencia que uno de sus servidores no responde, en este caso el servidor del nodo 2 con dirección IP 138.4.11.226 e internamente según lo configurado, utiliza otro nodo que mantiene integridad con los datos expuesto, ya que cumple con el parámetro de alta redundancia de datos en todos los nodos. El tiempo que demora el servidor en retomar su actividad es variable, pero según las pruebas realizadas se determinó que es menos de 3 segundos hasta determinar la conexión con otro nodo.

La configuración y funcionamiento de CoreOS con Apache Cassandra aseguran el perfecto funcionamiento del Servidor Web, que simplemente se encarga de tomar la IP que esté disponible y utilizarla para mostrarlo a través de su interfaz web, esto demuestra que la plataforma de segmento en Tierra del satélite UPMSat-2 cumple con los objetivos del proyecto y satisface las necesidades de los usuarios.

6 Conclusiones

El proyecto UPMSat-2 tiene gran importancia para diferentes grupos de investigación dentro de la Universidad Politécnica de Madrid, y el desarrollo de este trabajo ha supuesto un importante avance en el ámbito del proyecto, al implementar una plataforma funcional que almacenara los datos entregados por el satélite; además de detallar las especificación tanto en la arquitectura de la plataforma en Tierra como del sistema de comunicación del satélite y sus componentes.

El desarrollo de este proyecto ha conllevado a la investigación de sistemas que soporten alta disponibilidad, redundancia de datos y tolerancia a fallos. CoreOS se convierte en la plataforma base de este proyecto, por proporcionar la estabilidad en el área de sistemas orientados a servicios, y por la versatilidad del manejo en el área de contenedores, esto unido al amplio despliegue de nuevas versiones que han generado mejoras perceptibles en el sistema operativo.

La integración de CoreOS con los componentes del satélite UPMSat-2, es proporcionado por herramientas de programación que administran y controlan los sistemas del satélite y la plataforma, esta comunicación entre los dos sistemas genera un enlace entre la base de datos que se encuentra instalada y configurada en el clúster de CoreOS y el sistema del satélite, utilizando como medio de programación a Python. Esto genera un gran avance con respecto a la comunicación entre el satélite y la plataforma ya que Python es fácil de utilizar y tiene los complementos de programación necesarios para entablar esta comunicación.

CoreOS y la base de datos Apache Cassandra son parte de los componentes de la plataforma del satélite en Tierra que cumplen con el objetivo de administrar y mostrar los datos almacenados del satélite, estos dos componentes dan una solución óptima para determinar que la información esté disponible, redundante y tolerante a fallos dentro del sistema y así cumplir con los objetivos dispuestos en el proyecto.

Al tener en claro que la comunicación entre la plataforma terrena y el satélite se lo hará mediante Python, se consideró importante la demostración de los datos mediante un servidor web dedicado al despliegue en tiempo real y a la administración de los mismos, generando fiabilidad en la información proporcionada por el sistema hacia el usuario.

El prototipo desplegado cumple con las condiciones de diseño propuestas para este trabajo, sin embargo se recomienda que al tener lista y funcional la plataforma, se realicen pruebas utilizando una mayor cantidad de datos entregados por parte del satélite, y a futuro los datos del satélite ya en órbita.

Apéndice 1

CONFIGURACIÓN DE COREOS (Ejecutar en los 3 nodos)

- Configuración de Usuario y Clave en CoreOS

```
$ sudo openssl passwd -1 > cloud-config-file
Password:
Verifying – Password:
$ vim cloud-config-file
```

- Configuración Cloud-Config en CoreOS

```
#cloud-config
coreos:
  etcd2:
    name: coreos-cluster-etcd2-01
    initial-advertise-peer-urls: http://138.4.11.225:2380
    listen-peer-urls: http://138.4.11.225:2380
    listen-client-urls: http://138.4.11.225:2379,http://127.0.0.1:2379
    advertise-client-urls: http://138.4.11.225:2379
    initial-cluster-token: etcd-cluster-1
    initial-cluster:      coreos-cluster-etcd2-01=http://138.4.11.225:2380,coreos-cluster-
etcd2-02=http://138.4.11.226:2380,coreos-cluster-etcd2-03=http://138.4.11.224:2380
    initial-cluster-state: new
    #discovery: https://discovery.etcd.io/766e303fee99e03a03c5e47b0e74fc43
  units:
    - name: update-engine.service
      mask: true
    - name: locksmithd.service
      mask: true
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
    - name: systemd-networkd.service
      command: stop
    - name: 00-eth0.network
      runtime: true
      content: |
        [Match]
        Name=enp4s0
        [Network]
        Address=138.4.11.225/22
```

```
      Gateway=138.4.11.131
      DNS=8.8.8.8
  - name: down-interfaces.service
    command: start
    content: |
      [Service]
      Type=oneshot
      ExecStart=/usr/bin/ip link set enp4s0 down
      ExecStart=/usr/bin/ip addr flush dev enp4s0
  - name: systemd-networkd.service
    command: restart
  - name: docker.service
    command: start
  - name: docker-tcp.socket
    enable: start
    contents: |
      [Unit]
      Description=Docker Socket for the API
      [Socket]
      ListenStream=2375
      BindIPv6Only=both
      Service=docker.service
      [Install]
      WantedBy=sockets.targets
  - name: flanneld.service
    command: start
    drop-ins:
      - name: 50-network-config.conf
        content: |
          [Service]
          ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config '{
"Network": "10.1.0.0/16" }'
users:
  - name: coreos01
    passwd: $1$Inx1WXaK$pgCZfsVqlFKVu4/xSRKVy/
    groups:
      - sudo
      - docker
```

- Arranque del archivo Cloud-Config

```
$ sudo coreos-install –d /dev/sda –C stable –c cloud-config-file
```

CONFIGURACIÓN DE SERVICIO DOCKER Y APACHE CASSANDRA

- Fichero de Descubrimiento de Docker y Apache Cassandra

```
[Unit]
Description=cassandra
After=docker.service
Requires=docker.service

[Service]
Environment=CASSANDRA_CLUSTERNAME=cluster
CASSANDRA_SSL_STORAGE_PORT=7002
EnvironmentFile=-/etc/environment
ExecStartPre=-/usr/bin/docker kill %p-%i
ExecStartPre=-/usr/bin/docker rm %p-%i
ExecStartPre=/usr/bin/docker pull endocode/%p
ExecStartPre=/usr/bin/bash -c "echo $${COREOS_PUBLIC_IPV4:-$$(hostname -
i)} | /usr/bin/etcdctl set /cassandra_%i"
ExecStartPre=/usr/bin/bash -c "while [[ ! $$(/usr/bin/etcdctl get /cassandra_1)
]]; do echo 'Waiting for Cassandra Seed node'; sleep 1; done; echo 'Cassandra Seed
node is UP'; /usr/bin/etcdctl get /cassandra_1"
ExecStart=/usr/bin/bash -c "BROADCAST_ADDR=$${COREOS_PUBLIC_IPV4:-
$$(hostname -i)} && CASSANDRA_SEEDS=$(/usr/bin/etcdctl get /cassandra_1
| /usr/bin/tr -d '\n') && exec /usr/bin/docker run --rm --name %p-%i
-e CASSANDRA_CLUSTERNAME=${CASSANDRA_CLUSTERNAME}
-e CASSANDRA_SEEDS=\"$CASSANDRA_SEEDS\"
-e BROADCAST_ADDR=$BROADCAST_ADDR
-e CASSANDRA_SSL_STORAGE_PORT=$CASSANDRA_SSL_STORAGE_PORT
--publish 7000:7000 --publish
$CASSANDRA_SSL_STORAGE_PORT:$CASSANDRA_SSL_STORAGE_PORT --
publish 9160:9160 --publish 9042:9042 --publish 7199:7199 endocode/%p"
ExecStop=/usr/bin/docker stop %p-%i
ExecStopPost=/usr/bin/etcdctl rm /cassandra_%i
TimeoutStartSec=900s

[X-Fleet]
Conflicts=%p@*.service
```

- Fichero de Servicios de Docker y Apache Cassandra

```
$ docker run --name casscore01 -v /data/cassandra:/var/lib/cassandra -e
CASSANDRA_BROADCAST_ADDRESS="138.4.11.225" -p 7000-7001:7000-7001 -p
7199:7199 -p 9042:9042 -p 9160:9160 -d cassandra

$ docker run --name casscore02 -v /data/cassandra:/var/lib/cassandra -e
CASSANDRA_BROADCAST_ADDRESS="138.4.11.226" -e
CASSANDRA_SEEDS="138.4.11.225" -p 7000-7001:7000-7001 -p 7199:7199 -p
9042:9042 -p 9160:9160 -d cassandra

$ docker run --name casscore03 -v /data/cassandra:/var/lib/cassandra -e
CASSANDRA_BROADCAST_ADDRESS="138.4.11.224" -e
CASSANDRA_SEEDS="138.4.11.225" -p 7000-7001:7000-7001 -p 7199:7199 -p
9042:9042 -p 9160:9160 -d cassandra
```

Bibliografía

[1] Anónimo, *EcuRed*, Ecured, [En línea]. Available: https://www.ecured.cu/Cluster_de_alta_disponibilidad.

[2] J. A. Carrillo Ruiz, D. J. E. Marco de Lucas, J. C. Dueñas Lopez, F. Cases Vega, J. Cristino Fernandez, G. Gonzalez Muñoz de Morales y L. F. Pereda Laredo, *Big Data en los Entornos de Defensa y Seguridad*, Marzo 2013. [En línea]. Available: http://www.ieee.es/Galerias/fichero/docs_investig/DIEEEINV03-2013_Big_Data_Entornos_DefensaSeguridad_CarrilloRuiz.pdf.

[3] D. Kai, Z. Zhuxi, W. Huaimin y Y. Shuqiang, *RAIDB5: An Economical and High Available Database Cluster for Large-Scale Archived Stream*, 2008 Seventh International Conference on Grid and Cooperative Computing, vol. 1, nº 1, pp. 273-280, 2008.

[4] X. Noha, H. Harek y Y. Anis, *On Automated Cloud Bursting and Hybrid Cloud Setups Using Apache Mesos*, 2017 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech), vol. 1, nº 1, pp. 1-8, 2017.

[5] A. Cyrille, G. Quentin, R. Guillaume, B. Kazuaki, M. Lei y K. Takashi, *Model-based API Testing of Apache ZooKeeper*, 10th IEEE International Conference on Software Testing, Verification and Validation, vol. 1, nº 1, pp. 288-298, 2017.

[6] *CoreOS ETCD*, RedHat, 2017. [En línea]. Available: https://coreos.com/etcd/.

[7] *CoreOS Fleet*, RedHat, 2017. [En línea]. Available: https://coreos.com/fleet/docs/latest/.

[8] *Operating Coreos System*, Redhat, 2017. [En línea]. Available: https://coreos.com/why/.

[9] CoreOS, *CoreOS Blog*, Redhat, 1 December 2014. [En línea]. Available: https://coreos.com/blog/rocket.html.

[10] N. Naik, *Docker Container-Based Big Data Processing System in Multiple Clouds for Everyone*, 2017 IEEE International Systems Engineering Symposium (ISSE), vol. 1, nº 1, pp. 1-7, 2017.

[11] Docker, *Docker Community*, Docker Inc., 2017. [En línea]. Available: https://www.docker.com/what-docker#/container-platform.

[12] *ThoughtWorks NoSQL Databases: An Overview*, ThoughtWorks, 2 Octuber 2014. [En línea]. Available: https://www.thoughtworks.com/insights/blog/nosql-databases-overview.

[13] *MongoDB*, MongoDB Inc., 2018. [En línea]. Available: https://www.mongodb.com/what-is-mongodb.

[14] *Apache Cassandra*, The Apache Software Foundation, 2016. [En línea]. Available: casandra.apache.org.

[15] M. Bailey, *CoreOS in Action Running Aplications on Container Linux*, New York: Manning Publication Co., 2017.

[16] S. Makam, *Mastering CoreOS*, Birmingham: Packt Publishing, 2016.

[17] J.-F. Paris y D. D. E. Long, *Pirogue, a lighter dynamic version of the Raft distributed consensus algorithm*, 2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC), vol. 1, nº 1, pp. 1-8, 2015.

[18] O. Diego y O. John, *In Search of an Understandable Consensus Algorithm (Extended Version)*, 2014 USENIX Annual Technical Conference, vol. 1, nº 1, pp. 1-18, 2014.

[19] B. Johnson, *Raft: The Understandable Distributed Consensus Protocol*, Speaker Deck, 20 Septiembre 2013. [En línea]. Available: https://speakerdeck.com/benbjohnson/raft-the-understandable-distributed-consensus-protocol/.

[20] G. F. Tamás Kovácsházy y B. S. Csaba, *A Distributed Power Consumption Measurement System and its Applications*, 2011 12th International Carpathian Control Conference (ICCC), vol. 1, nº 1, pp. 224-229, 2011.

[21] D. Merkel, *Docker: lightweight Linux containers for consistent developmen and deployment*, Linux Journal, vol. 2, nº 2, 2014.

[22] *Docker overview*, Docker Inc., 2017. [En línea]. Available: https://docs.docker.com/engine/docker-overview/.

[23] *What is a Container*, Docker Inc., 2017. [En línea]. Available: https://www.docker.com/what-container.

[24] S. Vinod, C. Jeeva S., R. Pethuru y M. Russ, *Launching Applications Using Docker*, Packt Publishing, 2017.

[25] *Datastax Academy What is Apache Cassandra*, Datastax Academy Enterprise, 2018. [En línea]. Available: https://academy.datastax.com/planet-cassandra/what-is-apache-cassandra.

[26] H. Eben y C. Jeff, *Cassandra: The Definitive Guide, 2nd Edition Distributed Data at Web Scale*, Birmingham: O'Reilly Media, 2016.

[27] *Getting Started and User Information Apache Cassandra*, Datastax Inc, 2017. [En línea]. Available: https://docs.datastax.com/en/landing_page/doc/landing_page/current.html.

[28] *The Python Tutorial*, Python Software Foundation, 2017. [En línea]. Available: https://docs.python.org/3/tutorial/index.html.

[29] M. Venkitachalam, *Python Playground*, New York: No Strach Press, 2015.

[30] A. d. l. P. Juan, A. Alejandro, Z. Juan, G. Jorge y S. Angel, *UPMSat-2 Technical Specification Software Requirements Specification*, Madrid, 2018.

[31] C. Flannel, *Github Coreos Flannel*, 2017. [En línea]. Available: https://github.com/coreos/flannel.

[32] D. M. Pradales, *nobbot Tecnologias para las Personas*, 2017 07 18. [En línea]. Available: https://www.nobbot.com/general/python-lenguaje-programacion/.