**Experimento 3**

Grupo 6

Camilo Montenegro

Tomas Venegas

Juan Diego González

Juan Manuel Lovera

Carlos Peñaloza

Colmines

**Problemática**

Se quiere implementar un sistema con 2500 microcontroladores, cada uno con 4 sensores que envían información a intervalos con el fin de identificar condiciones de alerta en una mina. En este caso se tienen que asegurar las distintas partes relacionadas con la solución del problema para que la información, infraestructura o servicios no se vean afectados.

**Objetivo del experimento**

Se desea implementar un mecanismo de seguridad integrado, el cual evite cualquier tipo de ataque que pueda afectar el funcionamiento de la mina. Se debe asegurar las entidades físicas (autenticación y autorización para poder escribir en los tópicos del servidor de mensajería que se está utilizando), también el aseguramiento de las entidades virtuales que se tienen ( bridges, kafka, zookeeper, etc…) y por último los actores humanos, quienes son los que se deben autenticar y autorizar para que tengan acceso a ciertos servicios.

Por otro lado, se implementa la interfaz gráfica con la utilización de ciertas técnicas de usabilidad para que el usuario pudiera tener un mejor manejo del sistema. Para ello se utilizó la técnica de paginación para que los datos se vieran organizados. Debido a que se tiene una gran cantidad de datos, se logró realizar una paginación por parte del servidor. De esta manera se mitiga la problemática de la carga de datos por parte del cliente. Asimismo, se mantiene al usuario informado de en qué lugar de la aplicación en el momento. Esto es muy importante ya que el cliente sabe donde se encuentra y sabe a donde tiene que interactuar para llegar a su objetivo. También es importante aclarar que se puede llegar a cualquier módulo de la aplicación con mínimo tres clicks.

**Descripción del experimento**

En este experimento se probó la implementación de tácticas de seguridad en los distintos niveles manejados en la aplicación.

**Entidad Física:**

El envío de datos de bajo nivel desde los microcontroladores, pasando por el middleware y luego enviados al servidor de mensajería utilizando tópicos, maneja el protocolo de seguridad SSL.

Inicialmente, esta implementación se realiza utilizando SSL y haciendo publicaciones por tópico en CloudMQTT para que luego estos sean consumidos por un Bridge.

**Entidad Virtual:**

Al momento que los datos son recibidos desde el servidor de mensajería MQTT por el Bridge, esté se encarga de descifrar el mensaje para que pueda ser publicado en otro tópico en Kafka. Dicha comunicación entre el Bridge y Kafka+Zookeeper se realiza de igual forma con SSL utilizando un cifrado asimétrico y certificados digitales en ambos nodos de comunicación. De esta forma, se está asegurado tanto la lectura como escritura de los datos en Kafka+Zookeeper.

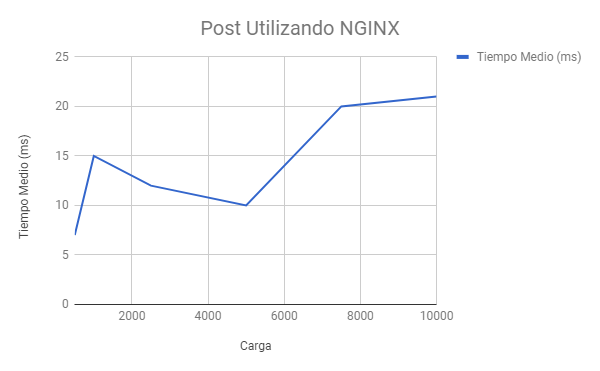
**Actores Humanos:**

Para el experimento, se implementó un sistema de autenticación por hashing. Existen diferentes grupos de usuarios y cada grupo cuenta con permisos diferentes en cuanto al uso de la aplicación. Las credenciales de los usuarios se almacenan en una base de datos luego de que se les realiza un hashing con el algoritmo SHA-256.

**Pruebas de Carga y Comparación**

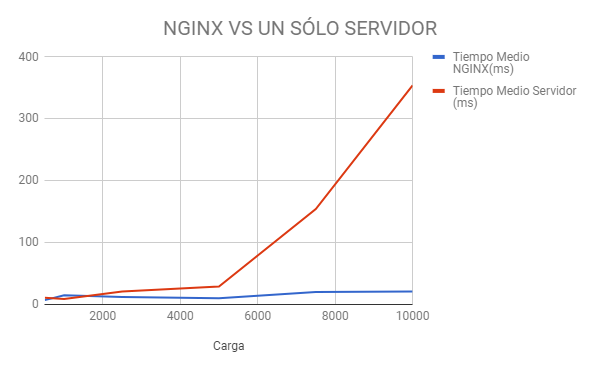
Para este experimento se realizaron pruebas de carga en dos ocasiones. Una, antes de la implementación de un balanceador de carga (NGINX) para ver cómo se comportaba la arquitectura y que tantos datos podía soportar. Posteriormente, después de haber implementado se realizaron pruebas de carga sobre este componente para ver como este respondía a las peticiones y cómo las distribuían. A continuación se pueden ver los datos recolectados con las pruebas realizadas:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Post usando NGINX** | | | | |
| **Carga** | **Tiempo Medio (ms)** | **Mínimo (ms)** | **Maximo (ms)** | **Error** |
| 500 | 7 | 7 | 93 | 0% |
| 1000 | 15 | 7 | 586 | 0% |
| 2500 | 12 | 5 | 101 | 0% |
| 5000 | 10 | 5 | 84 | 0% |
| 7500 | 20 | 5 | 201 | 0% |
| 10000 | 21 | 5 | 3002 | 0% |

****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Post con un sólo servidor** | | | | |
| **Carga** | **Tiempo Medio (ms)** | **Mínimo (ms)** | **Máximo (ms)** | **Error** |
| 500 | 11 | 3 | 121 | 0% |
| 1000 | 9 | 3 | 196 | 0% |
| 2500 | 21 | 3 | 1207 | 0% |
| 5000 | 29 | 3 | 2042 | 0% |
| 7500 | 154 | 3 | 3241 | 0% |
| 10000 | 354 | 3 | 4096 | 0% |





Se puede ver claramente que utilizando un balanceador de carga ayuda a mejorar el desempeño de la aplicación de una manera sustancial, debido que al tener redundancia de los nodos de procesamiento ayuda a que la carga se distribuya de manera equitativa y las tareas se realicen de manera más ágil. Es importante aclarar que cuando se tienen pocos datos, las dos alternativas se comportan de manera similar, pero al incrementar la carga, se ve que el servicio que utiliza el balanceador tiene un tiempo de respuesta menor a la que no tiene dicho balanceador.

**Decisiones de arquitectura**

Utilizamos tácticas de autorización y autenticación en todos los intercambios de mensajes. Y adicionamos tácticas de verificación y encriptación a los mensajes que tenían conexión con entidades externas a los servicios centrales.

La base de datos que contiene la información de autenticación está aislada de la base datos con la información del negocio. Esto lo hicimos con la intención de aumentar la seguridad, separar los objetivos y afectar el desempeño de la aplicación lo menos posible.

Se decidió implementar una redundancia de los nodos de procesamiento para poder mejorar el rendimiento y la disponibilidad de la aplicación. Para ello, se construyeron dos réplicas del servidor en dos máquinas distintas y además en una máquina aparte, se incorporó NGINX como balanceador de carga. Estos dos servidores se conectan con la misma base de datos.

**Artefactos construidos**

Debido a que la aplicación se basa en un patrón de diseño Model View Template implementado con Django, fue necesario definir los modelos y los views asociados a las entidades involucradas en el funcionamiento de la aplicación, en este caso, fue posible definir dos artefactos generados (Modelos y

views), estos dos artefactos fueron definidos en torno al funcionamiento de la aplicación, es decir, de acuerdo a los diferentes tipos de sensores que existen en cada uno de los microcontroladores.

Se implementaron servicios de autenticación y autorización en todos los niveles de la aplicación y además de esto se implementaron técnicas de encriptación y certificados.

**Análisis**

Los resultados indican que el servidor fue capaz de soportar la demanda de peticiones

máxima de alrededor de 7000 solicitudes por minuto, con un porcentaje de error mínimo. Esto significa que los patrones e implementación del diseño no fueron suficientes para cumplir totalmente con los requisitos del sistema bajo condiciones normales y tener un excelente desempeño bajo condiciones de estrés.

La pruebas de unidad fueron exitosas.

Las pruebas sobre el servidor de mensajería nos muestran dos cosas: la primera es que el servidor de mensajería es apto para recibir todas las peticiones requeridas y cumple con los atributos de calidad. La segunda es que el script publicador que hace las publicaciones en el servidor de Django no es lo suficientemente rápido.

**Conclusiones**

Vemos que la arquitectura implementada no satisface los requerimientos no funcionales.

Para mejorar el tiempo de respuesta de las peticiones POST proponemos utilizar una base de datos no relacional con una arquitectura hadoop.

Para solucionar el problema del publicador proponemos implementar en una segunda instancia un mecanismo multi-thread para el publicador,ya que la división en tópicos no parece ser suficiente para atacar este problema.

Es importante aclarar que la distribución de la carga afecta de manera positiva el desempeño de la aplicación ya que por medio de las tareas se distribuyen de manera equitativa de tal forma que se realizan de manera más ágil. Esto también ayuda a mejorar la disponibilidad de la aplicación .