Universidad Nacional de Educación a Distancia
Trabajo fin de máster (código 31105151)

MÁSTER UNIVERSITARIO DE INVESTIGACIÓN EN
INGENIERÍA DEL SOFTWARE Y SISTEMAS INFORMÁTICOS

Implementación de un sistema de videovigilancia basado en microservicios mediante el uso de la tecnología blockchain

Autor:
Helen Mariela Medina Chanca

Director:
Ismael Abad Cardiel

Curso: 2019/2020
Convocatoria: Julio de 2020

E.T.S. de Ingeniería Informática

Universidad Nacional de Educación a Distancia
Trabajo fin de máster (código 31105151)

MÁSTER UNIVERSITARIO DE INVESTIGACIÓN EN
INGENIERÍA DEL SOFTWARE Y SISTEMAS INFORMÁTICOS

Implementación de un sistema de videovigilancia basado en microservicios mediante el uso de la tecnología blockchain

Computación ubicua (tipo de trabajo B)

Autor:
Helen Mariela Medina Chanca

Director:
Ismael Abad Cardiel

E.T.S. de Ingeniería Informática

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha:22/06/2020

Quién suscribe:

Autor(a): Helen Mariela Medina Chanca
D.N.I/N.I.E/Pasaporte.: 53459617G

Hace constar que es la autor(a) del trabajo:

Implementación de un sistema de videovigilancia basado en microservicios mediante el uso de la tecnología Blockchain

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo. 

Impreso TFdM05_AutorPbl. Autorización de publicación y difusión del TFM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Juan del Rosal, 16
28040, Madrid

Tel: 91 398 89 10
Fax: 91 398 89 09
www.issi.uned.es

Agradecimientos

Después de un intenso periodo de nueve meses de trabajo, hoy es el día en el cual finalizo mi proyecto fin de máster. Ha sido un periodo de aprendizaje intenso no solo en el campo tecnológico sino también a nivel personal.
Realizar este proyecto ha requerido por mi parte un esfuerzo importante no solo por mi situación personal, sino también por las condiciones actuales en las que se ha desarrollado el mismo.

En primer lugar, me gustaría agradecer a mis suegros Enriqueta y Francisco por todo su apoyo y el sacrificio realizado, ya que sin su ayuda no me habría sido posible la finalización de este proyecto. Muchas gracias por todo.

A mis padres, Carlos y Aída, por el apoyo incondicional que siempre me han brindado.
A mi padre, por todos sus consejos y por guiarme en mi vida profesional y personal.
A mi madre, a la cual no tengo palabras para agradecerle todo su cariño y apoyo. Tú que me has acompañado y has estado siempre a mi lado a lo largo de mi vida en los mejores y peores momentos. Gracias a vosotros dos que sois mi modelo a seguir en cuanto al trabajo duro y sacrificio personal.

A Sergio, quien siempre me ha brindado su apoyo y comprensión. Especialmente en estos momentos.

A mi hermano Erik, por su apoyo y sus consejos.

A todos mis tíos y mis primos, especialmente por estar pendientes y cuidar de mis padres cuando yo no he podido estar presente. Gracias a vosotros, ya que de otra manera no habría tenido la tranquilidad emocional para terminar este proyecto.

Finalmente, me gustaría agradecer al director de este proyecto, Ismael, por orientarme en la realización del mismo.

Gracias a todos,

Helen

Dedico este proyecto a mi hijo, Eduardo,
con todo mi amor.

Resumen

En los últimos años hemos asistido a un aumento considerable de la aplicación de dispositivos *IoT* en diversos ámbitos. Uno de estos ámbitos son los sistemas de servicio de videovigilancia en ciudades "inteligentes". Se espera de estos sistemas que sean soluciones escalables y descentralizadas que permitan el uso simultáneo de varias cámaras distribuidas en distintas zonas dentro del área de videovigilancia.
Puesto que se trabaja con información sensible y se utilizan dispositivos *IoT*, que están siempre conectados a internet, se debe aportar soluciones que sean más seguras. Por lo tanto, no se pueden aplicar los mismos diseños y tecnologías empleadas en servicios de videovigilancia tradicionales, ya que estos servicios habitualmente utilizan arquitecturas monolíticas que son poco flexibles y poco escalables. Además, este tipo de sistemas tradicionales son más vulnerables ante un fallo del mismo o ataques realizados por usuarios externos, dado que utilizan un sistema centralizado con medidas de seguridad que dejan de ser válidas si se aplican a otros descentralizados.

Por ello, con el objetivo de solventar estos problemas, en este proyecto se propone la construcción de un sistema descentralizado principalmente con el uso de dos tecnologías: microservicios y *blockchain*.
Por un lado, se utiliza el diseño arquitectónico de microservicios que aporta ciertas ventajas que se consideran necesarias en la construcción de una solución descentralizada, ya que entre otras ventajas, aporta escalabilidad, acoplamiento e independencia en el despliegue y desarrollo de servicios con diferente funcionalidad.

Por otro lado, se utiliza la tecnología *blockchain* con la finalidad de proporcionar privacidad y seguridad. De esta forma se evita que usuarios externos tengan acceso al vídeo capturado por las cámaras y a los datos obtenidos en procesos realizados por los servicios.
Se considera que se pueden emplear los *smart contracts* como un mecanismo de control de acceso que permita acceder y compartir la información entre los distintos elementos del sistema, manteniendo siempre un nivel de seguridad riguroso necesario para este tipo de solucionaes.

Finalmente, se propone el uso de un modelo divido en cuatro capas que engloba los distintos componentes del sistema. Este modelo está basado en el paradigma *fog-edge-cloud computing* que solventa problemas habituales que se producen en modelos tradicionales como *cloud computing*. Siendo los principales problemas la centralización de las operaciones, el incremento de dispositivos *IoT*, el rendimiento (desde el punto de vista de procesamiento de datos) y la latencia. Donde estos últimos son elementos importantes a tener en cuenta en servicios de videovigilancia, dado que se trata de servicios que realizan el envío de información en tiempo real.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Introducción

En las últimas décadas se ha experimentado un importante desarrollo en el campo de la tecnología y de la comunicación, lo que ha permitido el desarrollo de la revolución industrial 4.0.
Por otra parte, se viene observando un aumento de la urbanización a nivel mundial; por ejemplo para el año 2025 se espera que el 58 % de la población mundial residirá en áreas urbanas [1].

Estos dos eventos han repercutido en la aparición del concepto "ciudades inteligentes" o también conocido como *smart cities*.
Estas ciudades con la adopción de soluciones *TIC* tienen la capacidad de crear, recolectar y procesar información con el objetivo de mejorar sus servicios y el uso eficiente de recursos.

Para llevar a cabo el desarrollo de una "ciudad inteligente", es necesaria la aplicación de otro concepto conocido como "Internet de las cosas" o IoT, que a su vez ha surgido recientemente y se encarga de conectar ciudades a través de millones de dispositivos conectados a la red.
Para que el sistema de estas ciudades sea sostenible se requiere de la capacidad de cálculo y de análisis de datos en la red local del dispositivo inteligente, especialmente en servicios que gestionan información en tiempo real para llevar a cabo decisiones críticas o con el objetivo de reaccionar inmediatamente ante cierto tipo de emergencias.

Con el objetivo de mejorar la calidad de vida de los ciudadanos de una "ciudad inteligente", se ve necesaria la implantación de servicios de videovigilancia en distintos lugares de la ciudad. Estos sistemas de videovigilancia, en comparación con los sistemas de videovigilancia tradicionales que normalmente utilizan una arquitectura monolítica, requieren la posibilidad de incorporar nuevas funcionalidades y que sean escalables, ya que se espera llevar a cabo el análisis de vídeo en tiempo real de una importante cantidad de vídeos obtenidos de múltiples dispositivos IoT.
En este caso, el uso de una arquitectura basada en microservicios facilitaría no sólo la incorporación de nuevos servicios y funcionalidades, sino que además proporcionaría una mayor escalabilidad al sistema.

Ciertos procesamientos de vídeo, como la monitorización en tiempo real, implican llevar a cabo tareas con muy alto coste computacional y medidas de decisión en tiempo real con el objetivo de evitar actividades delictivas, prevenir accidentes de tráfico u otras medidas de control y seguridad.
Para llevar a cabo estos procesos se ha estado empleando hasta el momento el paradigma de *cloud computing*, ya que resulta adecuado para realizar tareas con gran cantidad de datos (*Big Data*).
Sin embargo, *cloud computing* no es la solución óptima para servicios de videovigilancia en tiempo real debido al gran consumo de ancho de banda y latencia.
A lo largo del documento se verán otros paradigmas más apropiados para este tipo de procesos.

Otro problema a tener en cuenta es la falta de seguridad y privacidad en servicios de videovigilancia, ya que durante los últimos años se ha visto una mayor presencia de dispositivos IoT en este tipo de servicios. Por ello, es importante establecer servicios de seguridad descentralizados que permitan controlar el acceso a servicios o recursos de sistemas que utilizan este tipo de dispositivos "inteligentes".

Teniendo en cuenta lo anteriormente expuesto, el presente proyecto proporciona una solución de videovigilancia con tecnología segura basada en blockchain.

1.2. Objetivos

El objetivo principal de este proyecto es:

- Diseñar y construir un sistema de videovigilancia basado en microservicios con seguridad basada en blockchain.

Los objetivos secundarios son:

- Solucionar el problema de ancho de banda y de latencia del paradigma *cloud computing* empleando una infraestructura basada en el modelo *edge/fog processing* que traslade la responsabilidad de llevar a cabo ciertas tareas de decisión u otros procesamientos de la nube a los nodos *edge* y *fog*, y así, reducir significativamente el volumen de datos que se envían de los dispositivos a la nube.
- Utilizar la arquitectura de microservicios en el nodo *edge* y *fog* donde cada servicio encapsula un algoritmo de procesamiento independiente de vídeo o datos en implementación y despliegue al resto de servicios.
- Implementar un sistema de seguridad que permita impedir el acceso de usuarios no autorizados y la manipulación de vídeo o datos extraídos del proceso de análisis de vídeo.

1.3. Estructura de la memoria

El documento presenta la siguiente estructura:

- **Capítulo 1. Introducción**. Explica brevemente el contexto en el que se desarrolla el documento. Se enumeran los objetivos fundamentales que se pretenden cubrir y un breve comentario acerca de la estructura del documento.
- **Capítulo 2. Estado del arte**. Se describe el problema a solucionar en relación con otros trabajos existentes en el mismo ámbito. Se propone una solución que se compone de las tecnologías de microservicios y *blockchain* aplicado al ámbito de la videovigilancia.
- **Capítulo 3. Diseño del proyecto**. En este capítulo se describe la solución con más nivel de detalle. Se lleva a cabo el análisis, el diseño funcional y el diseño técnico de la solución propuesta.
- **Capítulo 4. Pruebas**. En este capítulo se realiza una batería de pruebas que muestran la funcionalidad básica del prototipo diseñado y la gestión de errores ante un mal uso de la aplicación.
- **Capítulo 5. Conclusiones y líneas futuras**. Se describen las conclusiones del proyecto y posibles trabajos futuros.
- **Apéndice**. Explica de forma resumida las tecnologías empleadas en el proyecto que no forman parte del estado del arte. Estas tecnologías son *blockchain* y *Kafka*, aplicados a la seguridad y al sistema de comunicación respectivamente.

Capítulo 2

Estado del arte

2.1. Introducción

A principios de la última década del siglo XX aparece el concepto de *smart cities*. Este reciente fenómeno ha ido extendiéndose rápidamente por el mundo en los últimos años, de modo que en las ciudades existe claramente una tendencia al uso de nuevas tecnologías para afrontar problemas relacionados con el tráfico urbano, contaminación, eficiencia energética o seguridad.

Se han aportado varias definiciones de *smart cities*, mostrándose a continuación una de las definiciones que se considera más apropiada para el contexto de desarrollo de este proyecto:

> "*A Smart City is a city that uses Smart Computing technologies to make the critical infrastructure components and services of a city – which include city administration, education, healthcare, public safety, real state, transportation, and utilities – more intelligent, interconnected and efficient*".[2]

2.2. Videovigilancia

Uno de los retos a alcanzar son los servicios de videovigilancia inteligentes, es decir, servicios que llevan a cabo de forma eficiente procesos en tiempo real de los vídeos recopilados de uno o varios dispositivos *IoT* conectados a la red.

Estos servicios son una parte esencial de la sostenibilidad de las ciudades inteligentes. Muchos de los procesos que se realizan en estos servicios requieren un rendimiento de alta precisión y toma de decisiones en tiempo real a partir de datos obtenidos o recopilados, con el objetivo de evitar resultados erróneos que generalmente son consecuencia de disponer de información insuficiente o debido a predicciones incorrectas.

Se han recopilado algunos ejemplos de videovigilancia con soluciones inteligentes para diversos servicios que se pueden encontrar fácilmente en una *smart city*.

En [3] se diseña un sistema basado en un modelo de tres niveles de jerarquía de nodos para detectar y monitorizar personas.
En este caso, las secuencias de vídeo son procesadas en tiempo real en el primer nivel (nodo *edge*[1]) y posteriormente se envía al segundo nivel (nodo *fog*[1]) o al tercer nivel (la nube) las características extraídas del vídeo junto con la información de la monitorización.

La arquitectura empleada es la siguiente:

- Primer nivel: Encargado de detectar personas, monitorizarlas y enviar dicha información al segundo nivel.
- Segundo nivel: Encargado de reconocer acciones y llevar a cabo procesos de descripción semántica.

[1] Para más información ver apartado Paradigmas de computación fog/edge

Después de llevar a cabo el análisis de la información obtenida del primer nivel, se obtienen modelos de la actividad de las personas, que son enviadas al tercer nivel o a la nube.

- Tercer nivel: Encargado de llevar a cabo tareas de mucho coste computacional, como algoritmos de decisión o algoritmos de aprendizaje automático.

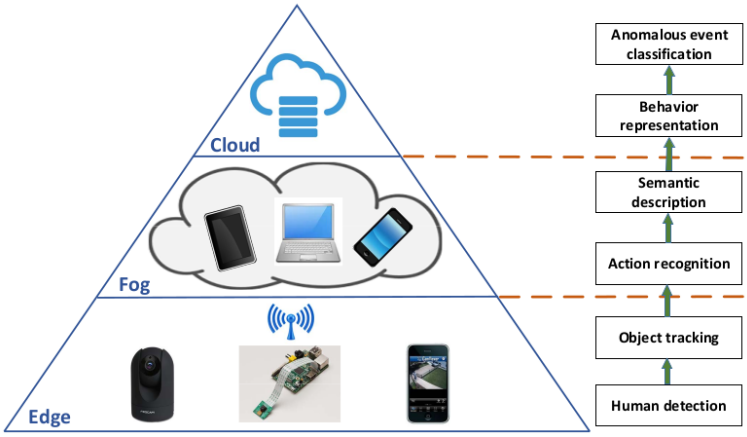

Figura 2.1: Arquitectura del modelo *cloud-fog-edge computing*

Con un modelo muy similar a [3], en el artículo [4] se diseña un sistema de monitorización de tráfico urbano en tiempo real con el objetivo de reducir el número de accidentes y la optimización de los servicios de emergencia.

En [5] se propone un sistema eficiente de gestión (basada en el modelo *cloud computing*) para recolectar la basura de ciudades *smart cities*.
Dicho sistema tiene los siguientes objetivos:

- Proveer servicios *SaaS* (*Software as a Service*), de tal manera que los clientes puedan conectarse a las aplicaciones proporcionadas por la nube.
- Permitir la comunicación entre los distintos participantes del sistema recolector de basura.

Este modelo dispone de varias funcionalidades:

- Compartir datos entre los conductores en tiempo real para llevar a cabo una optimización de la ruta.
- Control de las áreas inaccesibles de la ciudad.

Entre estas funcionalidades destaca el servicio de videovigilancia que utiliza una o varias cámaras *wireless*, cuyos propósitos son los siguientes:

- Recolección de evidencias para llevar a cabo análisis de accidentes.
- Reportar problemas.
- Evidencias del correcto trabajo realizado en la recolección de basura.

En la figura 2.2 se muestra el momento en el que un conductor de un camión de basura reporta un problema con la grabación de un vídeo de un área conflictiva.

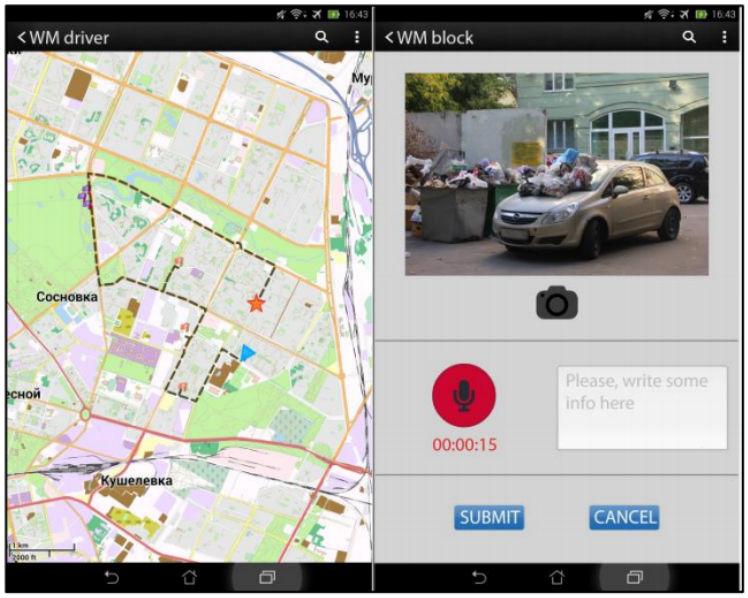

Figura 2.2: Aplicación de gestión de basura

En [6] se propone un sistema basado en un modelo con dos niveles de procesamiento de datos para mejorar los servicios de emergencia de un hospital.
Este sistema dispone de dos servidores:

- En el primer servidor se encuentra instalado el nodo *fog*, donde llega la información de los signos vitales del paciente en tiempo real vía *wifi*, empleando el protocolo *RTMP* (protocolo de transmisión en tiempo real).
  Estos datos se envían posteriormente a la nube.
- En el segundo servidor se encuentra instalada la nube.

La arquitectura del sistema (ver figura 2.3) se divide en tres partes:

- Cliente final o consumidor.
  Los médicos en el hospital pueden acceder a la información del paciente a través de un dispositivo móvil. Por otro lado, el doctor (cliente) puede conectarse al nodo *fog* para acceder a los datos obtenidos por los dispositivos *IoT* del dispositivo *Raspberry Pi* mediante el protocolo *RTMP*.
- Productor.
  Se envía al nodo *fog* la información obtenida por los sensores de temperatura y ritmo cardíaco a través de la cámara del dispositivo *Raspberry Pi 3*.
- Segundo y tercer nivel de procesamiento.
  El nodo *fog* envía la información recopilada a la nube y se vigilan constantemente las medidas obtenidas por el *Raspberry Pi* y, en caso de producirse una emergencia (signos vitales bajos), se envía una notificación al doctor vía *SMS*, reduciendo así la latencia en proceso de actuación de los servicios de emergencia.

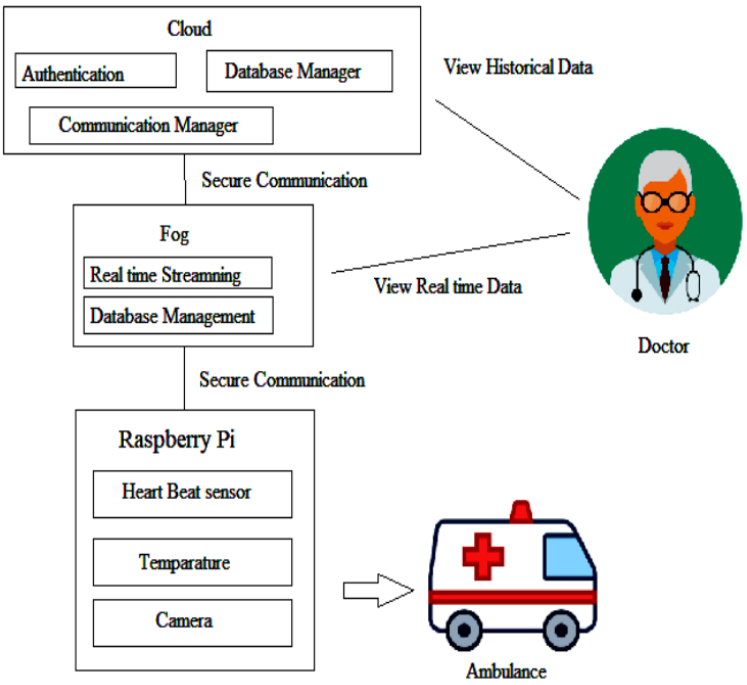

Figura 2.3: Sistema de servicio de emergencia "inteligente"

2.3. Paradigmas de computación fog/edge

"Internet de cosas" conocido como *IoT* es un concepto que se utiliza para definir un conjunto de objetos conectados [7] a través de la red. Estos dispositivos pueden ser agrupados en *clusters* ya sea de forma lógica o geográfica. Estos *clusters* de dispositivos *IoT* generan una gran cantidad de información desde distintas zonas geográficas, por consiguiente se necesita encontrar la forma de procesar estos datos de una manera eficiente.

Hasta el momento se ha estado utilizado el paradigma de *cloud computing* que proporciona de forma ubicua y a demanda acceso a recursos a través de una red que usualmente es internet.

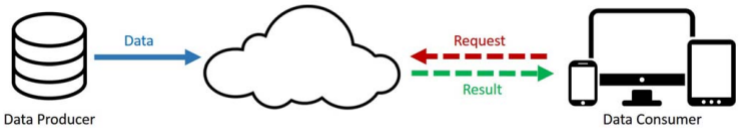

Figura 2.4: Modelo de computación en la nube

Como se puede ver en la figura 2.4 [8], en este tipo de modelo el productor genera los datos que se transfieren a la nube y posteriormente los consumidores realizan una petición (color rojo) a la nube para recibir como resultado (color verde) dichos datos procesados.

Aunque con este modelo se podría gestionar, dependiendo de la capacidad de procesamiento de la nube y del volumen de datos, la información generada por los *clusters* de dispositivos *IoT*, se presenta el problema de transferir la enorme cantidad de datos de los dispositivos a la nube y viceversa debido a la limitación del ancho de banda.

Otras formas eficientes de procesar la información pueden implicar la combinación de modelos computacionales tales como *in situ* [9] u *offloading* [10], donde el procesamiento de los datos se realiza cerca de la fuente de información. Este tipo de modelos pertenecen a los paradigmas *fog computing* y *edge computing*.

2.3.1. Fog computing

El paradigma de *fog computing* procesa los datos relativamente cerca de la fuente (distancia de red *LAN*) buscando reducir principalmente la latencia en procesos críticos como, por ejemplo, sistemas que requieran toma de decisiones en tiempo real.

El consorcio *OpenFog* [11] define el modelo *fog computing* como:

> *"Fog computing is a system-level horizontal architecture that distributes resources and services of computing, storage, control and networking anywhere along the continuum from Cloud to Things."*

El modelo *fog computing* se distingue de *cloud computing* en los siguientes aspectos [12]:

- Proximidad a los usuarios finales. El sistema de computación reside dentro del área de una red *LAN* (red de área local), mientras que en el modelo *cloud computing* el sistema de computación se podría localizar en una red *WAN* (red de área amplia) o *GAN* (red de área global). Es decir, la distancia del nodo *fog* al cliente es de un salto, mientras que con el paradigma de *cloud computing* la distancia del servidor al cliente es de varios saltos. Por tanto, con el paradigma de *fog computing* se mejora la eficiencia de la red en su conjunto y se reduce la latencia de la trasmisión de datos desde los dispositivos *IoT* al servidor. Por ejemplo, en [4] se realizan pruebas de envío de imágenes desde un *dron* a un controlador comparando el modelo *fog computing* y el modelo *cloud computing*. Los resultados muestran que el modelo *fog computing* supera al modelo convencional sin sufrir sobrecargas en la comunicación.
- Puede tener una distribución geográfica distribuida o localizada. El modelo de *cloud computing* dispone de una distribución geográfica centralizada.
- Comprende una mayor cantidad de nodos *fog* con relativamente pocos recursos. El modelo de *cloud computing* consiste de pocos servidores con bastantes recursos.
- A diferencia del modelo *cloud computing*, que en muchos de los casos es incapaz de proveer contenido, servicios y aplicaciones específicas, con el modelo *fog computing* los dispositivos pueden obtener personalización de los servicios, contenido y aplicaciones en función de su localización.
- Finalmente, la movilidad de los dispositivos finales está mejor soportado con el modelo *fog computing*.

A pesar de observar varias ventajas del modelo *fog computing* sobre el modelo *cloud computing*, no se puede reemplazar un modelo por el otro en el diseño de la arquitectura, ya que muchas veces el *cloud computing* es adecuado para tareas de procesamiento de lotes o *batchs* de coste computacional elevado [13].

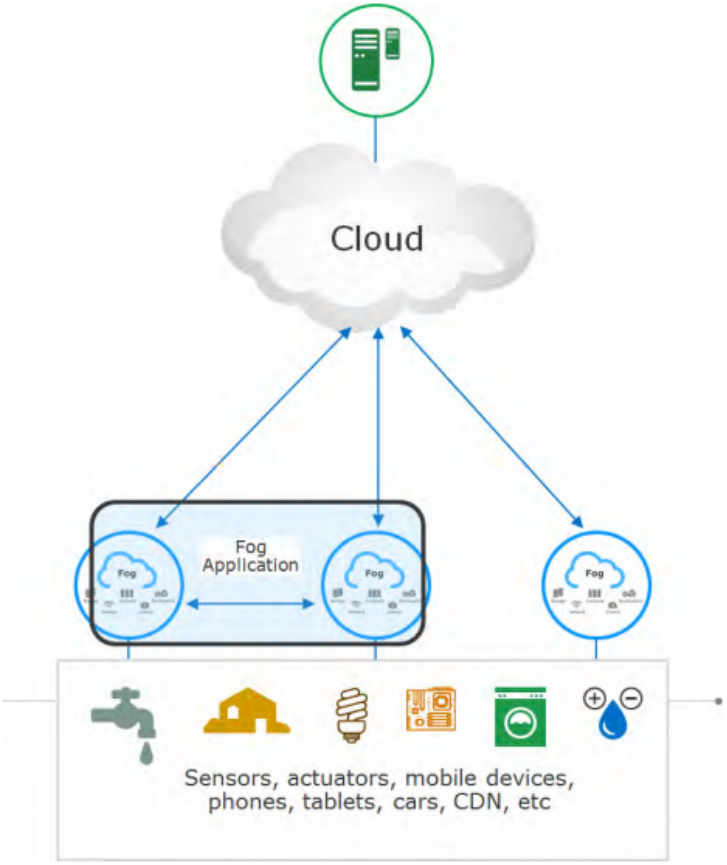

Figura 2.5: Modelo de computación en el nodo *fog*

En la figura 2.5 [14] se observan los distintos niveles de jerarquía de la infraestructura *fog-cloud computing* donde los datos de los dispositivos finales se envían al primer nivel de jerarquía para ser procesados en los nodos *fog* (que pueden o no comunicarse entre sí) y ser posteriormente transmitidos a la nube donde el cliente final tiene acceso.

2.3.2. Edge computing

El paradigma de *edge computing* procesa los datos cerca de la fuente. Por lo tanto, al reducirse la distancia cuando los datos son transmitidos se reduce la latencia, aumenta la velocidad y el rendimiento de los servicios y aplicaciones. En [8] se define el *edge computing* como *"any computing and network resources along the path between data sources and cloud data centers"*.

A continuación, se describen los motivos por los cuales el paradigma de *edge computing* es necesario:

- A pesar de considerar que el procesamiento de los datos en la nube es más eficiente computacionalmente en comparación con el de un dispositivo *IoT*, el modelo convencional de *cloud computing* resulta insuficiente para la gestión de todos los datos recibidos. Esto se debe a que se experimenta un aumento considerable de los datos generados por los dispositivos *IoT* al verse incrementado el número de los mismos en el nodo *edge*, derivando este hecho, a su vez, en problemas con el ancho de banda.
- En algunos casos, los dispositivos *IoT* tienen restricción de energía, por lo que resulta conveniente que se realicen ciertas tareas de procesamiento en el nodo *edge*, ya que una comunicación *wireless* requiere de un considerable consumo de energía.
- Hoy en día, muchos de los dispositivos juegan el papel de productor y consumidor, por lo que se requiere un tiempo de respuesta corto.

- Otro asunto importante es la protección de la privacidad. Por ejemplo, en el caso de los dispositivos de salud, *wearable*, los datos de salud del usuario son recolectados por el dispositivo y por motivos de privacidad el procesamiento de datos se realiza en el nodo *edge* en lugar de enviarlo a la nube para su procesamiento.

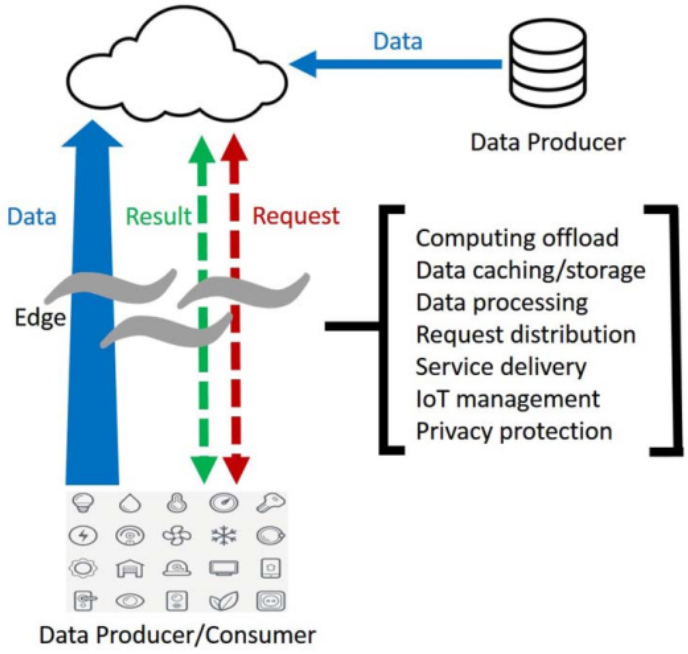

Figura 2.6: Modelo de computación en el nodo *edge*

En la figura 2.6 [8] observamos el paradigma de *edge computing* donde los dispositivos *IoT* juegan el doble rol de productores y consumidores. En el nodo *edge* los dispositivos pueden solicitar servicios y contenidos a la nube y también llevar a cabo ciertas tareas de procesamiento como almacenamiento de datos, protección de datos o gestión.

Los paradigmas *fog node* y *edge node* son muy similares, ya que en ambos el procesamiento de datos se realiza cerca de la fuente. La principal diferencia es dónde se realiza el procesamiento. En el modelo *fog processing* el procesamiento se realiza dentro del área de red local *LAN*, centrándose más del lado de la infraestructura, mientras que con el modelo *edge processing* el procesamiento se realiza directamente en el dispositivo o muy cerca de él, como un controlador de *smart home* [15]. Además, existen ciertas limitaciones en cuanto a análisis en tiempo real y *machine learning* que no se pueden alcanzar con el modelo *edge processing* y sí con el modelo *fog processing*.

Otros trabajos similares en cuanto al uso de paradigmas para el procesamiento de vídeo en tiempo real se pueden consultar en [16] y [17] que utilizan modelos *edge-cloud computing* y *edge computing* respectivamente.

De todas las soluciones mostradas anteriormente se observa que tienen en común la utilización de dispositivos *IoT* para la obtención de vídeo o imágenes que o bien son procesados utilizando modelos *SaaS*(*cloud computing*) o modelos que impliquen principalmente el procesamiento de datos cerca de la fuente (*fog computing* y *edge computing*).

Estos modelos buscan esencialmente maximizar el rendimiento, reducir latencia y solucionar problemas de ancho de banda. Sin embargo, en estas soluciones se observan algunos problemas como la seguridad y la escalabilidad. Por tanto, no tener en cuenta estos aspectos puede ocasionar problemas como comprometer la seguridad del recurso o del servicio o problemas en el diseño si en un futuro se modifica su funcionalidad o el sistema crece verticalmente incorporando nuevos servicios u horizontalmente incorporando nuevos servidores.

2.4. Blockchain

2.4.1. Introducción

A día de hoy, no se conoce con certeza el autor de *blockchain*, dado que esta tecnología fue inventada por una persona o un grupo de personas bajo el anonimato de Satoshi Nakamoto en 2008.

Blockchain es considerado como una base de datos distribuida donde los usuarios pueden acceder a los datos y realizar transacciones de forma segura. Todas estas transacciones se almacenan en bloques de datos después de ser aprobadas por un conjunto de nodos que utilizan un mecanismo de consenso, *mining* (ver apéndice A.1). En *blockchain* cada bloque es identificado por un identificador *hash* que se utiliza para controlar la integridad de los datos que almacena el bloque. Y debido a que en una cadena de bloques cada bloque contiene el identificador del bloque anterior se dificulta cualquier proceso de manipulación o falsificación después de ser almacenados en la red de *blockchain*.

Las características principales de *blockchain* son:

- Dispone de una arquitectura descentralizada, donde nadie tiene el control del sistema. Por esta razón, ninguna corporación o entidad puede censurarlo o pararlo.
- No se puede producir un fallo central y es muy resistente a cualquier intento de manipulación de los datos al no disponer de una arquitectura centralizada.
- Es accesible a todo el mundo.
- Permite la verificación de cada transacción desde el principio de creación de la red de *blockchain*.

Otra característica importante a tener en cuenta son los *smart contracts*, que básicamente son instrucciones que permiten la ejecución de un contrato sin intermediarios en la negociación. Por este motivo, son ideales en mecanismos de seguridad para la protección de datos y el control de acceso de usuarios a servicios o recursos del sistema.

Actualmente, *blockchain* es una tecnología que está en plena expansión y evolución. Hasta el momento han surgido varias variantes de esta tecnología. Algunas muy conocidas son *Ethereum* [18] e *Hyperledger fabric* [19]. Estas redes se pueden utilizar con diferentes propósitos, por ejemplo el envío de dinero (*Bitcoin* [20]), aumento de seguridad y compartir recursos, entre otros usos. Aunque se trata de una tecnología reciente y queda campo por explorar y descubrir otros escenarios donde se pueda aplicar, *blockchain* se considera una tecnología ideal para controlar la seguridad de sistemas descentralizados como puede ser un servicio de videovigilancia compuesto de servicios que procesan o transmiten datos recopilados o vídeos captados por dispositivos *IoT*.

2.4.2. Estudios realizados

Aunque en estos últimos años se han realizado muchos estudios de *blockchain* con la intención de aplicarlo en diversos contextos, se ha prestado especial interés al ámbito de la seguridad y la privacidad. Este interés en el campo de la seguridad se debe a la misma naturaleza de la tecnología, ya que gracias a las características comentadas anteriormente se considera seguro por diversos motivos. Entre ellos, el uso de técnicas de criptografía y mecanismos de seguridad. Al tratarse de un sistema descentralizado siempre está disponible, dado que se compone de múltiples nodos que contienen una copia de la cadena de bloques. Además, nos garantiza la integridad de los datos y la autenticidad. Respecto a la integridad de los datos se debe al hecho de que los datos registrados en un bloque no pueden ser eliminados ni modificados, puesto que esto implicaría comprometer la cadena de bloques restante llevando a cabo el proceso de *mining*. En cuanto a la autenticidad, gracias a la dirección pública siempre se conoce la identidad del usuario que realiza una transacción o ejecuta un *smart contract*.

Dentro del campo de seguridad, el mecanismo de control de acceso es uno de los campos más estudiados con el objetivo de proteger el acceso al dato por parte de usuarios no autorizados, especialmente en sistemas centralizados donde el dato se ve expuesto a un uso mal intencionado [21] y presentan problemas adicionales como el ser más sensibles ante un ataque o problemas

de cuello de botella en cuanto a rendimiento. Para ello, el propietario del dato puede establecer "normas" que controlen el acceso y el tipo de operaciones que un usuario puede realizar sobre un dato.

Existen diversos modelos de control de acceso donde cada uno de ellos tiene ventajas y limitaciones. Por ejemplo, modelos basados en identidad han evolucionado a lo largo del tiempo a modelos basados en roles o en atributos. En [22] los autores proponen un sistema de control de acceso basado en roles (ver figura 2.7) [22] que consiste principalmente en 2 fases:

- Publicar en la red de *blockchain* toda la información relevante del rol de usuario mediante el uso de *smart contracts*.

  El funcionamiento del sistema en esta fase es el siguiente:

  - En primer lugar, la organización es el encargado de generar la clave pública y privada del usuario, al cuál se le envía la clave a través de un canal de comunicación seguro.
  - Posteriormente, se despliega un *smart contract* con operaciones básicas, como por ejemplo, agregar y eliminar usuarios con un tipo de rol específico.
  - Por último, una vez se ha creado el usuario con un determinado rol, el usuario puede acceder a los servicios disponibles para dicho rol.

- Emplear un protocolo de autenticación para verificar si dicho usuario dispone del rol adecuado.

  El funcionamiento de la segunda fase consiste en los siguientes pasos:

  - El usuario solicita acceso a un servicio disponible para el rol del que dispone.
  - La organización, propietaria del servicio, verifica la autenticidad del usuario, solicitando la firma (con la clave pública y privada del usuario) de un dato que la misma organización genera.
  - Finalmente, la organización confirma la autenticidad del usuario comprobando la firma con la clave pública del usuario.

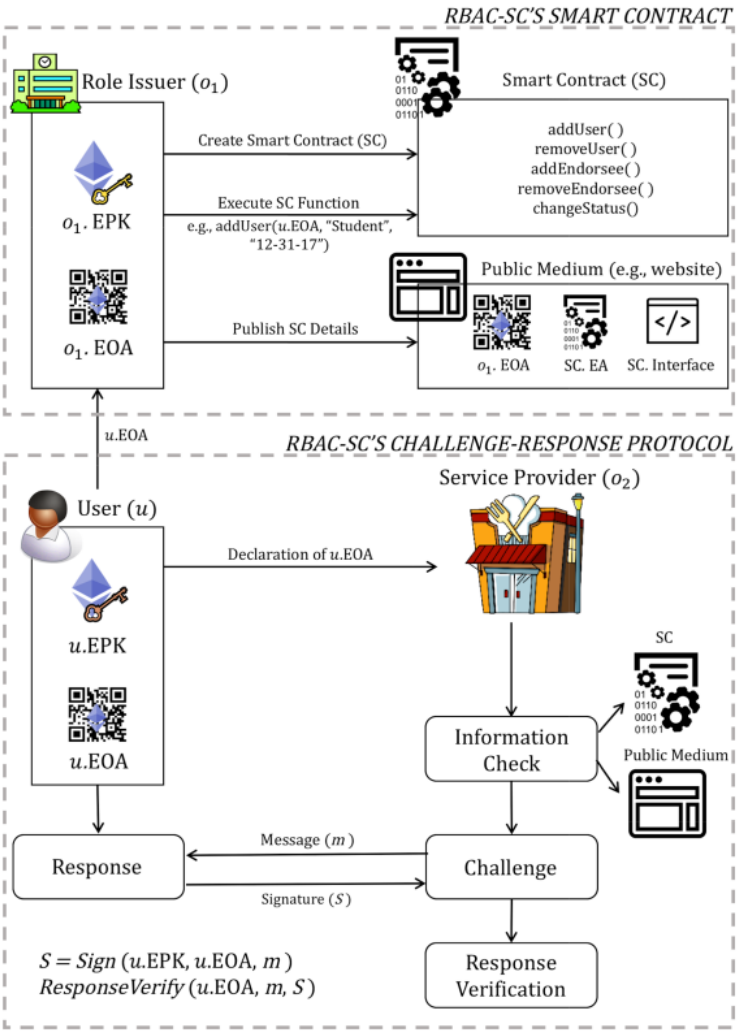

Figura 2.7: Diseño de la infraestructura de control de acceso basado en roles

Otros estudios de este modelo también se observan en [23] y [24]. En [23] se propone la implementación de dos sistemas de control de acceso utilizando el *blockchain Hyperledger Fabric*. Estos dos sistemas de acceso, que tienen un modelo de diseño similar, están basados en la entidad ($BIBAC$) y en el rol ($RBAC$) del usuario o nodo.

El funcionamiento de este sistema consiste en emitir una clave que se utiliza para identificar el usuario o nodo cuando dicho usuario se integra en la red de *blockchain*. Con esta clave, los usuarios o nodos de la red pueden solicitar al propietario el acceso a un recurso, y por tanto, realizar transacciones de forma segura. Además, el propietario del recurso tiene la capacidad de cancelar en cualquier momento el acceso a dicho recurso. Para llevar a cabo estas funcionalidades, los autores implementan 5 funcionalidades básicas con *smart contracts*: solicitar acceso, dar acceso, anular acceso, verificar el acceso y acceder al recurso.

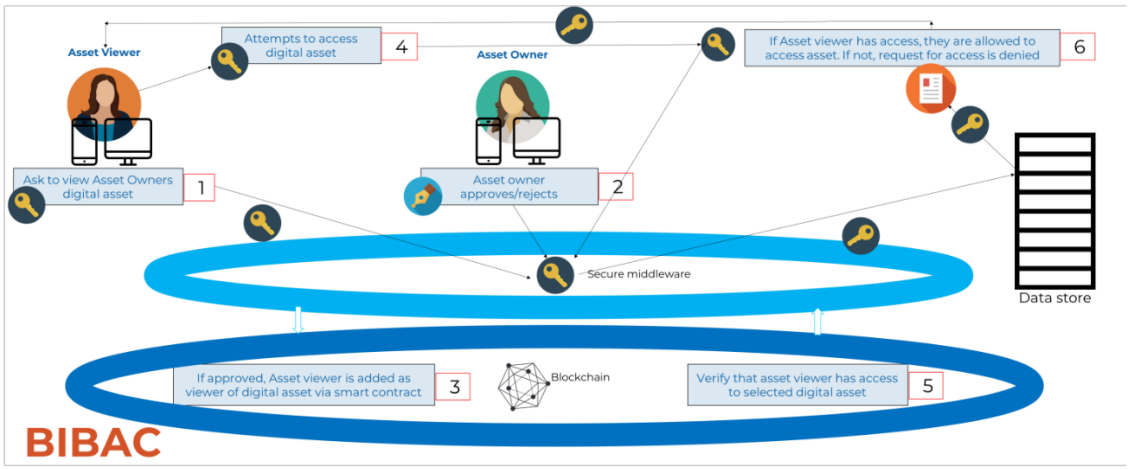

Figura 2.8: Modelo de diseño para el control de acceso *BIBAC*

En [24] el sistema almacena en una red de *blockchain* privada los tokens y las normas de acceso. Los tokens que se emiten por los propietarios del recurso se utilizan como firmas digitales que proporcionan acceso a dichos recursos. De esta manera, se garantiza la autenticación y la actualización de las normas de acceso. En este caso, los autores utilizan *blockchain* como una base de datos que almacena en forma de transacciones las normas de acceso para el par compuesto por el recurso y el usuario que solicita el acceso a dicho recurso.
En cuanto al modelo de control de acceso basado en atributos, consiste en la toma de decisiones según las normas de acceso al sistema y atributos asignados a un usuario. En [25] podemos observar este tipo de modelo donde se propone un control de acceso (*ABAC*) donde el usuario acumula tokens que obtiene de la validación de sus atributos por múltiples autoridades para posteriormente obtener acceso al recurso al cumplir el requisito de obtener un número suficiente de tokens.

En la figura 2.9 se puede ver de forma más detallada el funcionamiento del sistema.

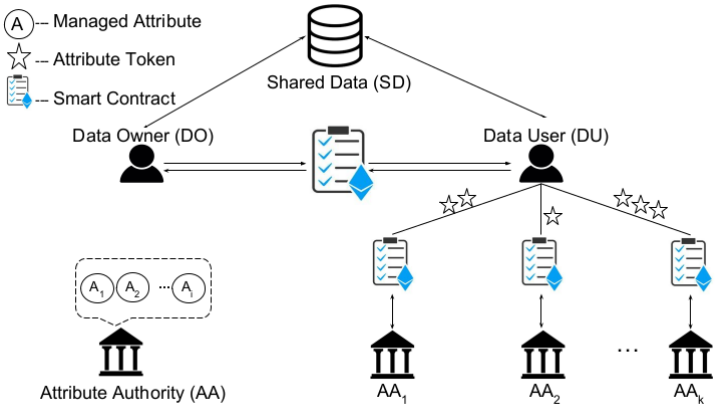

Figura 2.9: Esquema del sistema propuesto *ABAC*

En este sistema, el propietario del recurso (DO) utiliza una clave secreta para encriptar el dato, al cual sólo se puede acceder si se cumple unas normas de acceso (propuesto por el propietario) basado en atributos.
Por otro lado, el usuario (DU) solicita tokens a todas las autoridades propietarias del recurso a través de la ejecución de un *smart contract* para la validación de sus atributos. Una vez obtenido el número suficiente de tokens (el autor no lo especifica), el usuario solicita, utilizando *smart contracts* la obtención de la clave privada para la desencriptación del recurso o dato.
Finalmente, el propietario al comprobar la validez del usuario que ejecuta el *smart contract*, envía la clave secreta encriptada con la clave pública del usuario, de modo que el usuario es el único que puede acceder a esta clave encriptada.

Estos enfoques al igual que otros enfoques similares [26] no son adecuados en sistemas con presencia de IoT porque no proporcionan modelos escalables ni mecanismos eficientes para este tipo de sistemas. Por lo que los modelos de control de acceso deben tener en cuenta las siguientes características [27]:

- **Escalabilidad**. Debido al aumento importante de los dispositivos IoT en diversos sistemas.
- **Heterogeneidad**. Los sistemas IoT normalmente integran diferentes dispositivos físicos con diferentes tecnologías.
- **Causalidad**. Los modelos tradicionales como $RBAC$ y $ABAC$ se diseñan para un uso prolongado en el tiempo. Sin embargo, generalmente los dispositivos IoT se caracterizan por su corto tiempo de vida.
- **"Ligeros de peso"**. Los dispositivos IoT disponen recursos limitados, por lo que no pueden llevar a cabo tareas exhaustivas.

Teniendo en cuenta estas características se han encontrado otros modelos de acceso más adecuados para sistemas IoT. En [28] se propone un sistema descentralizado llamado *bubbles of trust* que consiste en la creación de zonas virtuales seguras donde los dispositivos pueden comunicarse de forma segura, ya que cada dispositivo puede comunicarse únicamente con otro dispositivo que pertenece a la misma zona virtual.

Con el objetivo de identificar y autentificar al usuario, el funcionamiento de este sistema se compone de las siguientes fases:

- Primera fase o fase de inicialización. Se designa un nodo como el nodo *master* encargado de crear el grupo virtual *bubble*.
- Segunda fase. Una vez creadas las zonas virtuales, el resto de nodos o *followers* solicitan su incorporación a dichas zonas. Para ello, el nodo *master* de la zona virtual genera un *ticket* que contiene el identificador del grupo "*grpID*", el identificador del nodo *follower*, "*objID*", su dirección pública en la red de *blockchain* y una firma digital (*Elliptic curve digital signature*) firmada con la clave privada del nodo *master*.

```
========================================
GroupID : XX
ObjectID: YY
PubAddr : @@
========================================
Signature (keccakhash(XX||YY||@@))
========================================
```

Figura 2.10: Ticket de acceso generado por el master de la zona virtual

- Tercera fase. El nodo *follower* realiza una transacción firmada con su clave privada. En esta transacción el nodo envía su *ticket* donde se verifica la autenticidad del mensaje con la clave pública del nodo y la validez del *ticket* con la clave pública del nodo *master*. Una vez validado el *ticket*, *blockchain* almacena el identificador del nodo junto con el identificador del grupo y la clave pública del nodo.
- Cuarta fase. En esta fase las transacciones realizadas por un nodo dentro de una zona virtual se verifican comprobando que la clave pública del nodo o usuario que realiza la transacción se corresponde con uno de los identificadores de nodo que tiene acceso a la zona virtual (identificado por el identificador de grupo).

Este modelo de acceso es ideal para un sistema que se componga de servicios o "usuarios" que se comunican todos entre sí. Pero este modelo no es apropiado en el uso de sistemas de videovigilancia, debido a que los servicios no siempre tendrán acceso a todas las cámaras y datos recopilados, sino

que se considera que el control de acceso al recurso debe ser evaluado en función de los derechos de acceso del solicitante.

Por otro lado, en [27], [29], [30] y [31] se ha encontrado otro modelo de control de acceso descentralizado basado en capacidades (*BlendCAC*) que en comparación con el resto de propuestas proporciona una solución escalable, ligera y de grano fino.

En este sistema se definen previamente los propietarios para cada dominio, el cuál es el encargado de controlar de forma local el acceso a los recursos de su dominio mediante una estrategia de gestión de tokens de capacidad basado en identidad. El funcionamiento es el siguiente:

- Una entidad solicita el acceso a un recurso iniciando un proceso de registro con el propietario del dominio donde se encuentra localizado dicho recurso.
- En base a la información local y a unas normas de seguridad, el propietario puede aprobar o denegar el acceso a la entidad. En caso de aprobar el acceso, el propietario genera un token de capacidades que una vez validado se envía a la red de *blockchain*.

$$ICap = f(VID_s, VID_o, D, AR, C) \tag{2.1}$$

Los parámetros de los que se compone el token son los siguientes:

- f: Función *hash* de mapeo
- VID_s: La dirección virtual del solicitante de los servicios.
- VID_o: La dirección virtual del objeto que proporciona el servicio.
- D: Almacena el siguiente contenido:
  - Las direcciones de las entidades a las cuales se les delega el token.
  - Derechos o acciones permitidas sobre el recurso para cada entidad.
  - Profundidad de delegación.
- AR: Acciones permitidas sobre el recurso.
- C: Otra información relacionada con el contexto, como el tiempo o la localización de la cámara.

Posteriormente, el recurso o proveedor del servicio verifica a partir del token que la entidad que solicita el servicio dispone de los permisos necesarios, en caso contrario, se deniega el servicio.

A continuación, se muestra en la figura 2.11 los distintos escenarios posibles de este modelo.

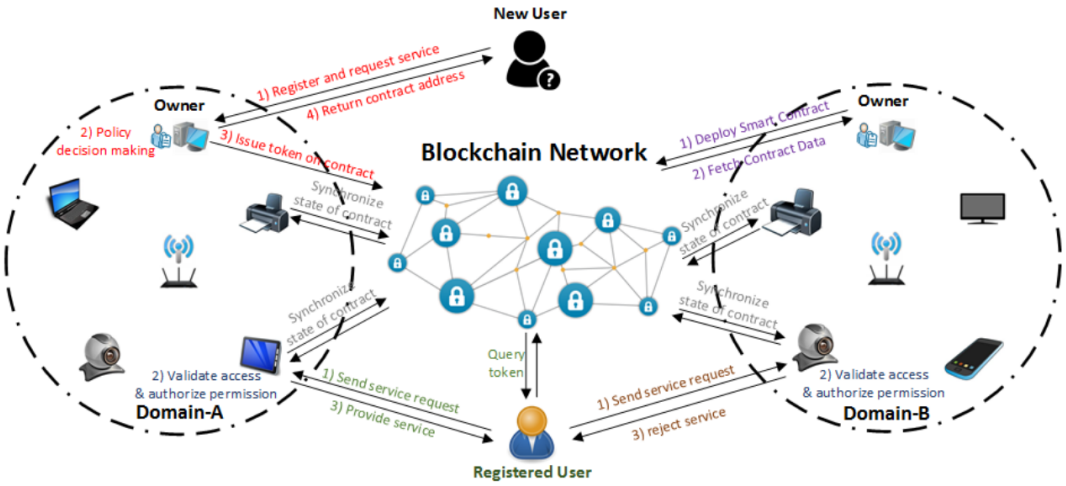

Figura 2.11: Modelo de control de acceso del sistema *BlendCAC*

Este modelo presenta los siguientes inconvenientes:

- Durante la fase de registro, el propietario del recurso permite el acceso al servicio en función de unas reglas o condiciones almacenadas en una base de datos, *Oracle*, por lo que se pueden producir las siguientes incidencias:
  - La base de datos se ve comprometida y por tanto, el token generado ya no es fiable.
  - Si el servidor de la base de datos no está disponible se deniega el registro.
- Todos los recursos que pertenecen a un determinado propietario deben tener acceso a la red de *blockchain* para la lectura del token generado. Sin embargo, algunos dispositivos debido a su limitación de recursos pueden verse excluidos de utilizar este modelo.
- Dificultad de la creación de una función *hash*, ya que actualmente el lenguaje por defecto, *solidity*, que se utiliza para la creación de los *smart contracts*, no soporta el parseo de datos almacenados en un archivo con extensión *json*, por lo que supone la creación de una librería que permita la escritura y lectura de este tipo de datos.

Finalmente, en otros estudios realizados, se han encontrado otras propuestas adicionales al control de acceso para proteger los datos o recursos del sistema. Concretamente, se mencionan dos propuestas en [32] y [33] para la protección de vídeo.

En [32] se propone un sistema de control automático que garantiza la integridad del vídeo. Este sistema está enfocado a la detección automática de colisiones en la carretera, de modo que cuando se detecta una colisión se lleva a cabo la grabación de la escena aproximadamente por unos 10 minutos. Este vídeo pasa por un algoritmo de procesamiento (*SHA256*) que calcula su *hash* y se envía a *blockchain* en forma de transacción para registrar la hora exacta de la colisión. De esta manera, se puede comprobar fácilmente la integridad del vídeo comparando el *hash* del vídeo original con el *hash* almacenado en el sistema.

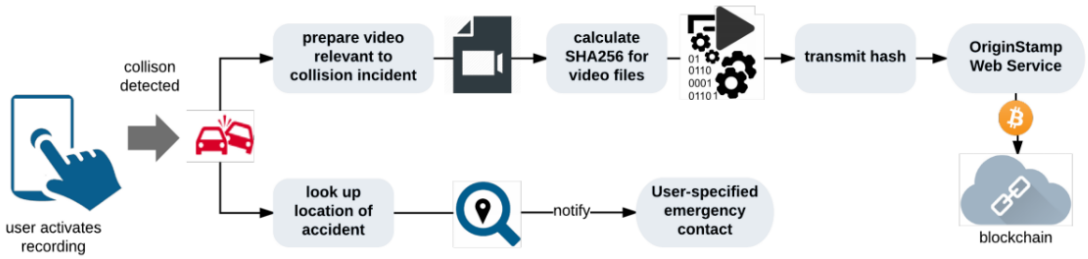

Figura 2.12: Sistema de detección automática de colisiones

En [33] se utiliza un sistema descentralizado, *Interplanetary File System* (*IPFS*) para el almacenaje seguro de ficheros pesados, tales como un vídeo. Donde se lleva a cabo un proceso de encriptación del fichero *AES* antes de ser almacenado. En este caso, el usuario que puede obtener la localización del fichero solicitándolo previamente al propietario, el cual podrá compartir una clave con el usuario para desencriptar la clave secreta (almacenada en la red de *blockchain*) que se utilizó para encriptar la localización del fichero.

2.5. Microservicios

2.5.1. Introducción

Se trata de un estilo de arquitectura empleado para el desarrollo de aplicaciones compuestas de múltiples servicios, donde cada uno de ellos se ejecuta en su propio proceso y emplea su propio mecanismo de comunicación.

En [34] se puede encontrar un modelo de diseño que se compone de 5 partes:

- **Servicios**. Son pequeños sistemas desacoplados que conforman el sistema completo.

- **Solución**. Se enfoca a dar solución al problema de forma global. Por tanto, debe tener presente las entradas y salidas de los servicios, la seguridad, enrutamiento y otras características con el fin de reducir la complejidad de los servicios.
- **Herramientas y procesos**. El comportamiento del sistema se ve afectado por las herramientas que se utilizan durante los procesos de desarrollo, despliegue y mantenimiento del producto.
- **Organización**. Considerado como uno de los puntos clave en el éxito de la implantación de una arquitectura basada en microservicios. Este componente hace referencia a los integrantes del equipo de trabajo y a la forma de trabajar.
- **Cultura**. También considerado importante, puesto que puede afectar a las decisiones de la organización.

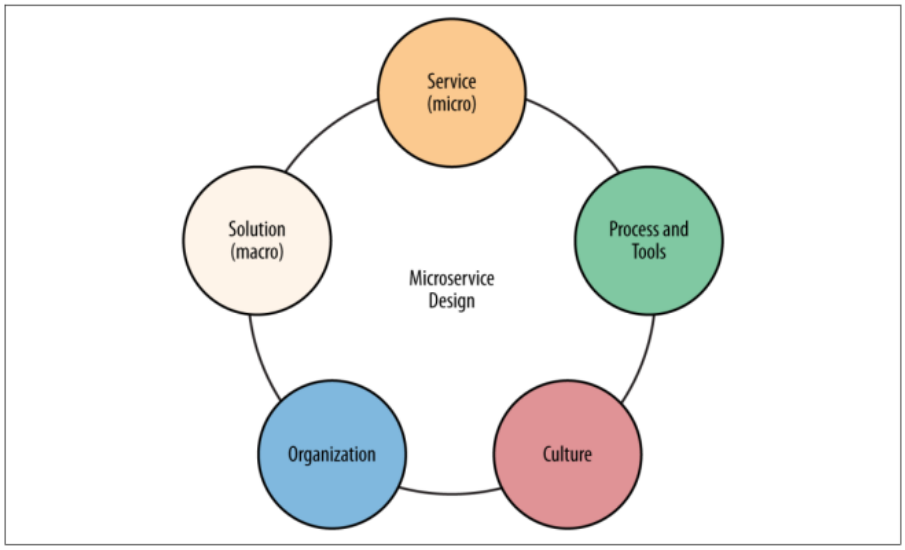

Figura 2.13: Modelo de diseño del sistema de microservicios

Lewis y Fowler [35] defienden que no todos los proyectos que utilizan microservicios comparten las mismas características, pero se pueden encontrar de forma recurrente algunas de ellas. En comparación con otros estilos arquitectónicos como el monolítico o como la arquitectura orientada a servicios, los servicios suelen tener un tamaño pequeño que generalmente vienen delimitados y agrupados por contexto. Los microservicios son generalmente autónomos durante todo su ciclo de vida, es decir, se utilizan procesos automáticos e integración continua desde su proceso de construcción hasta el despliegue de sus servicios. Además, una característica importante de esta arquitectura para el desarrollo de este proyecto es que se trata de una arquitectura descentralizada, ya que se compone de muchos servicios y, por consiguiente, el control del sistema no recae en una única parte central. En cuanto a comunicación, los servicios de este sistema se comunican entre sí mediante el uso de protocolos de comunicación *HTTP REST* o a través de mensajería de bus.

La organización que pretenda implantar este estilo arquitectónico en su negocio debe tener presente que los microservicios son apropiados para sistemas grandes donde cada servicio deberá estar enfocado a un determinado objetivo, para que puedan ser fácilmente reemplazables.

En la figura 2.14 se presenta un ejemplo de diseño de microservicios para una página web comercial que consta de 4 servicios donde el acceso a una instancia del servicio viene controlado por un módulo *load balancer*. También, como se puede apreciar, cada servicio es independiente del resto de servicios, es decir, están desacoplados y disponen de su propia base de datos para la gestión de información.

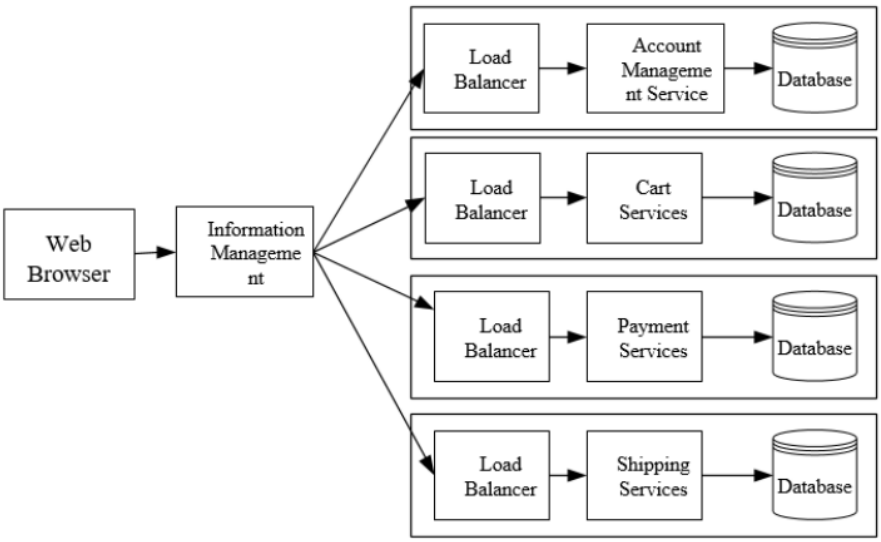

Figura 2.14: Arquitectura de microservicios de una web comercial

Las ventajas [34] que se pueden obtener de este tipo de arquitecturas son:

- Eficiencia en el lanzamiento de nuevos productos y funcionalidades.
- Capacidad de reutilización.
- Independencia en el despliegue, puesto que se integran rápidamente nuevas funcionalidades en el producto.
- Reducción de los costes de infraestructura debido a una mayor eficiencia en el software del sistema.
- El uso de herramientas adecuadas para una tarea específica acelera la introducción de la tecnología y aumenta las opciones de solución.
- Permite construir soluciones más complejas a partir de otras más simples.
- Los componentes de este tipo de sistema al ser reemplazables aumentan el tiempo de vida de un producto.
- Proporcionan mayor resistencia y disponibilidad.
- Mejor escalabilidad en el tiempo de ejecución que permite al sistema crecer o reducirse acorde con las necesidades del negocio.
- Mejora en la aplicabilidad de los procesos de testeo.

2.5.2. Estudios realizados

Las características previamente mencionadas hacen de la arquitectura de microservicios un candidato ideal en la construcción de sistemas flexibles que requieran sistemas de desarrollo y de mantenimiento más fáciles de aplicar (como el uso de procesos automáticos) y con un sistema descentralizado que permita añadir, modificar y eliminar servicios con el menor impacto en el sistema global.

Por todos estos beneficios muchos sistemas han optado por implantar esta arquitectura. Una de las tecnologías que nos podemos encontrar hoy en día y que apuestan por la arquitectura de microservicios es la tecnología *Uber* [36]. El funcionamiento general de este sistema consiste en la solicitud del servicio *Uber* a través de una aplicación para el transporte del usuario. En un principio se optó por la arquitectura monolítica porque era suficiente para el uso de esta aplicación al estar implantada únicamente en algunas ciudades, pero con el tiempo su uso se ha visto extendido por

todo el mundo. A medida que se introdujeron nuevos productos en el sistema, se experimentaron diversos problemas. Por ejemplo, la adición de nuevas funcionalidades y el aumento del desarrollo del trabajo. Por ello, en 2014 esta tecnología tomó la decisión de adoptar la arquitectura de microservicios que solventa muchos de los problemas que el negocio estaba experimentando.

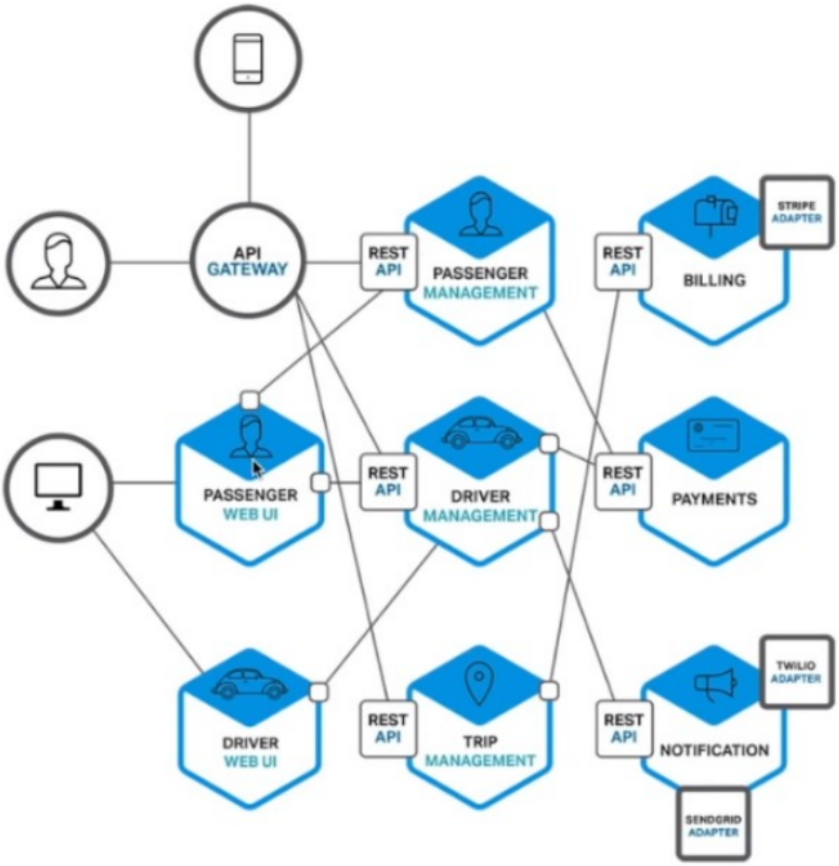

Figura 2.15: Arquitectura de microservicios de la tecnología *Uber*

Como se puede observar en la figura 2.15 la tecnología *Uber* se compone de muchos servicios que se comunican punto a punto utilizando el *API REST*. Algunos servicios como *billing* y *payments* pueden requerir una comunicación síncrona, es decir, el cliente espera una respuesta al realizar una petición al proveedor. Sin embargo, este tipo de diseño puede no ser el más apropiado para otros tipos de sistemas como el que se presenta en este proyecto, dado que un servicio de videovigilancia se podría componer de múltiples clientes y múltiples proveedores simultáneamente y, por tanto, se requiere procesar los datos de forma paralela.

Otros estudios realizados combinan la tecnología *blockchain* con microservicios con el objetivo de aumentar la seguridad en sistemas descentralizados.

En [37] se propone un sistema de videovigilancia basado en *blockchain* y en la arquitectura de microservicios. En este caso, los procesos de análisis pesados del vídeo son desacoplados en distintos servicios que pueden ser ejecutados en el mismo servidor o en distintos servidores. Cada servicio dispone de su propia base de datos con la información relevante al servicio. Los datos obtenidos del procesamiento del vídeo de cada servicio son recopilados a una base de datos *master* en el nivel *fog* del sistema. En este nivel se pueden encontrar más de una base de datos *master* en el caso de que el sistema disponga de más de un nodo *fog*. Las bases de datos de este nivel utilizan la tecnología *blockchain* para el control de acceso y la integridad del dato. Para garantizar la integridad del dato, se crean los bloques con el *hash* del resultado de análisis de vídeo que serán añadidos a la red de *blockchain* una vez sean validados.

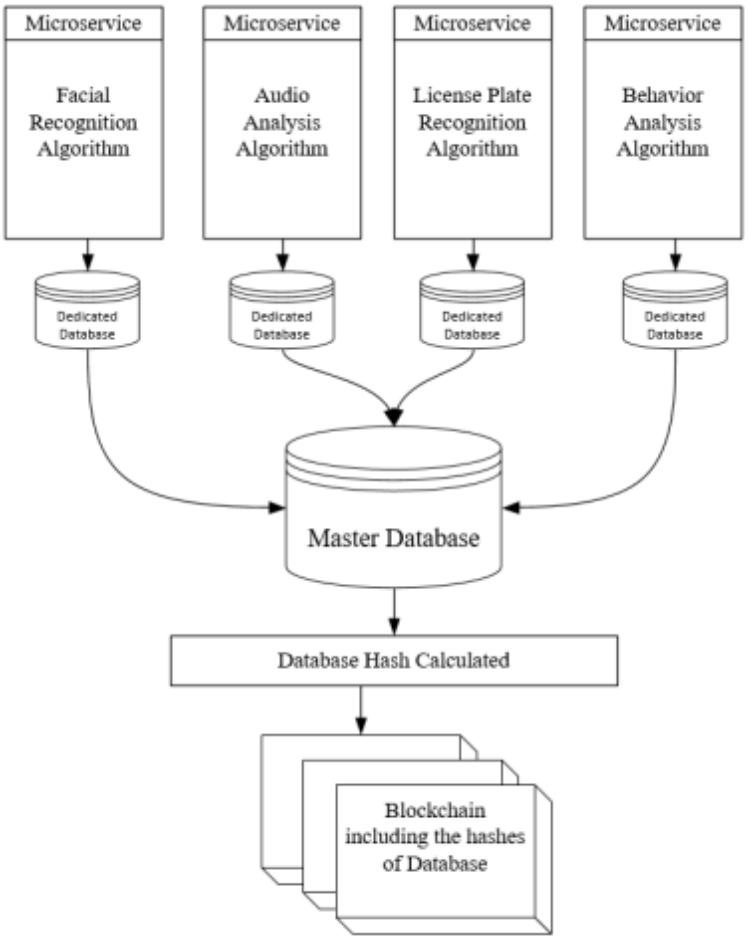

Figura 2.16: Propuesta de diseño de la arquitectura de microservicios para el procesamiento de vídeo

En este estudio se propone un diseño para el procesamiento de vídeo en tiempo real pero no se realiza su implementación. Más adelante, veremos otros estudios que se basan en este diseño.

Cabe destacar en este estudio que el diseño realizado es a alto nivel y, por tanto, no se tratan varios aspectos que se consideran relevantes para cualquier diseño básico de una arquitectura de microservicios. Algunos aspectos que se deberían tener en cuenta son:

- *Service discovery*. Para que un servicio pueda comunicarse con otro servicio debe conocer su dirección *IP* y el puerto. Tradicionalmente estos datos eran estáticos, por lo que solían estar embebidos en el código o almacenados en un fichero de configuración. Sin embargo, con la introducción de microservicios, estos datos son habitualmente dinámicos, ya que se pueden tener múltiples instancias de un mismo servicio. Por tanto, el sistema debe disponer de un servicio encargado de mantener un registro con ellos para que los servicios puedan ser localizables.
- *Load balancer*. Como se ha comentado en el punto anterior un servicio puede tener múltiples instancias ejecutándose en uno o varios servidores. Por tanto, el sistema debería disponer de un *load balancer* encargado de enrutar el tráfico en función de la disponibilidad (carga de trabajo) de las instancias de los servicios y de detectar la caída de un servicio para redirigir el tráfico a otro servicio disponible.
- Incluir o no un intermediario que se encargue de gestionar el *load balancer* y el *service discovery*. En el caso que no se incluya, cada servicio debe implementar la lógica necesaria para gestionar dichas funcionalidades.

Otro estudio, cuyo objetivo principal es asegurar los datos compartidos entre diferentes servicios,

lo podemos encontrar en [38] donde se propone una arquitectura de microservicios descentralizada (*BlendMAS*) *Blockchain-ENable Decentralized Microservices Architecture for SPS smart public safety*.

Este sistema consiste en 3 servicios:

- Servicios encargados de procesar el vídeo. Estos servicios se encargan principalmente de la extracción de características para su posterior análisis en servidores más potentes.
- Servicios que proporcionan una red de *blockchain* privada. Para ello, se basan en procesos de autenticación basados en la identidad. Por lo que un nuevo nodo que quiera ser registrado en la red debe solicitarlo al administrador del sistema, el cuál lleva a cabo procesos de identificación para permitir o denegar el permiso.
- Servicios de seguridad de la red *blockchain*. Se constituye principalmente de 2 tipos de servicios: *mining* y servicios de políticas de seguridad.
    - Los servicios *mining* son responsables de verificar las transacciones y de generar nuevos bloques en la red.
    - Servicios encargados de gestionar la seguridad del sistema:
        - Control de acceso al sistema
        - Servicio de gestión
        - Servicio de registro
        - Servicios de autenticación

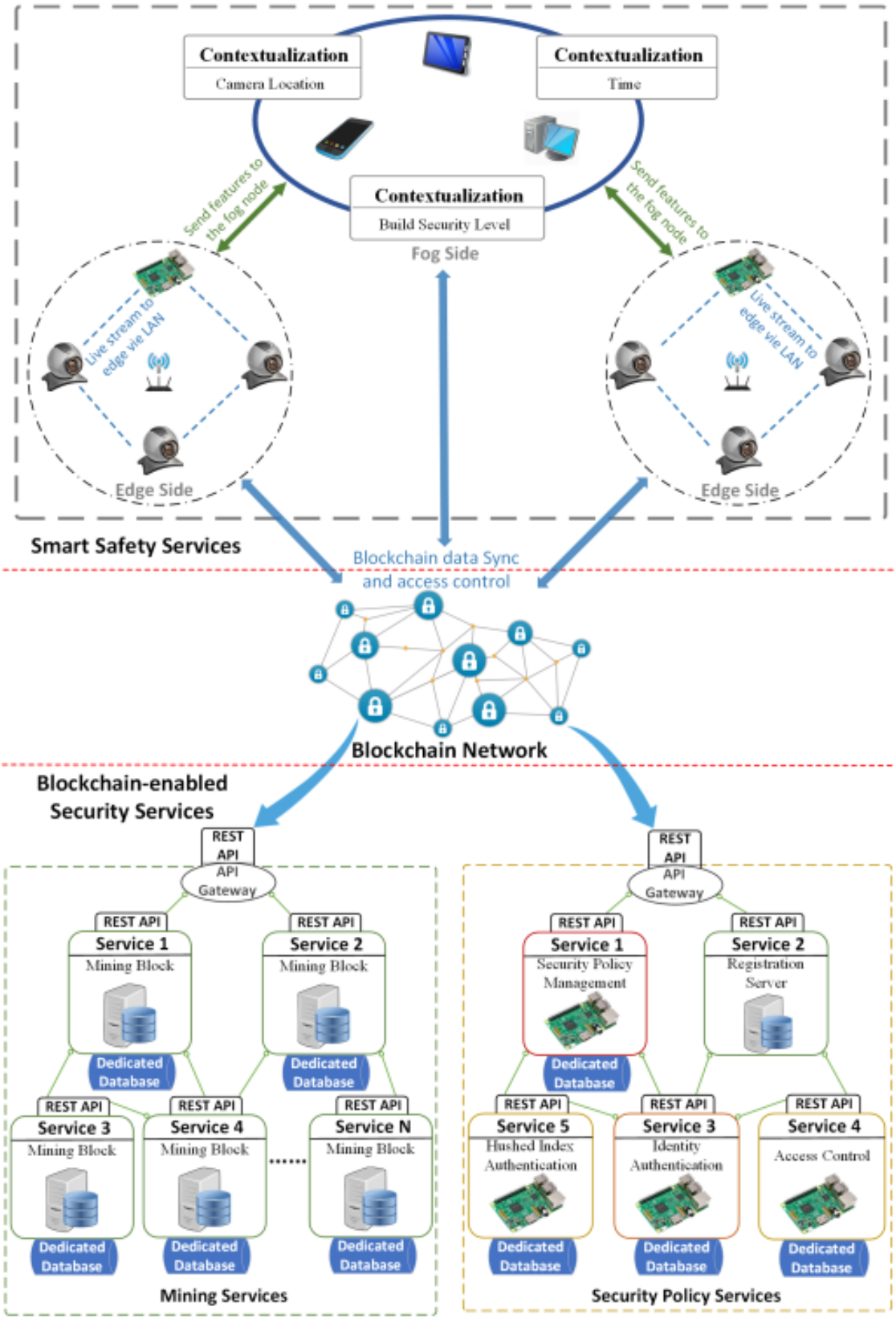

Figura 2.17: Diseño de la arquitectura *BlendMAS*

De forma similar en [39] se propone un sistema de seguridad pública de *IoT* ligera basado en una arquitectura de microservicios, *LIPS* (*Lightweight IoT based Smart Public Safety*), para servicios de videovigilancia. La diferencia principal con respecto al anterior diseño es que este sistema consta de 4 servicios, donde 3 de ellos siguen la misma línea de [38]. El cuarto servicio adicional se trata de servicios desplegados en el *fog* que realizan tareas más exhaustivas como la extracción de características a alto nivel o la toma de decisiones.

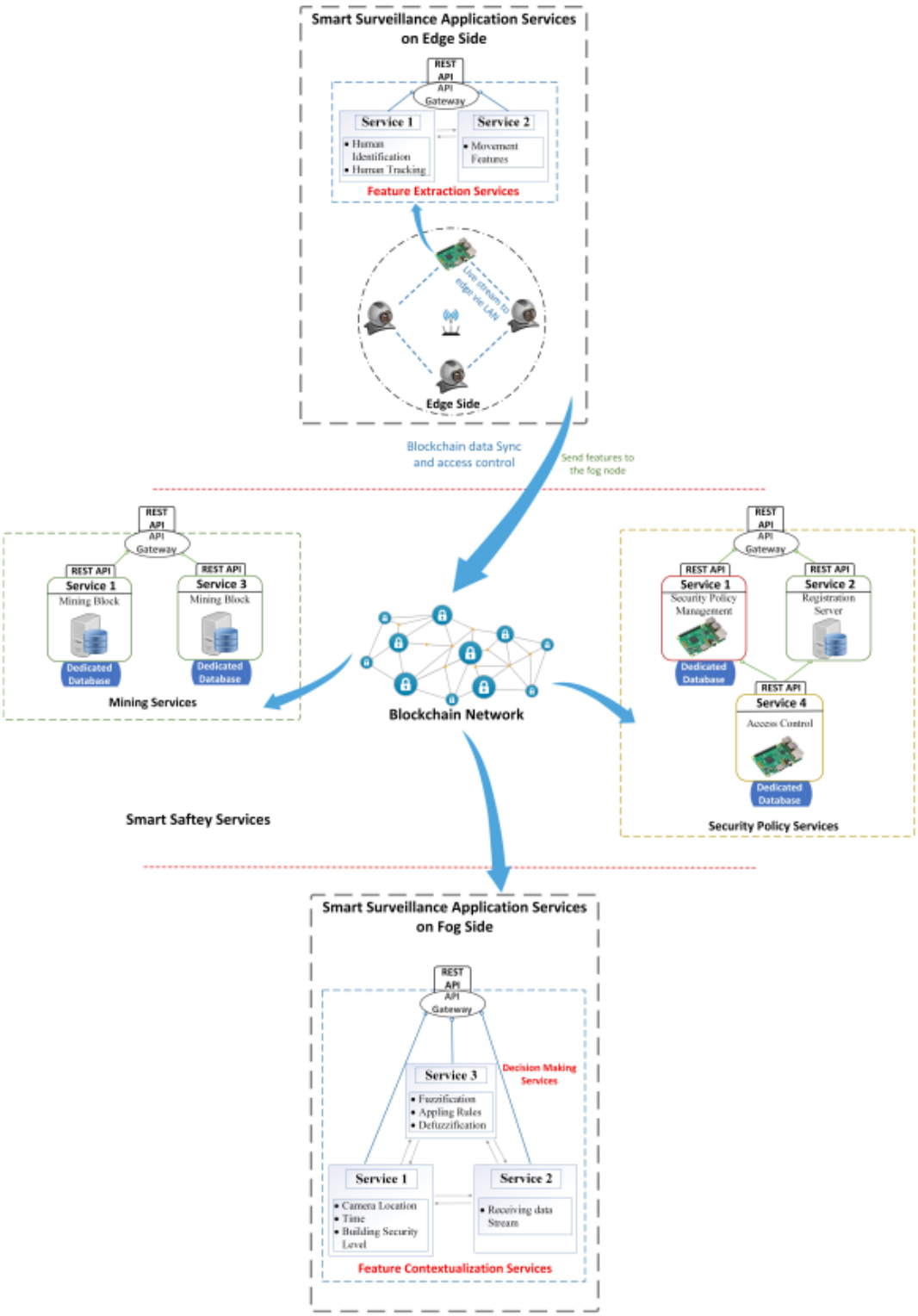

Figura 2.18: Diseño de la arquitectura LISPS

Las principales desventajas que se pueden observar de este tipo de diseño es que todos los servicios del sistema se comunican directamente sin ningún intermediario o *broker* que medie entre los distintos receptores y emisores. Algunas principales desventajas que se encuentran en este tipo diseño son:

- Los servicios del sistema necesitan conocer las localizaciones de los servicios con los cuales se quiere comunicar.
- Se reduce la disponibilidad del sistema, ya que el servicio emisor y el servicio receptor deben estar disponibles durante el proceso de comunicación.
- Aumento de la complejidad de la lógica de cada servicio, puesto que deben implementar ciertos mecanismos adicionales, por ejemplo, el reenvío de mensaje o la confirmación de la entrega de un mensaje.

Otro punto a comentar es el uso del *API REST* para la comunicación de los servicios. La principal desventaja de este estilo de comunicación es que soporta únicamente la comunicación "petición - respuesta". Por tanto, no se considera *REST* como un estilo de comunicación a utilizarse en todos los servicios presentes en este sistema de videovigilancia principalmente por 2 motivos:

- La transmisión de vídeo en tiempo real es imprescindible reducir la latencia, y un estilo de comunicación basado en petición-respuesta aumenta la latencia durante el proceso de comunicación.
- Se utiliza principalmente en una comunicación punto a punto. Sin embargo, en el sistema propuesto están presente varios dispositivos *IoT* que se comportan como clientes y varios consumidores. Por tanto, existen otros diseños más apropiados para su comunicación.

2.6. Motivaciones

De los anteriores antecedentes observamos que se utilizan diferentes arquitecturas para el desarrollo de un sistema de servicio de videovigilancia en tiempo real. Estas arquitecturas pueden ser más óptimas que otras en función del escenario al que se aplican. En el desarrollo de este proyecto se busca la implementación de una arquitectura que tenga presente varios aspectos que muy frecuentemente se pueden encontrar en una *smart city*:

- Incremento del número de dispositivos *IoT*. Al verse incrementado el número de estos dispositivos, experimentamos un aumento considerable de la información a procesar. Si además tenemos presente que muchos de los procesos en tiempo real, como los servicios de monitorización, son computacionalmente muy costosos en cuanto a recursos y tiempo de procesamiento, modelos como *cloud computing* pueden resultar no suficientes para procesar adecuadamente toda la información, produciéndose además problemas de latencia y limitación de ancho de banda. Por ejemplo, en [4] vemos que una cámara de videovigilancia con una resolución de un millón de píxeles produce aproximadamente en tiempo real 10 GB de datos por minuto. Por tanto, en un modelo *cloud computing* se ve comprometida la capacidad de respuesta al tener que procesar todos estos datos en tiempo real, y más aún si se incrementa el número de cámaras.

  Los problemas de latencia (crítico en un sistema de videovigilancia en tiempo real) y limitación de ancho de banda se podrían solucionar empleando modelos como *edge computing* o *fog computing*, ya que el procesamiento de vídeo se realiza cerca del dispositivo fuente.
  Sin embargo, con estos modelos tenemos un problema mayor que el modelo *cloud computing*, la capacidad de rendimiento. Ello es debido a que estos modelos, por norma general, disponen de menos recursos que la nube. Por lo cual, es conveniente el uso de un modelo con varios niveles o nodos (*cloud-fog-edge*) donde:

  - En el primer nivel o nodo *edge* se realicen procesos críticos en cuanto a latencia y ancho de banda.

  - En el segundo nivel o nodo *fog* se lleven a cabo procesos que requieran mayor capacidad de procesamiento, debido a las limitaciones de los dispositivos, procesos relacionados con temas de privacidad o ciertas tareas de decisión que requieran un tiempo de respuesta menor que del tercer nivel.
  - Finalmente, en el tercer nivel se lleven a cabo tareas de decisión y otros procesos con mayor coste computacional.

- Posibilidad de reemplazar o aumentar el número de funcionalidades del diseño. En servicios de videovigilancia, para la detección y monitorización de un objeto, normalmente se utilizan arquitecturas monolíticas, ya que no se espera realizar importantes modificaciones en este tipo de arquitecturas. Sin embargo, en soluciones de una *smart city* es necesario disponer de arquitecturas que sean más escalables, como microservicios [37]. Es necesario disponer de diseños arquitectónicos que permitan reemplazar funcionalidades o que permitan incorporar otras nuevas fácilmente, sobre todo ahora que con los nuevos avances tecnológicos van saliendo al mercado soluciones más eficientes y con mejores resultados.

- Con la aparición de los dispositivos *IoT* en sistemas de videovigilancia, la seguridad y la privacidad de estos sistemas se ve comprometida. Si además tenemos presente que la adopción de modelos centralizados como el paradigma de *cloud computing* pueden suponer un punto único de fallo, es necesario la adopción de medidas que permitan controlar el acceso al sistema y mantener la integridad del vídeo. De esta forma se evita que usuarios externos al sistema puedan acceder al mismo u otros datos recopilados o que sean manipulados de forma intencionada.

Capítulo 3

Diseño del proyecto

En este capítulo del proyecto se llevará a cabo a nivel alto la descripción, el análisis, el diseño funcional de la solución, y para finalizar se describirá el diseño técnico de la implementación del prototipo realizado para demostrar la funcionalidad del sistema propuesto.

3.1. Descripción

En este proyecto se propone una solución que combina las tecnologías *blockchain* y microservicios [37] para procesar en tiempo real el vídeo capturado por cámaras IoT, con el objetivo de solucionar los problemas recurrentes en un sistema de servicio de videovigilancia "inteligente" (apartado 2.2).

A continuación, se realiza la descripción general de la solución propuesta que solventa los problemas planteados en el apartado 2.6.

La figura 3.1 muestra una visión global del sistema de videovigilancia "inteligente" que se plantea en este proyecto. Como se puede observar, en este diseño existen varios grupos de dispositivos cámara, localizados en distintos lugares físicamente, que capturan vídeos que serán procesados en las capas *edge*, *fog* y *cloud*, donde los datos a procesar en cada capa provienen de la capa inferior.

En este sistema se distinguen 4 capas que se basan en el modelo 2.3:

- **Capa de dispositivos *IoT***. Formado por los dispositivos cámara. Estos dispositivos se encargan de capturar el vídeo desde el lugar de localización de cada cámara. El acceso al vídeo viene protegido por un sistema de control de acceso basado en *blockchain* que se ejecuta en el *backend* del sistema propuesto.
- **Capa *edge***. Se encarga de procesar el vídeo generado por las cámaras. Con el propósito de realizar distintos tipos de procesamiento al vídeo capturado por los dispositivos cámara, se propone el uso del estilo arquitectónico de microservicios. Estos servicios generarán unos datos de salida que se transmitirán a la capa superior *fog* (únicamente se enviarán a nodos con derecho de acceso los datos generados por el nodo en la capa inferior) para llevar a cabo procesos más pesados (que no pueden ser ejecutados en un nodo *edge*) u otros procesos críticos. El control de acceso de los servicios en esta capa utiliza una combinación de la tecnología *blockchain* junto con una técnica de grafos.
- **Capa *fog***. Esta capa puede estar formada por otras capas de tipo *fog*. Principalmente se encarga de procesar los datos generados por servicios localizados en capas inferiores. Al igual que la capa *edge*, el mecanismo de control de acceso de esta capa utiliza las tecnologías de microservicios y *blockchain*, donde los servicios utilizan el mismo sistema de control de acceso empleado en la capa *edge*.
  Posteriormente, los datos generados serán transmitidos a capas superiores que pueden ser otras de tipo *fog*, otros nodos pertenecientes a la misma capa *fog* o, finalmente, la capa *cloud*.
- **Capa *cloud***. Lleva a cabo procesos que no pueden realizarse en las capas inferiores debido al alto coste computacional de dichos procesos. En esta capa también se utilizan las tecnologías de *blockchain* y microservicios con la finalidad de controlar y realizar diversos tipos de

procesamiento en los datos obtenidos en capas inferiores.
Finalmente, los datos generados en esta capa pueden ser enviados al usuario final. Para acceder a estos datos se requieren técnicas habituales (fuera del ámbito de este proyecto) que sean viables y alcanzables por los usuarios finales.

Cabe resaltar que en el caso de servicios críticos, donde la latencia es un factor importante, se ejecutan en la capa *fog* o en la capa *edge* con la finalidad de transmitir la generación de dichos datos a capas superiores o a usuarios finales como se muestra en la figura 3.1.

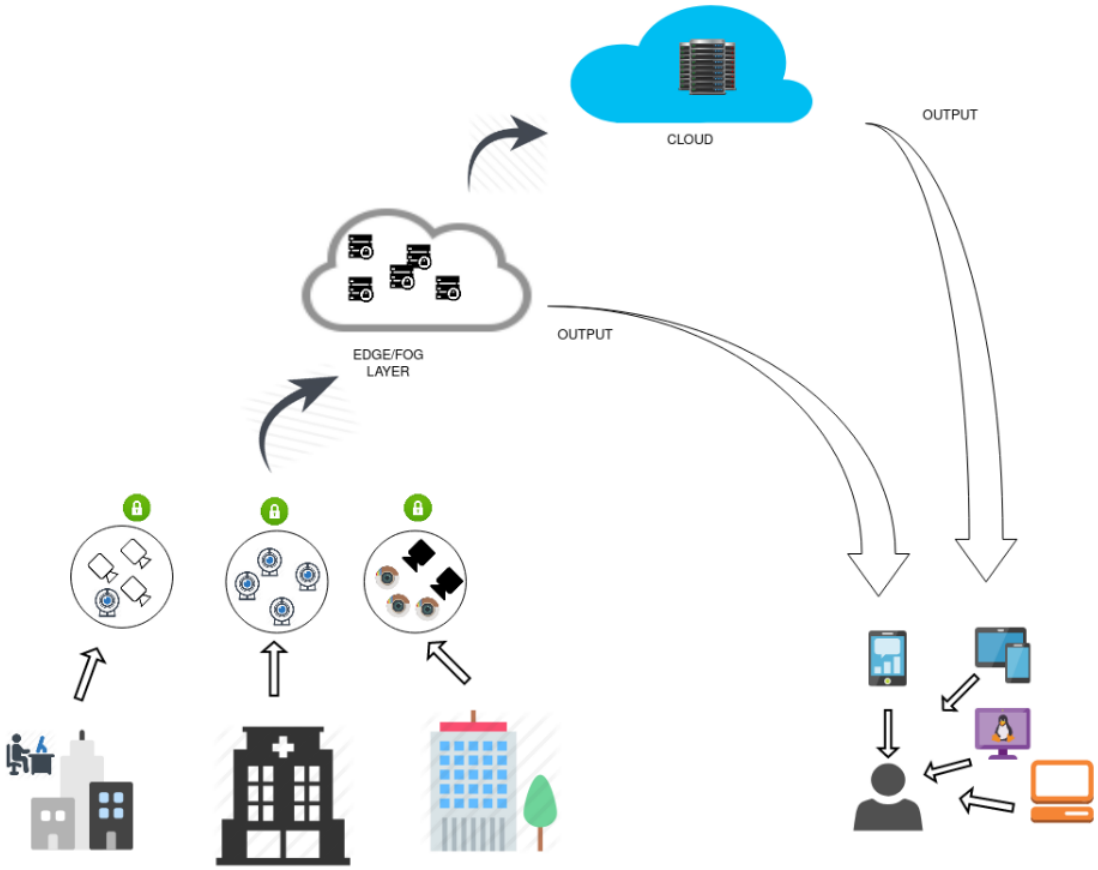

Figura 3.1: Visión global del sistema de videovigilancia

3.2. Análisis

En esta fase se realiza un análisis de los requerimientos funcionales y no funcionales del sistema a nivel alto.

3.2.1. Requerimientos funcionales

El objetivo de este apartado es la definición de los requerimientos funcionales que definen el alcance del sistema a nivel funcional.

A continuación se muestra un listado de los requerimientos funcionales y las tablas donde se especifica cada requerimiento.

- **RF01**: Alta dispositivo cámara.
- **RF02**: Baja dispositivo cámara.
- **RF03**: Alta servicio.

- **RF04**: Baja servicio.
- **RF05**: Alta nodo.
- **RF06**: Baja nodo.
- **RF07**: Alta usuario.
- **RF08**: Baja usuario.
- **RF09**: Captura del vídeo.
- **RF10**: Procesamiento del vídeo.
- **RF11**: Procesamiento de datos extraídos.
- **RF12**: Gestión de eventos.
- **RF13**: Visualización del vídeo.
- **RF14**: Control de acceso al vídeo.
- **RF15**: Control de acceso de datos.
- **RF16**: Almacenamiento de vídeo.
- **RF17**: Almacenamiento de datos.
- **RF18**: Gestión de servicios.

Cada requerimiento contiene un identificador, nombre, descripción y una secuencia del flujo normal de ejecución.

RF01	
Nombre	Alta dispositivo cámara
Descripción	Registro de la cámara en el sistema
Secuencia	▪ Creación de una cuenta digital[1]en *blockchain* ▪ Asignación de un identificador "id" único en el sistema ▪ Registro del identificador "id" de cada cámara en su correspondiente nodo *edge*

Tabla 3.1: Requerimiento funcional RF01

RF02	
Nombre	Baja dispositivo cámara
Descripción	Anulación del registro de la cámara en el sistema
Secuencia	▪ Eliminar la cuenta digital[1] en *blockchain* ▪ Eliminar el identificador "id" de la cámara del nodo

Tabla 3.2: Requerimiento funcional RF02

[1]Generación de un par de claves basada en un sistema criptográfico. La clave pública se utiliza para la identificación del usuario. La clave privada es secreta y se emplea para autentificación y encriptación

RF03	
Nombre	Alta servicio
Descripción	Registro del servicio en el sistema
Secuencia	▪ Instalación del servicio en un nodo *edge*, *fog* o *cloud* ▪ Creación de una cuenta digital[1] en *blockchain* ▪ Registro de la dirección virtual[2]del servicio en su nodo correspondiente

Tabla 3.3: Requerimiento funcional RF03

RF04	
Nombre	Baja servicio
Descripción	Anulación del registro del servicio en el sistema
Secuencia	▪ Desinstalar el servicio del nodo ▪ Eliminar la cuenta digital ▪ Eliminar el registro de la dirección virtual[2] del nodo correspondiente

Tabla 3.4: Requerimiento funcional RF04

RF05	
Nombre	Alta nodo
Descripción	Registro del nodo *edge*, *fog* o *cloud* en el sistema
Secuencia	▪ Crear una cuenta digital en *blockchain* ▪ Asignar un identificador "id" único al nodo ▪ Insertar el identificador "id" del nodo en el grafo de nodos

Tabla 3.5: Requerimiento funcional RF05

[2]Representación de la clave pública que permite operar en *blockchain*

RF06	
Nombre	Baja nodo
Descripción	Elimina el registro del nodo en el sistema
Secuencia	▪ Eliminar la cuenta digital ▪ Eliminar el identificador "id" del nodo del grafo de nodos

Tabla 3.6: Requerimiento funcional RF06

RF07	
Nombre	Alta usuario
Descripción	Registro del usuario en el sistema
Secuencia	▪ Asignar un usuario y contraseña

Tabla 3.7: Requerimiento funcional RF07

RF08	
Nombre	Baja usuario
Descripción	Elimina el registro del usuario en el sistema
Secuencia	▪ Eliminar el usuario del sistema

Tabla 3.8: Requerimiento funcional RF08

RF09	
Nombre	Captura del vídeo
Descripción	El sistema debe permitir la captura de vídeo de una o varias cámaras en tiempo real
Secuencia	▪ Configuración de parámetros de envío ▪ Envío de vídeo a otros servicios a través de un sistema de mensajería

Tabla 3.9: Requerimiento funcional RF09

RF10	
Nombre	Procesamiento de vídeo
Descripción	El sistema debe permitir realizar diferentes tipos de procesamiento de vídeo
Secuencia	▪ Captura del vídeo ▪ Envío del vídeo a otros servicios a través de un sistema de mensajería ▪ Extracción de características u otros datos ▪ Envío de los datos obtenidos a otros servicios a través del sistema de mensajería

Tabla 3.10: Requerimiento funcional RF10

RF11	
Nombre	Procesamiento de datos extraídos
Descripción	El sistema debe permitir realizar diferentes tipos de procesamiento o análisis a los datos extraídos del vídeo
Secuencia	▪ Recepción de datos ▪ Análisis y almacenamiento de los datos en una base de datos ▪ Envío de los datos obtenidos a otros servicios o usuarios a través del sistema de mensajería

Tabla 3.11: Requerimiento funcional RF11

RF12	
Nombre	Gestión de eventos
Descripción	Gestión de eventos de control de acceso
Secuencia	▪ Asignación de un cliente *listener* al evento registrado en *blockchain* ▪ Lanzamiento el evento al cumplirse ciertas condiciones ▪ Recepción del evento en el cliente *listener* ▪ Gestión del evento

Tabla 3.12: Requerimiento funcional RF12

RF13	
Nombre	Visualización del vídeo
Descripción	El sistema debe permitir al usuario visualizar el vídeo en tiempo real de una o varias cámaras
Secuencia	▪ El usuario inicia sesión ▪ Selección de las cámaras de las que se quiere recibir vídeo ▪ Visualización del vídeo

Tabla 3.13: Requerimiento funcional RF13

RF14	
Nombre	Control de acceso al vídeo
Descripción	Control del acceso al vídeo capturado por una o varias cámaras
Secuencia	▪ Registro del servicio en el nodo "propietario" (nodo donde la cámara está registrada) de la cámara ▪ Solicitud de acceso al nodo "propietario" ▪ Generación de un token *ticket* ▪ Lanza evento "AllowAccess" al cumplirse los requisitos del contrato (*smart contract*) ▪ Envío de los datos de conexión

Tabla 3.14: Requerimiento funcional RF14

RF15	
Nombre	Control de acceso de datos
Descripción	Control del acceso a los datos generados por un servicio
Secuencia	▪ Registro del servicio en su correspondiente nodo ▪ Solicitud de acceso al servicio ▪ Lanza evento "AllowAccess" al cumplirse los requisitos del contrato (*smart contract*) ▪ Comprobación que el nodo del servicio proveedor y el nodo del servicio solicitante están conectados en el grafo ▪ Envío de los datos de conexión

Tabla 3.15: Requerimiento funcional RF15

RF16	
Nombre	Almacenamiento de vídeo
Descripción	Almacena el vídeo por un tiempo determinado
Secuencia	▪ Captura de vídeo ▪ Especificación del tiempo de almacenamiento ▪ Almacenamiento del vídeo

Tabla 3.16: Requerimiento funcional RF16

RF17	
Nombre	Almacenamiento de datos
Descripción	Almacena los datos extraídos o procesados
Secuencia	▪ Captura de vídeo ▪ Procesamiento de vídeo ▪ Extracción de datos ▪ Recepción de los datos ▪ Almacenamiento de datos

Tabla 3.17: Requerimiento funcional RF17

RF18	
Nombre	Gestión de servicios
Descripción	Gestión de la comunicación entre los distintos servicios del sistema
Secuencia	▪ Registro del servicio en un servicio de registro ▪ Distribuir el envío de mensajes de forma balanceada

Tabla 3.18: Requerimiento funcional RF18

3.2.2. Requerimientos no funcionales

En contraste con los requerimientos funcionales, los requerimientos no funcionales nos sirven para representar características generales y de restricción del sistema.
Los requerimientos no funcionales que se han tenido en cuenta son:

- **RNF01**: Escalabilidad.
- **RNF02**: Flexibilidad.
- **RNF03**: Disponibilidad.
- **RNF04**: Seguridad y privacidad.

- **RNF05**: Tiempo de respuesta.
- **RNF06**: Rendimiento.
- **RNF07**: Cohesión.
- **RNF08**: Acoplamiento.
- **RNF09**: Recuperabilidad y confiabilidad.
- **RNF10**: Mantenimiento.

A continuación se detallan los requerimientos no funcionales compuestos de identificador, nombre, descripción y cláusula.

RNF01	
Nombre	Escalabilidad
Descripción	Capacidad del sistema para gestionar la carga de trabajo al aumentar sus recursos
Cláusula	Escalabilidad horizontal: debe permitir incluir más dispositivos y servicios

Tabla 3.19: Requerimiento no funcional RNF01

RNF02	
Nombre	Flexibilidad
Descripción	Capacidad de adaptación del sistema ante cambios funcionales y no funcionales
Cláusula	Debe permitir realizar los siguientes cambios con el menor impacto: ▪ Modificación de funcionalidad ▪ Modificación de estructura ▪ Migración a otros lenguajes y base de datos

Tabla 3.20: Requerimiento no funcional RNF02

RNF03	
Nombre	Disponibilidad
Descripción	Mide la frecuencia con el que el sistema proporciona la funcionalidad solicitada por un usuario
Cláusula	La recepción de vídeo en tiempo real, así como el resultado de los análisis de los vídeos, deben estar disponibles

Tabla 3.21: Requerimiento no funcional RNF03

RNF04	
Nombre	Seguridad y privacidad
Descripción	Protección del sistema ante un uso malintencionado y control del acceso
Cláusula	Debe cumplir los siguientes requisitos: ▪ Autenticación. Identificación del usuario mediante el uso de firma digital ▪ Autorización. Controla si el usuario tiene derecho a realizar una determinada acción ▪ Comunicación segura ▪ Mantener la integridad dato

Tabla 3.22: Requerimiento no funcional RNF04

RNF05	
Nombre	Tiempo de respuesta
Descripción	Tiempo medio desde que un suceso ocurre hasta su detección
Cláusula	Las incidencias deben notificarse

Tabla 3.23: Requerimiento no funcional RNF05

RNF06	
Nombre	Rendimiento
Descripción	Tiempo de respuesta ante una petición. En el sistema existirán 2 tipos de tareas clasificadas por prioridad: ▪ Tareas primarias. Ejemplo: transmisión de vídeo o envío de notificación ▪ Tareas secundarias. Ejemplo: análisis e creación de historial
Cláusula	Reducción del tiempo de respuesta para las tareas prioritarias

Tabla 3.24: Requerimiento no funcional RNF06

RNF07	
Nombre	Cohesión
Descripción	Mide el grado de relación de los elementos de un módulo
Cláusula	Se debe conseguir alta cohesión entre los diferentes componentes del sistema

Tabla 3.25: Requerimiento no funcional RNF07

RNF08	
Nombre	Acoplamiento
Descripción	Mide el grado de dependencia entre los distintos componentes
Cláusula	Se debe conseguir bajo acoplamiento entre los diferentes componentes del sistema

Tabla 3.26: Requerimiento no funcional RNF08

RNF09	
Nombre	Recuperabilidad y confiabilidad
Descripción	Capacidad de recuperación ante un fallo producido en el sistema
Cláusula	El análisis y el acceso al vídeo deben estar siempre disponibles. En el caso que se produzca un fallo del sistema, se debe continuar desde el momento en que se produjo el fallo

Tabla 3.27: Requerimiento no funcional RNF09

RNF10	
Nombre	Mantenimiento
Descripción	Mantenimiento del software
Cláusula	Posibilidad de emplear procesos de automatización en las diferentes fases de una mejora o corrección de defectos en el sistema

Tabla 3.28: Requerimiento no funcional RNF10

3.3. Diseño funcional

3.3.1. Introducción

En este apartado se realiza el diseño funcional del sistema de videovigilancia "inteligente".
En primer lugar, se mostrarán los modelos del sistema diseñado centrándonos en las tecnologías de microservicios y *blockchain*.
En segundo lugar, se presentan los diagramas de caso de uso del sistema.
Por último, se lleva a cabo el análisis de riesgos del sistema propuesto.

3.3.2. Paradigma de computación *fog-edge-cloud*

Se detalla el modelo de computación elegido para el sistema de videovigilancia compuesto por dispositivos *IoT*.

A diferencia de los estudios realizados en el capítulo 2, se propone un modelo de 4 capas basado en el paradigma de computación *fog-edge-cloud* (apartado 2.3).
Los modelos previamente vistos consideran a un dispositivo *IoT* como un nodo *edge*, ya que proponen el despliegue del servicio en el mismo dispositivo con el objetivo de la reducción de la latencia. Sin embargo, hay que tener en cuenta que muchos de estos dispositivos no disponen de los recursos necesarios para la instalación de programas pesados tales como el reconocimiento facial o detección de objetos que emplean técnicas de *machine learning*. Por consiguiente, en este diseño se añade una capa más que contendrá los dispositivos *IoT*, desplegando el servicio lo más cerca posible del dispositivo que captura el vídeo con la finalidad de reducir la latencia.
En resumen, los motivos que se tienen en cuenta para la adopción de un modelo de 4 capas (apartado 3.2) son:

- En el caso de disponer de una gran cantidad de dispositivos de cámara no sería rentable el despliegue de los servicios.
- Algunos tipos de servicios no pueden ser instalados en dispositivos con recursos limitados debido a su coste computacional.
- El mantenimiento del dispositivo se hace más complejo en el caso de que se modifiquen o añadan servicios adicionales.

Por consiguiente, se decide utilizar un nodo *edge* (cercano físicamente) para la gestión de los dispositivos cámara.
Con dicho modelo se solventan dos problemas:

- **Escalabilidad** [*RNF01*]. En el supuesto de que se requiera introducir nuevos servicios o se aumente el coste computacional de la lógica instalada en el nodo principal (nodo *edge*), siempre se puede añadir otro nodo *edge* secundario.
- **Coste computacional** [*RNF06*]. Dependiendo de la complejidad de la lógica que se encarga de la gestión de microservicios, balanceo o disponibilidad, la instalación se puede realizar en uno o varios nodos *edge* con el objetivo de mejorar el rendimiento del sistema.

Por otro lado, las capas *fog* y *cloud* siguen la misma línea que las capas del modelo computacional *edge-fog-cloud* visto en el capítulo 2. Cabe resaltar que la capa *fog* puede estar compuesta, a su vez, por otras capas *fog*, donde varios nodos *fog* pueden estar conectados a un mismo nodo *edge* con el propósito de mejorar el rendimiento del sistema y reducir el tiempo de respuesta [*RNF05*].

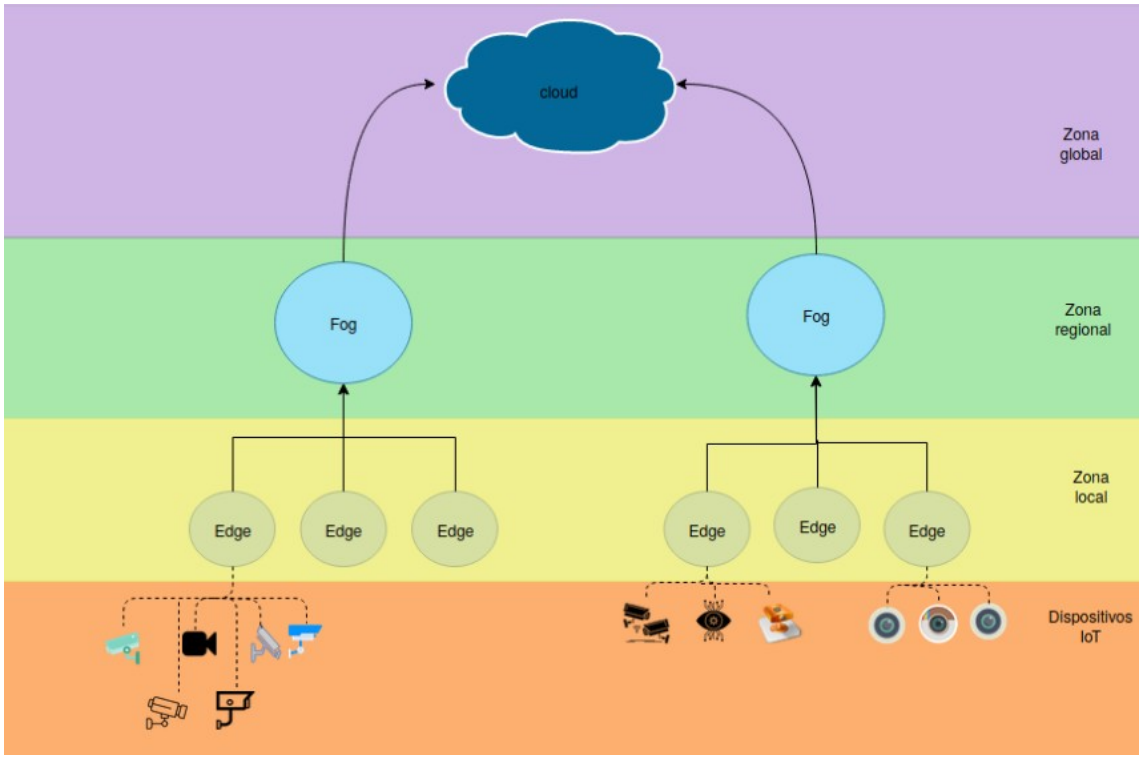

Figura 3.2: Modelo de computación de los nodos *edge*, *fog* y *cloud*

3.3.3. Arquitectura del sistema

El sistema de videovigilancia "inteligente" propuesto puede disponer de una o varias cámaras que envían vídeo de forma simultánea a uno o varios servicios para su correspondiente análisis.
A su vez, estos servicios envían los datos extraídos a otros servicios que se encuentran en la capa *fog* (relativamente cercanos a la fuente) para llevar a cabo procesos que requieran un coste

computacional mayor o procesos que realicen tareas críticas, donde el tiempo de respuesta sea importante. Por ejemplo, el envío de notificaciones en el caso de detección de un peligro.
Finalmente, se envía la información recopilada en la capa *fog* a la nube para otros procesos que no puedan realizarse en las capas *edge* o *fog*. Estos procesos pueden incluir procesos de decisión, datos históricos o algoritmos de tuneo.

Dado que el sistema propuesto está compuesto por varios clientes (*publisher*) que envían información a los servicios suscritos (*subscriber*), se ha elegido la arquitectura descrita en el apartado 3.4, basada en el patrón de diseño *publisher-subscriber* (ver figura 3.3). Como se puede ver, la gestión de las comunicaciones de este patrón se realiza a través de un sistema de mensajería que puede estar compuesto por uno o varios *brokers*. Como se verá más adelante, se pueden encontrar sistemas de mensajería que pueden o no utilizar un *broker*. En este diseño se opta por el uso de un sistema de mensajería basado en un *broker*.

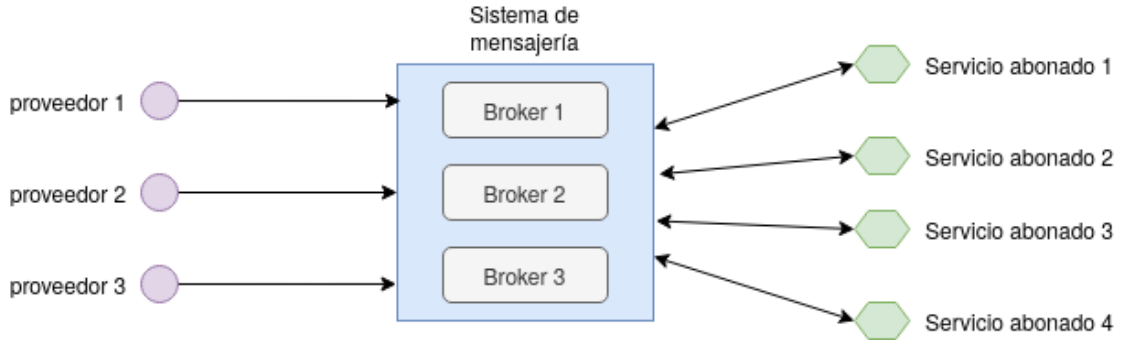

Figura 3.3: Patrón de diseño *publisher/subscriber*

En el diseño de la arquitectura del apartado 3.4 se pueden observar los siguientes componentes:

- **Cámaras *IoT***. Encargados de capturar vídeo y enviarlo a un sistema de mensajería. Este componente también hace uso del sistema de control de acceso proporcionado por el componente de seguridad.
- **Microservicios**. Este componente engloba los distintos servicios encargados de procesar vídeo y datos.
- **Seguridad**. Desarrolla un sistema de control de acceso basado en la tecnología de *blockchain* que cumple los requisitos [*RNF04*].
- **Sistema de mensajería**. Encargado de transmitir información en tiempo real a los servicios abonados.

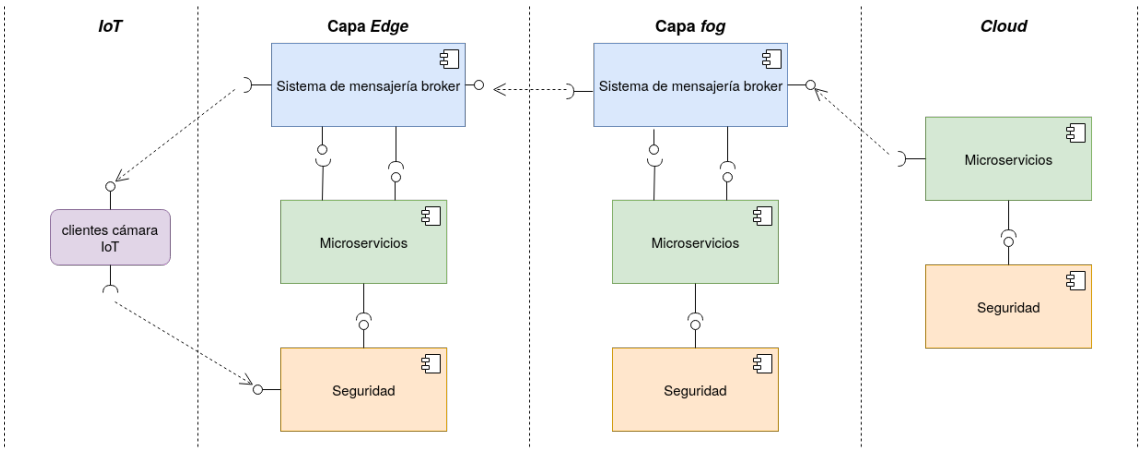

Figura 3.4: Diagrama de componentes de la arquitectura del sistema

En este diagrama también se puede observar que los servicios abonados pueden comportarse, a su vez, como clientes para el envío de información a los servicios desplegados en la capa superior. Para ello, cada capa debe disponer de su propio sistema de mensajería para poder comunicarse entre sí.

A continuación, se describe con más nivel de detalle la arquitectura del sistema de videovigilancia de la figura 3.5, centrándonos sobre todo en el componente de microservicios y el de seguridad.

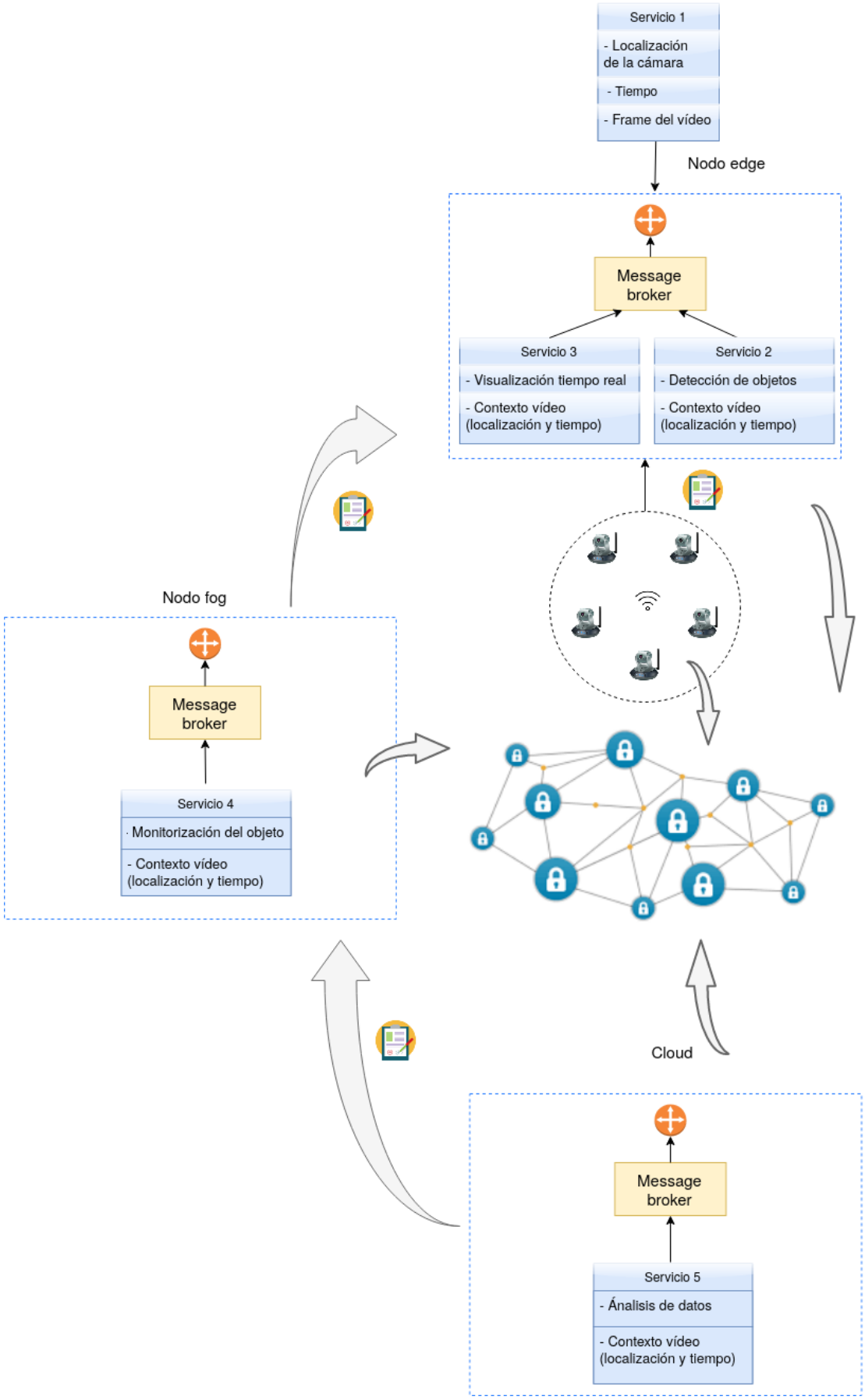

Figura 3.5: Arquitectura de sistema basado en 4 capas

Los dispositivos y los servicios en las distintas capas se encuentran comunicadas a través de un sistema de mensajería. A su vez, las cuatro capas del modelo están conectadas con la red de *blockchain*:

- Capa de los dispositivos cámara. Los dispositivos cámara se encuentran agrupados en un nodo *edge*, considerado como el "propietario" de los dispositivos. El acceso al vídeo capturado por estas cámaras está controlado por los *smart contracts*, de modo que si un servicio perteneciente al nodo *edge* desea acceder al vídeo de una cámara concreta, debe cumplir ciertas condiciones para que dicho dispositivo le conceda el acceso. Por otro lado, cada dispositivo cámara tendrá instalado un servicio que le permitirá enviar el vídeo y algunos datos de su contexto, tales como el tiempo y localización de la cámara.

- Nodo *edge*. Se encuentran diversos servicios con diferente funcionalidad cuyo propósito es el de realizar procesos críticos que requieran un tiempo de latencia relativamente pequeña, como por ejemplo, la visualización del vídeo y detección de objetos en tiempo real. Estos servicios también dispondrán de la información relativa al contexto del vídeo enviado por los servicios clientes de cada cámara. Dicha información será gestionada por cada servicio y reenviada a otros servicios para su procesamiento.
  En este nodo se generarán unos datos de salida al que tendrán acceso servicios pertenecientes a la capa *fog*. Por ello, si un servicio perteneciente a un nodo *fog* desea acceder a los datos generados por un servicio perteneciente a un nodo *edge*, debe cumplir unas condiciones que se controlan con los *smart contracts* y unas restricciones a nivel de nodo como se verá en el apartado de seguridad 3.3.3.

- Nodo *fog* y el nodo *cloud*. Tiene lugar un proceso similar al nodo *edge*. Por un lado, se despliegan los servicios que accederán a los datos generados por nodos pertenecientes a capas inferiores, donde tales servicios tienen diferente funcionalidad, generalmente con un mayor coste computacional. Por otro lado, los servicios que deseen acceder a servicios de las capas inferiores deben cumplir unas condiciones controladas por *smart contracts* y a nivel de nodo.

Diseño del componente de microservicios

Como ya se comentó en el apartado 2.5, la finalidad de este tipo de arquitectura es aportar a un sistema funcionalidades diferentes y muy bien definidas [*RNF07*] a través del uso de múltiples servicios. Además, cualquier modificación de la funcionalidad del sistema se puede realizar fácilmente actualizando, añadiendo o eliminando un servicio, proporcionando así, flexibilidad al sistema [*RNF02*] y, por lo tanto, facilitando el mantenimiento del mismo [*RNF10*].

Fases del funcionamiento del sistema:

- **Despliegue del servicio**. La instalación y el registro del servicio en el servicio de registro o *service registry* (figura 3.7) se puede realizar en cualquiera de los nodos de las 4 capas del sistema: *IoT*, *edge*, *fog* o *cloud* (ver requisito [*RF03*]). Cuando el servicio deje de estar operativo se desinstala y se elimina su registro del nodo correspondiente [*RF04*].

- **Ejecución de instancias del servicio**. Una vez el servicio está instalado en el nodo, se realiza la ejecución de un determinado número de instancias del servicio en paralelo.
  En el supuesto que se produzca la caída de una de las instancias, los datos pueden ser procesados por las instancias restantes. De este modo, se incrementa la fiabilidad de un sistema ante fallos inesperados de uno o varios elementos del sistema [*RNF09*].

- **Comunicación de los servicios**. La comunicación de los servicios se realiza a través de un sistema de mensajería basado en un *broker*. Este mecanismo de comunicación de los microservicios se verá con más detalle en el diseño del sistema de mensajería.

En el sistema de videovigilancia propuesto se presentan cuatro tipos de servicios (figura 3.6) distribuidos a lo largo de las cuatro capas del sistema.

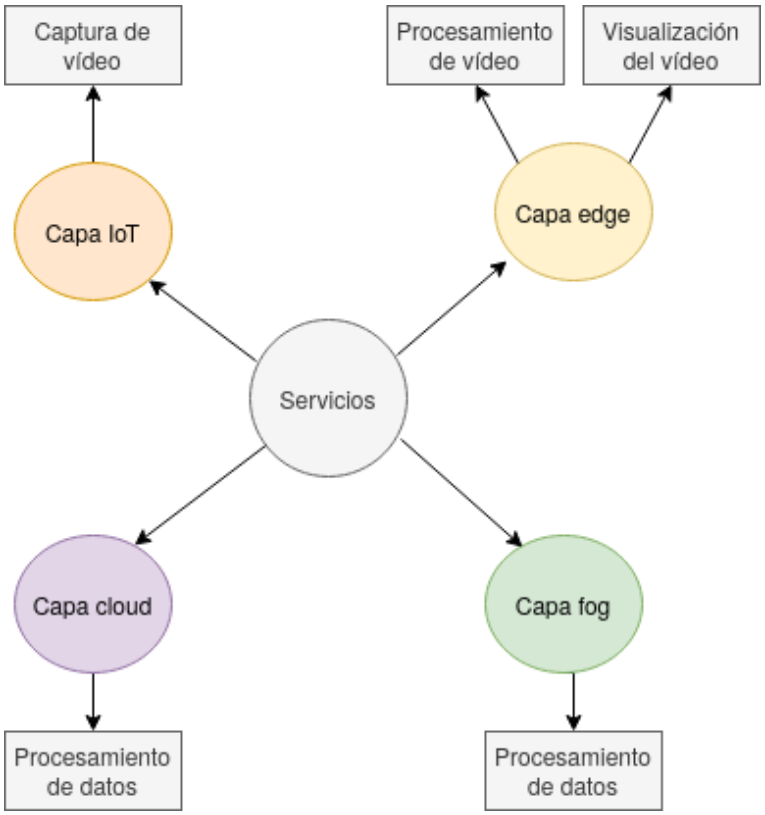

Figura 3.6: Esquema de los servicios del sistema

A continuación, se describe cada uno de estos servicios:

- **Captura de vídeo** [*RF09*]. Este servicio se ejecuta en el dispositivo cámara *IoT* correspondiente.
  Tiene las siguientes funcionalidades:

  - Creación de un identificador "id" único.
  - Activar la cámara del dispositivo.
  - Lectura del *frame* del vídeo capturado.
  - Empaquetar en un archivo con extesión *json* el identificador de la cámara, el *frame* y el instante en el cual el *frame* es capturado por la cámara.

  Se debe realizar el envío de vídeo y otros datos del contexto del vídeo al sistema de mensajería con dos objetivos:

  - Envío de vídeo para que pueda ser procesado por otros servicios en capas superiores.
  - Envío de datos del contexto del vídeo para que los datos obtenidos en cada capa siempre estén ligados al vídeo procesado en un instante determinado.

- **Procesamiento de vídeo** [*RF10*]. Este servicio se encarga de extraer características de los objetos detectados. Por ejemplo, en el caso de tratarse de un servicio detector de objetos los datos extraídos serían:

  - Número de detecciones
  - Categoría a la que pertenece el objeto
  - Precisión en la detección
  - Posición el objeto detectado.

Es posible realizar otras operaciones como el almacenamiento del vídeo o características del vídeo con la finalidad de comprobar cualquier manipulación del vídeo por usuarios ajenos al sistema.

- **Procesamiento de datos extraídos** [*RF11*]. Este tipo de servicio puede estar desplegado en un nodo *fog* o en un nodo *cloud*, ya que realiza procesos más exhaustivos. Principalmente, tiene dos funcionalidades:
  - Análisis de los datos obtenidos por otros servicios.
  - Almacenamiento de datos [*RF17*] en una base de datos propia de cada servicio con otros fines, por ejemplo, llevar a cabo procesos de monitorización del objeto detectado.
- **Visualización del vídeo** [*RF13*]. Este tipo de servicio puede estar desplegado en un nodo *edge* o *fog*. Tiene como funcionalidad mostrar la visualización del vídeo en tiempo real de una o varias cámaras.

Diseño del sistema de mensajería

En el sistema de mensajería empleado en el diagrama 3.5 se opta por utilizar un sistema de mensajería asíncrono basado en un *broker*, pero hay que tener presente que existen otros mecanismos de comunicación que carecen de *brokers*. Por ejemplo, en el capítulo 2 algunos investigadores proponen el uso del protocolo *REST*, sin embargo, se considera que este protocolo no es adecuado para el sistema propuesto, puesto que exige que el cliente y el servidor estén disponibles durante el proceso de intercambio de información. Además, al tratarse de un protocolo síncrono, un cliente *HTTP* debe esperar a recibir una respuesta por parte del servidor, y por tanto, la disponibilidad del sistema se ve reducida.

Podemos encontrar otros sistemas de mensajería que no utilizan un *broker* y que se podrían tener en cuenta en este proyecto, pero se ha encontrado que este tipo de sistemas tienen, a su vez, algunas desventajas que no son adecuadas parar el mismo. Por ejemplo, los servicios que utilizan este sistema de comunicación deben conocer la localización del servicio con el que quieren comunicarse, por lo que estos servicios deben utilizar algún mecanismo de *service discovery*. Por otro lado, estos servicios deben hacer uso de otros mecanismos que también son importantes en un proceso de comunicación, como el garantizar la entrega de un paquete de información. Como consecuencia, el uso de estos mecanismos incrementa la complejidad de la lógica de los servicios.

Las ventajas de un sistema de mensajería basado en un *broker* son:

- **Comunicación asíncrona**. Siempre que se desee maximizar la disponibilidad del sistema, debe reducirse en lo posible las cantidades de comunicación síncrona.
- **Acoplamiento reducido** [*RNF08*]. El cliente realiza una petición simplemente enviando los mensajes o datos al canal apropiado del sistema de mensajería. Por consiguiente, el cliente desconoce las instancias de los servicios que solicitan sus servicios o recursos. Por este motivo, el cliente no necesita disponer de un mecanismo de *service discovery* para encontrar y aplicar balanceo de carga de dichas instancias, ya que el sistema de mensajería es el encargado de realizar estas tareas, facilitando así el mantenimiento del sistema [*RNF10*].
- **Almacenamiento de mensajes**. Este sistema almacena los mensajes hasta que puedan ser procesados por los consumidores. En consecuencia, los datos que se envíen no se perderán en el caso de que el consumidor no esté disponible temporalmente debido a una caída o carga de trabajo.
  Como se verá más adelante, este tipo de sistemas también aportan cierta fiabilidad, puesto que no se perderán datos como se ha comentado anteriormente y además pueden ser configurados para emplear múltiples *brokers* con el propósito de no bloquear el sistema ante la caída de uno, cumpliéndose así el requisito [*RNF09*].

El sistema de mensajería propuesto dispone de los siguientes componentes:

- **Descubrimiento de servicios o *service discovery***. Se compone del balanceo de carga y del servicio de registro [*RF18*].

- **Balanceo de carga o *load balancer***. Es el encargado de distribuir la carga de trabajo entre las distintas instancias de los servicios, de modo que dicha instancia no se vea sobrecargada con peticiones.
  Se garantiza la continuación del proceso global del sistema en el caso de que una de las instancias de los servicios deje de funcionar. Por lo tanto, se dejan de enviar datos a esa instancia para enviarlos a otras instancias disponibles. En el supuesto de que dicha instancia esté otra vez disponible, el balanceo de carga puede volver a enviarle tráfico de datos.
  De igual forma, es capaz de gestionar distintas situaciones como añadir o eliminar un servicio en el sistema.

- **Servicio de registro**. Encargado de registrar los servicios. Almacena las direcciones *IP* de las instancias para que sean localizables por el balanceo de carga.
  Es el encargado de actualizar los datos de los servicios o eliminar un servicio del registro si dicho servicio no está operativo.

- **Gestor del sistema**. Es el encargado de gestionar los distintos componentes del *broker*. Es posible disponer de múltiples *brokers*.

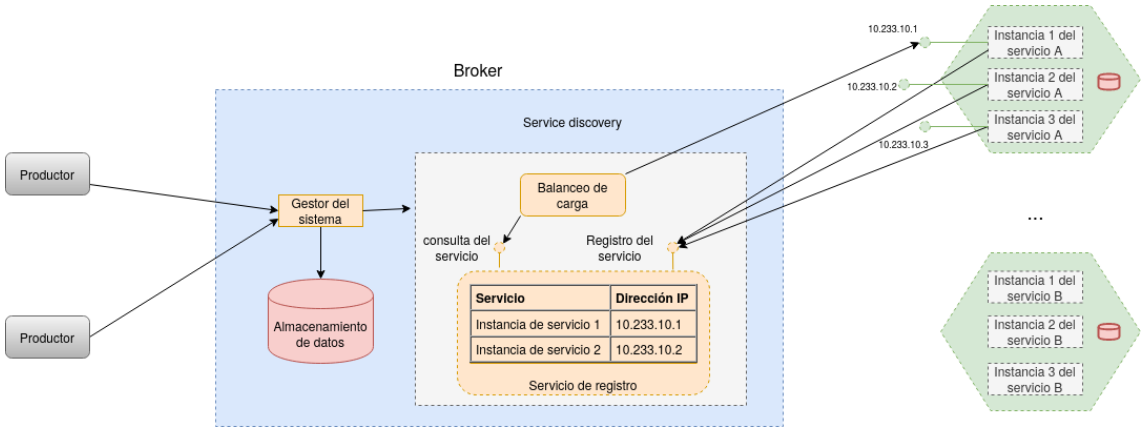

Figura 3.7: Diseño del sistema basado en microservicios

En la imagen 3.7 se puede observar los distintos elementos implicados en un proceso de comunicación.

- **Sistema de mensajería**. Compuesto del balanceo de carga, servicio de registro y un gestor. El servicio de registro almacena las direcciones *IP* de las distintas instancias de cada servicio y el balanceo de carga consulta al servicio de registro las instancias disponibles con la finalidad de distribuir los mensajes entre dichas instancias.

- **Consumidores**. Se tratan de servicios completamente independientes que disponen de su propia base de datos. Se puede ejecutar más de una instancia del mismo servicio.
  Estos servicios consumidores pueden convertirse en productores cuando envíen datos recopilados a otros servicios para otro tipo de análisis. Para más detalle ver el apartado de microservicios 3.3.3.

- **Productores**. Dispositivos cámara o servicios que envían datos en tiempo real.

- **Sistema de almacenamiento**. Opcionalmente se puede configurar el sistema de mensajería para almacenar los datos durante el proceso de comunicación con el propósito de gestionar dichos datos de forma independiente al resto de servicios del sistema. Por ejemplo, se podría configurar para almacenar los datos o vídeo [*RF16*] indefinidamente o eliminarlos pasado un tiempo determinado.

Diseño del componente de seguridad

Se propone el uso de la tecnología *blockchain* con el propósito de controlar el acceso (a partir del uso de eventos [*RF12*]) a los diversos recursos y servicios del sistema, puesto que proporciona mecanismos de seguridad como la criptografía y *smart contracts*.

La figura 3.8 muestra a nivel alto el mecanismo de control de acceso al vídeo y a los datos el sistema de videovigilancia.

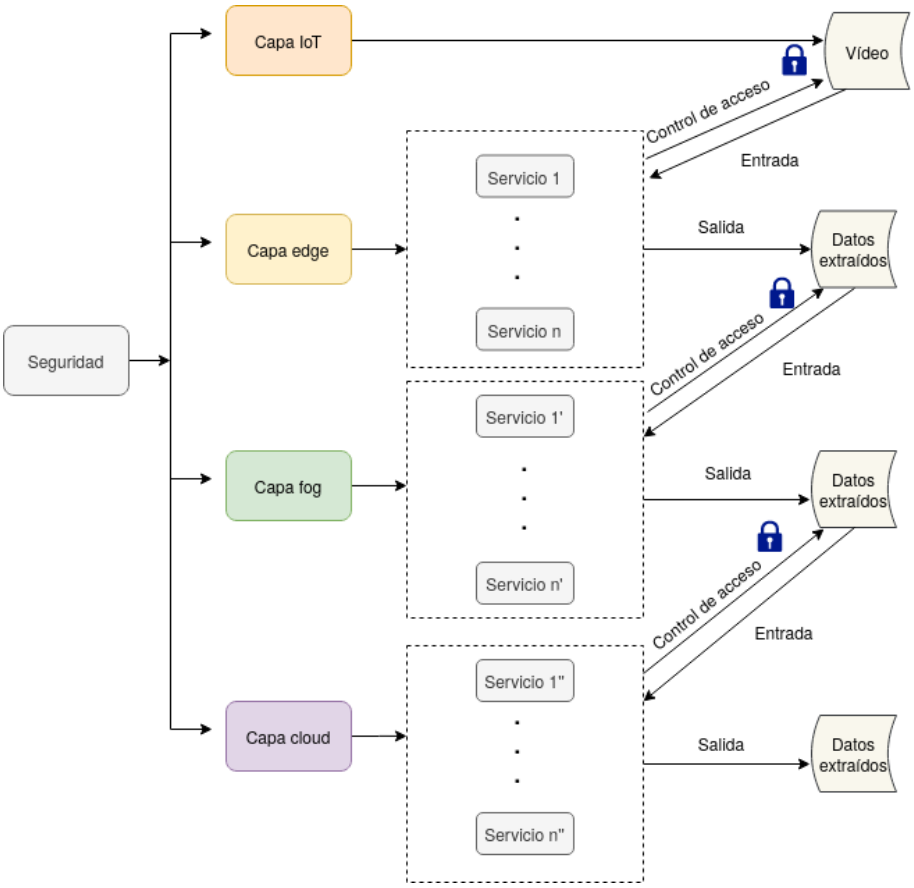

Figura 3.8: Esquema del control de acceso del sistema

En el componente de seguridad se diseñan las siguientes funcionalidades:

El alta y la baja de los dispositivos cámara en la red de *blockchain*.

El alta de la cámara [*RF01*] implica la ejecución de las siguientes operaciones:

- La creación de una cuenta digital en *blockchain*.
- Registrar el identificador de la cámara (creado en el apartado de microservicios) en la lista interna del nodo *edge* "propietario".

La baja de la cámara [*RF02*] implica la ejecución de las siguientes operaciones:

- Eliminar la cuenta digital de la cámara en *blockchain*.
- Eliminar el identificador de la cámara de la lista interna del nodo *edge* "propietario".

El alta y la baja de los nodos en la red de *blockchain*

El alta del nodo [*RF05*] consiste en:

- Crear la cuenta digital del nodo en *blockchain*.
- Desplegar en *blockchain* una lista interna (basada en *smart contracts*) con el identificador de las cámaras del nodo.
- Añadir el identificador del nodo en el grafo de nodos.

La baja del nodo [*RF06*] consiste en:

- Eliminar la cuenta digital del nodo en *blockchain*.
- Eliminar el *smart contract* correspondiente al nodo desplegado en *blockchain*.
- Eliminar el nodo del grafo de nodos.

Alta y baja de los servicios en *blockchain*.

El alta del servicio [*RF03*] se realiza en dos fases para tener un mayor control del proceso de registro, puesto que un servicio podrá acceder a datos y recursos del sistema.
Las dos fases para registrar el servicio en *blockchain* son:

- **Despliegue de la dirección virtual del servicio**. Esta fase tiene como finalidad registrar el servicio en la lista interna de verificadores o *approvers* basados en *smart contracts* con el objetivo final de incrementar la seguridad y tener un mayor control del proceso de registro. En la red de *blockchain* se puede disponer de uno o varios verificadores que pueden ser configurados por el sistema. El papel del verificador consiste en realizar procesos de validación en su propio entorno y aprobar el despliegue de la dirección virtual del servicio en su lista interna.

  El proceso del despliegue, mostrado en la figura 3.9, consiste en los siguientes pasos:

  - Crear una cuenta digital del servicio en *blockchain* y obtener la dirección virtual a partir de la clave pública del servicio.
  - Realizar una transacción para añadir la dirección virtual del servicio en la lista interna (basado en un *smart contract*) del verificador.
  - El verificador emite el evento "*deployService*" vía *smart contracts*. Este evento contiene el identificador del servicio, el identificador del nodo donde el servicio está instalado y la dirección virtual del servicio.
  - El evento es recibido por los observadores o *listeners* de los verificadores restantes para repetir el proceso de registro de la dirección virtual del servicio después de realizar las validaciones necesarias para comprobar que el servicio y sus datos son correctos. De esta manera, cada verificador mantiene una lista con las direcciones virtuales de los servicios del sistema.
    En esta transacción se envían los datos del servicio y del nodo.

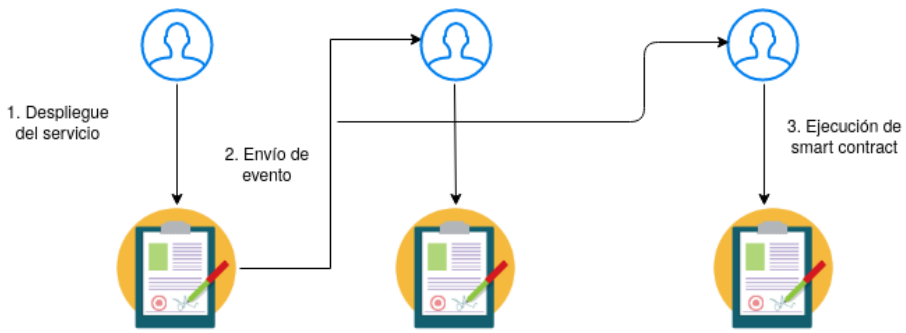

Figura 3.9: Despliegue de un servicio

- **Registro de la dirección virtual del servicio en un nodo**. Se utiliza *smart contracts* para almacenar en una lista interna las direcciones virtuales de los servicios que tienen acceso a los recursos o servicios disponibles en este nodo.
  Consiste en los siguientes pasos:
  - Cada nodo tiene un propietario o gestor que posee el control de los dispositivos o servicios conectados a él.
  - El servicio solicita el registro al propietario del nodo para tramitar su registro.
  - El propietario o gestor del nodo consulta a los verificadores si la dirección virtual de dicho servicio está registrado en la lista interna de los verificadores.
  - Se lleva a cabo el registro de la dirección virtual del servicio en el nodo después de ser aprobado por todos los verificadores. En caso de ser rechazado por uno de ellos, se realiza el registro del servicio con estado pendiente y se emite un evento, "*UpdateService*", para que otro proceso del sistema repita el proceso del registro en un tiempo determinado con la finalidad de validar o rechazar el servicio. En el último caso, se elimina completamente la dirección virtual del servicio de la lista interna del nodo.

La baja del servicio [*RF04*] se realiza en dos fases:

- Se elimina la dirección virtual del servicio de la lista interna de los verificadores. El proceso de eliminación también emplea el mismo sistema (basado en eventos) utilizado en el registro de la figura 3.9. De este modo, se elimina el registro del servicio de todos los verificadores.
- Posteriormente, se elimina el registro del servicio del nodo en un proceso de sincronización del nodo con la lista interna de los verificadores.

Control de acceso al vídeo.

El control de acceso tiene como objetivo controlar el acceso de los servicios al vídeo capturado por las cámaras [*RF14*]. El proceso se divide en 3 fases:

- Registro de la dirección virtual del servicio en el nodo correspondiente (proceso explicado en el apartado anterior).
- El servicio solicita al nodo propietario de la cámara la generación de un token[3], llamado *ticket* (figura 3.10), que contiene los siguientes datos:
  - Identificador de la cámara.
  - Dirección virtual del propietario del *ticket* (nodo que genera el *ticket*).
  - Dirección virtual del servicio que solicita el acceso.
  - Derecho de acceso: lectura, escritura o lectura y escritura.
  - Contador que indica el número de veces que el *ticket* puede ser utilizado. Cuando el contador llega a cero, el *ticket* es destruido.
  - Fecha de expiración del *ticket*. A partir de esta fecha el *ticket* no es válido y, por tanto, es destruido.

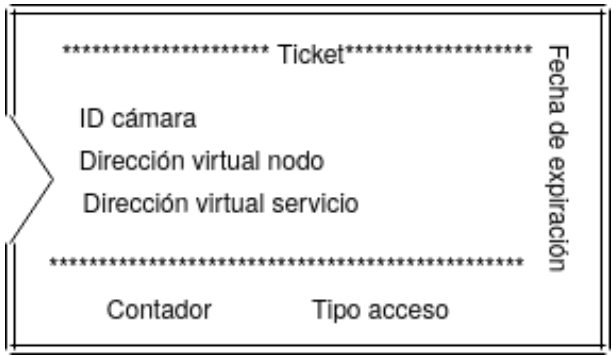

Figura 3.10: Ticket de acceso

- El servicio solicita el acceso al dispositivo cámara para tener acceso al vídeo capturado por el mismo. Para ello, el dispositivo emplea un *smart contract* para controlar el acceso al vídeo. Este control se basa en leer el *token* generado por el nodo y comprobar si dicho servicio tiene acceso.
  En el caso de tener acceso se emite un evento, "*AllowAccess*" (contiene el identificador de la cámara y la dirección virtual del servicio) para que se le conceda. Este evento, como se verá más adelante en el apartado 3.4.2, será gestionado por el *listener* u observador de este contrato para el envío de datos de conexión y la creación de una transacción que servirá como recibo del proceso realizado.

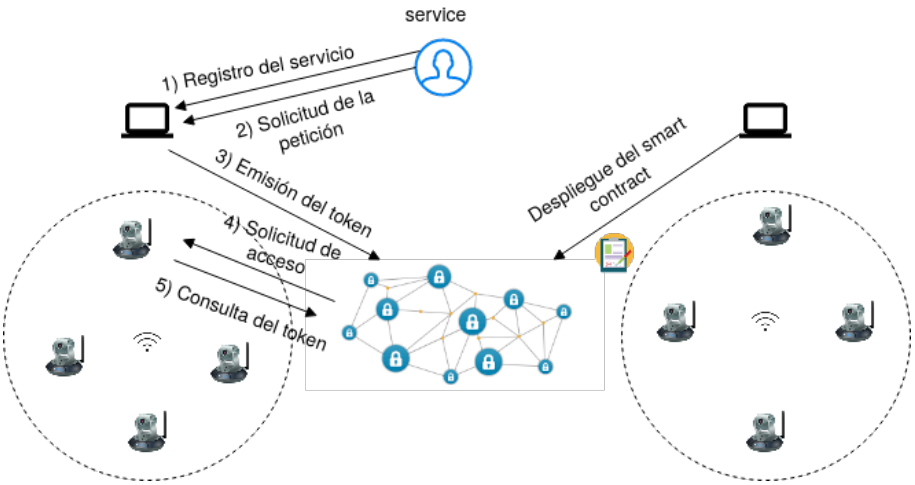

Figura 3.11: Control de acceso del vídeo

La figura 3.11 muestra el proceso completo del control de acceso del vídeo de un servicio. En primer lugar, el gestor del nodo despliega el *smart contract* con la lista interna de las cámaras de las que es propietario. Posteriormente, se realiza el registro del servicio(1) (registro de su dirección virtual), la solicitud de generación del token(2) y, finalmente, la petición de acceso(4) y lectura del *token*(5).

Control de acceso a datos.

Este proceso está formado por 3 fases [*RF15*]:

- Registro de la dirección virtual del servicio en el nodo correspondiente.
- El servicio solicita el acceso al servicio (vía *smart contracts*) que genera los datos. Como puede verse en la figura 3.12, el proceso es similar al control de acceso del vídeo, pero en este caso no se genera un *token ticket* sino que simplemente se comprueba que el servicio solicitante (nodo B) esté registrado en uno de los nodos conectados con el nodo del servicio proveedor (nodo A). Por lo tanto, un servicio que quiera acceder a otro debe solicitar el acceso al mismo vía *smart contracts*. En el supuesto que se le conceda el acceso, el servicio proveedor emitirá el evento, "*AllowAccess*", con la dirección virtual del servicio solicitante, el identificador del nodo del servicio proveedor y el identificador del nodo del servicio solicitante.

[3]Token es un tipo de *smart contract* que representa un activo digital en *blockchain*, pudiendo ser transferible.

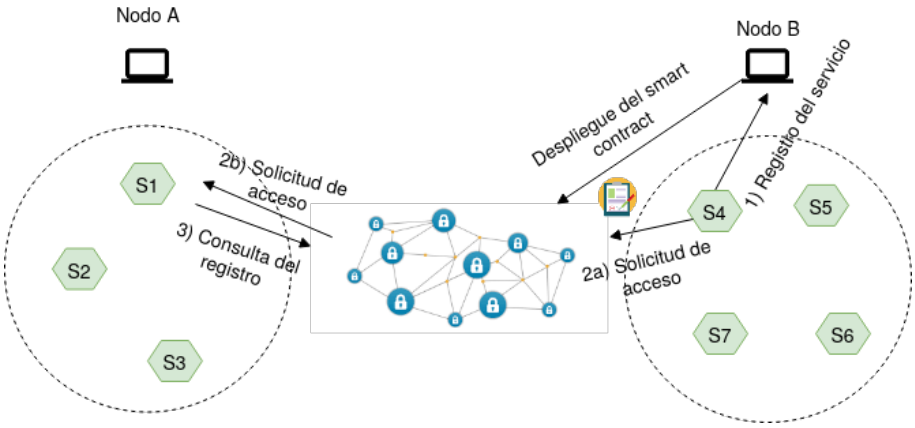

Figura 3.12: Control de acceso a datos

- Al emitirse el evento "*AllowAccess*", se comprueba que el nodo del servicio solicitante tiene acceso al nodo del servicio proveedor. El control de acceso se basa en el uso de un grafo dirigido (figura 3.13), donde los servicios de un nodo pueden acceder a los servicios disponibles en los nodos que estén conectados a dicho nodo a través del grafo.
  Puesto que se trata de un grafo dirigido, la dirección de comunicación de los servicios es unidireccional, ya que carece de sentido que un servicio instalado en un nodo reciba datos de servicios instalados en los nodos pertenecientes a capas superiores. Por ejemplo, un servicio del nodo *edge* no tiene sentido que reciba datos de un servicio perteneciente a un nodo *fog*. Sin embargo, sí es posible la comunicación entre nodos pertenecientes a la misma capa. En este último escenario, dicha comunicación entre nodos debe ser incluida en el grafo de nodos.

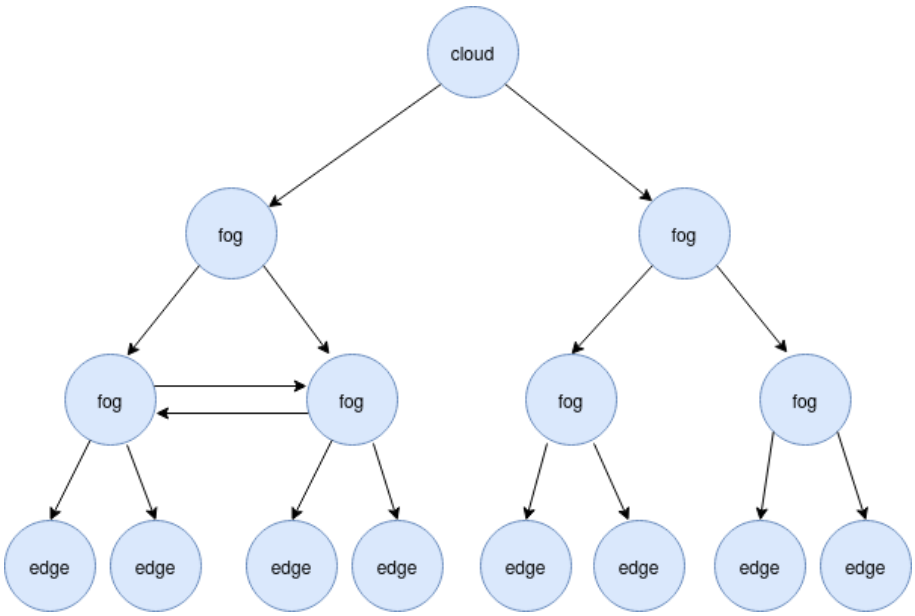

Figura 3.13: Ejemplo de un grafo dirigido de nodos del sistema

- El *listener* del contrato emisor gestiona el evento, se encarga de enviar los datos de conexión y de generar la transacción para reflejar en *blockchain* el proceso, al igual que el control de acceso de vídeo.

3.3.4. Casos de uso

Los casos de uso son una herramienta para el análisis de proyectos software, permitiendo analizar la interacción entre los distintos componentes del sistema.

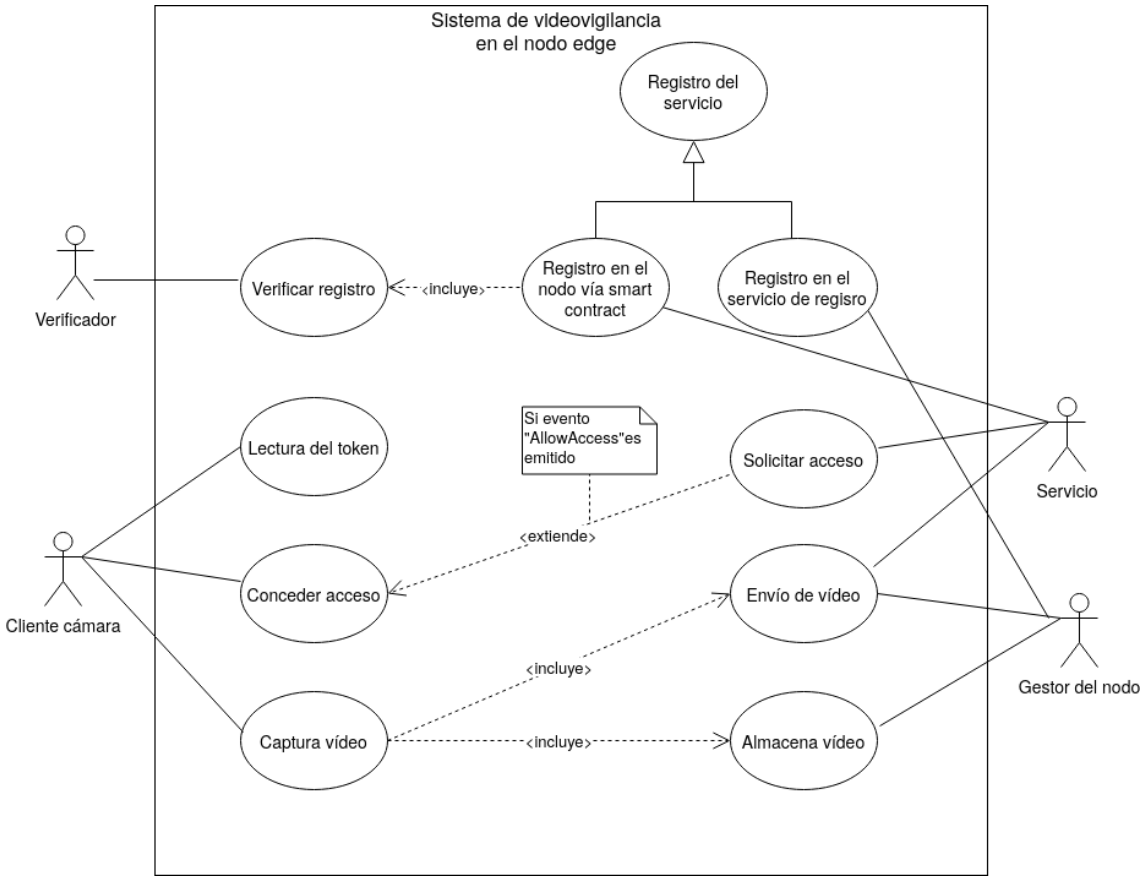

Figura 3.14: Caso de uso en el nodo *edge*

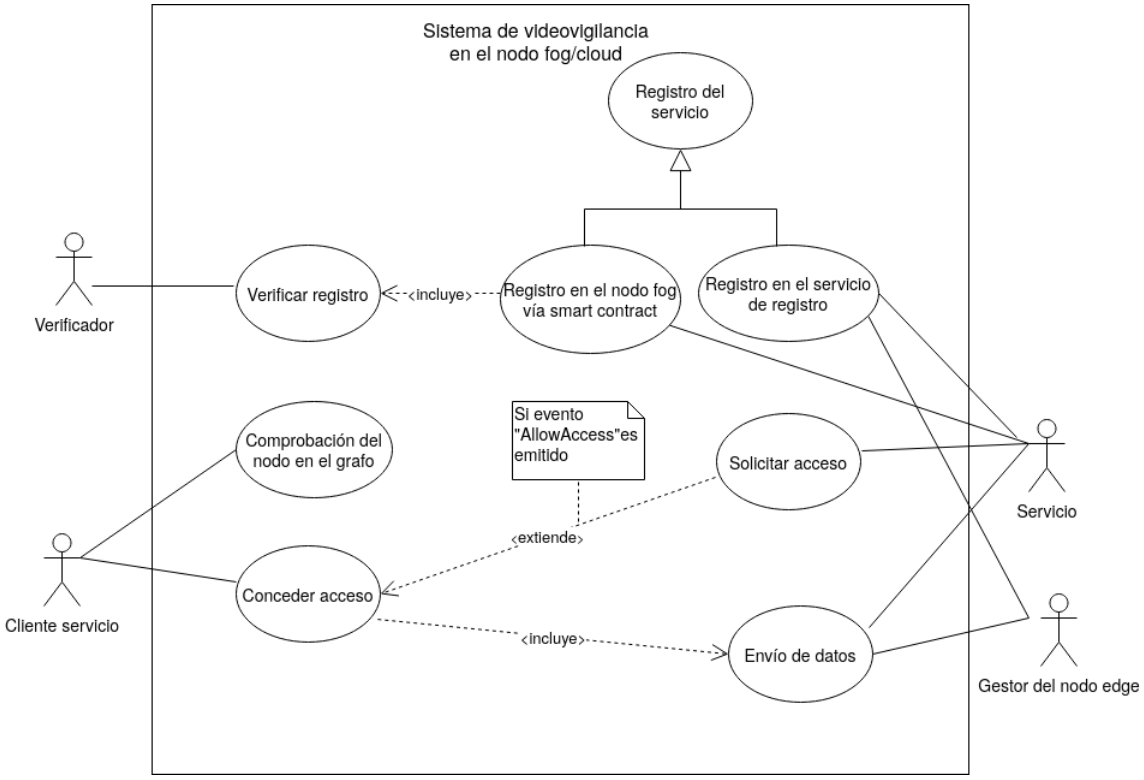

Figura 3.15: Caso de uso en el nodo *fog/cloud*

3.3.5. Casos de riesgo

Los factores de riesgo del sistema, posibles amenazas y consecuencias son descritos a continuación.
Para cada amenaza incluida, se considera el objetivo de la misma, el grado del riesgo, descripción, técnicas que se aplican para llevarla a cabo y las medidas a tomar para prevenir que el sistema sea atacado.

Amenaza 1	
Objetivo	Acceso a la clave privada
Riesgo	Alto
Descripción	Robo de la clave privada o de la contraseña para recuperar la clave. Se pueden realizar operaciones no deseables como desencriptar mensajes o realizar transacciones maliciosas
Técnicas	Ataque de fuerza bruta, ataque de canal lateral, ataque de *replay*, engaño, etc.
Contramedidas	▪ No compartir la clave privada ▪ Emplear contraseñas seguras y que se actualicen frecuentemente ▪ Uso de tiempo de expiración en los mensajes ▪ Informar al usuario ▪ Utilizar otras técnicas especializadas en la protección de la clave

Tabla 3.29: Riesgo 1: Acceso a la clave privada

Amenaza 2	
Objetivo	Acceso a vídeo o datos
Riesgo	Medio
Descripción	Se accede al vídeo transmitido o datos almacenados en bases de datos
Técnicas	Ofuscación
Contramedidas	▪ Control de acceso ▪ Encriptado de la información ▪ Almacenamiento de los datos en un sistema descentralizado

Tabla 3.30: Riesgo 2: Acceso al vídeo trasmitido o datos almacenados

Amenaza 3	
Objetivo	Comprometer la red de *blockchain*
Riesgo	Alto
Descripción	Realizar operaciones que comprometan la seguridad de las transacciones y de los *smart contracts*.
Técnicas	Ataque del 51 %, ataque *DAO*, *underflow*, *race condition*, etc.
Contramedidas	▪ Mantener un número considerable de nodos en el sistema ▪ Controlar los bucles del *smart contract* ▪ Verificar que los datos primitivos están dentro del rango permitido ▪ Uso de técnicas especializadas: uso de *mutex* para bloquear el estado del *smart contract* hasta que el propietario pueda realizar las verificaciones correspondientes y actualizar correctamente el estado.

Tabla 3.31: Riesgo 3: Ataque a la red de *blockchain*

Amenaza 4	
Objetivo	Interceptación de la comunicación
Riesgo	Alto
Descripción	El atacante puede acceder a los datos del vídeo o datos transmitidos
Técnicas	*Hooking*, *MITM* (*man in the middle attack*)
Contramedidas	▪ Encriptación de la información ▪ Empleo de firma digital

Tabla 3.32: Riesgo 4: Interceptación de la comunicación

Amenaza 5	
Objetivo	Acceder al sistema con privilegios de administrador
Riesgo	Medio
Descripción	El atacante puede realizar operaciones no permitidas, por ejemplo, desplegar un servicio sin autorización
Técnicas	Robo de identidad
Contramedidas	▪ Ciertas operaciones deben recaer únicamente en el propietario del *smart contract* ▪ Limitación de los privilegios del administrador ▪ Autorización múltiple

Tabla 3.33: Riesgo 5: Acceder al sistema con privilegios de administrador

En la tabla 3.34, localizada en la siguiente página, se muestra la clasificación de estas amenazas utilizando el modelo *STRIDE*[40].

En la tabla 3.35, localizada en la siguiente página, se califica cada amenaza siguiendo el modelo *DREAD*[41].
El rango de ponderaciones empleado para definir el grado de riesgo es la siguiente:

- (12 a 15): Riesgo alto
- (8 a 11): Riesgo medio
- (5 a 7): Riesgo bajo

Por último, se muestra el caso de mal uso del sistema.

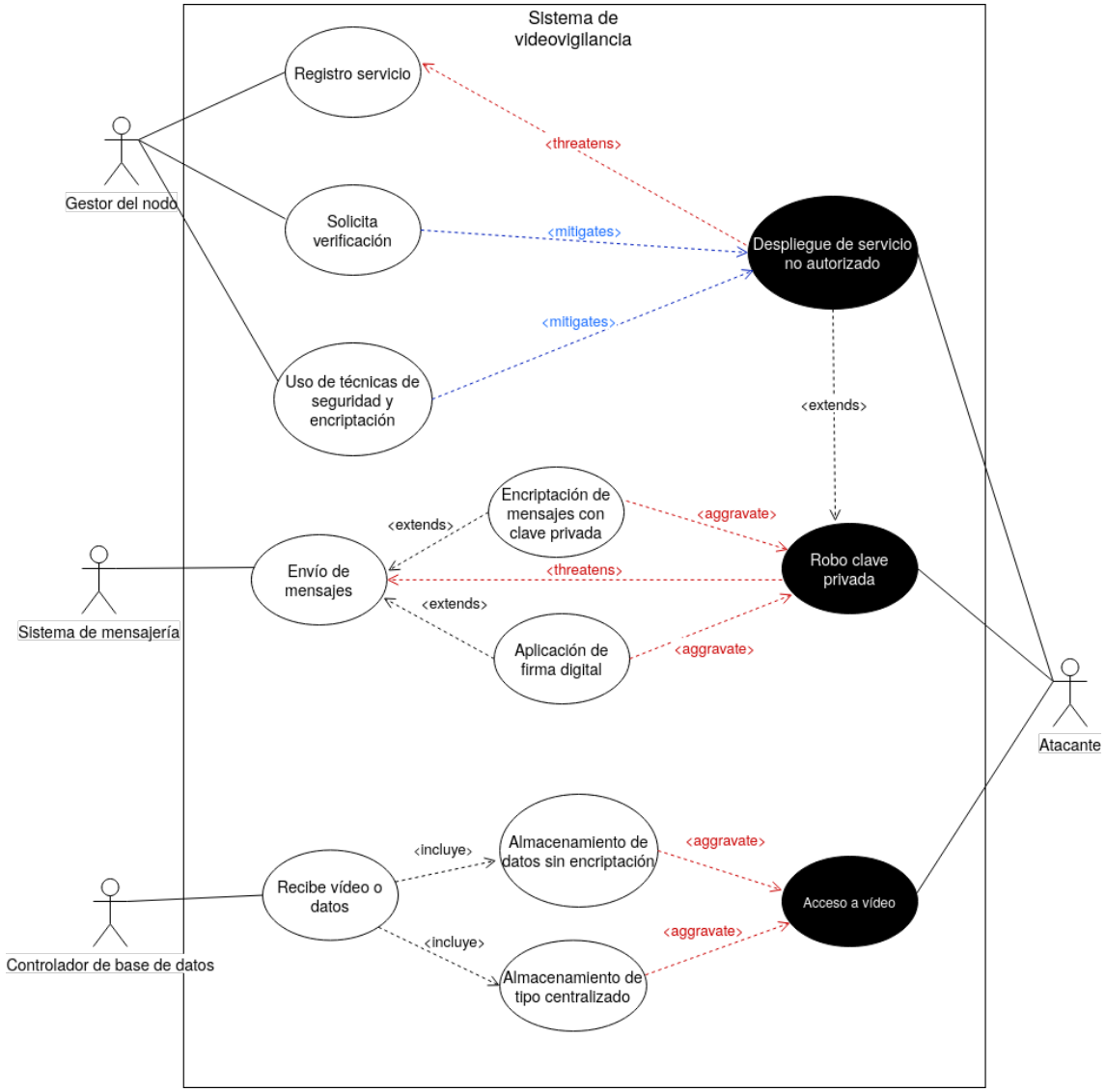

Figura 3.16: Caso de mal uso

Amenazas	Suplantación de identidad	Manipulación de datos	Repudio	Divulgación de información	Negación de servicio	Elevación de privilegio
Amenaza 1	✓	✓		✓		
Amenaza 2		✓		✓	✓	
Amenaza 3			✓	✓	✓	
Amenaza 4		✓	✓	✓	✓	
Amenaza 5	✓	✓		✓	✓	✓

Tabla 3.34: Riesgo 5: Acceder al sistema con privilegios de administrador

Amenazas	D	R	E	A	D	Total	Calificación riesgo
Amenaza 1	2	1	1	2	1	7	Bajo
Amenaza 2	3	2	2	3	2	12	Alto
Amenaza 3	2	1	1	2	1	7	Bajo
Amenaza 4	2	2	1	2	2	9	Medio
Amenaza 5	2	1	1	3	1	8	Medio

Tabla 3.35: Riesgo 5: Acceder al sistema con privilegios de administrador

3.4. Diseño técnico

3.4.1. Introducción

Se presenta el diseño de la implementación del prototipo realizado.
El diseño incluye únicamente los diagramas y el pseudocódigo de los fragmentos de código que se consideran relevantes para el entendimiento del flujo de los diversos procesos.

La figura 3.17 representa el diagrama de componentes de los módulos que conforman el sistema de videovigilancia del prototipo implementado con un sistema de mensajería constituida por un único *broker*. El diseño es una simplificación de la arquitectura 3.4 para reducir el nivel de complejidad de la implementación y la ejecución de pruebas.

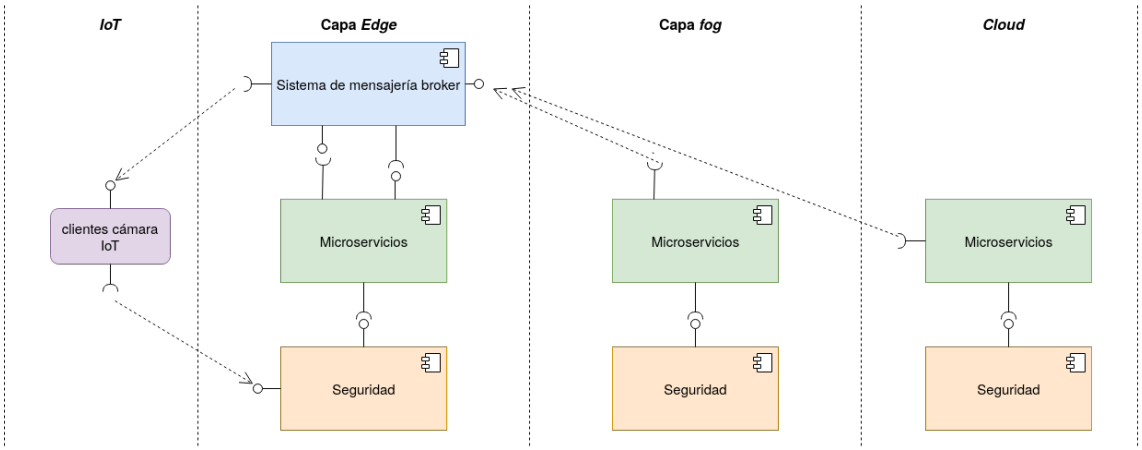

Figura 3.17: Diagrama de componentes del sistema de videovigilancia del prototipo implementado

A continuación, se describe la funcionalidad y el diseño técnico de cada uno de los componentes arriba mostrados.

3.4.2. Seguridad

Se implementa la lógica de control de acceso y encriptación en el presente módulo. La lógica es definida en los *smart contracts* y en los clientes *Javascript*.
Para comunicar los clientes en *Javascript* con *Ethereum*, se hace uso del *API web3*. Para realizar el desarrollo de los *smart contracts*, se emplea el lenguaje de programación *Solidity*, puesto que actualmente es uno de los lenguajes que más se utiliza en la creación de los *smart contracts* en *blockchain*.

API web3

El *API web3.js* permite comunicar clientes en *Javascript* con *Ethereum blockchain* a través de llamadas remotas(RPC).

Para comunicar el cliente con *Ethereum* se opta por emplear dicha *API* debido a que actualmente es la más estable y dispone de una gran comunidad de desarrolladores que dan soporte *online*.

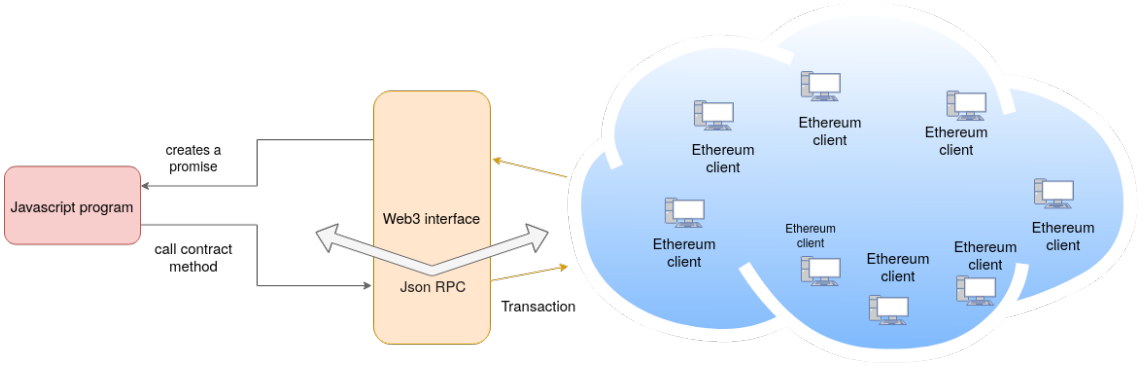

Figura 3.18: Diagrama de comunicación del *API web3* con *blockchain*

Como se puede ver en la figura 3.18, un cliente en *Javascript* puede realizar llamadas a los métodos públicos de un *smart contract*. En el supuesto de que se quisiera realizar una transacción, se utilizaría el método *"send"* y en el caso de una consulta, se podría emplear el método *"call"* del *API web3*.
En ambos casos, con clientes en *Javascript* se puede emplear la lógica de *promises* (se comportan como receptores de eventos para recibir de forma asíncrona si dicha llamada se ha ejecutado con o sin éxito) para comunicarse con *Ethereum*.

Control de acceso

Se procede a presentar los diagramas de clases, diagramas de secuencia y pseudocódigo de la lógica que controla el control de acceso a los datos y vídeo.
Posteriormente, se muestra el diagrama de clases y pseudocódigo de los distintos *smart contracts* implementados en el desarrollo del prototipo.

La figura 3.19 representa las clases que se han implementado haciendo uso de los *smart contracts*. En dicho diagrama podemos resaltar los siguientes detalles:

- Cada nodo dispone de su propia clase de registro, y aunque la lógica de esta clase varía del nodo *fog* al del nodo *edge*, ambas clases comparten una interfaz similar, por lo que heredan de la clase *"Register"*.
- La creación de *"tickets"* solo puede ser realizada por un nodo de tipo *edge*, por lo que el método que crea el *"ticket"* se define como *internal* en la clase *"Permission"* para que sólo pueda ser llamado por las clases hijas, en este caso, la clase *"EdgeRegister"*.
- La lógica de control de acceso dependerá de si el proveedor se trata de un servicio o un dispositivo cámara. Por tanto, cada cámara cliente dispondrá de su propio control de acceso que pertenece a la clase *"EdgeNodeControlAccess"*. De igual manera, cada servicio (desplegado en cualquiera de los nodos: *edge*, *fog* y *cloud*) dispondrá de su propio control de acceso que pertenece a la clase *"FogNodeControlAccess"*.
- La clase *FogRegister* se emplea para registrar servicios en los nodos *edge*, *fog* y *cloud*.

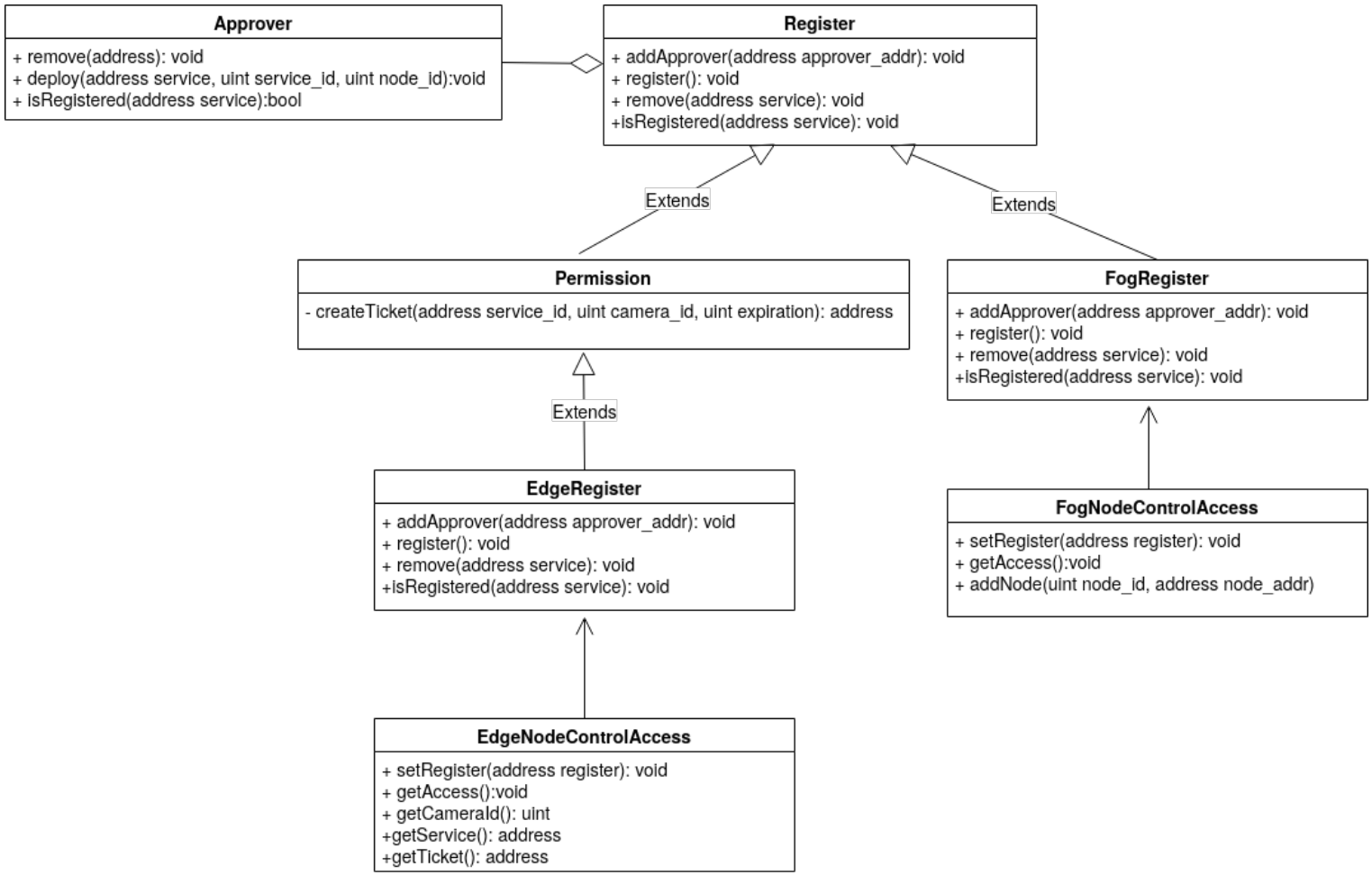

Figura 3.19: Diagrama de clases del control de acceso basado en *smart contracts*

En el algoritmo 1 y en el algoritmo 2 se describe la lógica del despliegue y la eliminación de un servicio en un verificador en la red de *blockchain*. En este despliegue se debe indicar el tipo del nodo en el que el servicio va a ser desplegado, es decir, debe indicar si el nodo es un nodo de tipo *fog*, *edge* o *cloud*.

Algorithm 1 Approver:deploy service

```
1: procedure DEPLOY(service, service id, node id)
2:     ServiceConfiguration ← data structure: service_id, node_id, is_registered
3:     _service_data_deployment ← mapping: (address =>ServiceConfiguration)
4:     if service is not registered then
5:         if msg.sender is owner then
6:             _service_data_deployment ← add service
7:             emit event deployService
8:         else
9:             revert changes
```

Algorithm 2 Approver:remove service

```
1: procedure REMOVE(service)
2:     if service is registered then
3:         if msg.sender is owner then
4:             delete service
5:             emit event removeService
6:         else
7:             revert changes
```

El algoritmo 3 describe la lógica para realizar el registro de un servicio en un nodo.
Esta fase de registro del servicio implica que cada nodo debe disponer de su propio *smart contract* para realizar el registro, ya que el único que puede ejecutar el método *"register"* es el propietario del *smart contract*, en este caso, el nodo. En este método se llevan a cabo dos funcionalidades principales:

- Registro del servicio al ser aprobado por los distintos *approvers* o verificadores.
- En el caso de que el registro no se haya podido completar, el servicio se guarda con estado *"Pending"* y se emite un evento que puede ser gestionado por un *listener*. Posteriormente, mediante un proceso *batch*, se puede iniciar otra vez el proceso de registro hasta que el registro del servicio se complete con éxito o se alcance el límite máximo de tiempo (*"_deadline"*) establecido para dicho servicio.

Algorithm 3 Register:register

```
1: procedure REGISTER
2:     if msg.sender is owner then
3: loop(min_approvers):
4:         if msg.sender is registered in approver then
5:             node_id ← approver.getNodeId()
6:             if register id == node_id then
7:                 state service is Pending
8:                 service id is approver.getServiceId()
9:                 num_approvers ← num_approvers + 1
10:            else
11:                revert changes
12:        goto loop
13:        if num_approvers == min_approvers then
14:            state register is Registered
15:        else
16:            _deadline ← 1 day
17:            emit event UpdateService
18:    else
19:        revert changes
```

El algoritmo 4 define la lógica que controla la generación del *token* que permite al servicio acceder a los servicios proporcionados por el dispositivo cámara. Este *token* generado, conocido como *"ticket"*, se compone de los siguientes datos:

- El identificador del dispositivo cámara al que el servicio tiene acceso.
- Identificador del servicio que solicita el acceso a la cámara.
- Un contador que permite reutilizar el *"ticket"* un número de veces definido por este contador.
- Fecha de caducidad, donde el *"ticket"* podrá ser eliminado por el controlador de acceso.
- Dispone de tres tipos de acceso: modo lectura, modo escritura o ambas a la vez.

Algorithm 4 Register:getAccess

```
1: procedure GETACCESS(service id, camera id)
2:     if service is registered in node then
3: loop:
4:         _camera_id ← list of cameras in node
5:         if camera id == _camera_id then
6:             generate ticket
7:             break
8:         goto loop
```

El algoritmo 5 describe la lógica del control de acceso del recurso, en este caso, el dispositivo cámara. Por tanto, el servicio tendrá acceso al vídeo después de que se verifique la validez del *"ticket"* generado para este servicio.

Algorithm 5 EdgeControlAccess:getAccess

```
1:  procedure GetAccess
2:      ticket ← getTicket(msg.sender, camara id)
3:      if ticket does not exist then
4:          revert changes
5:      else
6:          if ticket expiration date < now then
7:              destroy ticket
8:          else
9:              counter ← getCounter()
10:             if counter > 0 then
11:                 camera_id ← ticket.getCameraId()
12:                 service_id ← ticket.getService()
13:                 if (camara id == service_id) and (msg.sender == service_id) then
14:                     if ticket rigth isREAD or READ_WRITE then
15:                         counter ← counter - 1
16:                         emit event AllowAccess
17:                         if counter <= 0 then
18:                             destroy ticket
19:                 else
20:                     revert changes
21:             else
22:                 revert changes
```

El algoritmo 6 muestra la lógica que permite a un servicio solicitante (servicio generalmente desplegado en el nodo *fog* o en el nodo *cloud*) acceder a los datos extraídos por los servicios en capas inferiores.

Algorithm 6 FogControlAccess:getAccess

```
1:  procedure GetAccess
2:      if msg.sender is registered in node then
3:          emit eventAllowAccess
4:      else
5:          revert changes
```

Como puede observarse, el controlador de acceso genera un evento que será gestionado por un *listener* del servicio proveedor para verificar si el servicio solicitante puede acceder a estos datos (ver figura 3.25). Este requisito se cumple cuando el nodo del servicio solicitante tiene en su lista al nodo del servicio proveedor (diagrama 3.20).

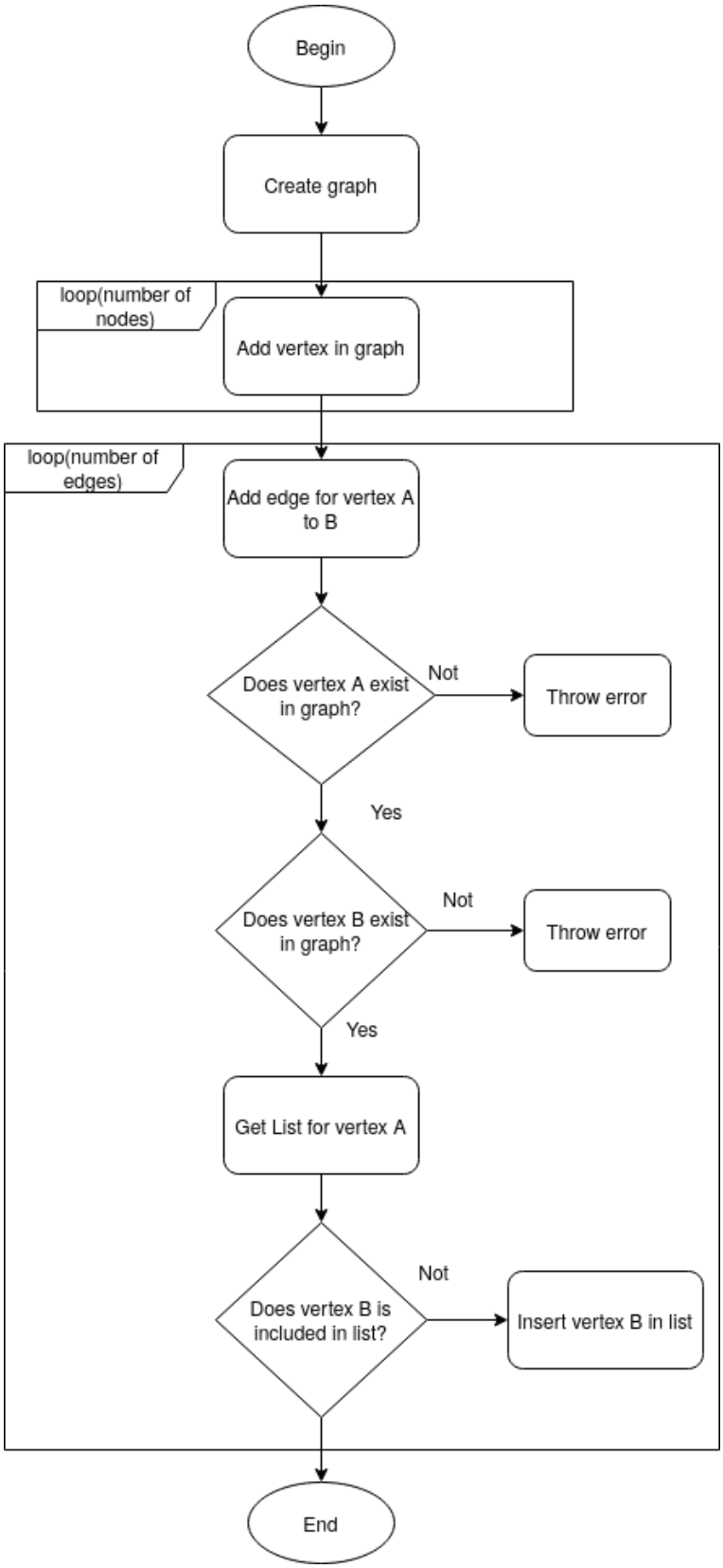

Figura 3.20: Diagrama de flujo del grafo de nodos

La figura 3.21 muestra el proceso de despliegue de un servicio en la red de *blockchain*. Como ya se ha comentado en la sección de diseño, el número de verificadores, *"approvers"*, puede ser configurado en el sistema. En este prototipo se han utilizado tres verificadores.

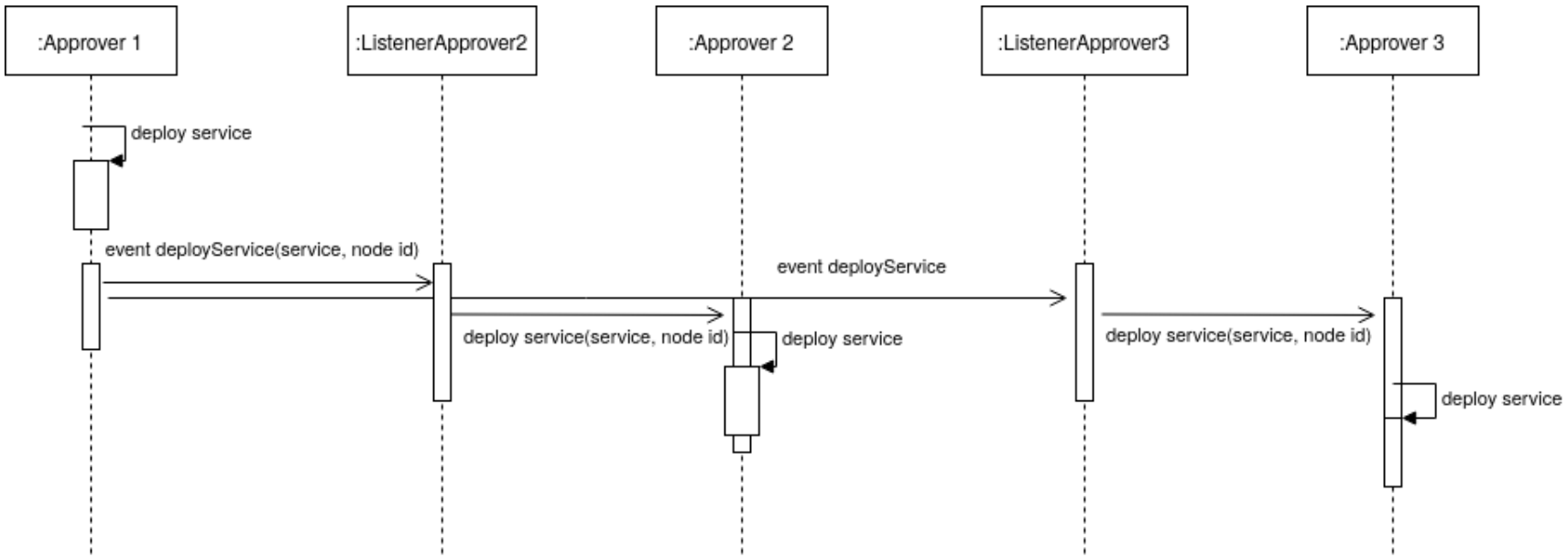

Figura 3.21: Diagrama de secuencia del proceso de despliegue de un servicio

La figura 3.22 representa el proceso completo del control de acceso al vídeo capturado por una cámara.
El evento *"AllowAccess"* es emitido después de validar el *"ticket"* del servicio solicitante. Este evento es gestionado por el *listener* del dispositivo cámara, *"ListenerControlAccess"*. Su función es empaquetar y enviar los datos de conexión (datos que se verán con más detalle en el apartado de *Kafka*), siendo transmitidos al servicio solicitante para que pueda inscribirse como abonado a la recepción del vídeo. Con este proceso de empaquetación se pretende aumentar la seguridad del proceso de comunicación (ver amenaza 4, apartado 3.3.5), que consiste en lo siguiente:

- Obtener la firma digital del servicio proveedor a partir de la clave privada y el mensaje que almacena los datos en formato *JSON* con el objetivo de asegurar la autenticidad y la integridad del mensaje.
- Encriptar la firma digital y el mensaje con la clave pública del servicio solicitante. De esta forma, se asegura que el servicio solicitante es el único que puede leer el mensaje.

Por otro lado, el *listener* también realiza una transacción en *blockchain* que servirá como "recibo" del acceso concedido. Esta transacción contiene la siguiente información:

- El emisor de la transacción. En este caso, el dispositivo cámara.
- El receptor de la transacción que hace referencia al servicio solicitante.
- El dato transmitido en esta transacción contiene la firma digital obtenida en el apartado anterior. Por tanto, el servicio puede realizar comprobaciones internas con los datos recibidos vía *HTTP*.

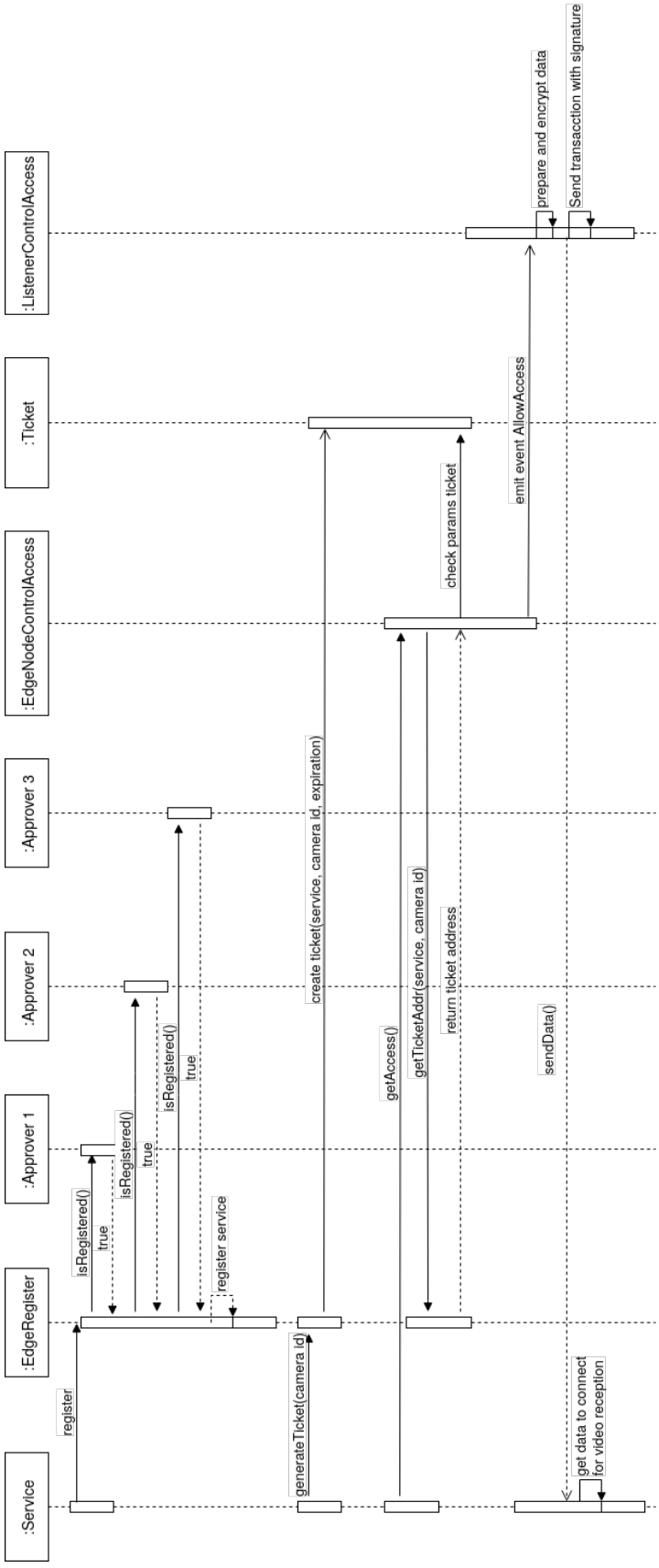

Figura 3.22: Diagrama de secuencia de una petición de acceso al vídeo con éxito

En el diagrama de secuencia 3.23 se muestra un proceso de petición de acceso fallido. En este caso, se realiza el registro del servicio sin éxito debido a que no todos los verificadores tienen registrado dicho servicio. De esta forma, se aumenta la seguridad en el caso de que un atacante consiga comprometer los datos almacenados de un verificador (ver amenaza 5 del apartado 3.3.5).

Otro proceso que se realiza cuando falla el registro del servicio es la emisión de un evento *"updateService"* que permitirá completar el proceso de registro.
Finalmente, también se puede observar que al no estar el servicio registrado en el nodo, se rechazarán los procesos de generación del *"ticket"* y de acceso, protegiendo así el acceso al dispositivo cámara por parte de otros usuarios no autorizados.

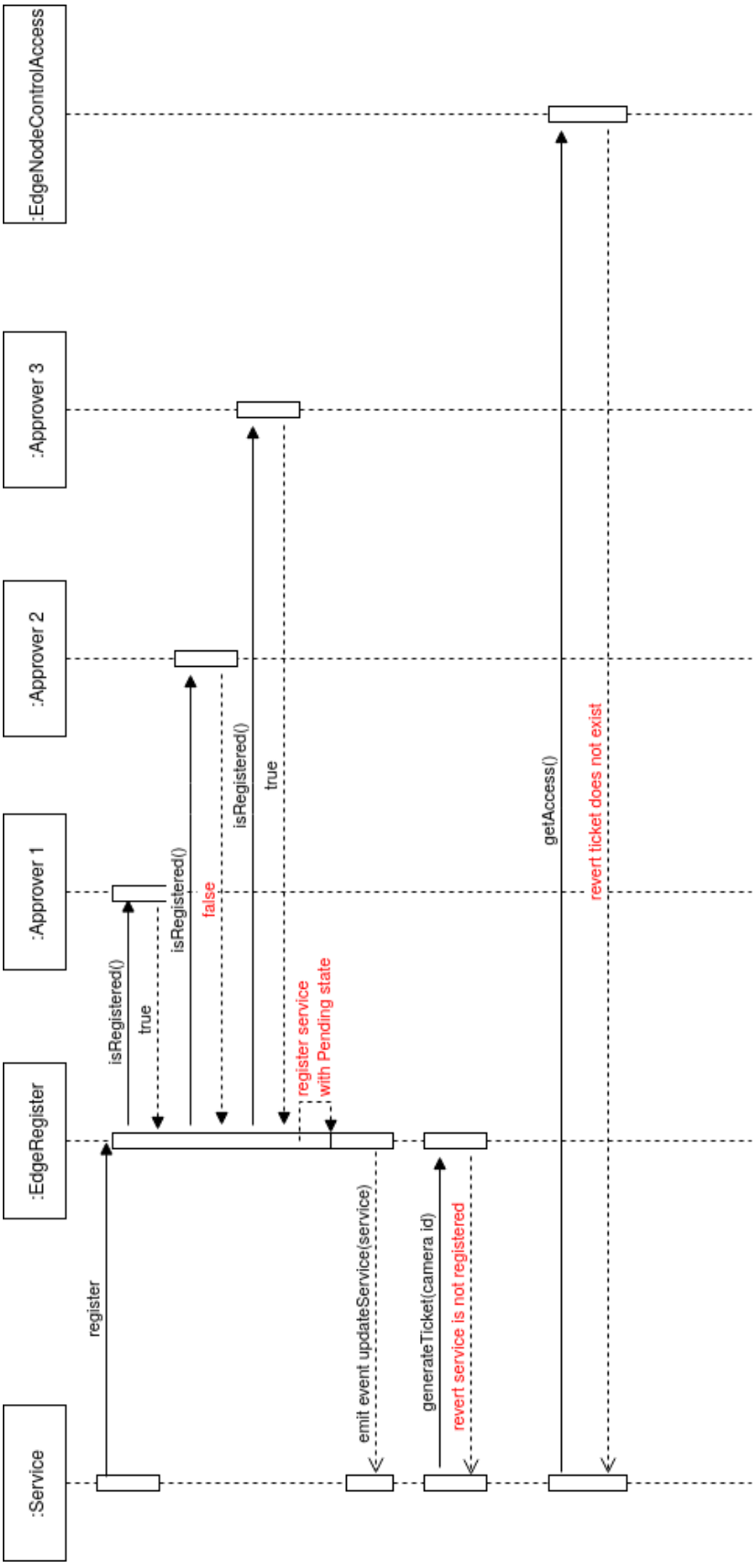

Figura 3.23: Diagrama de secuencia de una petición de acceso fallido

En la figura 3.24 se representa otro proceso de petición de acceso a la cámara que finaliza con error, pero en este caso se rechaza el acceso debido a que el *ticket* ya no es válido en esta conexión. Por tanto, el servicio debe solicitar a su nodo respectivo la generación de otro *ticket* que sí le permita el acceso. De este modo, se evita que dichos *tickets* circulen en la red de *blockchain*, ya que podrían ser manipulados y reutilizados por otros usuarios.

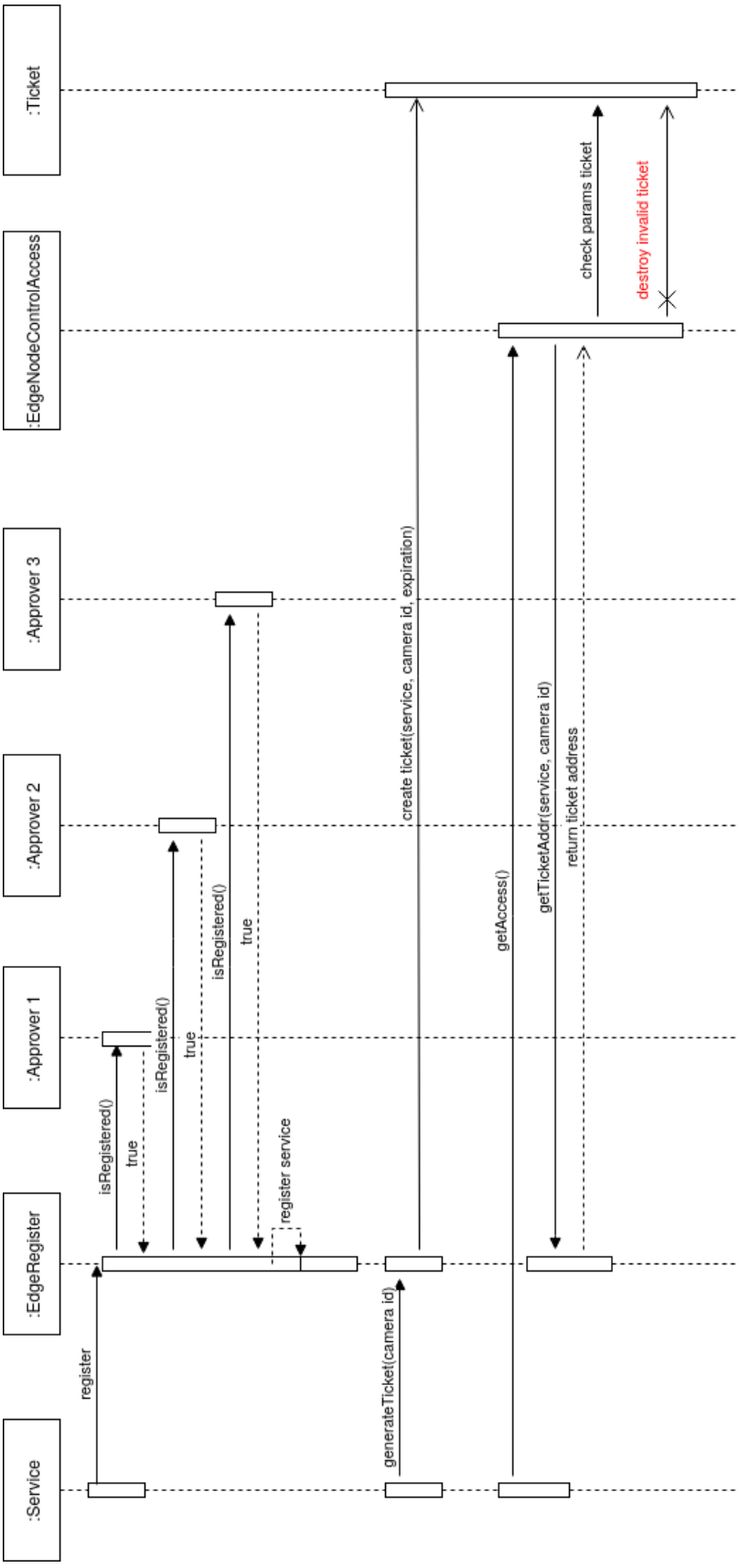

Figura 3.24: Diagrama de secuencia de la generación y destrucción de un *ticket* que deja de tener validez

En la figura 3.25 podemos observar el proceso de control de acceso de un servicio que solicita el acceso a los datos extraídos previamente por un servicio desplegado en el nodo *edge*. Este flujo del proceso es similar a la petición de acceso al vídeo, pero difiere en que el servicio no solicita la generación de un *ticket*, sino que directamente solicita el acceso al controlador de acceso del servicio proveedor, llamado *"FogNodeControlAccess"*. Este controlador comprueba que dicho servicio se encuentra registrado en el nodo "propietario" del servicio solicitante. De modo que si se verifica este registro, el controlador de acceso emite un evento *"AllowAccess"* que es capturado por el *listener "ListenerControlAccess"*. Dicho *listener* comprueba si existe un camino en el grafo dirigido entre el nodo del servicio solicitante y el nodo del servicio proveedor (para más detalle ver sección 3.3.3). Una vez se haya validado el acceso del servicio, se lleva a cabo la empaquetación (proceso explicado en apartados anteriores) de los datos de conexión para la recepción de datos.

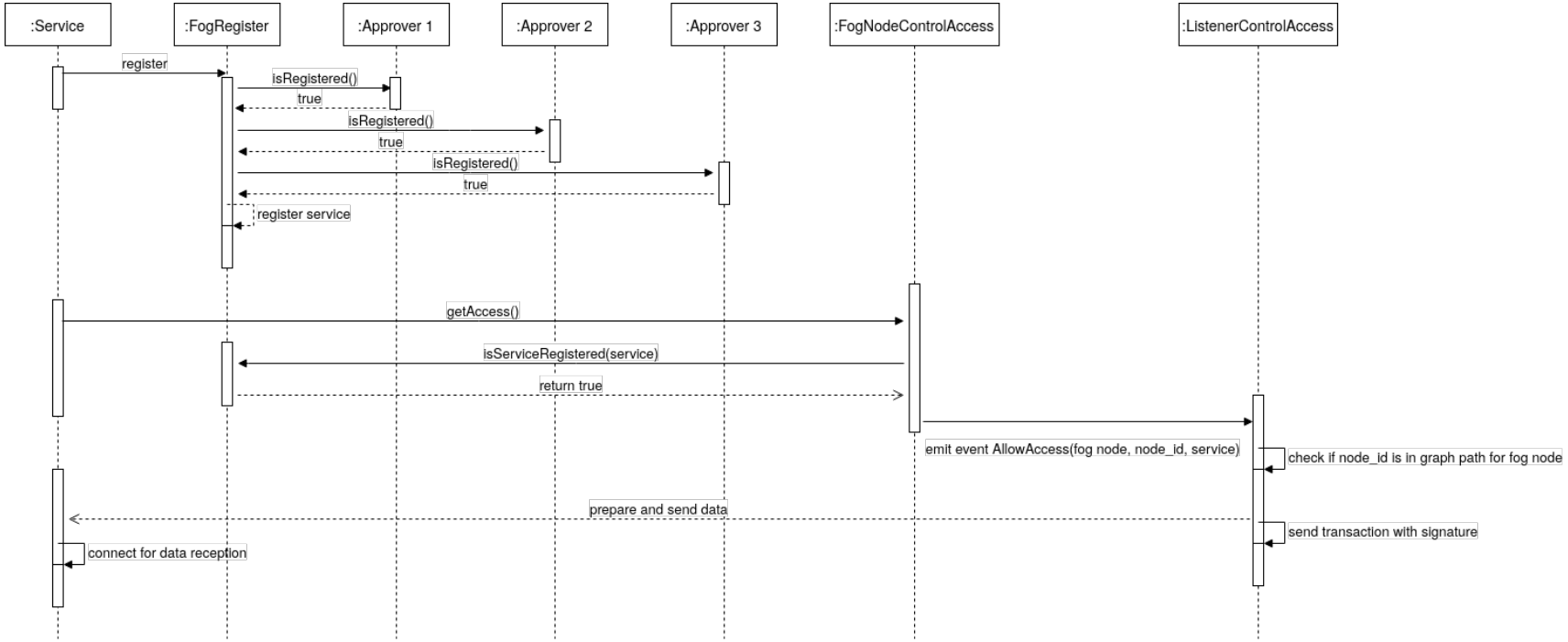

Figura 3.25: Diagrama de secuencia de la petición de acceso de un servicio a los datos extraídos por un servicio en el nodo *edge* o en un nodo *fog*

En último lugar, el *listener* también realiza una transacción en *blockchain* del acceso concedido. Esta transacción tiene las siguientes características:

- El emisor de la transacción, en este caso, es el servicio proveedor.
- El receptor de la transacción que hace referencia al servicio solicitante.
- La firma digital del servicio proveedor.

El diagrama 3.26 muestra la petición de acceso fallido de un servicio solicitante que está desplegado en un nodo *fog* y, siguiendo las reglas del grafo dirigido (explicado en el apartado anterior), no tiene acceso al nodo del servicio proveedor (nodo *edge*).

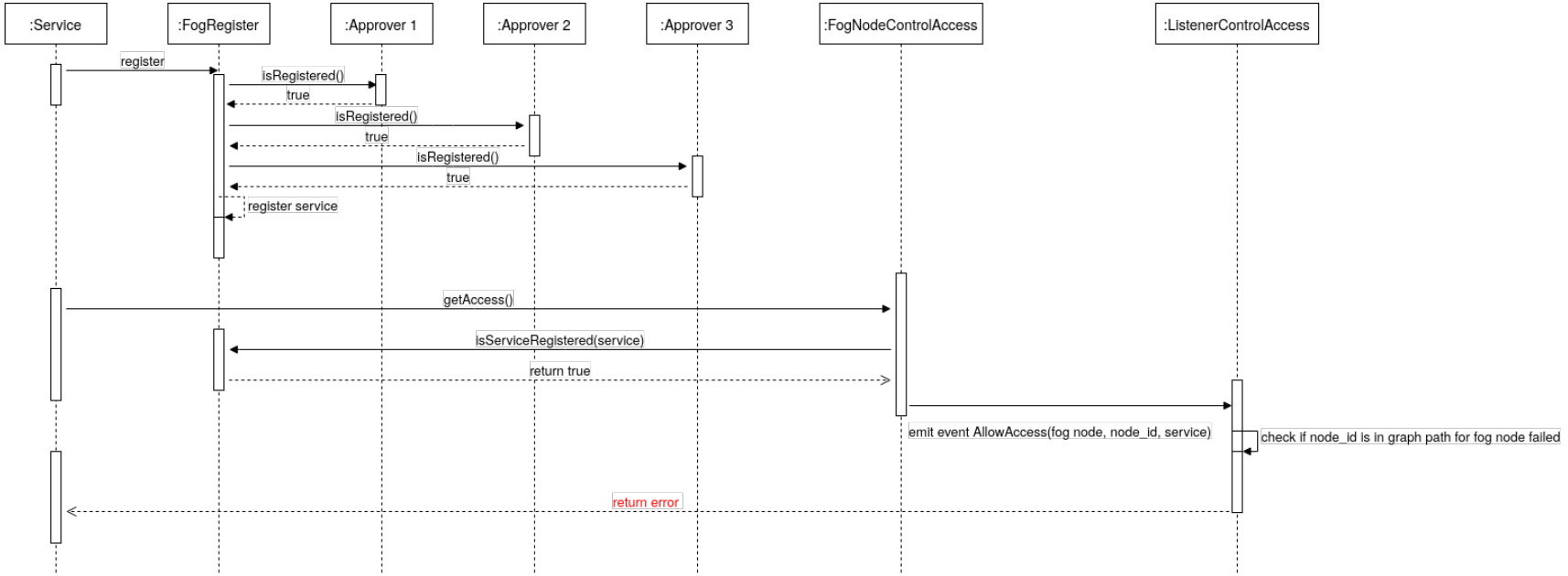

Figura 3.26: Diagrama de secuencia de una petición de acceso fallido de un servicio a los datos extraídos por un servicio en un nodo *edge* o un nodo *fog*

3.4.3. Sistema de mensajería

Para llevar a cabo el desarrollo del prototipo se opta por utilizar un sistema de mensajería asíncrono basado en un *broker* (apartado 3.3.3). Sin embargo, este tipo de sistemas también tienen algunas desventajas:

- El rendimiento del sistema puede verse afectado si se produce un cuello de botella.
- Se trata de un mecanismo centralizado, por tanto, puede impactar en la fiabilidad del sistema si se produce un fallo.
- Aumento de la complejidad del sistema, ya que se trata de otro componente del sistema que debe ser instalado, configurado y mantenido.

Kafka

Diversos sistemas de mensajería habitualmente empleados han sido estudiados: *ActiveMQ*, *RabbitMQ* y *Apache Kafka*.
Se ha optado por el uso de *Apache Kafka*, ya que ofrece abundantes funcionalidades útiles en un sistema de videovigilancia y soluciona los problemas que suelen experimentar este tipo de sistemas. Las ventajas de este sistema frente a los sistemas de mensajería *ActiveMQ* y *RabbitMQ* son:

- Permite mantener el orden de los mensajes.
- Proporciona persistencia de datos, es decir, puede ser configurado para retener los mensajes durante un periodo de tiempo o de forma indefinida. Por lo tanto, el mismo mensaje puede ser reproducido múltiples veces. Sin embargo, en *RabbitMQ* los mensajes son eliminados una vez han sido consumidos.
- Soporta el modelo *pull messages*: los consumidores son los encargados de solicitar y consumir los mensajes. Sin embargo, el modelo soportado habitualmente por otros sistemas es el *push message*, que podría abrumar al consumidor si la tasa de llegada de los mensajes es más alta que la tasa de procesamiento por parte del consumidor.
- Soporta un gran número de consumidores y ofrece un buen rendimiento en el envío de gran cantidad de datos en tiempo real.
- Es altamente escalable, ya que pueden ser añadidos más *brokers* a su sistema, convirtiéndole en un sistema más resistente a fallos.

Kafka dispone de su propio *service discovery*. *Kafka* proporciona balanceo de carga a partir del uso de grupos y se protege de forma automática de un posible fallo en el sistema mediante el uso del factor de replicación. Hace uso del servicio centralizado *Zookeeper* para monitorizar su estado. *Zookeeper* proporciona otras funcionalidades como el registro de nombres(*service registry*) y el mantenimiento de la configuración y sincronización.

Información adicional sobre *Kafka* está disponible en el apéndice A.2.

A continuación, se muestran los esquemas del productor y de los consumidores implementados.

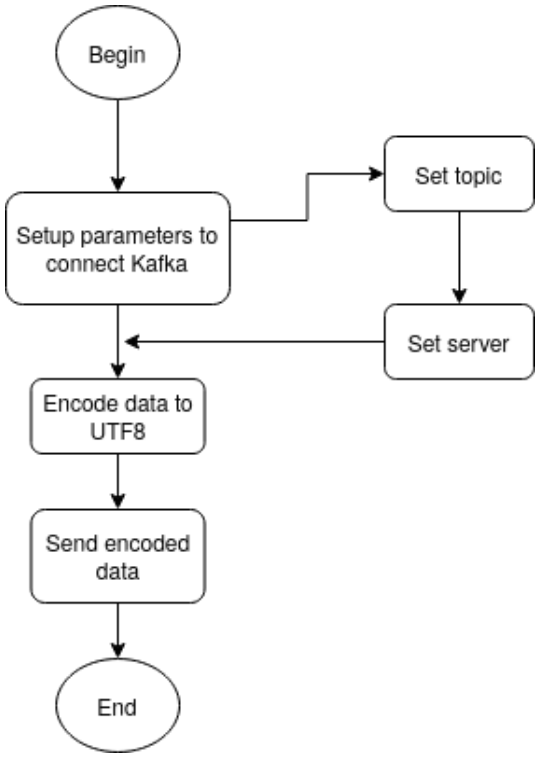

Figura 3.27: Diagrama de flujo del productor

La figura 3.27 muestra el funcionamiento completo del productor. Este productor es el encargado de configurar los parámetros de conexión y gestionar los datos previos a la transmisión de los mismos al sistema de mensajería *Kafka*.
Las principales funcionalidades son:

- Configurar los parámetros de conexión con *Kafka* para enviar los datos. Es necesario configurar el *topic* (canal donde se va a transmitir la información), el puerto y la dirección *IP* del servidor *Kafka*.
- Codificar los datos a un *array* de *bytes* para realizar el *streaming* de dicho mensaje.
- Envío de los datos codificados al canal correspondiente.

Otro dato importante a destacar es el uso del *topic* para la transmisión de datos a *Kafka*. Este dato es usado por los consumidores para conectarse a *Kafka*, de modo que sin el mismo, el consumidor no podrá recibir los datos. El *topic* es conocido únicamente por el productor que crea el *topic* y *Kafka*. Por ello, en este diseño, el productor es el encargado de transmitir el *topic* y la dirección *IP* de *Kafka* encriptados con la dirección pública del servicio solicitante (ver el apartado de control de acceso 3.3.3).

El diagrama 3.28 muestra el prototipo empleado para el consumidor. Como se puede observar, el consumidor realiza dos funciones principales:

- Configurar los parámetros de conexión con *Kafka* para recibir datos. Es necesario configurar el *topic* (canal de dónde recibir la información), el identificador del grupo al que pertenece la instancia y la dirección *IP* del servidor *Kafka*.
- Lanzar las instancias del consumidor que se conectarán a un determinado *topic* y grupo para recibir la información.

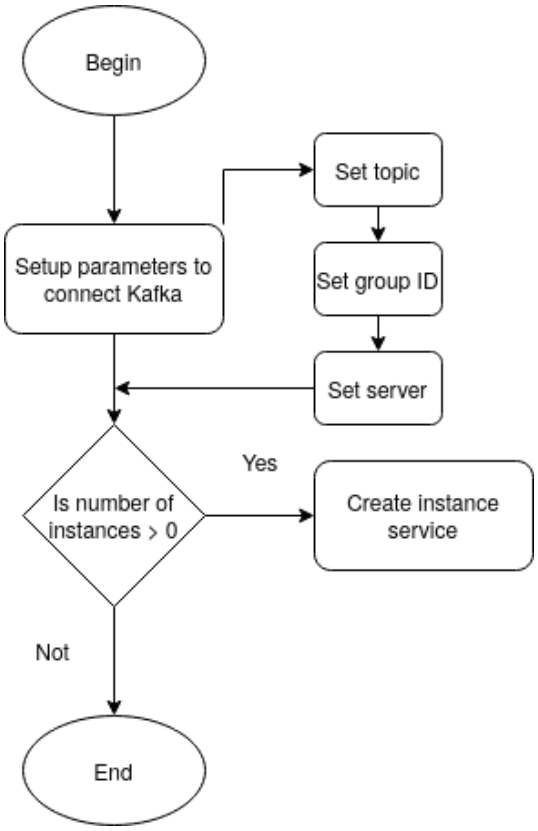

Figura 3.28: Diagrama de flujo del consumidor

3.4.4. Microservicios

Introducción

En esta sección se describen los servicios desarrollados en un sistema de vigilancia que detecta objetos. Para ello, se han creado cuatro servicios:

- Un servicio responsable de la captura y envío del vídeo en tiempo real.
- Un servicio detector de objetos encargado de procesar la imagen capturada del vídeo para enviar los datos de salida a otro servicio localizado en el nodo *fog*.
- Un servicio web que funciona como un servidor web para transmitir el vídeo en tiempo real.
- Un servicio de almacenamiento que guarda en una base de datos los datos procesados por un servicio en el nodo *edge*.

Servicios

Se presentan los diagramas de flujo de los cuatro servicios implementados.

En primer lugar, se muestra el diagrama de flujo 3.29 del servicio (instalado en un dispositivo IoT) encargado principalmente de transmitir el vídeo al servicio de mensajería para la distribución del mismo al resto de servicios.

Este servicio tiene un rol productor y dispone de las siguientes funcionalidades:

- Activación de la cámara.
- Lectura de los *frames* de la cámara.
- Codificación del *frame* del vídeo.

- Creación de un fichero con formato *JSON* con los elementos vistos en el apartado de microservicios 3.3.3:
  - El identificador de la cámara. Será utilizado para identificar la procedencia del vídeo.
  - El tiempo en milisegundos. Se empleará para identificar el instante en el que se capturó el *frame* del vídeo.
  - El *frame* del vídeo codificado.
- Transmisión de los datos empaquetados en un archivo con formato *JSON* al sistema de mensajería.

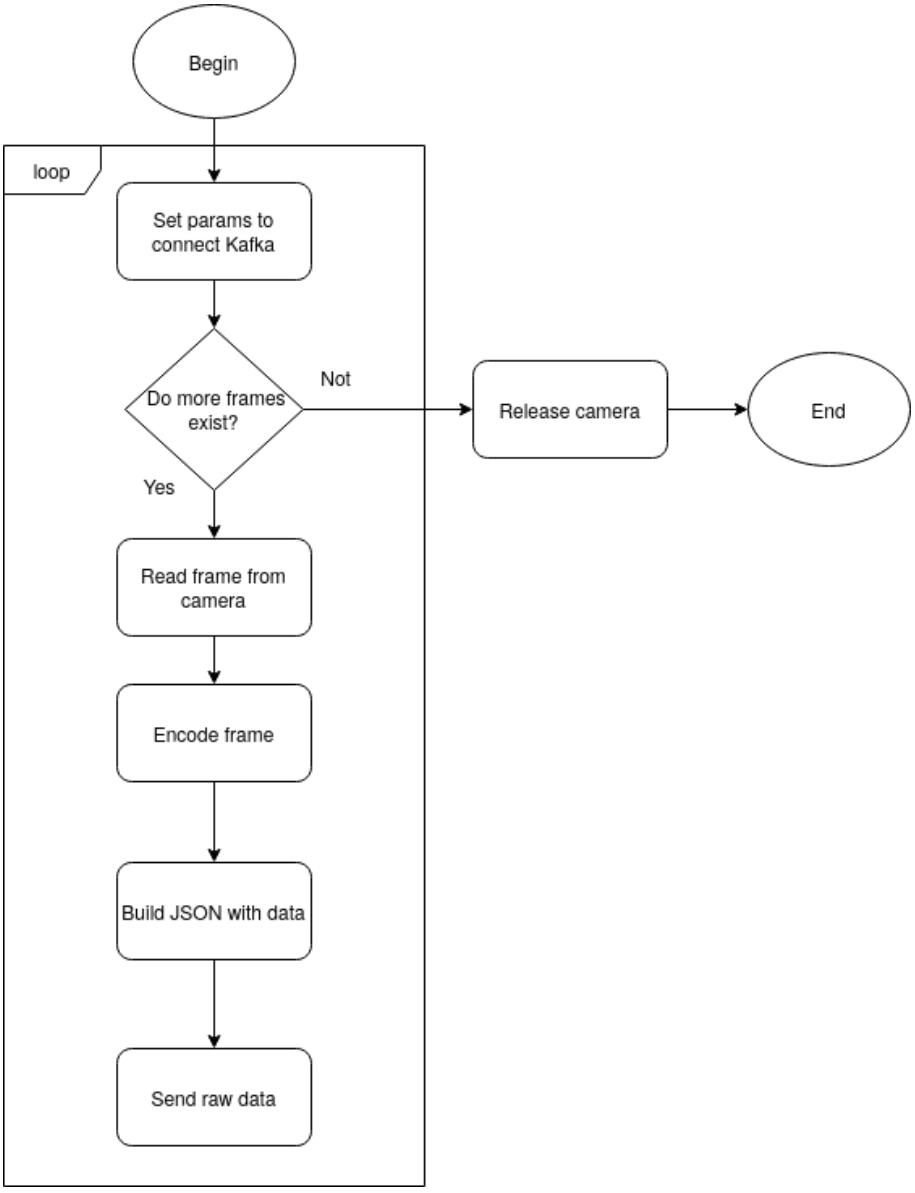

Figura 3.29: Diagrama de flujo del servicio productor del vídeo

En segundo lugar, se muestra el diagrama de flujo 3.30 del servicio (desplegado en el nodo *edge*) encargado de procesar el vídeo con el objetivo de detectar objetos para finalmente transmitir estos datos de detección a un servicio en el nodo *fog*.

Este servicio, que tiene un rol de consumidor, recibe los datos transmitidos por un cliente cámara. Estos datos, almacenados en un archivo con formato *JSON*, contienen la identificación de la cámara, el tiempo y el *frame* capturado en dicho instante. Posteriormente, el servicio decodifica

el *frame* y lleva a cabo un proceso de análisis (ver figura 3.31) basado en *machine learning* para detectar objetos en dicho *frame*. Este proceso de análisis genera unos parámetros de salida que identifican el objeto y que pueden ser empleados para monitorizar dicho objeto.

Este servicio también tiene rol de productor, puesto que envía los datos obtenidos a *Kafka* con otro *topic* con el objetivo final de transmitir estos datos (obtenidos en 3.31 y que son empaquetados en un archivo con formato *JSON*) a un servicio en el nodo *fog* para la monitorización del objeto.

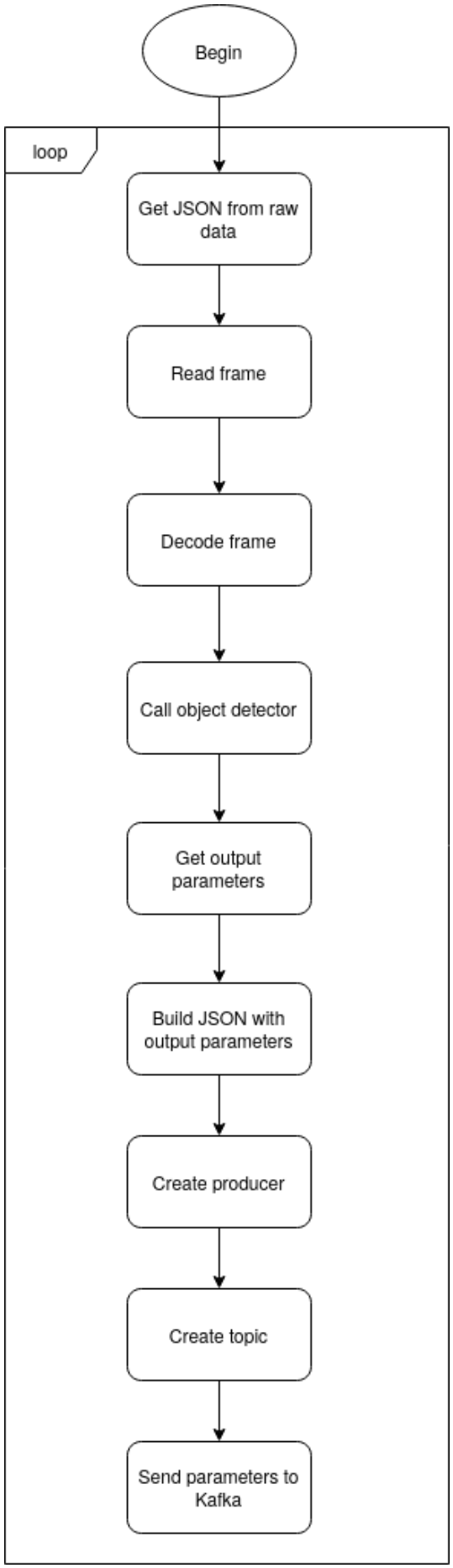

Figura 3.30: Diagrama de flujo del servicio detector de objetos

La figura 3.31 muestra de forma resumida el proceso de detección de objetos utilizando las librerías de *Tensorflow*. Para más información sobre este proceso de detección ver Tensorflow-API .

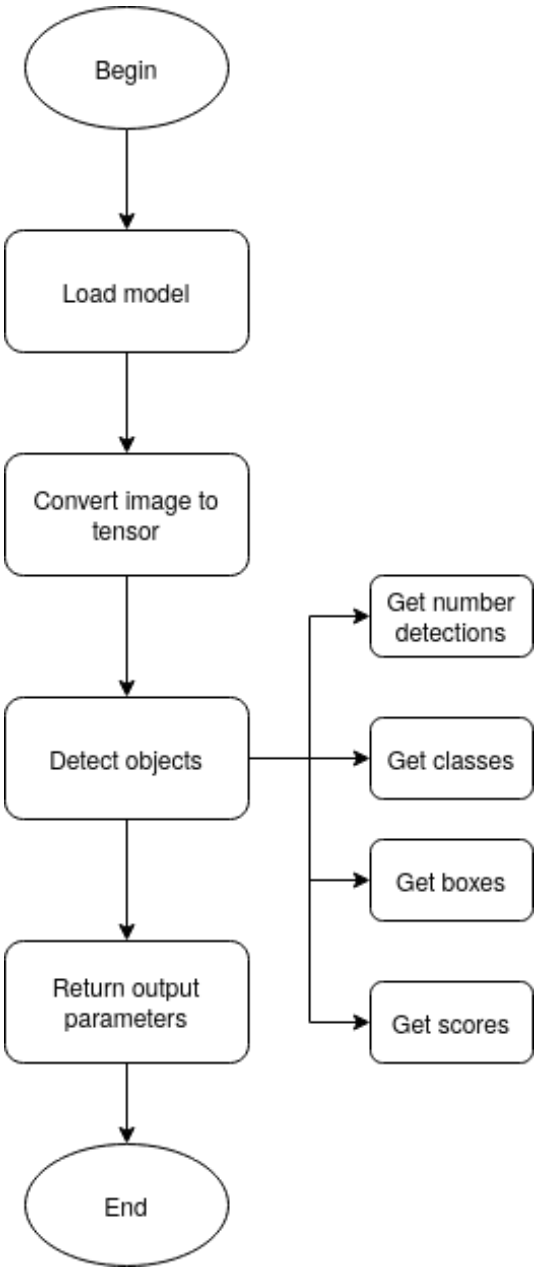

Figura 3.31: Diagrama de flujo del detector

El proceso genera los siguientes parámetros de salida:

- Número de objetos detectados en la imagen.
- La clase a la que pertenece el objeto.
- La precisión de detección del objeto. Este valor está dentro del rango [0, 1].
- La posición normalizada del objeto detectado. Esta posición viene delimitada por posición en "x" y en "y":
  - La posición mínima en el eje "x".
  - La posición máxima en el eje "x".

- La posición mínima en el eje “y”.
- La posición máxima en el eje “y”.

En tercer lugar, se muestra el diagrama de flujo 3.32 del tercer servicio (dispone de un rol de consumidor), desplegado en el nodo *edge*. Se trata de un servicio web encargado de proporcionar vídeo en tiempo real al usuario que lo solicite y de almacenar el *hash* del vídeo en una base de datos.

Como puede observarse en el diagrama 3.32, este servicio sigue un proceso similar al servicio anterior. En primer lugar, desempaqueta los datos almacenados en un archivo con formato *JSON* para obtener la identificación de la cámara, el tiempo y un *frame* del vídeo capturado en dicho instante.

Posteriormente, realiza dos procesos: el almacenamiento de los *frames* en un *buffer* para la visualización del vídeo en el cliente y el cálculo del *hash* del *frame* para su almacenamiento en una base de datos. Este último paso se realiza como una medida de protección de la integridad del vídeo (ver amenaza 2 en el apartado 3.3.5), por lo que cualquier intento de manipulación del vídeo puede ser detectado comparando de forma iterativa el *hash* de un *frame* en un instante determinado con el *hash* almacenado en la base de datos para el mismo instante.

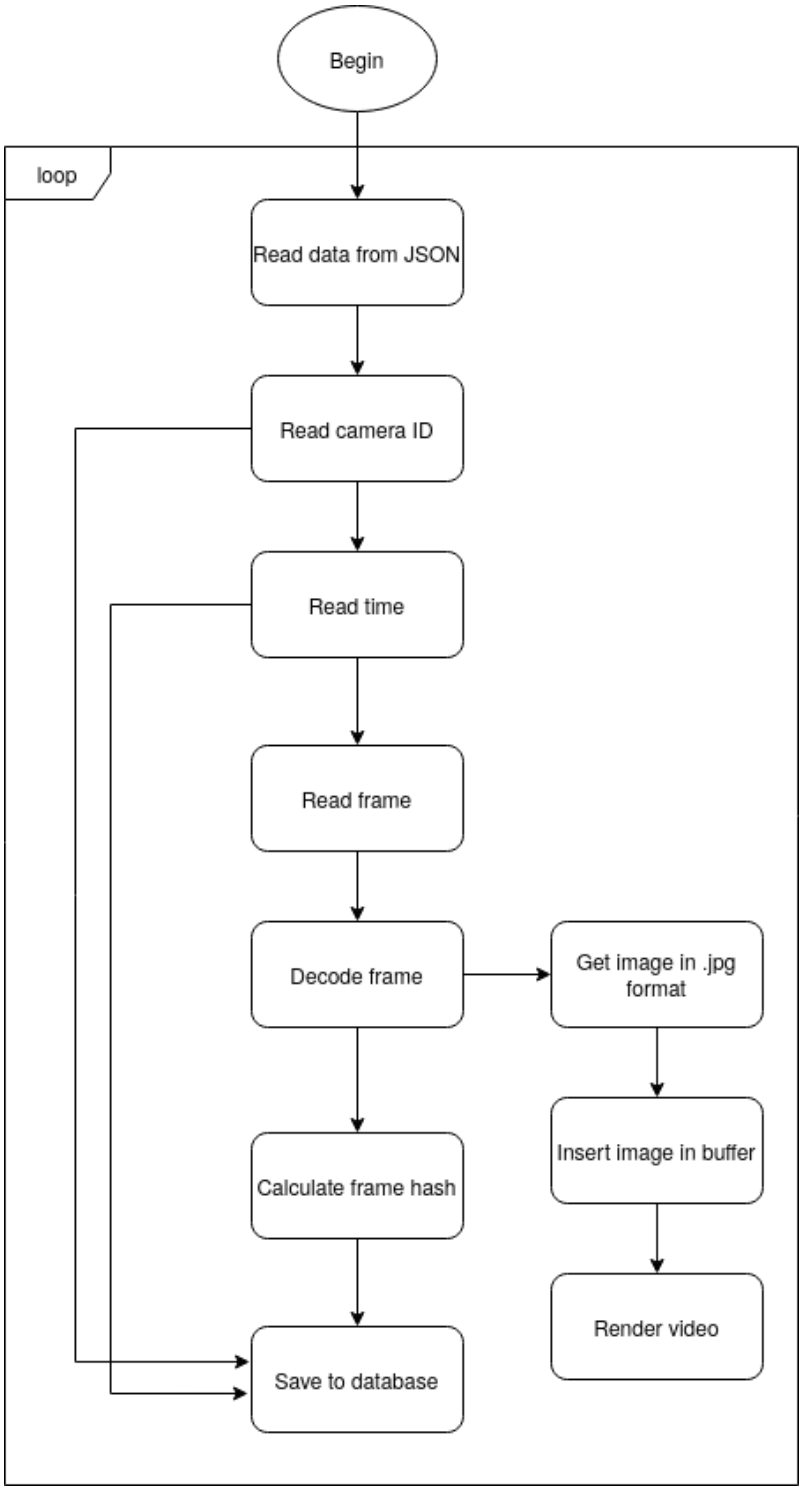

Figura 3.32: Diagrama de flujo del servicio web

El diagrama 3.33 muestra la comunicación síncrona entre el cliente y el servidor web utilizando el protocolo *HTTP/REST* para la visualización del vídeo en tiempo real.

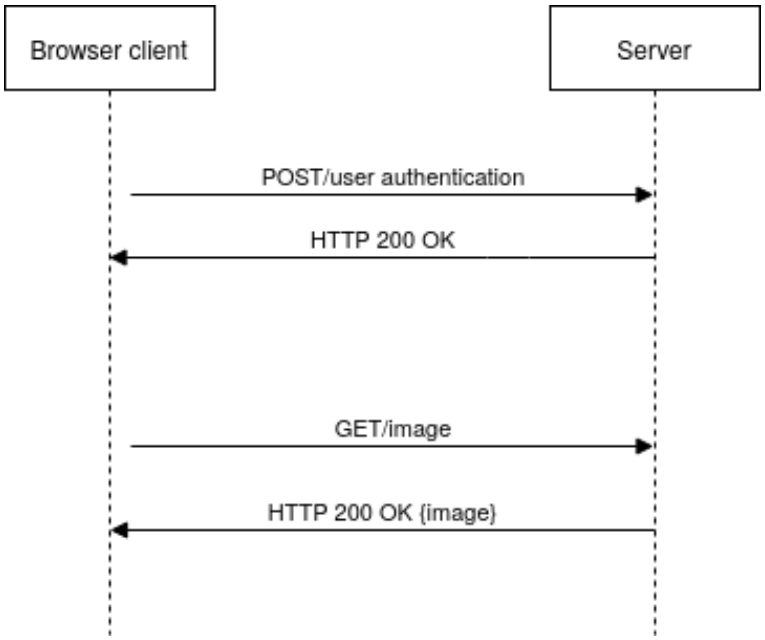

Figura 3.33: Diagrama del servicio de almacenamiento

La figura 3.34 muestra el diseño de la tabla *HASH_ VIDEO* para el almacenamiento de los *hashes* de un vídeo. Está compuesta por 3 campos:

- ***camera_ id***. Identificador único del cliente cámara. Este campo almacena los 4 dígitos del identificador.
- **time**. Instante en el que un *frame* es capturado por la cámara. Este tiempo tiene el formato "*Y-M-D H:M:S.f*" y se define como *VARCHAR* variable de longitud 30. Este campo y el anterior son definidos como clave primaria (*primary key*) de la tabla.
- ***hex_ hash_ image***. *Hash* de un *frame*. Campo definido como *VARCHAR* de longitud variable hasta 128 caracteres para almacenar *hashes* más seguros como el *sha512* que generan una huella digital de 128 caracteres.

Figura 3.34: Tabla *HASH_ VIDEO*

Por último, las figuras 3.35 y 3.36 muestran el diseño del cuarto servicio desarrollado. Este servicio con rol de consumidor está desplegado en el nodo *fog*. El objetivo de este servicio es almacenar en una base de datos los parámetros obtenidos por el servicio detector con la finalidad de monitorizar los objetos detectados. Los parámetros obtenidos son: el instante en el que un *frame* es capturado, la identificación de la cámara, el número de objetos detectados, las clases a la que pertenecen los objetos, el porcentaje de confianza de detección y la localización de los objetos detectados en el *frame*.

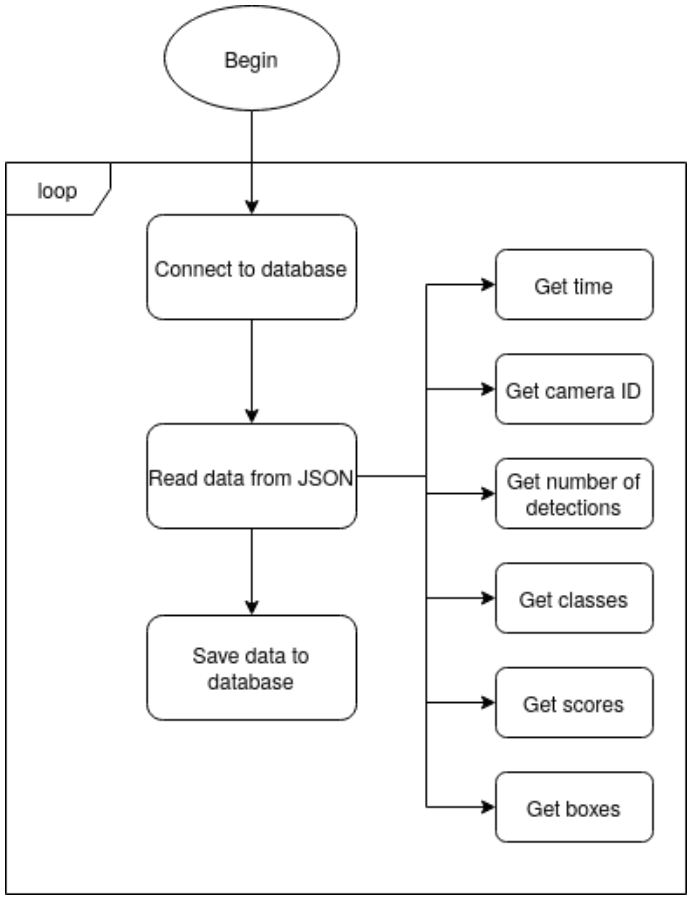

Figura 3.35: Diagrama de flujo del servicio de almacenamiento de datos

La figura 3.36 muestra el diseño de la tabla empleada para el almacenamiento de los datos enviados por el servicio detector de objetos. La tabla *PARAMS_FRAME* se compone de los siguientes campos:

- ***camera_id***. Identificador de la cámara. Consta de 4 dígitos.
- ***time***. Instante en el que el *frame* es capturado. Definido como *VARCHAR* variable hasta longitud máxima de 30 caracteres.
- ***detection_id***. Identificador del objeto detectado en un *frame*. Definido como un entero de 2 dígitos. Este campo junto con los campos *camera_id* y *time* se emplean como clave primaria de la tabla.
- ***class_detection***. Clases de los objetos detectados. Definido como un entero de 1 dígito.
- ***detection_score***. Indica el porcentaje de confianza de si una detección es correcta. Definido como un decimal de 10 dígitos hasta 5 decimales.
- ***box_x1***. Indica la posición mínima en el eje "x" del objeto detectado.
- ***box_x2***. Indica la posición máxima en el eje "x" del objeto detectado.
- ***box_y1***. Indica la posición mínima en el eje "y" del objeto detectado.
- ***box_y2***. Indica la posición máxima en el eje "y" del objeto detectado.

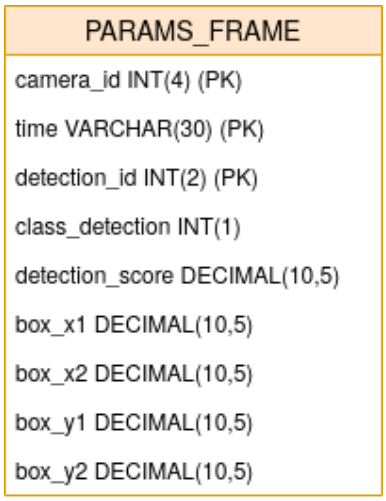

Figura 3.36: Tabla *PARAMS_FRAME*

Capítulo 4

Pruebas

4.1. Introducción

En este capítulo se lleva a cabo seis pruebas para demostrar la funcionalidad básica del sistema propuesto en el proyecto.

Para probar la funcionalidad del prototipo implementado, se han empleado diversas herramientas y recursos.

Respecto a las herramientas empleadas, se listan las más relevantes:

- ***Ganache***. Posibilita la ejecución de *Ethereum blockchain* en local. Permite al usuario el despliegue, ejecución e inspección de los *smart contracts*. *Ganache* proporciona unas direcciones virtuales con un balance inicial de 100 *ethers* (apartado 4.1).
- ***VirtualBox***. Software de virtualización de sistemas operativos. En estas pruebas se han utilizado 4 contenedores de este tipo para la instalación del sistema de mensajería *Kafka*, el servicio de detección de objetos, el servicio web y el servicio de almacenamiento.

  Las características de los contenedores son:

  - Sistema operativo: Ubuntu 18.04.
  - Memoria: 4096MB.
  - Número de procesadores: 2.
- ***PyCharm***. Entorno de desarrollo para *Python* que permite el despliegue de funcionalidad en entornos remotos.
- ***SQLite***. Base de datos para la gestión del almacenamiento de la información de las tablas *HASH_ VIDEO* Y *PARAMS_ FRAME*.

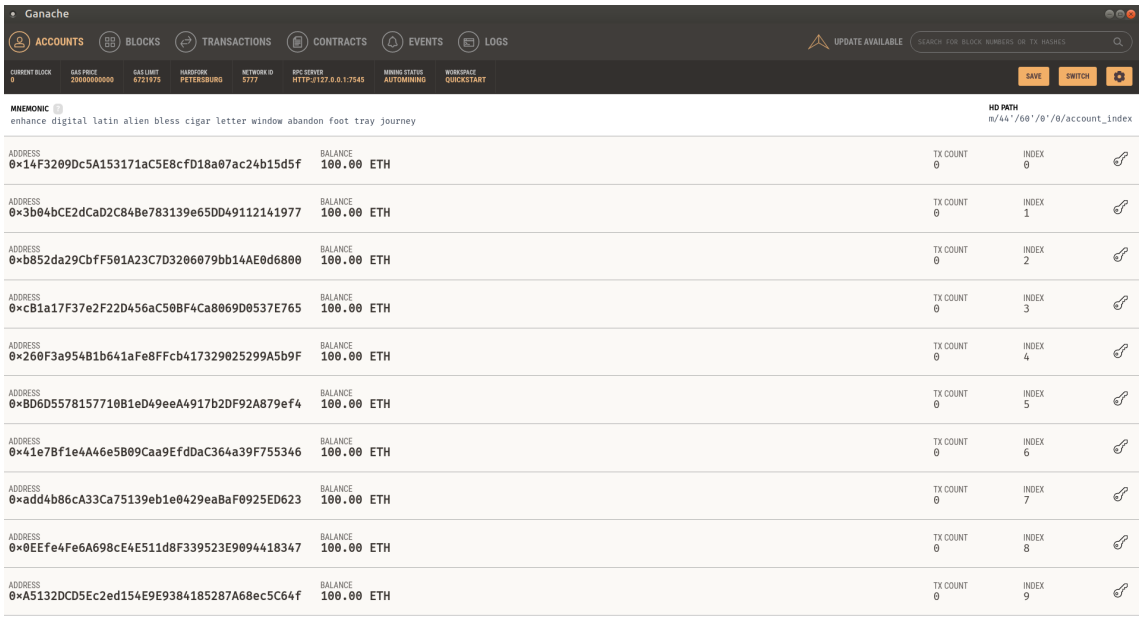

Figura 4.1: Estado inicial de las direcciones virtuales en *Ganache*

A continuación, se listan los recursos empleados:

- Dos portátiles para el nodo *edge* y el nodo *fog*, y un dispositivo *IoT*.
  Para el nodo *fog* se ha empleado un portátil con las siguientes características:

  - Modelo: Lenovo Z50-70.
  - Sistema operativo: *Ubuntu. Release 18.04.4 LTS (Bionic Beaver) 64-bit.*
  - Kernel: Linux 4.15.0-99-*generic* x86_64.
  - Memoria RAM: 8 GiB.
  - Procesador: Intel®Core™ i5-4210U CPU @ 1.70GHz x 4.

  Para el nodo *edge* se ha empleado un portátil con las siguientes características:

  - Modelo: MSI CX612QC
  - Sistema operativo: *Ubuntu. Release 18.04.4 LTS (Bionic Beaver) 64-bit.*
  - Kernel: Linux 5.30-46-*generic* x86_64.
  - Memoria RAM: 15.6 GiB.
  - Procesador: Intel®Core™ i7-4712MQ CPU @ 2.30GHz x 8.

  El dispositivo *IoT* empleado para la captura de vídeo cumple las siguientes características.

  - Modelo: *Raspberry Pi* 3 modelo B+.
  - CPU: 4x *ARM Cortex*-A53, 1.2GHz.
  - GPU: *Broadcom VideoCore IV*.
  - RAM: 1GB LPDDR2 (900 MHz).
  - Red: 10/100 *Ethernet*, 2.4GHz 802.11n *wireless*.
  - Incluye un módulo camera que se conecta al *Raspberry Pi* a través de un puerto serie.

La figura 4.2 muestra la representación de los recursos empleados para la realización de las pruebas.
El portátil de marca *MSI*, empleado como nodo *edge*, contiene los contenedores del sistema de mensajería, servicio web y el servicio de detección de objetos.

El portátil *Lenovo*, empleado como nodo *fog*, contiene el contenedor del servicio de almacenamiento y la herramienta *Ganache* para la ejecución de los *smart contracts* localmente.

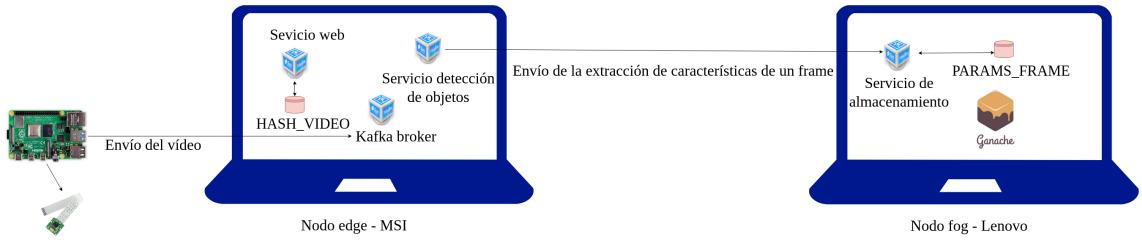

Figura 4.2: Esquema de los recursos empleados para la realización de las pruebas

4.2. Preparación del sistema

Se describe la configuración y los pasos previos a realizar antes de ejecutar las pruebas del sistema.

Los rangos de valores con las que se va a trabajar son los siguientes:

- El identificador de la cámara está dentro del rango [1 - 100].
- El identificador del nodo está dentro del rango [101 - 1000]. En estas pruebas se va a tener en cuenta únicamente el nodo *edge* y el nodo *fog*.
- El identificador del servicio está dentro del rango [1001 - 10000].

4.2.1. Despliegue de los verificadores

La prueba está compuesta por 3 verificadores para el registro de las direcciones virtuales de los servicios implementados:

- Servicio de detección de objetos instalado en el nodo *edge*.
- Servidor web instalado en el nodo *edge*.
- Servicio de almacenamiento que almacena en una base de datos las características extraídas por el detector.

La figura 4.4 muestra las transacciones de los *smart contracts* o contratos creados y desplegados en la red de *blokchain Ganache*.

Cada creación o despliegue de un *smart contract* en *blockchain* conlleva las operaciones de compilación del contrato, la generación del interfaz del contrato, la creación del *hash* que identifica la transacción, la dirección virtual del contrato y el número de *miners* empleados para la creación de un bloque en la red de *blockchain*.
En la figura 4.3 se muestra el proceso de despliegue de un contrato o *smart contract* en *blockchain*.

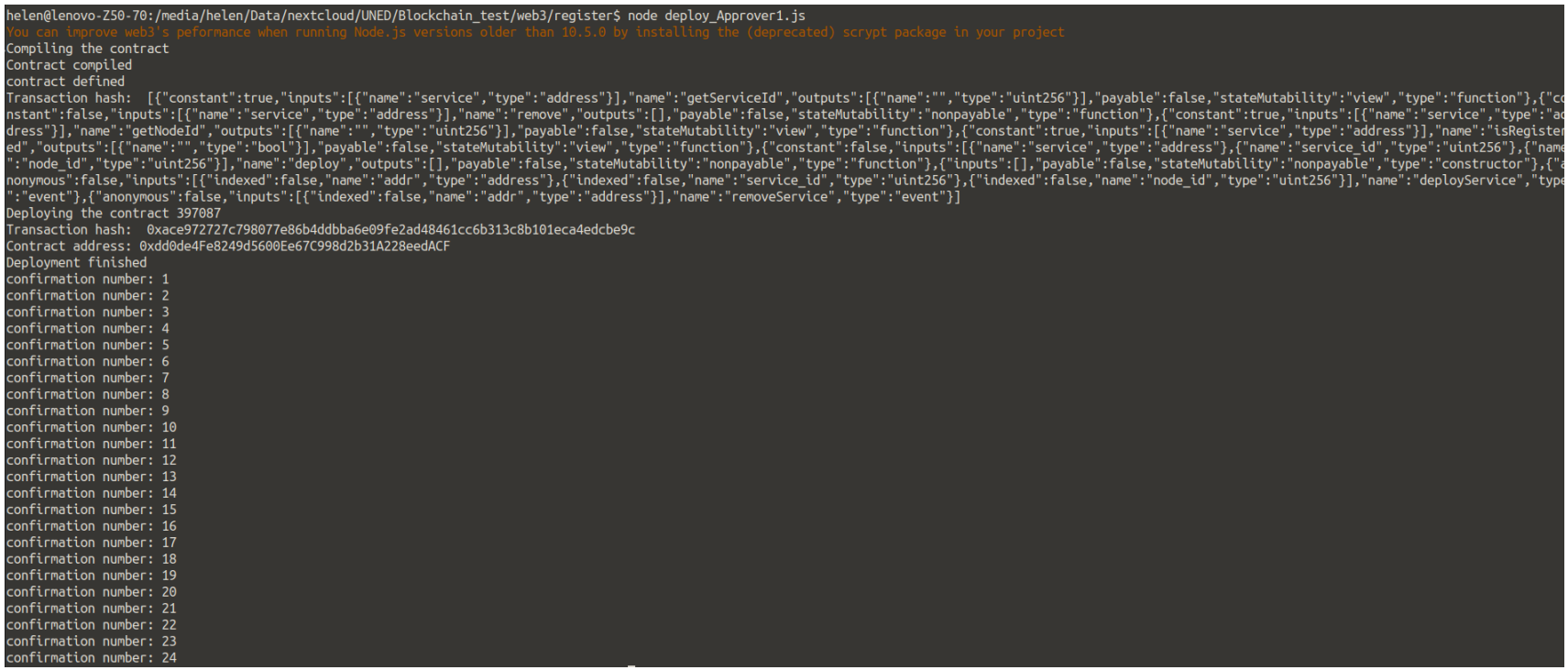

Figura 4.3: Despliegue de un *smart contract* en *blockchain Ganache*

Cada una de estas transacciones (figura 4.5) contiene el *hash* que identifica dicha transacción, la cuenta digital del verificador propietario del *smart contract*, la dirección virtual del *smart contract* creado, el gas[1]empleado en dicha transacción, el precio del gas[1], el límite del gas[1] y el número del bloque y el dato transmitido.

A continuación, se muestran las direcciones virtuales de los verificadores y de sus respectivos *smart contracts*:

	Verificador 1
Dirección virtual de la cuenta	0x14F3209Dc5A153171aC5E8cfD18a07ac24b15d5f
Dirección virtual del contrato	0xdd0de4Fe8249d5600Ee67C998d2b31A228eedACF

Tabla 4.1: Datos del primer verificador en la red de *blockchain*

	Verificador 2
Dirección virtual de la cuenta	0x3b04bCE2dCaD2C84Be783139e65DD49112141977
Dirección virtual del contrato	0xF2ef5385Da191202B706780C9916D1c5CfF94b50

Tabla 4.2: Datos del segundo verificador en la red de *blockchain*

	Verificador 3
Dirección virtual de la cuenta	0xb852da29CbfF501A23C7D3206079bb14AE0d6800
Dirección virtual del contrato	0x7A9A6188DBfBC740B05c48c46Fc702E9ed7D7bf6

Tabla 4.3: Datos del tercer verificador en la red de *blockchain*

[1]El gas es la unidad que mide el trabajo computacional requerido para ejecutar transacciones o *smart contracts*

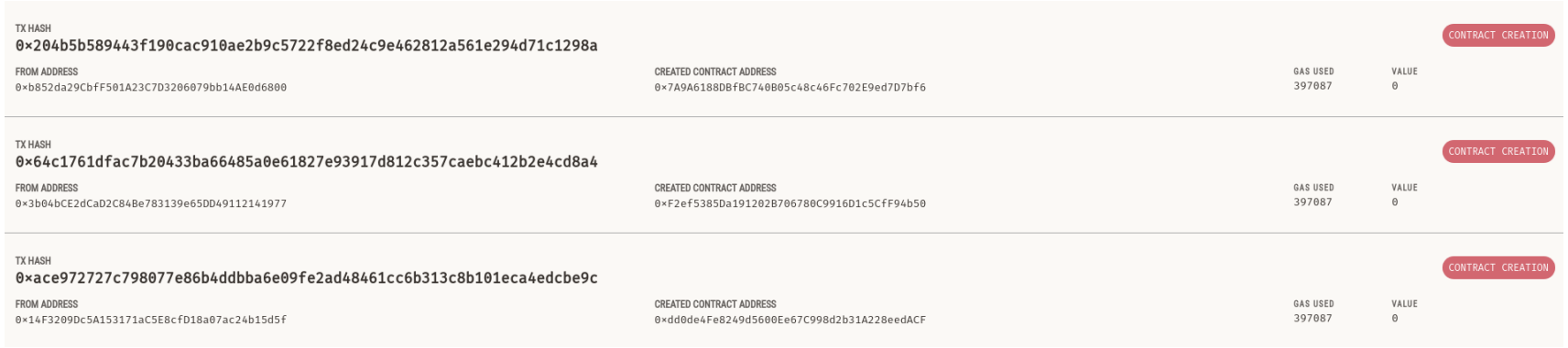

Figura 4.4: Despliegue de los verificadores en la red de *blockchain*

Figura 4.5: Datos de una transacción de la creación de un *smart contract*

4.2.2. Despliegue de los nodos

En esta sección, se realiza el despliegue de los nodos *fog* y *edge* en la red de *blockchain*. Como se ha comentado en la introducción, únicamente se va a tener en cuenta el nodo *fog* y nodo *edge* para las pruebas, ya que el uso de estos nodos es suficiente para demostrar la funcionalidad del prototipo.

Este proceso se realiza en dos pasos: la creación de los contratos y la asignación de los verificadores del registro del servicio.
En primer lugar, se despliega los *smart contracts* (figuras 4.6 y 4.7) de los nodos a la red de *blockchain*. En el caso del nodo *edge* el contrato contiene registrado en una lista interna los identificadores de las cámaras de las que el nodo es propietario.

A continuación, se muestran las direcciones virtuales de los nodos *edge* y *fog* y de sus respectivos contratos o *smart contracts*.

Nodo *edge*	
Dirección virtual del nodo	0x260F3a954B1b641aFe8FFcb417329025299A5b9F
Dirección virtual del contrato	0x4f0508B68318865e0714383f08852C66E231d9A9

Tabla 4.4: Datos del nodo *edge* en la red de *blockchain*

Nodo *fog*	
Dirección virtual del nodo	0xcB1a17F37e2F22D456aC50BF4Ca8069D0537E765
Dirección virtual del contrato	0xE2f782061FBC80818595Acad9C92245afAFaDb2E

Tabla 4.5: Datos del nodo *fog* en la red de *blockchain*

En segundo lugar, se asigna a cada nodo los 3 verificadores creados en el apartado anterior. Se puede observar en las figuras 4.9 y 4.10 las transacciones realizadas por cada nodo para asignar los verificadores en sus respectivos contratos *0x4f0508B68318865e0714383f08852C66E231d9A9* y *0xE2f782061FBC80818595Acad9C92245afAFaDb2E*.

4.2.3. Despliegue de la cámara

En esta prueba se realiza el despliegue del *smart contract* de la cámara en la red de *blockchain*. Este contrato es empleado para controlar el acceso a los recursos de la cámara. Ver figura 4.8.

Cliente cámara	
Identificador de la cámara	1
Dirección virtual de la cuenta	0xBD6D5578157710B1eD49eeA4917b2DF92A879ef4
Dirección virtual del contrato	0x86Ee1465835f578c2aAc0CdA50993483595B6562

Tabla 4.6: Datos de la cámara

Figura 4.6: Creación del contrato del nodo *edge*

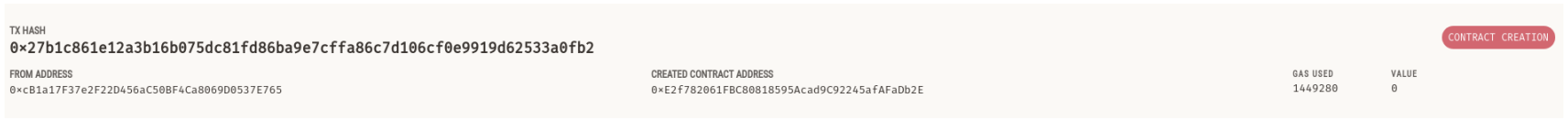

Figura 4.7: Creación del contrato del nodo *fog*

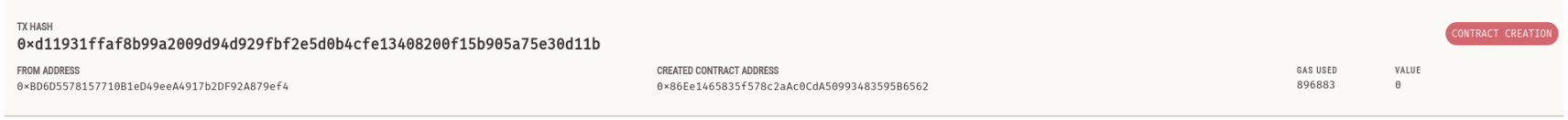

Figura 4.8: Creación del contrato de la cámara en *blockchain*

Figura 4.9: Asignación al nodo *edge* de los 3 verificadores del sistema

Figura 4.10: Asignación al nodo *fog* de los 3 verificadores del sistema

4.3. Prueba 1

La prueba 1 se encarga de comprobar el funcionamiento básico del prototipo diseñado.
Se compone de varias fases:

- Registro de los servicios en los nodos.
- Solicitud de acceso al vídeo.
- Recepción y gestión del vídeo.
- Solicitud de acceso a los datos.
- Recepción y gestión de los datos.

4.3.1. Registro de los servicios en los nodos

En relación con el apartado de seguridad (apartado 3.3.3), este proceso se realiza en dos partes.

La primera consiste en registrar las direcciones virtuales públicas de los servicios en los 3 verificadores o *approvers* del sistema. Para ello, se realiza la transacción en uno de los verificadores y, automáticamente, el registro de dichos servicios se realiza en el resto de verificadores a través de los *listeners* de cada verificador.

```
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/register$ node interact_Approver.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Deploy detector service address: 0x41e7Bf1e4A46e5B09Caa9EfdDaC364a39F755346 service id: 1001 node id: 101
Owner address: 0x14F3209Dc5A153171aC5E8cfD18a07ac24b15d5f
Contract address: 0xdd0de4Fe8249d5600Ee67C998d2b31A228eedACF
service_id:  1001  node_id:  101  registered:  true
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/register$ node interact_Approver1_deploy_fog_service.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Deploy detector service address: 0xadd4b86cA33Ca75139eb1e0429eaBaF0925ED623 service id: 1002 node id: 102
Owner address: 0x14F3209Dc5A153171aC5E8cfD18a07ac24b15d5f
Contract address: 0xdd0de4Fe8249d5600Ee67C998d2b31A228eedACF
service_id:  1002  node_id:  102  registered:  true
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/register$ node interact_Approver1_deploy_web_service.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Deploy detector service address: 0x0EEfe4Fe6A698cE4E511d8F339523E9094418347 service id: 1003 node id: 101
Owner address: 0x14F3209Dc5A153171aC5E8cfD18a07ac24b15d5f
Contract address: 0xdd0de4Fe8249d5600Ee67C998d2b31A228eedACF
service_id:  1003  node_id:  101  registered:  true
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/register$
```

Figura 4.11: Despliegue de los servicios en el verificador 1

El despliegue de todos los servicios se realiza en el verificador 1 (ver figura 4.11). Como se puede observar, el despliegue de dichos servicios se ha realizado con éxito, ya que el valor de la variable *registered* es "*true*" si el servicio es registrado con éxito o "*false*" si el servicio se ha registrado con estado pendiente o no ha sido registrado.

Con respecto al resto de verificadores se puede comprobar que el despliegue de los servicios en el segundo (ver figura 4.12) y tercer verificador (ver figura 4.13) también se han realizado con éxito.
En estas pruebas también se muestran direcciones virtuales de los elementos implicados en la transacción, es decir, la dirección virtual de los contratos de los verificadores, la dirección virtual de cada verificador y las direcciones virtuales de los servicios.

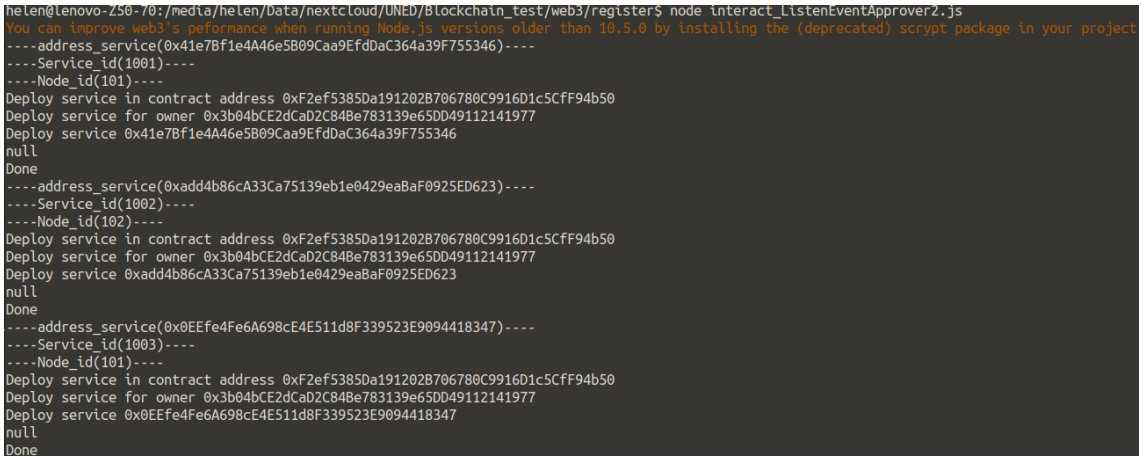

Figura 4.12: Despliegue de los servicios en el verificador 2

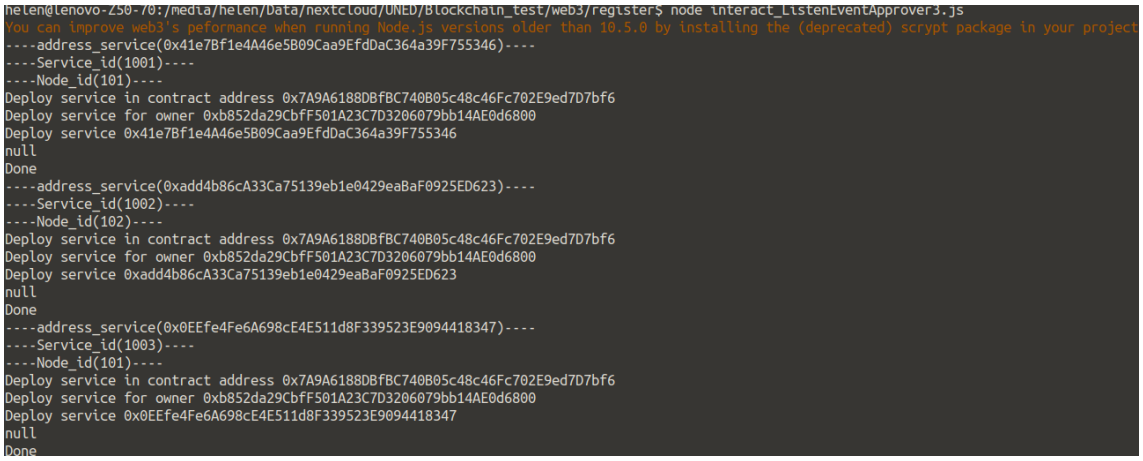

Figura 4.13: Despliegue de los servicios en el verificador 3

Los datos de los servicios desplegados en la red de *blockchain* se muestran en la tabla 4.7. Como se puede ver, se han desplegado dos servicios en el nodo *edge* con identificador 101 y un servicio en el nodo *fog*, cuyo identificador es 102.

Servicios

Nombre	Servicio	Nodo	Dirección virtual
Servicio detector	1001	101	0x41e7Bf1e4A46e5B09Caa9EfdDaC364a39F755346
Servicio web	1003	101	0x0EEfe4Fe6A698cE4E511d8F339523E9094418347
Servidor de almacenamiento	1002	102	0xadd4b86cA33Ca75139eb1e0429eaBaF0925ED623

Tabla 4.7: Datos de los servicios del sistema

El resultado de estas transacciones de registro también puede verificarse en *Ganache* (ver figuras 4.14, 4.15 y 4.16).

Figura 4.14: Transacciones del despliegue del servicio detector

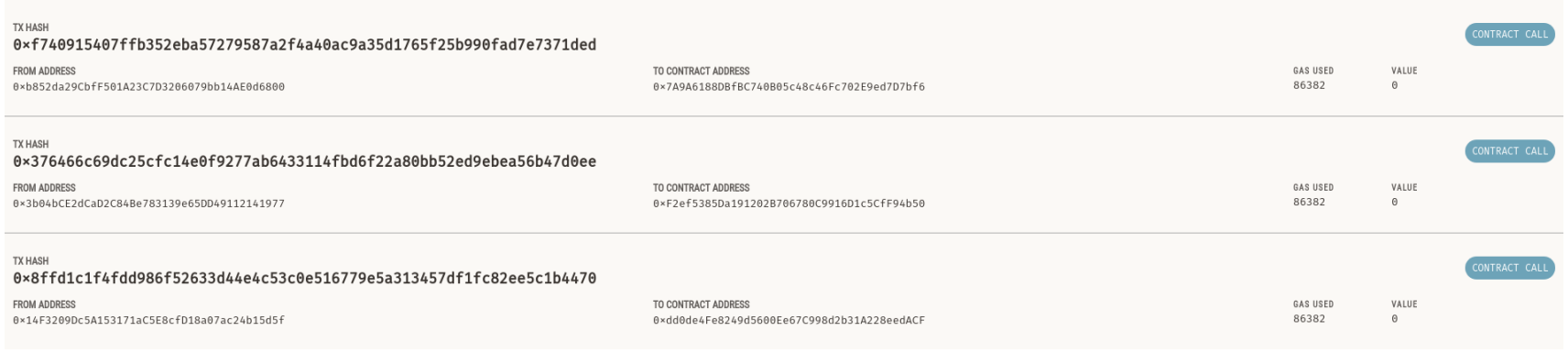

Figura 4.15: Transacciones del despliegue del servicio de almacenamiento

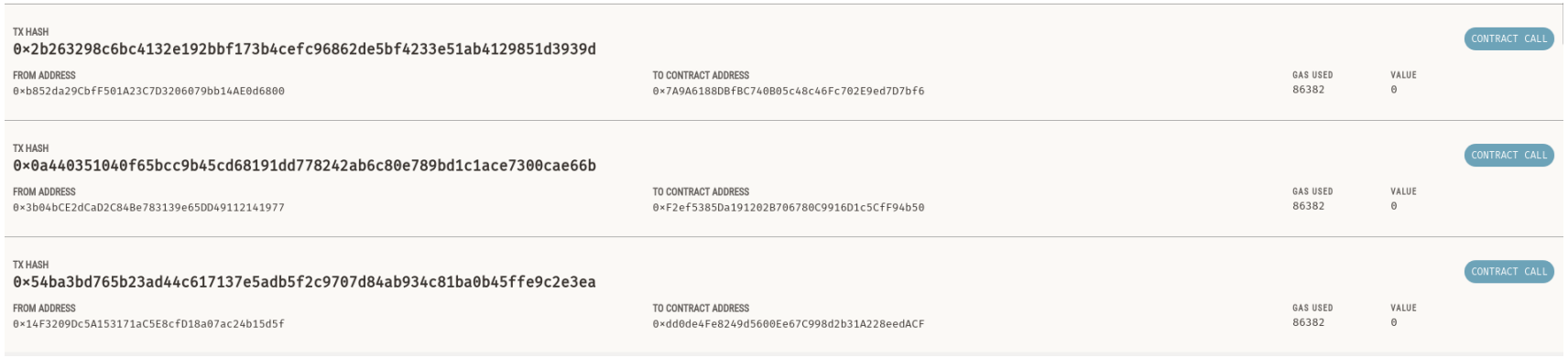

Figura 4.16: Transacciones del despliegue del servicio de web

Figura 4.17: Transacción del servicio detector de objetos

En la figura 4.17 se muestra con más nivel de detalle la transacción realizada para el verificador 1, *0x14F3209Dc5A153171aC5E8cfD18a07ac24b15d5f*, donde el campo *contract address* corresponde a la dirección del *smart contract* del verificador, el campo *sender address* corresponde a la dirección virtual de la cuenta del verificador y el dato resaltado corresponde al dato transmitido a *blockchain*, es decir, la dirección del servicio que se ha registrado en el verificador.

La segunda parte de este proceso es realizar el registro de la dirección virtual de los servicios en sus respectivos nodos. Dado que el registro de estos servicios se realizó con éxito en los verificadores, el proceso de registro en el nodo debe realizarse sin problemas.

El proceso de registro (al igual que el proceso de registro en los verificadores) puede ser realizado únicamente por el propietario del *smart contract*. Por lo tanto, en esta transacción, el gestor del nodo es el único que puede realizar este proceso. La transacción del nodo *edge* es realizada por la cuenta *0x260F3a954B1b641aFe8FFcb417329025299A5b9F* y la transacción en el nodo *fog* es realizada por la cuenta *0xcB1a17F37e2F22D456aC50BF4Ca8069D0537E765*.

A continuación, se muestra el proceso de registro realizado para los tres servicios con sus respectivas transacciones.

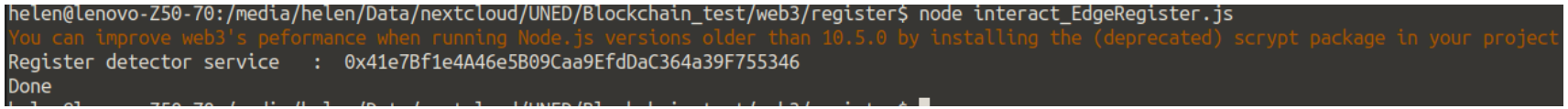

Figura 4.18: Registro del servicio detector en el nodo *edge*

Figura 4.19: Transacción del registro del servicio detector

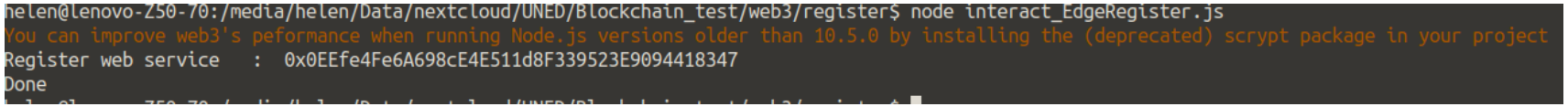

Figura 4.20: Registro del servicio web en el nodo *edge*

Figura 4.21: Transacción del registro del servicio web

```
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/register$ node interact_FogRegister.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Register service  :  0xadd4b86cA33Ca75139eb1e0429eaBaF0925ED623
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/register$
```

Figura 4.22: Registro del servicio de almacenamiento en el nodo *fog*

Figura 4.23: Transacción del registro del servicio de almacenamiento

4.3.2. Solicitud de acceso al vídeo

La solicitud de acceso al vídeo se divide en 2 partes (ver apartado 3.3.3):

- Solicitud del *token* al nodo *edge*.
- Solicitud de los datos de conexión al cliente cámara.

Solicitud del token

Se solicita al nodo *edge* la creación del *token ticket* en la red de *blockchain*. La figura 4.24 indica la creación del *ticket* (se resalta la dirección virtual).

La creación del *ticket* se realiza con los siguientes valores:

- Contador del *ticket*: 1.
- Derecho: *READ*.
- Dirección virtual del servicio que solicita el recurso: *0x41e7Bf1e4A46e5B09Caa9EfdDaC364a39F755346*. Esta dirección virtual corresponde al servicio detector.
- Identificador de la cámara: 1.
- Dirección del propietario del *ticket*: *0x260F3a954B1b641aFe8FFcb417329025299A5b9F* (gestor del nodo *edge*).
- Tiempo de expiración: 30 minutos.

Solicitud de los datos de conexión

En esta fase se solicita el acceso a los datos de conexión para la recepción de vídeo.

En primer lugar, el servicio detector realiza una petición de acceso al contrato del cliente cámara. Como se puede ver en la figura 4.26, esta petición se realiza con éxito, por lo que el contrato emite el evento "*AllowAccess*" con el identificador de la cámara y la dirección virtual del servicio.

En segundo lugar, el *listener* de este contrato recibe y gestiona el evento. Se encarga de enviar los datos de conexión y generar una transacción que sirve como recibo del acceso concedido. Estos dos últimos pasos están reflejados en la figura 4.27.

En la primera parte del proceso se crea el mensaje con el *topic*, la dirección *IP* y el puerto del *broker*. A continuación, el programa crea la firma digital del mensaje resaltado en color amarillo. En esta captura, también se aprecia la transacción realizada por el *smart contract* al emitir el evento "*AllowAccess*" con el identificador de la cámara y la dirección virtual del servicio.

Posteriormente, después de encriptar el mensaje con los datos de conexión y la firma digital, se transmiten estos datos vía *socket* al cliente del servicio (ver figura 4.28), el cual es el único que puede desencriptar el mensaje y la firma digital, y así verificar la autenticidad y la integridad de los datos. Para más detalle sobre este proceso ver apartado 3.4.2.

Finalmente, se genera una transacción con la firma digital calculada previamente que servirá como recibo de la concesión del acceso (ver figura 4.29). En esta transacción se muestran los datos del emisor y del servicio solicitante y la firma digital calculada en el paso anterior (ver figura 4.27).

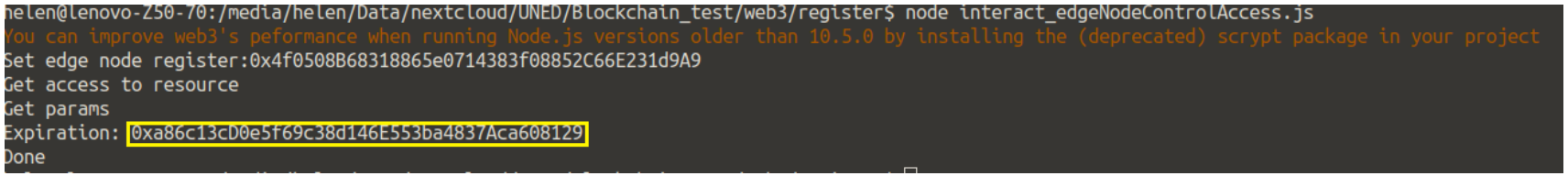

Figura 4.24: Creación de la dirección virtual del *ticket*

Figura 4.25: Transacción de la solicitud del *ticket*

Figura 4.26: Transacción de la solicitud de acceso del servicio detector

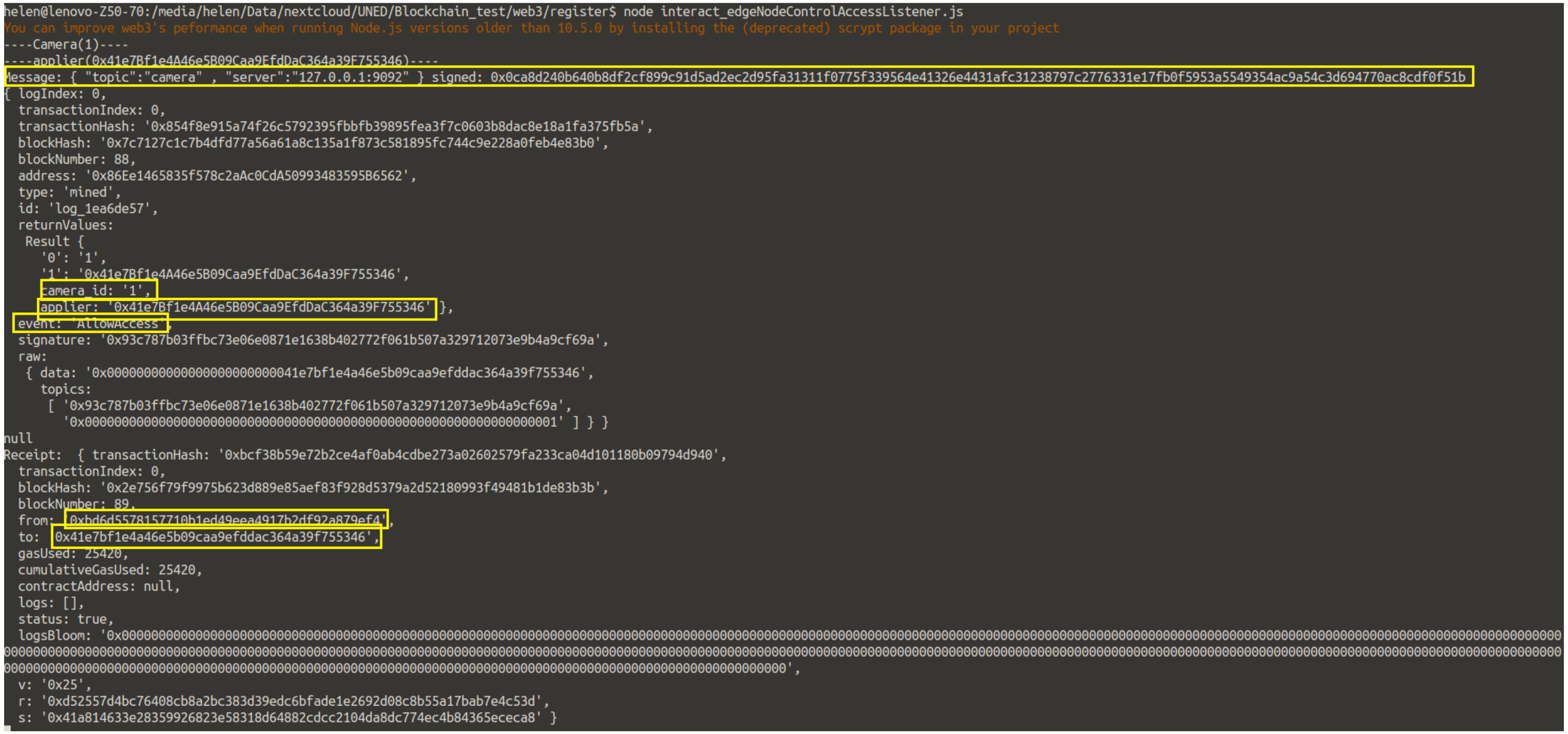

Figura 4.27: Concesión del acceso a los datos de conexión

Figura 4.28: Recepción de los datos de conexión por el servicio detector

Figura 4.29: Transacción del recibo de concesión de acceso

4.3.3. Recepción y gestión del vídeo

El servicio detector de objetos se subscribe como consumidor del vídeo capturado por la cámara con los datos de conexión recibidos en el paso anterior. Para ello, el servicio se conecta con el broker del sistema de mensajería de Kafka con la siguiente configuración:

- *Topic*: "camera".
- Identificador del grupo del servicio detector: "*object_ detection_ group*".
- Dirección *IP Kafka*: 192.168.1.70.
- Puerto: 9092.

En la figura 4.30 se muestra el objeto detectado del vídeo capturado por el Raspberry Pi 3.

Figura 4.30: Proceso de detección de un objeto

4.3.4. Solicitud de acceso a los datos

El proceso de obtención de los datos de conexión para acceder a los datos generados por el servicio detector es similar al proceso de control de acceso de vídeo, con la diferencia de que este proceso se realiza en una única fase y no se emplean *tokens* para la comprobación del acceso, sino que es controlado por el registro del servicio en el nodo y el acceso es proporcionado por el grafo de nodos en el lado del cliente del servicio detector.

En primer lugar, el servicio detector despliega el contrato (ver figura 4.31) que gestiona su control de acceso.

Figura 4.31: Creación del contrato de control de acceso del servicio

Figura 4.32: Solicitud de acceso del servicio en el nodo *fog*

Posteriormente, el servicio de almacenamiento solicita el acceso a dicho contrato realizado con éxito (ver figura 4.32). En esta transacción, el servicio de almacenamiento solicita acceso al servicio detector a través del contrato del servicio detector que emite el evento "*AllowAccess*" con los datos del servicio solicitante (servicio localizado en el nodo *fog*) y los identificadores de los nodos que se utilizarán para verificar si el nodo *fog* tiene acceso al nodo *edge* en el grafo de nodos.

Finalmente, el *listener* del control de acceso del servicio recibe el evento y empaqueta los datos necesarios para la conexión al sistema de mensajería para la recepción de los datos extraídos por este servicio.

La figura 4.34 muestra cómo se recibe el evento "*AllowAccess*" con los datos del nodo *fog* y nodo *edge*. De igual forma, se puede visualizar la transacción 4.35 que se empleará como recibo de la concesión de acceso del servicio detector (*0x41e7Bf1e4A46e5B09Caa9EfdDaC364a39F755346*) al servicio de almacenamiento (*0xadd4b86cA33Ca75139eb1e0429eaBaF0925ED623*).

La figura 4.36 muestra el envío de los datos de conexión encriptados al servicio de almacenamiento. Se verifica que al desencriptar el mensaje el emisor de dicho mensaje es el servicio detector.

4.3.5. Recepción y gestión de los datos

Se muestra el acceso a los datos de salida obtenidos por el servicio detector.

El servicio de almacenamiento se conecta al sistema de mensajería *Kafka* con los datos de conexión obtenidos en el apartado anterior:

- *Topic*: "analyze".
- Dirección *IP Kafka*: 192.168.1.70.
- Puerto: 9092.

Como se puede observar en la figura 4.33, este servicio de almacenamiento localizado en el nodo *fog* recibe los datos del objeto detectado (ver apartado 3.4.4) y los almacena en una base de datos con la finalidad de monitorizar dicho objeto. La descripción de los datos almacenados en esta base de datos se localiza en el apartado (ver apartado 3.4.4).

```
sqlite> select * from params_frame;
1|2020-05-17 17:55:58.075754|1|[52]|[0.6036014]|0.5024751|0.24259868|0.7645995|0.6362791
1|2020-05-17 17:55:58.819539|1|[52]|[0.58357304]|0.50306827|0.24074487|0.76348203|0.6365882
1|2020-05-17 17:55:59.417776|1|[52]|[0.54533756]|0.5037003|0.24106699|0.76302546|0.6358766
1|2020-05-17 17:55:55.294839|1|[52]|[0.56191504]|0.50357735|0.24260598|0.76411664|0.6352545
1|2020-05-17 17:55:55.368480|1|[52]|[0.5843575]|0.5030892|0.24150822|0.7638459|0.6351017
1|2020-05-17 17:55:56.339441|1|[52]|[0.56907886]|0.50321984|0.24275264|0.7640207|0.6366339
1|2020-05-17 17:55:56.559517|1|[52]|[0.5474493]|0.5035604|0.24201062|0.764382|0.6359433
sqlite>
```

Figura 4.33: Almacenamiento de las características del objeto detectado en el nodo *edge*

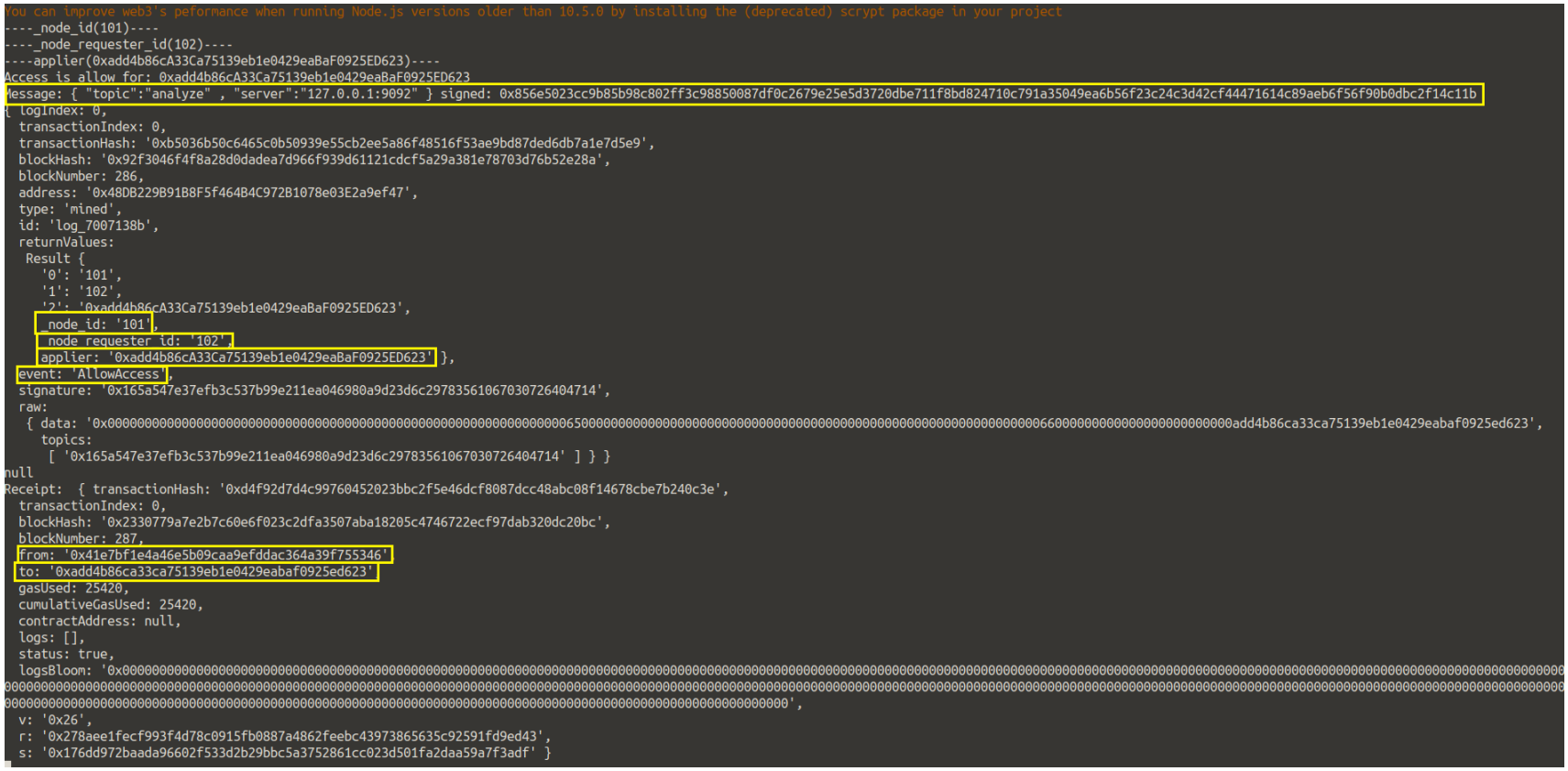

Figura 4.34: Concesión del acceso al servicio de almacenamiento

Figura 4.35: Transacción del recibo generado por el servicio detector

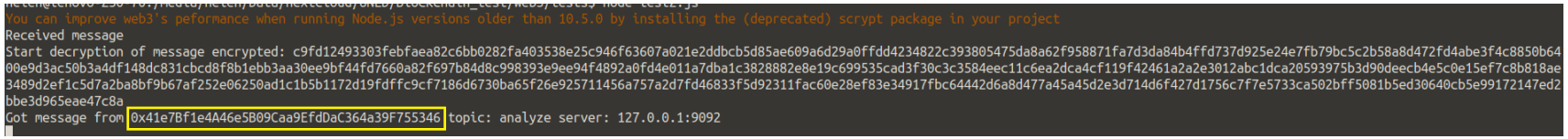

Figura 4.36: Recepción de los datos de conexión por el servicio de almacenamiento

4.4. Prueba 2

El objetivo de la prueba 2 consiste en comprobar el proceso de visualización de vídeo en tiempo real.

El servicio *web* debe tener acceso a los datos de conexión para conectarse al *broker Kafka* y poder así recibir el vídeo para ponerlo a disposición del usuario para su visualización.

El proceso de control de acceso para el servicio *web* sigue el mismo flujo que la prueba del apartado 4.3. El proceso se realiza en dos fases: solicitud de la generación del *ticket* y la solicitud de acceso.

En la primera fase, el nodo *edge* genera el *ticket* para el servicio *web* (ver figura 4.37). La transacción de este proceso se muestra en la figura (ver figura 4.38).

A continuación, se realiza con éxito la solicitud de acceso al contrato de la cámara 4.39.

Los detalles de esta concesión de acceso los podemos visualizar en la imagen 4.40. En este proceso de control de acceso se llevan a cabo las siguientes acciones:

- Recepción y gestión del evento emitido por el contrato de la cámara. La cámara emite el evento "*AllowAccess*" con la dirección virtual del servicio web y el identificador de la cámara.
- Creación del mensaje que contiene los datos necesarios para que un consumidor pueda conectarse a *Kafka*.
- Empaquetación de los datos de conexión. Este proceso consiste en la encriptación y generación de la firma digital. Para más detalle ver el apartado 3.4.2.
- Generación de la transacción con el envío de la firma digital (ver figura 4.41) obtenida previamente para reflejar en la red de *blockchain* el proceso de envío de los datos de conexión.

Por otro lado, la imagen 4.42 muestra la recepción de los datos de conexión por parte del cliente del servicio solicitante. Como se puede ver en esta imagen, el emisor de los datos se trata de la cámara que concede el acceso.

Este servicio se conecta con el sistema de mensajería empleando los datos de conexión recibidos:

- *Topic*: "camera".
- Dirección *IP Kafka*: 192.168.1.70.
- Puerto: 9092.

```
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_edgeNodeControlAccess.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Set edge node register:0x4f0508B68318865e0714383f08852C66E231d9A9
Get access to resource
Get params
Ticket: 0xCDeDBCe98F78CCce948d9Dc128590aF60CC8b548
```

Figura 4.37: Generación del *ticket* del servicio *web*

← BACK **TX 0×84d034e39f5b100f5c340587710359d08607596976786d3cc265ba2b23659bf0**

SENDER ADDRESS	TO CONTRACT ADDRESS	
0×260F3a954B1b641aFe8FFcb417329025299A5b9F	0×4f0508B68318865e0714383f08852C66E231d9A9	CONTRACT CALL

VALUE	GAS USED	GAS PRICE	GAS LIMIT	MINED IN BLOCK
0.00 ETH	339485	20000000000	600000	308

TX DATA

0×ebd6b1860000000000000000000000000eefe4fe6a698ce4e511d8f339523e90944183470000000000000000000000000000000000000000000000000000000000000001

Figura 4.38: Transacción de la generación del *ticket* del servicio *web*

← BACK **TX 0×66988c37476e3b42859dc8b688b855980e74e4eea532d4ca096ad88946737250**

SENDER ADDRESS	TO CONTRACT ADDRESS	
0×0EEfe4Fe6A698cE4E511d8F339523E9094418347	0×86Ee1465835f578c2aAc0CdA50993483595B6562	CONTRACT CALL

VALUE	GAS USED	GAS PRICE	GAS LIMIT	MINED IN BLOCK
0.00 ETH	102404	20000000000	600000	310

TX DATA

0×f7ca8431

Figura 4.39: Solicitud de acceso a la cámara

Figura 4.40: Concesión del acceso a los datos de conexión del servicio web

Figura 4.41: Transacción del recibo de concesión de acceso del servicio web

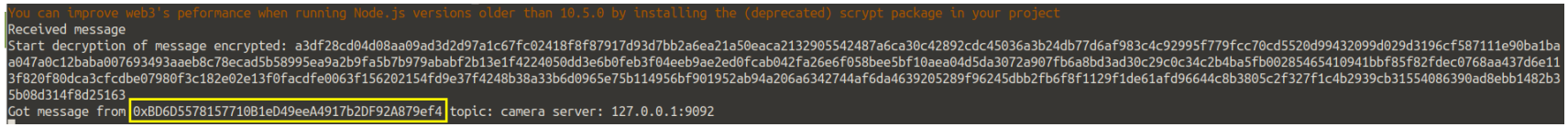

Figura 4.42: Recepción de los datos de conexión por el servicio web

El funcionamiento básico del servicio *web* es mostrado a continuación.

En el apartado de microservicios, se describió su funcionalidad: la visualización del vídeo en tiempo real y el almacenamiento del *hash* de los *frames* del vídeo en una base de datos.

La figura 4.43 muestra la página de inicio del servicio *web* para el acceso al vídeo en tiempo real. El usuario que quiera acceder a la visualización del vídeo, debe seleccionar la opción de "*Streaming*" que le redirigirá a otra *url* para la recepción del mismo (figura 4.44).

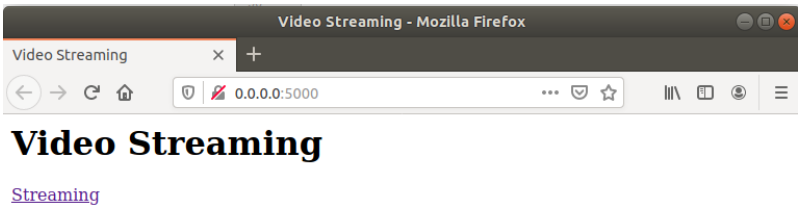

Figura 4.43: Página de inicio del servicio *web*

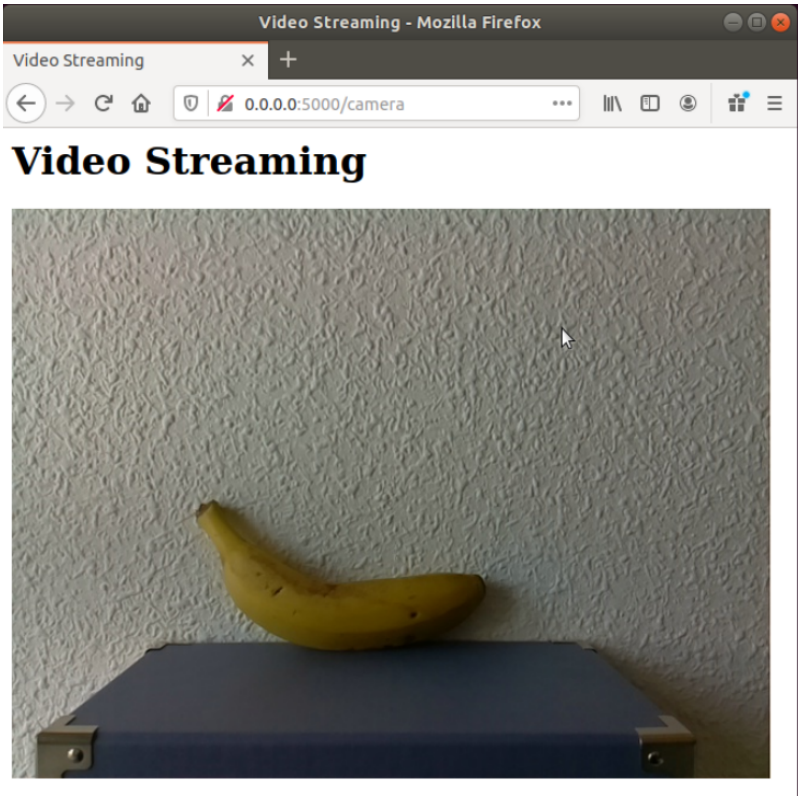

Figura 4.44: *Streaming* en tiempo real del vídeo capturado por el *Raspberry Pi 3*

Por otro lado, este servicio lleva a cabo el almacenamiento del *hash* de cada *frame* en una base de datos (figura 4.45) como sistema de seguridad ante cualquier posible modificación del mismo. Para más información sobre los campos de esta tabla ver el apartado 3.4.4.

Figura 4.45: Almacenamiento del *hash* de cada *frame* del vídeo capturado

4.5. Prueba 3

La prueba 3 consiste en verificar la seguridad del sistema en el supuesto de que los datos del contrato de un verificador sean comprometidos. Para probar esta características, se realiza el despliegue en un único verificador de un servicio de prueba con cuenta digital *0xA5132DCD5Ec2ed154E9E9384185287A68ec5C64f*.

En la figura 4.46 se muestra el despliegue de este servicio con el identificador 1004 en el nodo edge (101)

```
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_Approver1_deploy_service.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Deploy detector service address: 0xA5132DCD5Ec2ed154E9E9384185287A68ec5C64f service id: 1004 node id: 101
Owner address: 0x14F3209Dc5A153171aC5E8cfD18a07ac24b15d5f
Contract address: 0xdd0de4Fe8249d5600Ee67C998d2b31A228eedACF
service_id:  1004  node_id:  101  registered:  true
Done
```

Figura 4.46: Despliegue del servicio de prueba en el verificador 1

A continuación, se realiza el proceso de registro del servicio en el nodo *edge*. Como se puede ver en la figura 4.47, el proceso de registro no es realizado correctamente, por lo que el servicio es registrado con estado pendiente y la consulta del estado de registro será falsa ("*false*") (figura 4.48).

```
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_edgeNodeListener.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
----applier(0xA5132DCD5Ec2ed154E9E9384185287A68ec5C64f)----
{ logIndex: 0,
  transactionIndex: 0,
  transactionHash: '0x570e56660a4e132809be9163ee99b2676c52e44e9b51c11b8187ddaff6c9ebd1',
  blockHash: '0x6ed4cfbf4a833aef68d11168ff89194ac6c912d960f89f58453763dd36a0a355',
  blockNumber: 281,
  address: '0x4f0508B68318865e0714383f08852C66E231d9A9',
  type: 'mined',
  id: 'log_20315043',
  returnValues:
   Result {
     '0': '0xA5132DCD5Ec2ed154E9E9384185287A68ec5C64f',
     addr: '0xA5132DCD5Ec2ed154E9E9384185287A68ec5C64f' },
  event: 'UpdateService',
  signature: '0x5b17fda697d7a133c7f1b0e52273c1423d5b24aa9a7b18560fa73109b22a5fa4',
  raw:
   { data: '0x000000000000000000000000a5132dcd5ec2ed154e9e9384185287a68ec5c64f',
     topics:
      [ '0x5b17fda697d7a133c7f1b0e52273c1423d5b24aa9a7b18560fa73109b22a5fa4' ] } }
null
```

Figura 4.47: Proceso de registro en el nodo *edge* fallido

```
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_EdgeRegister.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Register service   :  0xA5132DCD5Ec2ed154E9E9384185287A68ec5C64f
Register service is registered :  false
Done
```

Figura 4.48: Consulta del estado de registro del servicio de prueba

4.6. Prueba 4

La prueba 4 verifica la validez del *token ticket* para un número máximo de usos. Para realizar esta prueba, se crea un *ticket* para el servicio *web* con un contador máximo de 3 usos.
En primer lugar, se solicita al nodo *edge* la generación del *ticket*. En la figura 4.49 se muestra la dirección virtual del *ticket* generado y en la figura 4.50 se muestra la transacción de la creación de dicho *ticket*.

```
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_EdgeRegister.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
 Apply access camera: 1 service:0x0EEfe4Fe6A698cE4E511d8F339523E9094418347
 Ticket generated: 0x7FEe06B8ABb531C17f6995efa9d1B73c6f4d8be5
Done
```

Figura 4.49: Creación del *ticket* para el servicio *web*

← BACK

TX 0×0ab29974b88a5e28b558594eada8a70c1ebd84af802582286145814477e22dce

SENDER ADDRESS 0×260F3a954B1b641aFe8FFcb417329025299A5b9F		TO CONTRACT ADDRESS 0×4f0508B68318865e0714383f08852C66E231d9A9		CONTRACT CALL
VALUE 0.00 ETH	GAS USED 339485	GAS PRICE 20000000000	GAS LIMIT 600000	MINED IN BLOCK 250

TX DATA

0×ebd6b1860000000000000000000000000eefe4fe6a698ce4e511d8f339523e90944183470000000000000000000000000000000000000000000000000000000000000001

Figura 4.50: Transacción de la creación del *ticket*

Una vez el *ticket* es creado, se realiza varias veces la solicitud de acceso con dicho *ticket*. La figura 4.51 muestra varias acciones:

- La asignación del contador del *ticket* a 3.
- La solicitud de acceso iterativo al cliente cámara.

El *ticket* sólo puede ser reutilizado 3 veces, pasado este límite el *ticket* es destruido por el contrato de la cámara.

Los intentos de uso del *ticket* se pueden verificar en las transacciones 4.52, 4.53, 4.54 y 4.55. En cada transacción se visualiza la solicitud del servicio *web* *0x0EEfe4Fe6A698cE4E511d8F339523E9094418347* al contrato de la cámara y el recibo de la concesión de acceso de la cámara (*0xBD6D5578157710B1eD49eeA4917b2DF92A879ef4*) al servicio web. Sin embargo, en la última transacción sólo se muestra la transacción de solicitud de acceso, pero no se genera ningún recibo, ya que no se le concede el acceso debido a que el *ticket* ya no es válido.

```
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_EdgeRegister.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
 Apply access camera: 1 service:0x0EEfe4Fe6A698cE4E511d8F339523E9094418347
 Ticket generated: 0xA8fb850B3B4490f48Df33Be88E5B330D0236bE82
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_Ticket.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
 Set counter: 3
 Counter defined: 3
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_edgeNodeControlAccess.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Get access to resource
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_Ticket.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
 Counter: 2
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_edgeNodeControlAccess.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Get access to resource
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_Ticket.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
 Counter: 1
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_edgeNodeControlAccess.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Get access to resource
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_Ticket.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
 Counter: 0
Done
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_edgeNodeControlAccess.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Get access to resource
Error found
Error: Returned error: VM Exception while processing transaction: revert Ticket does not exist
    at Object.ErrorResponse (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/web3-core-helpers/src/errors.js:29:16)
    at /media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/web3-core-requestmanager/src/index.js:140:36
    at XMLHttpRequest.request.onreadystatechange (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/web3-providers-http/src/index.js:96:13)
    at XMLHttpRequestEventTarget.dispatchEvent (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/xhr2-cookies/dist/xml-http-request-event-target.js:34:22)
    at XMLHttpRequest._setReadyState (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/xhr2-cookies/dist/xml-http-request.js:208:14)
    at XMLHttpRequest._onHttpResponseEnd (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/xhr2-cookies/dist/xml-http-request.js:318:14)
    at IncomingMessage.<anonymous> (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/xhr2-cookies/dist/xml-http-request.js:289:61)
    at emitNone (events.js:111:20)
```

Figura 4.51: Proceso de reutilización del *ticket*

Figura 4.52: Primer intento con éxito de uso del *ticket*

Figura 4.53: Segundo intento con éxito de uso del *ticket*

Figura 4.54: Tercer intento con éxito de uso del *ticket*

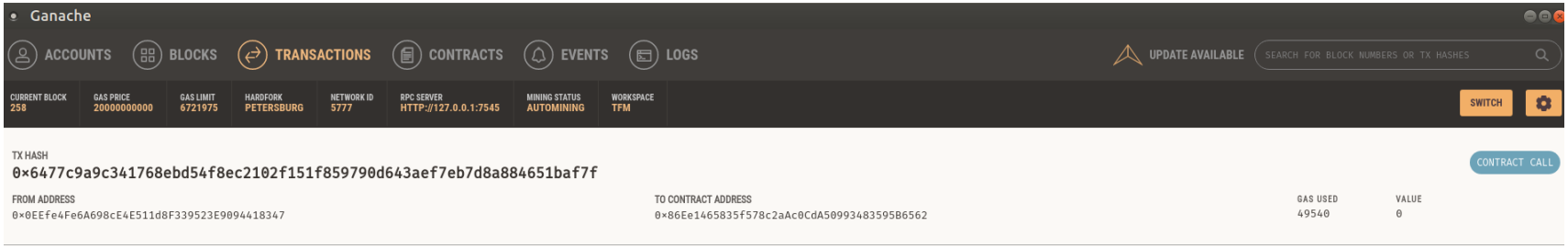

Figura 4.55: Cuarto intento fallido de uso del *ticket*

4.7. Prueba 5

La prueba 5 tiene como finalidad comprobar la seguridad del control de acceso cuando usuarios con cuenta en la red de *blockchain* intentan realizar operaciones a las que no están autorizados. Por ejemplo, en la figura 4.56 se muestra como un usuario intenta registrar una dirección virtual en un nodo, pero el contrato genera un error, ya que la cuenta que realiza dicha operación no es la propietaria del contrato.

4.8. Prueba 6

La prueba 6 consiste en ejecutar dos instancias del servicio de detección de objetos con los siguientes datos de conexión:

- *Topic*: camera
- Dirección *IP Kafka*: 127.0.0.1
- Puerto: 9092
- Id del grupo en *Kafka*: "*object_ detection_ group*"

Las imágenes 4.57 y 4.58 muestran cómo estos procesos tienen los identificadores 140147470714624 y 139863551346432 respectivamente.

```
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$ node interact_EdgeRegister.js
You can improve web3's peformance when running Node.js versions older than 10.5.0 by installing the (deprecated) scrypt package in your project
Register service   :  0xA5132DCD5Ec2ed154E9E9384185287A68ec5C64f
Error capturado
Error: Returned error: VM Exception while processing transaction: revert Only owner can register a service
    at Object.ErrorResponse (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/web3-core-helpers/src/errors.js:29:16)
    at /media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/web3-core-requestmanager/src/index.js:140:36
    at XMLHttpRequest.request.onreadystatechange (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/web3-providers-http/src/index.js:96:13)
    at XMLHttpRequestEventTarget.dispatchEvent (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/xhr2-cookies/dist/xml-http-request-event-target.js:34:22)
    at XMLHttpRequest._setReadyState (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/xhr2-cookies/dist/xml-http-request.js:208:14)
    at XMLHttpRequest._onHttpResponseEnd (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/xhr2-cookies/dist/xml-http-request.js:318:14)
    at IncomingMessage.<anonymous> (/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests/node_modules/xhr2-cookies/dist/xml-http-request.js:289:61)
    at emitNone (events.js:111:20)
    at IncomingMessage.emit (events.js:208:7)
    at endReadableNT (_stream_readable.js:1064:12)
    at _combinedTickCallback (internal/process/next_tick.js:138:11)
    at process._tickCallback (internal/process/next_tick.js:180:9)
helen@lenovo-Z50-70:/media/helen/Data/nextcloud/UNED/Blockchain_test/web3/tests$
```

Figura 4.56: Intento fallido de registro al ser ejecutado por un usuario no autorizado

```
(kafka_python) helen@ubuntu-1804:~/programs/Apache-Kafka-Video-Streaming/consumer$ python3 detector_consumer.py
adding consumers
starting process id:  140147470714624
raspberry
1591027500.882278
detection
2020-06-01 18:32:37.663390: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
2020-06-01 18:32:37.828363: I tensorflow/core/platform/profile_utils/cpu_utils.cc:94] CPU Frequency: 2394455000 Hz
2020-06-01 18:32:37.831230: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x7f7674c37d60 executing computations on platform Host. Devices:
2020-06-01 18:32:37.831270: I tensorflow/compiler/xla/service/service.cc:175]   StreamExecutor device (0): <undefined>, <undefined>
2020-06-01 18:32:38.221243: W tensorflow/compiler/jit/mark_for_compilation_pass.cc:1483] (One-time warning): Not using XLA:CPU for cluster because envvar TF_XLA_FLAGS=--t
f_xla_cpu_global_jit was not set.  If you want XLA:CPU, either set that envvar, or use experimental_jit_scope to enable XLA:CPU.  To confirm that XLA is active, pass --vm
odule=xla_compilation_cache=1 (as a proper command-line flag, not via TF_XLA_FLAGS) or set the envvar XLA_FLAGS=--xla_hlo_profile.
model loaded
[<tf.Tensor 'image_tensor:0' shape=(None, None, None, 3) dtype=uint8>]
```

Figura 4.57: Instancia 1 del servicio de detección


```
(kafka_python) helen@ubuntu-1804:~/programs/Apache-Kafka-Video-Streaming/consumer$ python3 detector_consumer.py
adding consumers
starting process id:  139863551346432
raspberry
1591027500.5553174
detection
2020-06-01 18:37:56.845563: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
2020-06-01 18:37:56.942132: I tensorflow/core/platform/profile_utils/cpu_utils.cc:94] CPU Frequency: 2394455000 Hz
2020-06-01 18:37:56.942697: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x7f3454c38520 executing computations on platform Host. Devices:
2020-06-01 18:37:56.942855: I tensorflow/compiler/xla/service/service.cc:175]   StreamExecutor device (0): <undefined>, <undefined>
2020-06-01 18:37:57.308534: W tensorflow/compiler/jit/mark_for_compilation_pass.cc:1483] (One-time warning): Not using XLA:CPU for cluster because envvar TF_XLA_FLAGS=--t
f_xla_cpu_global_jit was not set.  If you want XLA:CPU, either set that envvar, or use experimental_jit_scope to enable XLA:CPU.  To confirm that XLA is active, pass --vm
odule=xla_compilation_cache=1 (as a proper command-line flag, not via TF_XLA_FLAGS) or set the envvar XLA_FLAGS=--xla_hlo_profile.
model loaded
[<tf.Tensor 'image_tensor:0' shape=(None, None, None, 3) dtype=uint8>]
```

Figura 4.58: Instancia 2 del servicio de detección

Capítulo 5

Conclusiones y líneas de investigación futuras

5.1. Conclusiones

Se concluye que en este proyecto se ha alcanzado su objetivo principal, consistente en el diseño y la construcción de un prototipo de un sistema de videovigilancia basado en microservicios y *blockchain*.

A continuación, se detalla cada uno de los objetivos secundarios que han servido para alcanzar el objetivo principal.

En primer lugar, se ha conseguido construir un sistema de videovigilancia empleando la arquitectura de microservicios. No existe una única solución para el diseño de la arquitectura, pero en este proyecto se considera que se aporta un diseño que ofrece las características propias y generalmente habituales de este tipo de arquitecturas orientada a servicios. Estas características son:

- **Escalabilidad y Flexibilidad**. En comparación con un sistema monolítico propio de este tipo de sistemas, la arquitectura de microservicios permite fácilmente la actualización y la creación de nuevos servicios, puesto que cada servicio se ejecuta en su propio proceso y con su diseño se busca reducir al máximo el acoplamiento, manteniendo una fuerte cohesión entre los componentes del servicio.
- **Disponibilidad**. Se pueden ejecutar múltiples instancias de los servicios, de modo que si una instancia no está disponible o hay saturación de carga de trabajo se puede ejecutar más instancias del mismo servicio, mejorando su disponibilidad y su rendimiento.
- **Cohesión**. Característica propia de los microservicios, donde se busca su modularidad teniendo en cuenta la lógica de negocio en su creación.
- **Acoplamiento**. Se busca reducir el acoplamiento al máximo. Para ello, se emplea un sistema de mensajería para el envío de mensajes. En consecuencia, los servicios de este sistema no tendrán que compartir ni funcionalidad, ni recursos al estar completamente desacoplados.
- **Mantenimiento**. El desarrollo de los servicios y su despliegue se pueden realizar de forma independiente. Es posible el uso de procesos automatizados, facilitando el mantenimiento de los servicios.
- **Compatible con la estrategia de descentralización**. La descentralización de la funcionalidad y el almacenamiento de datos se realiza en diversos nodos de las cuatro capas del modelo propuesto.

En segundo lugar, en el diseño del sistema propuesto se ha conseguido construir un sistema de control de acceso basado en la tecnología *blockchain*. *Blockchain* se trata de una tecnología reciente que cada vez va tomando más presencia en el control de acceso de sistemas, donde la privacidad y la seguridad son importantes.

Las principales características que se obtienen con el uso de esta tecnología son:

- **Transparencia**. Todas las operaciones que se realicen en la red de *blockchain* vía *smart contracts* o a través de transacciones o eventos quedan reflejadas en *blockchain*. Por lo que, cualquier intento de registro, ya sea en los verificadores o en los nodos, u operaciones relacionadas con la concesión de acceso a cualquier cuenta digital se podría comprobar fácilmente en la red de bloques.
- **Seguridad**. Por su propia naturaleza blockchain ha demostrado ser una tecnología muy segura. Además, con el uso de la clave par digital (clave pública y clave privada), encriptación y otros mecanismos de seguridad como los *smart contracts*, se consigue en este proyecto los objetivos de confidencialidad, autenticación, no repudio e integridad del dato.
- **Descentralización**. En comparación con otros sistemas de seguridad, donde generalmente toda la información está almacenada en un único servidor y todo el procesamiento de datos es centralizado, esta tecnología nos permite descentralizar el control de acceso a través de las cuatro capas mencionadas anteriormente. Por ello, en esta solución el control de acceso al vídeo y a los datos está controlado por las cámaras y por los servicios del sistema. En consecuencia, cada servicio o recurso podrá aplicar medidas de seguridad adicionales si lo requiere necesario. Y en el supuesto de que se comprometa el control de acceso o datos de un servicio o recurso no afectará a los elementos restantes al estar distribuidos.

En tercer lugar, se considera que el uso de los *smart contracts* junto con las transacciones y las operaciones de firma digital y encriptación mejora la seguridad del sistema del servicio de vigilancia propuesto. Estos mecanismos de seguridad nos permiten alcanzar la protección de la privacidad y la integridad del dato deseados (ver apartado 3.3.5):

- **Protección de las cuentas digitales**. Aunque el robo de las cuentas digitales depende en gran medida del conocimiento y del uso realizado por parte del usuario, *blockchain* ofrece técnicas para mejorar su protección, por ejemplo, mediante el uso de frases de recuperación de contraseña (*passphrase*). Por otro lado, en el diseño del sistema propuesto también se han tomado medidas sobre las operaciones "sensibles" como es el registro de un servicio.

  En el caso del registro de las cuentas de los servicios cuyos usuarios pueden acceder a los distintos recursos del sistema, se ha aumentado la seguridad del proceso de registro mediante el uso de verificadores, donde la seguridad aumentará conforme se incrementa el número de los mismos. De esta manera, se protege el sistema ante cualquier ataque que comprometa los datos de un determinado nodo con cuenta digital en la red, robo de una cuenta o se haga un uso indebido de privilegios.
- **Protección de la integridad del vídeo y de los datos**. Se consigue mediante el uso de las siguientes metodologías:
  - **Uso del control de acceso**. El acceso al vídeo y a los datos está únicamente restringido a los usuarios autorizados que han superado con éxito el control de acceso. En la concesión de este acceso, se envían los datos de conexión del *Kafka broker* y del *topic* al servicio que lo solicite, donde el proceso de solicitud de servicios en la capa *edge* se valida por medio de un *ticket* que tiene validez temporal. En el caso de servicios en el nodo *fog* y el nodo *cloud*, la solicitud se valida por medio de permisos gestionados por un grafo de nodos donde la dirección de acceso de recursos va desde el nodo raíz hasta el nodo final.
  - **Descentralización de los datos**. A diferencia de otros sistemas, la información de los datos extraídos por el vídeo se encuentra distribuida en diferentes bases de datos a lo largo de las distintas capas.
  - **Generación de los *hashes* del vídeo**. El *hash* de cada *frame* del vídeo se almacena en una base de datos. De esta forma, se controla que dicha información no haya sido manipulada.
- **Protección de la red de *blockchain***. Esta tecnología, al ser una tecnología emergente, se ve expuesta constantemente a ataques que buscan nuevas debilidades. Debido a ello, esta

tecnología reduce sus debilidades buscando nuevas soluciones a estas amenazas.
Por ejemplo, el ataque del 51 % se ve reducido, debido a que plataformas como *Ethereum* quieren implantar otro mecanismo, *proof of stake*, que reemplace el actual proceso de *mining*, *proof of work*, donde a diferencia del *proof of work*, el nuevo mecanismo no da recompensas al primer minero que resuelve el problema.
Por otro lado, otros riesgos pueden verse reducidos empleando redes privadas como el que se ha propuesto en este proyecto o redes de tipo consorcio que son redes parcialmente privadas con un número considerable de nodos en la red.

- **Protección de la comunicación**. Se consigue mediante la encriptación de los datos a transmitir con la clave pública del destinatario, de modo que el receptor del mensaje es el único que puede desencriptarlo y, además, puede comprobar la integridad y la autenticación del mensaje con la firma digital, que, a su vez, se transmite en el mensaje.
  En el supuesto de que estos datos se pierdan o se comprometan durante el proceso de comunicación o que el servicio repudie la recepción del mensaje, el sistema puede verificar en cualquier momento los accesos concedidos consultando la firma digital, el emisor y receptor del mensaje en las transacciones que se generaron cuando se concedió el acceso a un recurso determinado.

Por último, se considera que con el diseño de 4 capas propuesto se solventan algunos problemas descritos en el apartado 3.3.2. Se comentan dos casos considerados principales:

- **Incremento de dispositivos *IoT***. Este tipo de modelo de 4 capas es ideal en sistemas que dispongan de varios dispositivos *IoT*, que pueden llegar a ser decenas o cientos de dispositivos. Por lo tanto, con un modelo de 3 capas, cada dispositivo debería realizar operaciones de alto coste computacional, además de otras funcionalidades, que permitan la comunicación con otros servicios. Un modelo de 3 capas se considera válido para sistemas con pocos dispositivos *IoT* o con dispositivos *IoT* con un procesador lo suficientemente potente que permita realizar operaciones más costosas. En este último caso, aumentaría el coste y la complejidad del mantenimiento de la funcionalidad instalada en todos los dispositivos que forman parte del sistema.
- **Rendimiento**. Con el uso de recursos con mayor potencia en el nodo *edge* y en el nodo *fog*, se podría aumentar el rendimiento de las operaciones realizadas en comparación con otros dispositivos como el *Raspberry Pi*.

Se ha observado una limitación del sistema con respecto a la capacidad de procesamiento de vídeo, principalmente debido al uso de recursos limitados. En la gráfica 5.1 se muestra la velocidad de procesamiento del servicio de detección en el nodo *edge* y el servicio de almacenamiento en el nodo *fog*. Como se puede observar, estos valores tienen una media de 0.1 fps (*frames* por segundo) y 0.07 fps respectivamente, los cuáles son relativamente bajos para un sistema de videovigilancia que requiera una acción rápida ante la ocurrencia de un suceso. Estos valores se deben a los siguientes motivos:

- Uso de una máquina virtual con las limitaciones indicadas en el apartado 4.1 para realizar la instalación del servicio de detección.
- Uso de redes neuronales para la detección de objetos con un alto coste computacional. Se puede mejorar el rendimiento con el uso de la *GPU* (unidad de procesamiento gráfico) en lugar de la *CPU* (unidad de procesamiento central), lo cual requiere tarjetas gráficas potentes para la reducción del tiempo de procesamiento. Este hecho se verifica con la ejecución de una prueba del servicio de detección de objetos donde se compara las velocidades de procesamiento del vídeo cuando la funcionalidad de detección se encuentra desactivada y activada. Estas velocidades de procesamiento alcanzan una media de 275 fps y de 0.07 fps respectivamente, para un vídeo capturado por una cámara incorporada en el portátil como nodo *edge* (donde todos los procesos del sistema se ejecutan en una única máquina virtual). Por lo tanto, con esta prueba se comprueba que la funcionalidad de detección puede tener un gran impacto en la latencia percibida durante el procesamiento de detección.
- Encontrar el número óptimo de instancias del servicio acorde con las características del servidor.

- No se ha optimizado el proceso de detección de objetos, ni el proceso de sistema de mensajería *Kafka*, ya que para alcanzar dichas optimizaciones se requiere la ejecución de una batería de pruebas con distintas configuraciones y modelos. Por ejemplo, en el caso de *Kafka* se puede reducir el número de particiones para mejorar el rendimiento.
- No se ha optimizado la velocidad de captura de vídeo del *raspberry pi*, ya que las pruebas se han realizado con una captura de vídeo de 15 fps aproximadamente.

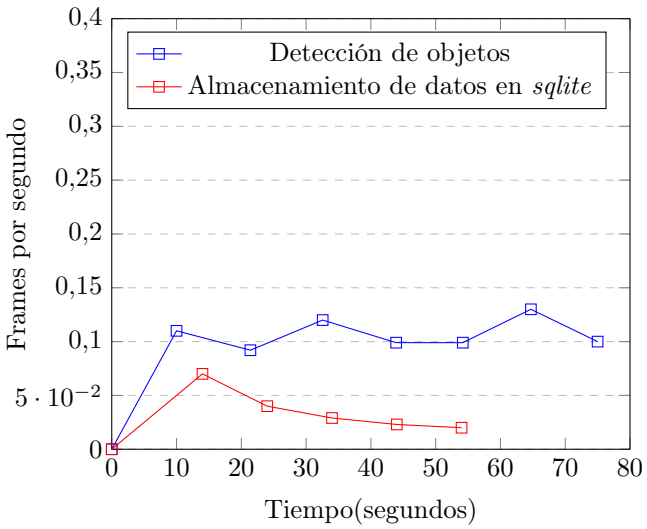

Figura 5.1: Velocidad de procesamiento del *frame* y de sus características extraídas, donde la captura de vídeo se realiza con una cámara de *Raspberry Pi* con una velocidad de 15 fps

5.2. Líneas de investigación futuras

El sistema de servicio de videovigilancia inteligente de este proyecto abarca una gran área de trabajo. Por lo tanto, en la solución propuesta hay muchos aspectos que no se han podido explorar. A continuación, se enumeran algunas líneas de trabajo que servirían para completar la investigación de este proyecto.

- Optimización de los distintos componentes del sistema. Como se ha explicado en el apartado anterior, se observa una latencia considerable para un servicio de videovigilancia en tiempo real. Se considera que esta latencia puede verse reducida con el uso de mejores recursos y con la búsqueda de optimización en el sistema de mensajería, en la funcionalidad del servicio y finalmente en el proceso de captura de vídeo.
- Implementar el desarrollo de comunicación entre los sistemas de mensajería *Kafka* de las 4 capas del sistema.
- Envío de eventos entre los *smart contracts*. Actualmente, se ha visto que la tecnología *blockchain* se puede aplicar a diversas áreas y se están explorando nuevos ámbitos distintos a la seguridad donde esta tecnología podría ser también utilizada. Sin embargo, esta tecnología se encuentra aún en fase de maduración, concretamente en el caso de los lenguajes de programación de los *smart contracts*, como *Solidity* o *Serpent*, que permiten una funcionalidad limitada. Por ejemplo, algunas funcionalidades como el paso de argumentos de tipo *string* están aún en fase de experimentación y otras funcionalidades como el envío de eventos entre *smart contracts* no está aún soportado. Por ende, sería interesante aplicar el envío de eventos entre *smart contracts* cuando plataformas como *Ethereum* soporten dicha funcionalidad.
- Envío de los parámetros de conexión vía *smart contracts* para que dichas operaciones queden registradas en *blockchain*. Estos parámetros son privados, por lo que no deben ser visibles en la red. El problema radica en que esta tecnología aún no soporta métodos de encriptación y,

por lo tanto, cualquier dato o transacción en *blockchain* es visible. Una posible solución sería encriptar estos parámetros previamente antes de su envío a la red y utilizar algún mecanismo para compartir la clave fuera de la red de *blockchain* o emplear la cuenta digital (conocida por el emisor y el receptor) como clave de encriptación. Para más información ver apéndice A.2.

- Almacenamiento de los *hashes* del vídeo procesado en la red de *blockchain*. En la red de *blockchain* no es aconsejable que se almacenen los *frames* del vídeo porque *blockchain* no está enfocado para ello, sino que generalmente se emplea para la creación de transacciones que no pueden ser modificadas, ni eliminadas en la red. Por lo tanto, se propone como mecanismo de protección del dato, la creación de transacciones que contengan los *hashes* de los *frames* del vídeo al detectarse un suceso determinado.

Bibliografía

[1] Glasmeier A. y Christopherson S. *Thinking about smart cities*. Vol. 8. 14 de feb. de 2015, págs. 3-12. URL: https://doi.org/10.1093/cjres/rsu034.

[2] Balandin S.; Andreev S.y Koucheryavy Y. *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*. [15^{Th} International Conference, NEW2AN 2015]. 2015.

[3] Xu R.; Nikouei S.; Chen Y.; Song S.; Polunchenko A.; Deng C. y Faughnan T. *Real-time human object tracking for smart surveillance at the edge*. [IEEE International Conference on Communications (ICC), Selected Areas in Communications Symposium Smart Cities]. 2018.

[4] Chen N.; Chen Y.; Blasch E.; Ling H.; You Y. y Ye X. *Enabling Smart Urban Surveillance at The Edge*. [IEEE International Conference on Smart Cloud]. 2017.

[5] Medvedev A.; Thodoris Anagnostopoulos T.; Viktorovich Fedchenkov P.; Khoruzhnikov S. *Waste Management as an IoT-Enabled Service in Smart Cities*. 2015. URL: https://www.researchgate.net/publication/279196726.

[6] Shesherao Mete S. y Pundlikrao A. *A Smart EMS Enhancing Prehospital Quality with Cloud Fog Operability*. Vol. 2. 2019. URL: www.rjetm.in/.

[7] Munir A.; Kansakar P.y Khan S. *IFCIoT: Integrated Fog Cloud IoT Architectural Paradigm for Future Internet of Things*. [arXiv: Distributed, Parallel, and Cluster Computing]. 1 de ene. de 2017.

[8] Shi W. ; Cao J.; Zhang Q; Li Y. Y Xu L. *Edge Computing: Vision and Challenges*. Vol. 3. 2016.

[9] Pignataro C.; Kumar N.; Morrisville Nainar. y Morrisville Rajiv. *In-situ operations, administration and managment and network event correlation for internet of things*. Estados Unidos, 29 de ago. de 2019.

[10] Soudris D. y Henkel J. Samie F.;Tsoutsouras V.; Bauer L; Xydis S. *Computation Offloading Management and Resource Allocation for Low-power IoT Edge Devices*. [Conferencia: IEEE World Forum on Internet of Things (WF-IoT)]. 2016.

[11] [The OpenFog Consortium]. 2017. URL: http://www.openfogconsortium.org/.

[12] M. Abdelshkour. *Iot, from cloud to fog computing*. 2015. URL: https://goo.gl/7zNxEd.

[13] Gai K.; Qiu M. y Zhao H. *Energy-aware task assignment for mobile cyber-enabled applications in heterogeneous cloud computing*. 2017.

[14] OpenFog. *OpenFog Reference Architecture for Fog Computing*. USA, 2017. URL: https://www.iiconsortium.org/index.htm.

[15] Risteska B. y Trivodaliev K. *A review of Internet of Things for smart home: Challenges and solutions*. 5 de oct. de 2016.

[16] Wu H.; Hu J.; Sun Jiexiang y Sun Danfeng. *Edge Computing in an IoT Base Station System: Reprogramming and Real-Time Tasks*. 5 de mar. de 2019. URL: https://doi.org/10.1155/2019/4027638.

[17] Nikouei S. Y.; Chen Y.; Song S.; Xu R.; Choi B. y Faughnan T. R. *Smart Surveillance as an Edge Network Service: from Harr-Cascade, SVM to a Lightweight CNN*. 1 de oct. de 2018.

[18] URL: https://ethereum.org/.

[19] URL: https://www.hyperledger.org/projects/fabric.

[20] URL: https://bitcoin.org.

[21] Mollah M.; Azad M. y Vasilakos A. *Security and privacy challenges in mobile cloud computing. Survey and way ahead.* Vol. 84. 2017, págs. 38-54.

[22] Cruz Jason P.; Kaji Y. y Yanai N. *RBAC-SC: Role-Based Access Control Using Smart Contract.* [Disponible en IEEE Access]. 2018.

[23] Ugobame Uchibeke Uchi.; Hosseinzadeh Kassani S.; Deters R. y Schneider K. *Blockchain Access Control Ecosystem for Big Data Security.* Canadá: IEEE, 2018.

[24] Ouaddah A.; Elkalam A. y Ouahman A. *FairAccess: a new Blockchain-based access control framework for the Internet of Things.* Marruecos: Wiley, 2017. URL: `wileyonlinelibrary.com`.

[25] Guo H.; Meamari E. y Shen Chien-Chung. *Multi-Authority Attribute-Based Access Control with Smart Contract.* [Department of Computer and Information Sciences, University of Delaware]. USA, 2019.

[26] Mendez D. y Yang B. *Blockchain-Based Whitelisting for Consumer IoT Devices and Home Networks.* 2018. URL: `https://www.researchgate.net/publication/328345831`.

[27] Yu Chen R.; Blasch E. y Chen G. *BlendCAC: A BLockchain-ENabled Decentralized Capability-based Access Control for IoTs.* USA, 2018.

[28] Hammi M. T.; Bellot P.; Hammi B. y Serhrouchni A. *Bubbles of Trust: a decentralized Blockchain-based authentication system for IoT.* 2018. URL: `https://www.researchgate.net/publication/326094774`.

[29] Yu Chen R.; Blasch E. y Chen G. *BlendCAC: A Smart Contract Enabled Decentralized Capability-Based Access Control Mechanism for the IoT.* USA, 2018.

[30] Yu Chen R.; Blasch E. y Chen G. *An Exploration of Blockchain Enabled Decentralized Capability based Access Control Strategy for Space Situation Awareness.* USA, 2018.

[31] Yu Chen R.; Blasch E. y Chen G. *A Federated Capability-based Access Control Mechanism for Internet of Things (IoTs).* 2018.

[32] Gipp B.; Kosti J. y Breitinger C. *Securing Video Integrity Using Decentralized Trusted Timestamping on the Bitcoin Blockchain.* [Disponible en la librería electrónica AIS (AISeL)]. 2016.

[33] Wang S.; Zhang Yinglong y Zhang Yaling. *A Blockchain-Based Framework for Data Sharing With Fine-Grained Access Control in Decentralized Storage Systems.* [Publicado en IEEE Access con identificador 10.1109/ACCESS.2018.2851611]. 2018.

[34] Nadareishvili I.; Mitra R.; McLarty M. y Amundsen M. *Microservice architecture. Aligning Principles, Practices, and Culture.* Editorial O'reilly, 2016.

[35] URL: `https://martinfowler.com/articles/microservices.html`.

[36] URL: `https://medium.com/@narengowda/uber-system-design-8b2bc95e2cfe`.

[37] Nagothu D.; Xu R.; Nikouei S. Y. y Chen Y. *A Microservice-enabled Architecture for Smart Surveillance using Blockchain Technology.* [Dept. of Electrical & Computer Engineering, Binghamton University, SUNY, Binghamton, NY 13902]. USA, 2018.

[38] Xu R.; Nikouei S.; Chen Y.; Blasch E. y Aved A. *BlendMAS: A BLockchain-ENabled Decentralized Microservices Architecture for Smart Public Safety.* USA, 2019.

[39] Xu R.; Nikouei S.; Chen Y.; Blasch E. y Aved A. *Decentralized Smart Surveillance through Microservices Platform.* 2019.

[40] URL: `https://es.wikipedia.org/wiki/STRIDE_(seguridad)`.

[41] URL: `https://en.wikipedia.org/wiki/DREAD_(risk_assessment_model)`.

[42] Singhal B.; Dhameja G.y Sekhar Panda P. *Beginning blockchain.* Editorial Apress, 2018.

[43] Antonopoulos A. y Wood G. *Mastering Ethereum Building Smart Contracts and Dapps.* Editorial O'reilly, 2018.

[44] Bruyn A. *Blockchain an introduction.* [University Amsterdam. Supervisor Rob van der Mei]. 2017.

[45] Parth G. *Typescript microservices: Build, deploy, and secure microservices using typescript combined with Node.js.* [Disponible en www.packt.com]. 2018.

Palabras clave

IoT:

Internet de las cosas

RTMP:

Protocolo de transmisión en tiempo real

SMS:

Servicio de mensajes cortos

LAN:

Red de área local

WAN:

Red de área amplia

GAN:

Red de área global

RCP:

Llamada a procedimiento remoto

SHA:

Algoritmo de hash seguro

SHA256:

Algoritmo de hash seguro de 256 bits

AES:

Estándar de cifrado avanzado

HMAC:

Código de autenticación de mensaje hash con clave

ABAC:

Control de acceso basado en atributos

IPFS:

Sistema de archivos interplanetarios

BIBAC:

Control de acceso basado en la identidad en blockchain

RBAC:

Control de acceso basado en roles

MITM:

Man in the middle attack

DREAD:

Daño potencial, reproducibilidad, explotabilidad, clientes afectados, descubrimiento

STRIDE:

Suplantación de identidad, manipulación de datos, repudio, divulgación de información, negación de servicio, elevación de privilegio

RPC:

Protocolo de llamada a procedimiento remoto

API:

Interfaz de un programa aplicación

GPU:

Unidad de procesamiento gráfico

CPU:

Unidad de procesamiento central

Apéndice A

Apéndice

A.1. Blockchain

En este apéndice se presenta información considerada relevante para un mejor entendimiento del proyecto. Más información sobre esta tecnología se puede encontrar en las referencias [42], [43] y [44].

A.1.1. Tipos de blockchain

Pueden ser públicas o privadas:

- **Pública**:
  - Está disponible para todo el mundo que tenga conexión a internet.
  - Está basada en la ideología de que el usuario debe pagar por almacenar y realizar transacciones.
  - Dado que los nodos se encuentran distribuidas de forma extensa por todo el mundo, este tipo de *blockchain* es resistente a ataques, ya que no se trata de una única organización a la que se pueda atacar.
  - Dispone de una comunidad estable para su soporte.
- **Privada**:
  - Similar a base de datos tradicionales. No considerada muchas veces como una autentica *blockchain*, puesto que no se encuentran realmente distribuidas por todo el mundo.
  - Permite un mayor control del coste, dado que una o varias organizaciones controlan la infraestructura.
  - Mayor control del almacenamiento de datos.
  - Control de las entidades que tienen acceso a la red.
  - Menos resistente que las redes de *blockchain* públicas, ya que en relación con la pública disponen de muy pocos nodos.

A.1.2. Operaciones

En *blockchain* se pueden realizar dos operaciones, transacciones y *smart contracts* o *code chain*.

Transacciones

Una transacción consiste en el envío de datos en forma de un contrato a una red de ordenadores donde la comunicación se produce entre dos usuarios que se encuentran en la red. Cada ordenador en la red de *blockchain* es conocido como nodo, donde cada nodo dispone de una copia de los datos

existentes en *blockchain*. Las transacciones se ejecutan y validan en base a unos *scripts* y contratos definidos. De esta manera, se asegura que todos los nodos ejecuten las transacciones utilizando la misma reglas. El resultado de esta transacción se añade al *blockchain*, y debido a que cada nodo dispone de la misma copia, se dificulta que una entidad maliciosa comprometa la integridad de la transacción.

Características de las transacciones son:

- Las transacciones son atómicas, es decir, se realiza el proceso completo o en caso de fallo se deshacen todas las operaciones realizadas hasta el momento.
- Las transacciones se ejecutan de forma independiente, por lo tanto, las operaciones no pueden interactuar o interferir con otras.
- Las transacciones no se eliminan, es decir, no pueden ser modificadas y tampoco pueden ser eliminadas.
- Son inspeccionables. Cada ejecución de una transacción se realiza con la dirección del usuario que realiza la transacción, proporcionando más seguridad y mayor control sobre las transacciones realizadas.

Smart contract o code chain

Engloba líneas de código que hacen cumplir la negociación o ejecución de un contrato sin la necesidad de terceras partes. Dispone de su propia cuenta, dirección y balance. Además, es capaz de enviar y recibir mensajes, y como cualquier otra transacción debe pagar una cuota por almacenamiento y ejecución.

Cualquier nodo en la red puede acceder a todas las transacciones y estado reciente del *smart contract*. Para ello, el nodo debe interactuar con el *smart contract* a través de su dirección e interfaz *RPC* (*Remote Procedure Call*).

A.1.3. Hashing

El hasing se emplea comúnmente en protocolos de encriptación. Consiste, básicamente, en la ejecución de un algoritmo matemático que genera siempre un resultado con la misma longitud independientemente del parámetro de entrada. Este proceso se ejecuta en un único sentido, es decir, siempre genera el mismo *hash* para la misma entrada, pero no se puede regenerar la entrada a partir del *hash*.

El algoritmo *SHA 256* (pertenece a la familia de algoritmos *SHA*) y es habitualmente utilizado en *blockchain* para la creación de los bloques.

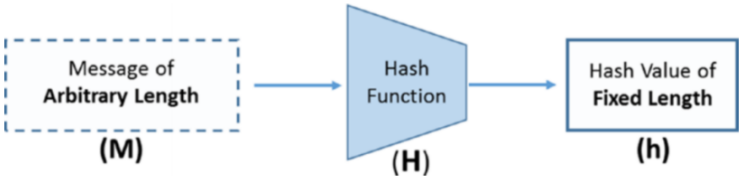

Figura A.1: Diagrama del algoritmo hash

A.1.4. Cadena de bloques

Un bloque se conforma de datos y de su *hash* que se obtiene como resultado del *hashing* de los datos del bloque, de modo que en el supuesto de que se cambie cualquier dato del bloque, producirá un *hash* diferente invalidando el bloque.

Los datos relevantes que puede almacenar un bloque son:

- **Nonce**. Se trata de un número arbitrario que se utiliza en combinación con el *hash* como un elemento de control para evitar que se manipule la información que almacenan los bloques.

El nonce asignado al bloque está ligado al resultado del algoritmo *hashing*, que debe cumplir unos requisitos. Por ejemplo, los primeros caracteres del *hash* deben contener un número determinado de ceros. Por lo tanto, no se puede predecir el nonce y es considerado como un mecanismo de prueba, *proof of work*, para la máquina que genera el *hash*. El caso de que se requiera que los primeros 4 caracteres del *hash* contengan ceros, implica que para cualquier cambio realizado en el bloque, el algoritmo *hashing* se tendrá que ejecutar las veces necesarias hasta que se encuentre un nonce que produzca un *hash* que cumpla dicho requisito. Este proceso es conocido como *mining*.

- **Número de bloque**. Hace referencia al orden del bloque en la cadena de bloques.
- **Hash del bloque previo**. Cualquier cambio en uno de los bloques modificaría el *hash* del bloque, por lo que ya no coincidiría con el *hash* de este campo del bloque posterior, y como consecuencia, se invalidarían todos los bloques siguientes en la cadena. La única solución a este problema sería realizar el proceso de *mining* a todos estos bloques, lo cuál implica un cómputo bastante considerable.
- **Timestamp**. Registro de la creación del bloque.

En el caso de una cadena de bloques distribuida como la del *blockchain*, se mantiene un copia de esta cadena en múltiples nodos de la red (dependiendo de la implementación de *blockchain*), de modo que en el supuesto de que una cadena de bloques se vea comprometida en uno de los nodos, es decir, se haya modificado y llevado a cabo el proceso *mining* por la entidad maliciosa, el nodo comprometido se podría detectar fácilmente y ser eliminada de la red de *blockchain*.

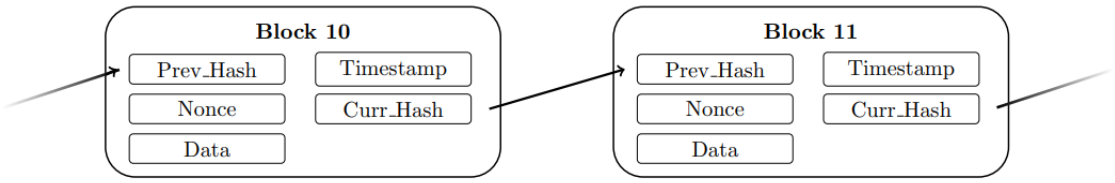

Figura A.2: Estructura del bloque de una cadena de bloques blockchain

A.1.5. Criptografía

La criptografía es el componente más importante de una red blockchain. Tiene los siguientes objetivos:

- **Confidencialidad**. Sólo el destinatario puede entender el mensaje.
- **Integridad del dato**. Los datos no pueden ser falsificados o modificados, ya sea por error o de forma intencional.
- **Autenticación**. La autenticidad del emisor puede ser verificada por el receptor del mensaje.
- **No repudio**. El usuario que envía el mensaje no puede negarse más tarde de que es el emisor del mensaje.

Los datos en una red *blockchain* normalmente se encuentran disponibles para cualquier usuario que tenga acceso a la red. Por ello, es importante disponer de otros sistemas para controlar quién o quiénes tienen acceso a ciertos datos o recursos de la red. En *blockchain* se suele utilizar el método de encriptación. Se distinguen 2 tipos:

- **Simétrica**. Este método se emplea para encriptar el dato o mensaje que sólo puede ser descifrado por aquellos usuarios que disponen de la clave de encriptación. Algoritmos a destacar de este tipo de encriptación son el algoritmo *AES* (*Advanced Encryption Standard*) y *HMAC*, por ser considerados como unos de los más seguros.

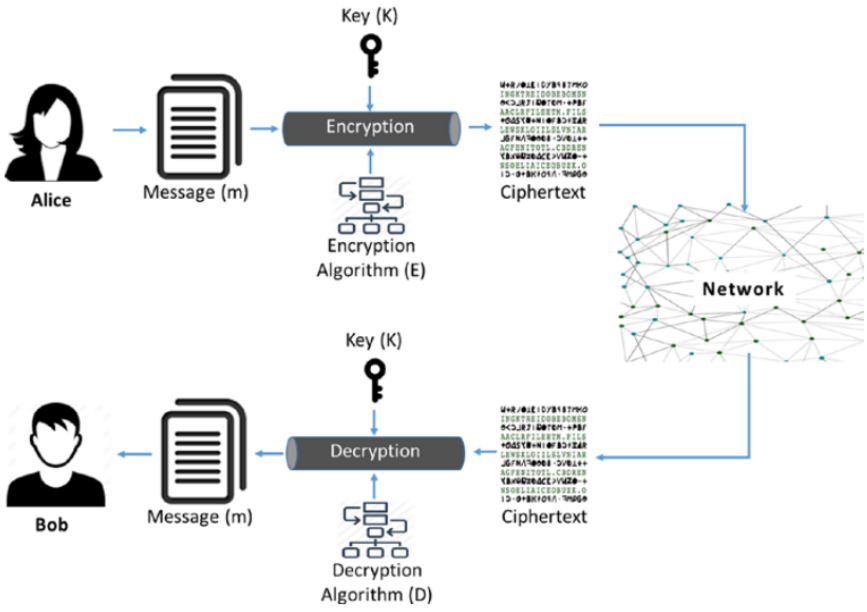

Figura A.3: Proceso de un método de encriptación normal

- **Asimétrica**. Estos sistemas de encriptación emplean dos claves diferentes. Una de las claves es privada, la cual no se debe compartir con nadie y la otra clave es pública, como su nombre indica, puede ser compartida con otros usuarios en la red. En este tipo de encriptación, el remitente utiliza la clave pública del destinatario para encriptar el mensaje, por lo que únicamente el destinario puede descifrar el mensaje haciendo uso de su clave privada. Dentro de esta variante, destaca la firma digital, consistente en encriptar el *hash* del mensaje con la clave privada del emisor, de tal manera que el destinatario que recibe el mensaje junto con la firma puede verificar la autenticidad y la integridad del mismo.

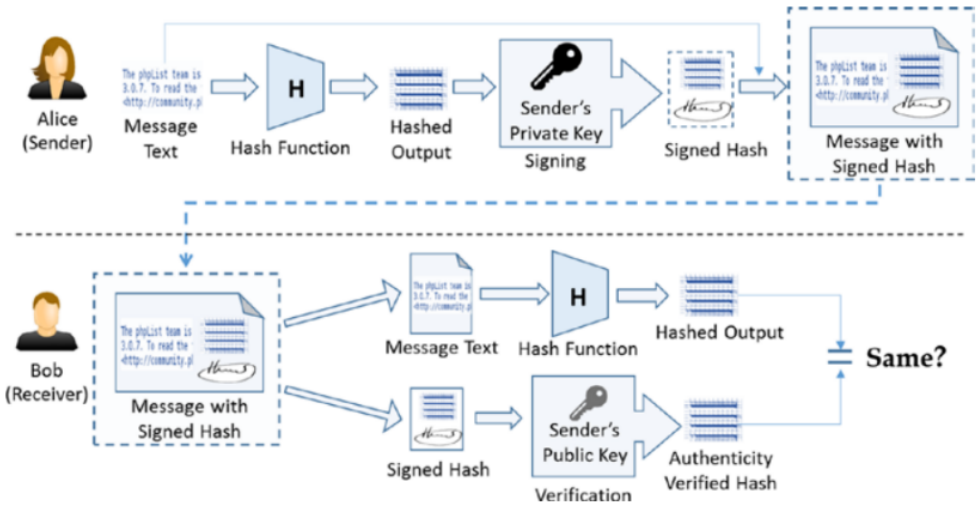

Figura A.4: Proceso de envío de un mensaje con firma digital

A.2. Kafka

Kafka es un sistema de mensajería asíncrono basado en un *broker* que se encarga de gestionar la comunicación y los datos entre los productores y consumidores, donde los productores, o *producers*,

envían datos de entrada a este sistema de mensajería y los consumidores, o *consumers*, leen los datos de salida.

La figura A.5 muestra el uso de *Kafka* con *microservicios* con múltiples productores y múltiples consumidores. En este ejemplo, se emplean 2 productores y 5 consumidores.

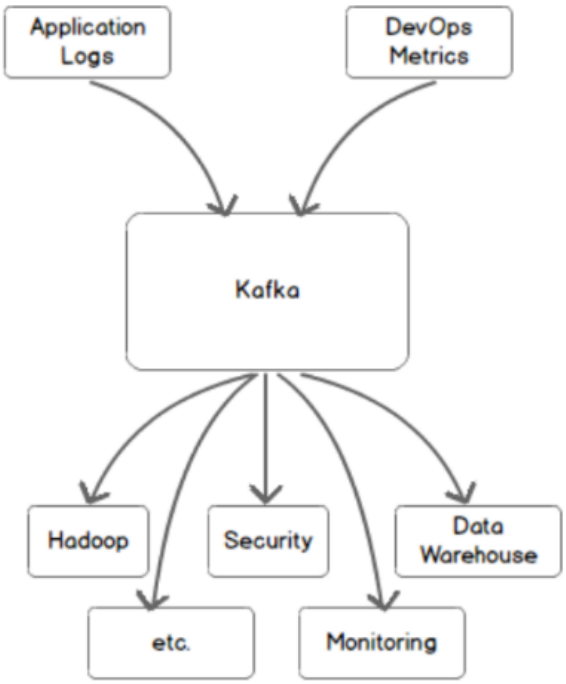

Figura A.5: Sistema de mensajería distribuido

Este sistema se conforma de los siguientes módulos:

- **Broker**. Encargado de gestionar todas las peticiones de los clientes, productores y consumidores. Una instancia de *Kafka broker* puede gestionar una gran cantidad de información sin que su rendimiento se vea afectado. Por otro lado, este sistema puede realizar balanceo de carga con múltiples *brokers*, también conocido como *Kafka cluster*.
- **Zookeeper**. Es un servicio de registro que mantiene el estado de los *brokers*, *topics* y usuarios. Otras de las funcionalidades de *Zookeeper* es la de notificar al productor y al consumidor del fallo o de la presencia de un nuevo *broker*.
- **Productor**. Envía datos, también conocidos como *records*, al *broker* empleando el modelo *push messages*.
- **Consumidor**. Consume lotes de datos de un *broker* utilizando el modelo de *pull messages*.

A.2.1. Topic

Los topics son flujos de datos que son divididas en particiones (ver figura A.6). Estas particiones permiten paralelizar un *topic* dividiendo el dato de un *topic* entre los múltiples *brokers* del sistema, por lo que permite que varios consumidores puedan leer datos en paralelo. Cada mensaje dentro de una partición tiene un identificador, conocido como offset, que permite mantener el orden de los mensajes, así como ofrecer la posibilidad de leer el mensaje desde cualquier punto que el consumidor elija. Otra característica a destacar es que los productores pueden crear tantos *topics* como deseen.

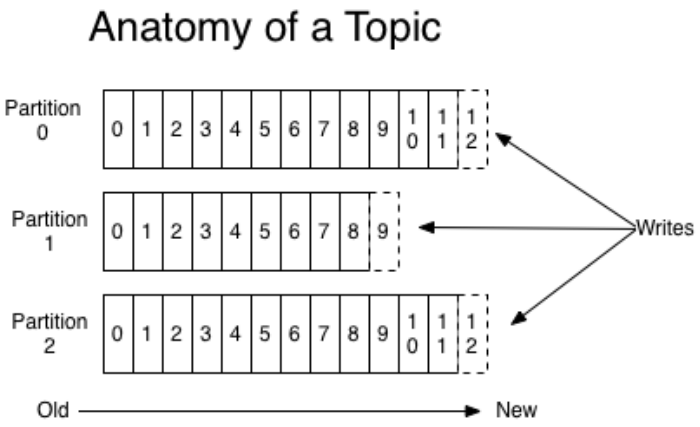

Figura A.6: Anatomía de un *topic*

Se puede aumentar el número de copias de un *topic* incrementando el factor de replicación en un *Kafka cluster* para asegurar la disponibilidad del sistema si un *broker* deja de estar disponible para recibir peticiones. En el caso de tener un factor de replicación mayor a uno, se designa un líder para cada partición. Este líder es el responsable de recibir y enviar datos a su partición donde el resto de réplicas (*followers*) son sincronizadas con dicha partición.

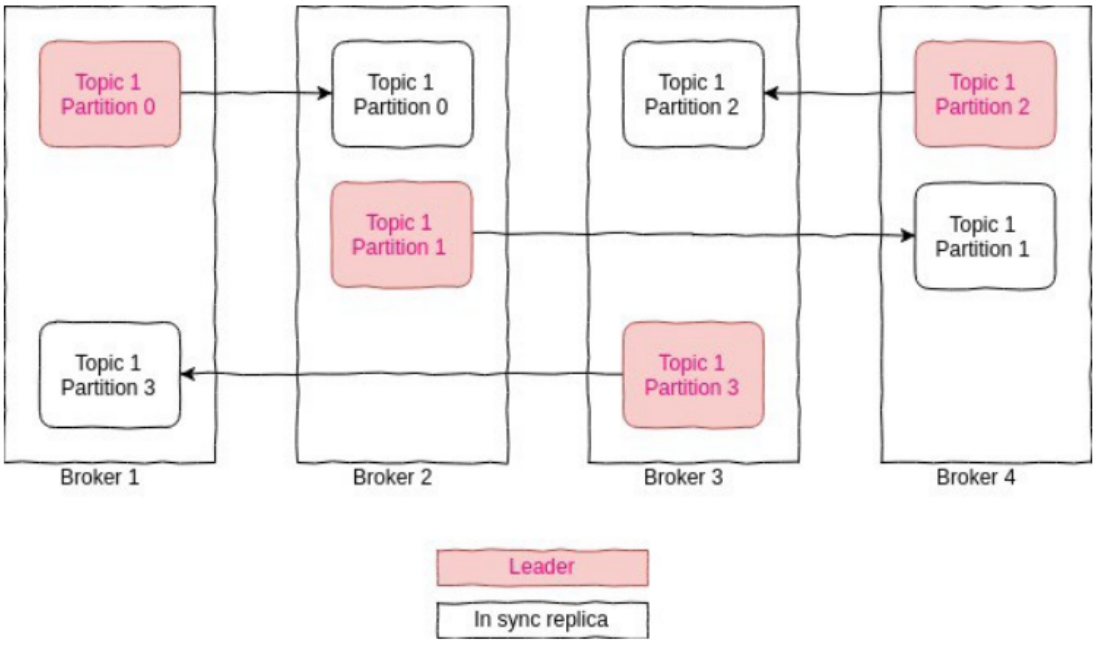

Figura A.7: *Kafka cluster* compuesto de 4 *brokers* con un factor de replicación 2

A.2.2. Grupos

Los consumidores pueden ser organizados en grupos, donde cada grupo consume todos los mensajes de un determinado *topic* y cada consumidor de un grupo lee de una única partición. De este modo, *Kafka* puede alcanzar el balanceo de carga creando grupos de consumidores y variando el factor de réplica que permite a un grupo consumir datos de múltiples particiones, asegurando la disponibilidad del dato. Por lo que si aumenta el número de consumidores se puede aumentar el número de particiones teniendo en cuenta lo siguiente:

- Si se dispone de más consumidores que particiones, los consumidores podrían estar desocu-

pados, ya que no tienen datos que leer de una partición.

- Si se dispone de más particiones que consumidores, los consumidores recibirán mensajes de múltiples particiones.

En la figura A.8 se muestra un *Kafka cluster* compuesto de dos *brokers*. Estos *brokers* disponen de 4 particiones con un factor de réplica 1. En este ejemplo, se designan 2 grupos de consumidores, donde cada grupo, en total, leerá los mensajes de todas las particiones de un *topic*, pero en ningún caso los consumidores del mismo grupo leerán mensajes de la misma partición.

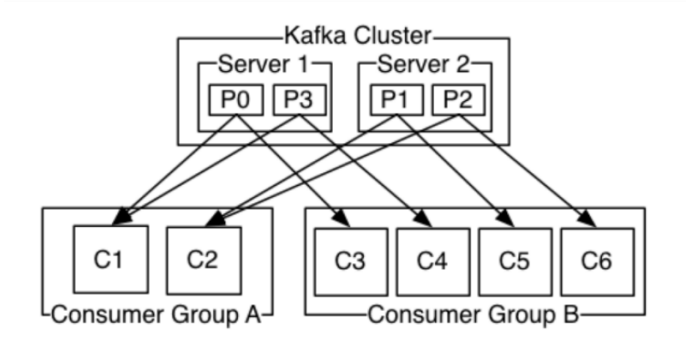

Figura A.8: *kafka cluster* compuesto de 2 *brokers* y dos grupos: *Consumer group A* y *Consumer group B*