Universidad Zaragoza
1542

Trabajo Fin de Grado

Toma de decisiones descentralizada en sistemas distribuidos con recursos computacionales de capacidad heterogénea

Decentralised decision making in distributed systems with computational resources of heterogeneous capacity

Author

Raúl Logroño Arteaga

Directores

Rafael Tolosona Calasanz
Jose Ramón Gállego Martínez

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2021

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe entregarse en la Secretaría de la EINA, dentro del plazo de depósito del TFG/TFM para su evaluación).

D./Dª. Raúl Logroño Arteaga ,en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Gr. Ing. de Tecnologías y Serv. de Telecomunicación (Título del Trabajo)

Toma de decisiones descentralizada en sistemas distribuidos con recursos computacionales de capacidad heterogénea

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 25 de Noviembre de 2021

Fdo: Raúl Logroño Arteaga

RESUMEN

Durante los últimos años, debido al enorme desarrollo de los dispositivos electrónicos, la capacidad para generar datos ha aumentado significativamente, lo que ha dado como resultado la producción de una amplia variedad de información. Los sistemas de computación de datos necesitan adaptarse a esta nueva ola creciente, para ello es necesario cambiar de un paradigma tradicional centralizado a uno descentralizado. Esto resolvería el principal problema de los sistemas actuales, su escalabilidad, pero generaría otros problemas típicos del paradigma distribuido debido a la escasa investigación en este campo durante las últimas décadas. En este Trabajo Final de Grado vamos a diseñar e implementar un sistema distribuido descentralizado, que explore cómo el paradigma de programación declarativo (en particular, los sistemas expertos, un tipo de inteligencia artificial que busca emular el comportamiento de un experto en su área de conocimiento) puede ayudar a construir sistemas distribuidos con mayor capacidad para auto adaptarse bajo condiciones de ejecución cambiantes.

En general, en este contexto de gran generación de datos, las fuentes de datos o los nodos cercanos a ellas tienen baja capacidad de computación, mientras que los nodos con mayor capacidad están más alejados en la nube. Las aplicaciones que recopilan datos suelen tener asignados unos requisitos temporales para llevar a cabo el procesamiento asociado, que suele denominarse QoS de la aplicación. El problema consiste en decidir dónde deben procesarse esos datos, cumpliendo con esos requisitos de QoS. Para ello, los nodos que componen el sistema han de ser capaces de enviar datos a través de la red, conocer el estado de la red, y el estado de otros nodos que pueden procesar datos por él. Una vez los nodos tienen este conocimiento, son capaces de estimar el coste en tiempo del procesado de los datos, lo que les sirve para saber si están violando el QoS, y en caso de hacerlo buscar otra opción que sí lo cumpla. Es aquí donde entra el sistema experto. Comparamos sus decisiones con las de un sistema de reglas estáticas (la forma tradicional de toma de decisiones) para comprobar si realmente puede aportar algo más de lo que aporta una solución a priori más sencilla. Tras recopilar datos de ambos tipos de decisiones sobre dos escenarios distintos durante veinticuatro horas, podemos concluir que los sistemas expertos poseen las cualidades necesarias para ser una alternativa válida y flexible en la implementación de soluciones bajo un paradigma distribuido.

Índice

Capítulo 1

Introducción y objetivos

Durante los últimos años, debido al enorme desarrollo de los dispositivos electrónicos, la capacidad para generar datos ha aumentado significativamente, lo que ha dado como resultado la producción de una amplia variedad de información que va desde mediciones de fenómenos naturales hasta comportamientos relacionados con los seres humanos. Las infraestructuras computacionales, para poder procesar toda esa información a gran escala, tienen que evolucionar del control tradicional centralizado a un paradigma en el que la toma de decisiones sobre la gestión de los recursos computacionales se realiza de forma local, es decir, descentralizada.

Tradicionalmente, un sistema distribuido puede verse como un conjunto de máquinas o nodos, posiblemente heterogéneos, interconectados a través de la red y capaces de cooperar unos con otros para resolver tareas. Partiendo de esta base se plantean dos tipos de arquitecturas: centralizada y descentralizada. La diferencia principal entre estas es cómo y dónde se realiza la toma de decisiones y cómo la información se comparte entre los nodos del sistema. En la descentralizada, no hay un punto único de decisión, los nodos no tienen conocimiento del estado global del sistema, cada nodo toma la decisión que más le conviene en función de las reglas establecidas y la información que tiene en ese momento. En cambio, en la arquitectura centralizada las decisiones las toma un único nodo central con conocimiento total del estado del sistema.

La arquitectura centralizada es la solución más extendida por ser más sencilla y rápida de aplicar, aunque plantea una serie de retos. El principal de ellos es la escalabilidad del sistema porque el nodo central no dispone de recursos ilimitados y tiene un límite en el numero de conexiones o máquinas que puede coordinar al mismo tiempo. Esto nos lleva al siguiente problema: su fragilidad; ya que si el nodo central experimenta algún tipo de incidencia repercutirá al funcionamiento global del sistema.

Estos serían los principales problemas que plantea la arquitectura centralizada,

pero que resuelve la arquitectura descentralizada. Al no haber ningún nodo central que tome todas las decisiones, el sistema no tiene ningún problema en escalar el número de máquinas. Así queda resuelto el problema de la escalabilidad y la fragilidad del mismo. Eso sí, esta arquitectura plantea otros muchos problemas o retos que la arquitectura centralizada no tiene. Se pueden resumir en cómo los nodos comparten los recursos: datos, CPU, memoria RAM, memoria disponible en disco, ancho de banda disponible, etc. y cómo toman decisiones sin disponer de información completa del sistema, porque al final, el principal problema del sistema distribuido frente al centralizado es que es subóptimo.

En este Trabajo Final de Grado hemos diseñado e implementado un sistema distribuido descentralizado a pequeña escala, que a modo de prueba de concepto nos permita comparar distintas soluciones a los problemas planteados por esta arquitectura. La clave está en las reglas que deciden cómo gestiona los recursos cada nodo, así que hemos llevado a cabo dos posibles soluciones: reglas estáticas (la forma más tradicional de programar) y reglas dinámicas. Las reglas estáticas son una sucesión de condicionales: IF THEN ELSE. Mientras que las reglas dinámicas las aplicamos a través de un sistema experto [10] o sistema de mantenimiento de la verdad, es una de las aplicaciones de la inteligencia artificial, pretende simular el proceso de razonamiento de un profesional humano en su campo.

El documento está organizado de la siguiente manera: el capítulo dos presenta las tecnologías usadas en el proyecto y su configuración; en el capítulo tres se ve el diseño del sistema ordenado en partes según se han ido implementado; el cuatro muestra en profundidad el funcionamiento de las reglas dinámicas en nuestro sistema; el cinco expone un análisis de los datos, así como de los resultados de las pruebas y las comparaciones entre las dos soluciones; y por último, el capítulo seis expone las conclusiones y líneas futuras de trabajo.

El código desarrollado para este proyecto está disponible en el repositorio citado[12].

Capítulo 2

Tecnologías subyacentes

2.1. Docker

Uno de los grandes retos de la gestión e instalación de sistemas distribuidos es cómo distribuir el software y las versiones de sus dependencias (bibliotecas). Para tal fin, surgieron los contenedores, aunque últimamente, también se utilizan como alternativa o complemento a las máquinas virtuales (puesto que son más ligeros y generan menos sobrecarga): los contenedores, por tanto, aspiran a crear un entorno aislado de ejecución para un programa, junto con las dependencias software que necesita para su ejecución. Docker[1] fue el primer proyecto en desarrollar el concepto de contenedor. Docker automatiza el despliegue de aplicaciones dentro de contenedores. Esto permite ejecutar aplicaciones en cualquier máquina utilizando un único archivo de configuración.

2.1.1. Contenedores

Los contenedores encapsulan las dependencias de software necesarias para el funcionamiento de una aplicación. Para lanzar un contenedor es necesaria una imagen sobre la que apoyarse. Esta imagen se puede descargar del repositorio de imágenes de docker, donde la comunidad las publica para lanzar determinadas tecnologías o aplicaciones sobre contenedores, o, en caso de necesitar algo más personalizado, se puede elaborar un DockerFile, donde se especifica el sistema operativo que ejecutará el contenedor y todos los paquetes y configuraciones que sean necesarias. Incluso es posible partir de una imagen del repositorio para modificar su configuración. En la figura 2.1 se muestra uno de los dos DockerFiles elaborados para este proyecto, así conseguimos configurar e instalar todos los paquetes de software que la aplicación desarrollada necesita para funcionar dentro del contenedor.

[1] `https://www.docker.com/`

```
FROM ubuntu:18.04


# install requirements

RUN apt-get -qq update && apt-get -qq install --no-install-recommends -y python3.8 \
 python3-dev \
 python-pil \
 python-lxml \
 wget \
 unzip \
 protobuf-compiler \
 ffmpeg \
 libsm6 \
 libxext6 \
 iperf

RUN wget -q -O /tmp/get-pip.py --no-check-certificate https://bootstrap.pypa.io/get-pip.py && python3 /tmp/get-pip.py

RUN  pip install -U pip \
 jupyter \
 matplotlib \
 tensorflow==1.5 \
 opencv-python

# define working directory within docker image
WORKDIR /opt/object_detection_video
ENV LD_LIBRARY_PATH=/opt/object_detection_video/:$LD_LIBRARY_PATH

CMD ./main
```

Figura 2.1: Dockerfile

2.1.2. Networking

Para interconectar los contenedores y, por tanto, los procesos que se ejecutan dentro de los contenedores, Docker ofrece varios tipos de red, en nuestro caso hemos usado una red de tipo overlay, lo que nos permite comunicar contenedores de diferentes ubicaciones entre sí. La alternativa sería encaminar nosotros mismos los paquetes conociendo las direcciones IPs publicas de las máquinas reales y, configurar los firewalls para permitir el tráfico de la aplicación. Aun así tendríamos que crear una red en Docker de tipo puente para reenviar los paquetes de la máquina real al contenedor. La red overlay de Docker usa puertos reservados (TCP 2377, TCP/UDP 7946, UDP 4789)[3], lo que da algo más de seguridad y en caso de no fiarnos de los nodos intermedios podríamos encriptar los datos usando la opción –opt encrypted al crear la red. Esta claro que la opción de la red overlay es más robusta y nos encaja mejor.

Para desplegar una red overlay Docker usa varias técnicas, pero la tarea principal la hace VXLAN[17], es un protocolo de red que encapsula el tráfico MAC (capa dos) en paquetes UDP creando una red virtual de capa dos sobre las redes de encaminamiento de capa tres. En la figura 2.2 [4] podemos ver un ejemplo de como configura Docker la red overlay. Tenemos los contenedores conectados por la interfaz eth0 a un dispositivo veth, actúa como una conexión ethernet virtual con el fin de cambiar el espacio de nombres (NS) del contenedor (red LAN virtual) al de la red. Ahí la interfaz VXLAN le añade al paquete ethernet una cabecera VXLAN con el identificador de esa red virtual

(VNI), se encapsula en un paquete UDP y se usa IP como protocolo de encaminamiento de capa tres. En el otro extremo el proceso es el mismo pero al contrario, cuando el paquete le llega al contenedor no se ha enterado ni de la encapsulación ni de la interfaz VXLAN, para él es como si estuviera en local con el otro contenedor.

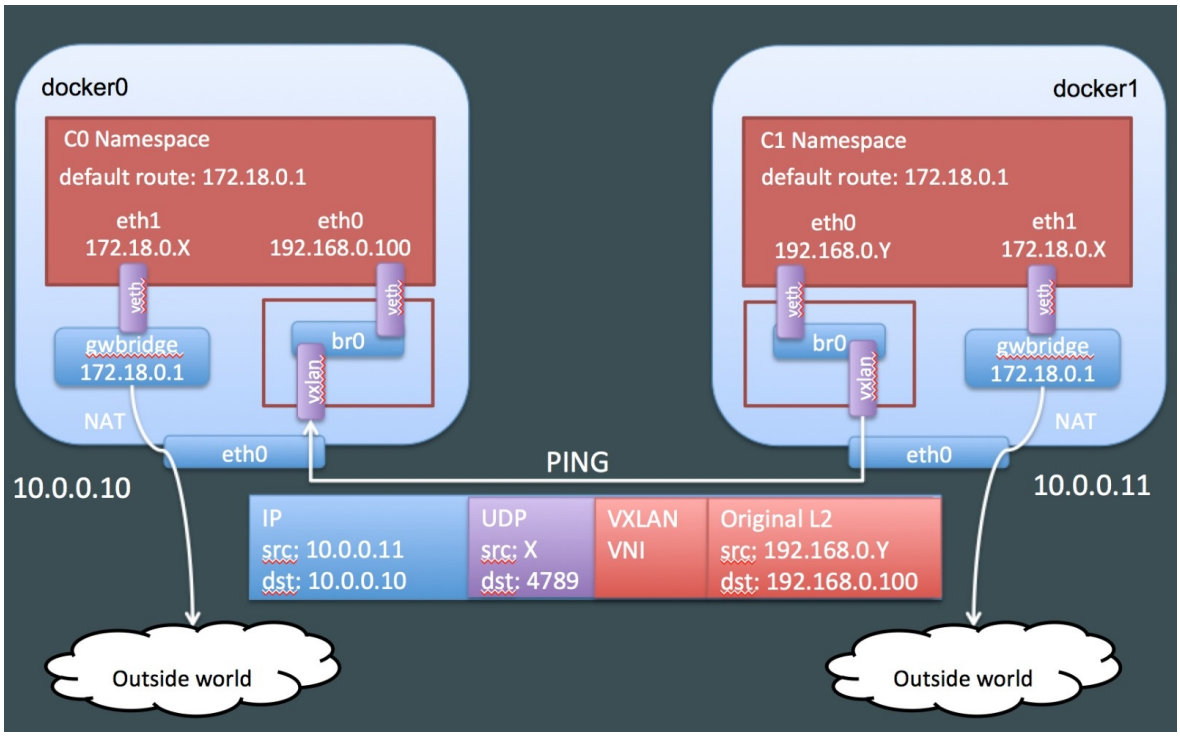

Figura 2.2: Diagrama de conexiones en una red overlay entre dos contenedores

La otra interfaz, eth1, que tiene el servidor funciona como un NAT para darle conectividad a internet al contenedor. Si quisiéramos restringir la conexión al exterior de los contenedores eliminando esta interfaz eth1 usaríamos la opción, –internal, al crear la red.

2.2. Herramienta de análisis de vídeo

En la introducción hemos hablamos de datos que van desde mediciones de fenómenos naturales hasta comportamientos relacionados con los seres humanos, pero en este proyecto para obtener resultados comparables debemos elegir un tipo de dato concreto. Hemos optado por procesar vídeo utilizando una herramienta [8] que analice estos vídeos. La aplicación captura los fotogramas de un vídeo uno a uno y a cada uno de ellos le aplica un modelo de red neuronal que detecta un objeto, como resultado da un archivo de texto con los objetos que ha detectado clasificados por tipos y con su posición, tanto temporal(nº de fotograma) como su localización en el fotograma(da dos coordenadas trazando un rectangulo que encierra

al obejeto). La herramienta está programada en python usando la librería de OpenCV para el análisis de la imagen. Para adaptarla a nuestro proyecto le hemos hecho pequeños cambios al script de lanzamiento de la herramienta, de forma que nos deje elegir el vídeo a analizar y la calidad de ese análisis (numero de fotogramas analizados).

La herramienta requiere de unos paquetes de software para su funcionamiento, la mayoría de ellos son paquetes de python a excepción de tensorflow. Este paquete ha ocasionado problemas durante la realización del TFG, puesto que su compilación en nodos con arquitecturas ARM, como las de las Raspberrys utilizadas en el despliegue (como se describirá posteriormente), es un proceso lento y que puede fallar debido a la capacidad limitada de estas máquinas. Por este motivo, el dockerfile que crea la imagen de los contenedores para las Raspberrys es distinto al del resto de máquinas, en él descargamos los paquetes ya compilados para este tipo de arquitectura.

2.3. Golang

Go[2] es un lenguaje de programación opensource, creado por Google en 2009. Tiene una sintaxis parecida a la de C salvando algunas diferencias como por ejemplo, las declaraciones al revés, donde el tipo de dato se escribe después del nombre de la variable (por Ej. function(texto string)). Admite el paradigma de programación orientada a objetos, pero no tiene herencia de tipos y tampoco palabras clave que demuestren que soporta este paradigma. Por último Go esta orientado a aprovechar sistemas con múltiples procesadores y procesamiento en red.

Elegimos Go en lugar de, por ejemplo, java que se ve en el grado principalmente por ésta última característica. Si los comparamos en una de las tareas básicas de concurrencia como lanzar un hilo, en Go basta con escribir "go function()", y con eso habríamos terminado. Mientras que en Java tendríamos que hacer una clase que implemente la interfaz "runnable", luego construir el objeto, crear un objeto thread y por ultimo lanzarlo. Es evidente que Go supone una simplificación enorme a este proceso.

2.4. CLIPS

CLIPS es una herramienta de desarrollo de sistemas expertos. Los sistemas expertos son una de las aplicaciones de la inteligencia artificial, pretende simular el

[2] `https://go.dev/`

razonamiento humano de la misma manera que lo haría un experto en su área. De esta forma se pretende simular un escenario en el que en cada nodo haya un experto tomando las decisiones correspondientes sobre los datos que recibe. La principal diferencia con los lenguajes de programación tradicionales es que estos expresan el conocimiento de forma imperativa (*existe para ello el operador de composición secuencial*), mientras que CLIPS es un lenguaje declarativo. La diferencia es enorme, porque en un código declarativo es el sistema el que busca la solución, no hay un flujo programado en sí mismo (una instrucción a continuación de la otra), sino que hay un conjunto de reglas y es el sistema el que realiza una inferencia a partir de ellas.

CLIPS estructura el conocimiento en hechos y reglas. Los hechos son información sobre el entorno que usa para razonar. Mientras que las reglas son los elementos que permiten que el sistema evoluciones, normalmente modificando hechos. Esa modificación puede ser directa sobre la base de hechos almacenada o como consecuencia de cambios en el entorno (por ejemplo, si una regla decide a quien enviar un vídeo los hechos serían la monitorización de los enlaces, la capacidad disponible en los nodos, etc.). Así pues CLIPS es capaz de *inferir* conocimiento complejo de alto nivel a partir de métricas de bajo nivel.

Para integrar las funciones de CLIPS en Go hemos utilizado CLIPSGo [15], es un proyecto de código abierto que permite una integración bidireccional entre CLIPS y Go. *Fue desarrollado originalmente por la empresa Keysight en 2020.*

2.5. IPerf

Es una herramienta utilizada para crear flujos de datos TCP y UDP y con ello medir el rendimiento de la red. En nuestro caso la usaremos para medir el ancho de banda disponible entre los enlaces.

Antes de elegir IPerf estudiamos utilizar otras herramientas como Yaz. Yaz es una herramienta más especializada en realizar una aproximación del ancho de banda rápidamente. Utiliza la técnica de Packet train, midiendo la dispersión de varios paquetes es capaz de estimar el ancho de banda disponible. Mientras que IPerf manda paquetes de un nodo a otro, midiendo la cantidad de datos que ha sido capaz de transmitir. Esto lo hace más intrusivo por estar ocupando el ancho de banda disponible aunque sea por un periodo corto de tiempo. Lo que nos ha hecho elegir IPerf en lugar de Yaz es, la inconsistencia en sus estimaciones y la complejidad de su configuración.

Los parámetros para medir el ancho de banda en Yaz varían en función del escenario, por lo tanto, cuanto peor escojamos los parámetros mayor error habrá en la estimación. Algunos de estros parámetros son: el número de paquetes por stream, el tamaño del payload de los paquetes, la distancia temporal entre dos paquetes, los streams por medida, etc. A la hora de estimar el ancho de banda Yaz variaba mucho su resultado frente a pequeñas variaciones en estos parámetros, cometiendo algunas un gran porcentaje de error. Es por esto que escogemos IPerf como primera aproximación para tener medidas consistentes del ancho de banda.

2.6. Traffic Control - tc

Traffic Control (a partir de ahora tc) [7] es una herramienta que nos permite configurar el tráfico en el kernel de linux. En nuestro proyecto la hemos utilizado para limitar el ancho de banda de los enlaces generando distintos escenarios para poner a prueba el sistema desarrollado.

Para mostrar como se introduce una limitación del ancho de banda vemos el siguiente ejemplo: **tc qdisc add dev eth0 root tbf rate 1mbit burst 32kbit latency 400ms**

Si ejecutamos este comando en el shell de linux estableceremos una limitación de un megabit por segundo en la interfaz eth0. Para entender mejor que estamos haciendo lo desglosaremos por partes: **qdisc** hace referencia a la cola FIFO que hay entre el kernel y la tarjeta de red, por lo que la regla que estamos añadiendo se refiere únicamente al tráfico saliente. Seguimos, **add dev eth0 root**, indican que añadimos una regla con la prioridad del usuario root para la interfaz eth0. **tbf** usa el algoritmo de Token Bucket [6] para ralentizar el tráfico. A partir de aquí son parámetros del algoritmo:

- **rate**: Es la velocidad máxima permitida en la interfaz.
- **burst**: Es el tamaño máximo permitido para el buffer, a partir de ahí se empezaran a descartar los paquetes.
- **latency**: Es el tiempo máximo que puede estar un paquete en el buffer antes de ser descartado.

Capítulo 3

Diseño e Implementación

En este apartado veremos cómo se ha diseñado y desarrollado el sistema en el orden de implementación de las funcionalidades. Luego pasaremos a detalles más concretos de la implementación viendo las estructuras de datos definidas y qué funciones cumplen.

3.1. Diseño de red

En un sistema distribuido de este tipo lo normal es que los nodos que forman la red estén en diferentes localizaciones, por lo que, deberemos hacer uso de redes externas para encaminar nuestros paquetes. En un escenario típico los nodos próximos a la fuente de datos tienen una capacidad limitada (pueden ser la propia fuente), los nodos con altas capacidades de computación se encuentran más alejados en la nube, donde podríamos mandar lo datos generados para procesarlos sin limitaciones de computación y también posibles nodos de capacidad intermedia, también ubicados en una localización intermedia. Las ubicaciones es un parámetro a tener en cuenta, es posible que tengamos que mandar gran cantidad de datos a ubicaciones remotas a través de la red. Para modelar este escenario hacemos uso de tres tipos de nodos con capacidades diferentes: una máquina muy potente que actúa a modo de centro de datos; un macmini antiguo de capacidad intermedia; cuatro raspberrys pi model 3.

- Nodo con gran capacidad de recursos: una estación de trabajo Workstation Intel Xeon SkyLake-SP 3106 dual con 480GB de disco SSD, 8TB de disco duro y 128 GB de memoria RAM.
- Nodo de capacidad intermedia: Macmini antiguo de hace 9 años.
- Nodo de recursos limitados: cuatro Raspberry Pi 3 model B, cuentan con un procesador Quad Core 1.2GHz y 1GB de memoria RAM.

Las Raspberry al ser las máquinas con menor capacidad de computo simulan estar al lado de la fuente de datos. Esta fuente podrían ser por ejemplo el conjunto de cámaras de vigilancia de un edificio, y las Raspberrys reciben los vídeos de las cámaras o incluso las cámaras podrían estar directamente conectadas a las Raspberrys, siendo estas la fuente directamente. El Macmini emula un posible nodo de capacidad intermedia ubicado más cerca de la fuente que el centro de computación en la nube, que es emulado por el Intel Xeon.

Por otro lado, la conexión de red entre los distintos nodos se realiza mediante la red overlay (descrita en el capítulo 2.1.2), creando túneles entre los los distintos contenedores haciéndoles ver que están en una red local, pero en realidad sus paquetes se encaminan a través de la red, por lo que, cada una de las conexiones de red tendrá características diferentes. En nuestro caso, todos los nodos están ubicados en la misma red física, pero hemos emulado las distintas características de red mediante el software tc (descrito en el capítulo 2.6). En la imagen 3.1 podemos ver un resumen gráfico de todos los componentes del sistema en tres niveles, físico, virtual y de ejecución.

3.2. Diseño y funcionamiento de un nodo

Como se ha comentado en la introducción, las aplicaciones suelen tener unas restricciones de QoS que se reflejan en una fecha límite o deadline. Por diversos motivos, tales como aspectos de privacidad, resulta interesante el procesado cerca de la fuente de los datos. Sin embargo, generalmente esos nodos son de baja capacidad de computación, por lo que dependiendo del deadline existente será viable el computo cerca de la fuente o será necesario el envío hacia nodos de mayor capacidad computacional.

Antes de ponernos a implementar nada debemos saber cuáles son las tareas que puede realizar un nodo dentro del sistema. La finalidad de la red es el procesado distribuido y descentralizado de datos, por lo que cuando un nodo tiene los datos, ha de decidir que hace con ellos. Esta decisión se toma en base a la urgencia de los datos, es decir, cada dato tiene una fecha limite (deadline) que cumplir, cuanto mas alejada sea la fecha menos urgente será, y por lo tanto puede ser procesado por nodos menos potentes. O por el contrario, si la fecha es próxima debería procesarse con mayor brevedad. Para que un nodo sepa si los datos son urgentes se lleva a cabo una estimación del tiempo que puede transcurrir para procesar los datos sin exceder esa fecha límite. Para ello, y como se ve en la figura 3.2, se tiene en cuenta: el tiempo de

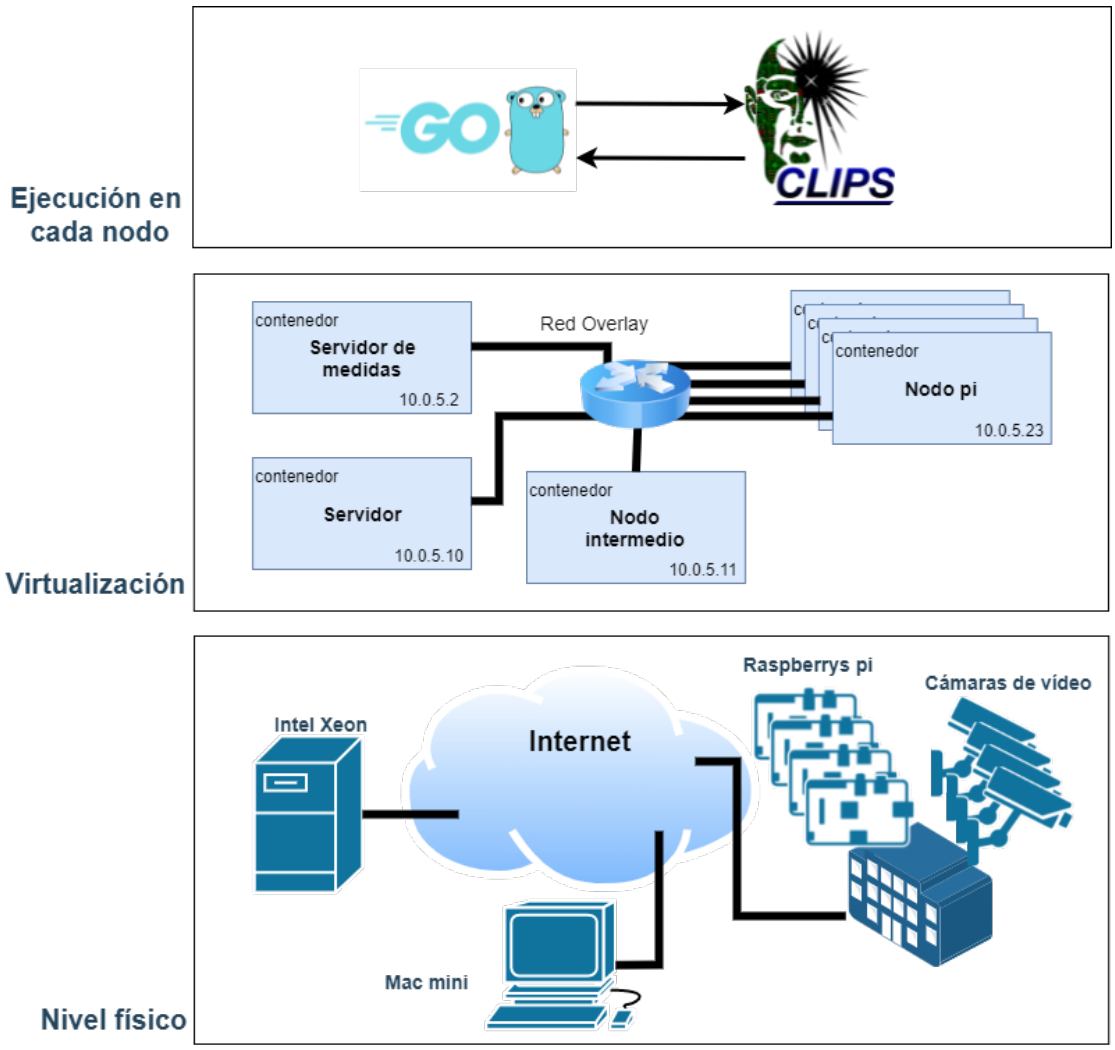

Figura 3.1: Resumen gráfico de la implementación del sistemas

procesado *(Tpr)*; el tiempo de transmisión *(Tx)*; el tiempo de espera en cola *(Tq)*. La suma de los tres ha de ser menor que el tiempo hasta el deadline, e idealmente el momento en el que los datos terminen de procesarse debería ser lo mas cercano posible al deadline.

La complejidad de la decisión está en la planificación, es decir, lo extenso que sea el abanico de acciones. En nuestro caso el sistema a implementar es simple porque tenemos dos posibles acciones: procesar en local; procesar en remoto. Y una tercera que no se ha llegado a implementar totalmente como se explicará en el capítulo 5, procesado parcial, que sería una mezcla de las dos anteriores.

En la figura 3.3 vemos cómo se estructuran las funcionalidades que requiere el nodo para tomar las decisiones y llevarlas a cabo. Para procesar en remoto necesitamos las mismas capacidades que en local: ser capaces de procesar los datos y estimar cuanto

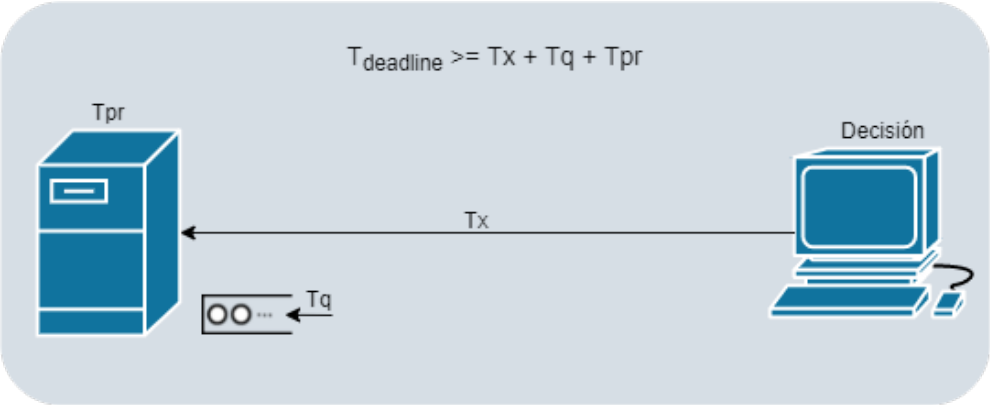

Figura 3.2: Estimación del tiempo para cumplir el deadline

tiempo va a costar ese procesado. Además para procesar en remoto el nodo necesita poder enviar y recibir datos, saber quiénes son sus nodos vecinos y sus capacidades. Por otra parte, también necesitamos monitorizar las métricas que influyen en el tiempo de envío (ancho de banda disponible) y de cola (ocupación del nodo), para ser capaces de hacer una estimación de estos tiempos.

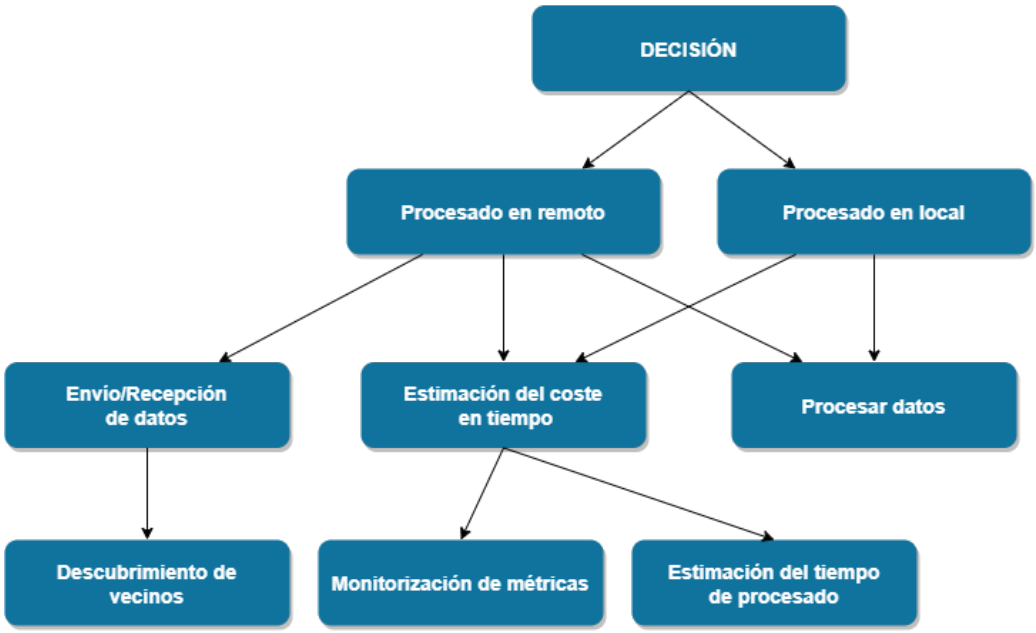

Figura 3.3: Estructura de funcionalidades

3.2.1. Envío y Recepción de vídeo y descubrimiento de vecinos

Antes de que un nodo envíe datos debe saber a quienes puede mandárselos. Este descubrimiento de vecinos se realiza a través de un archivo de configuración en el que vienen especificados cuatro parámetros:

- La dirección IP de los nodos vecinos.
- Un alias a modo de identificador de cada nodo.
- El tipo de nodo que es dentro de los tres que ya hemos mencionado.
- Los slots de procesamiento de los que dispone ese nodo, es decir, cuantos vídeos puede procesar en paralelo.

Todos estos datos se guardan en un mapa de pares "key-value", donde la llave es el alias del nodo y el valor una estructura de datos tipo "nodo"donde se almacenan estos valores y también las métricas correspondientes a ese nodo.

Este proceso de configuración sucede una vez, de modo que al iniciar el contenedor y ejecutar la aplicación lo primero que hace es crear el mapa de nodos. Con este conocimiento el nodo ya puede elegir a quien mandar vídeo en caso de no procesarlo en local. Así pues, cuando se ha tomado la decisión de procesar en remoto y ha de mandar el vídeo tenemos dos posibles escenarios de partida:

- El nodo tiene el vídeo completo en disco.
- Se esta recibiendo el vídeo de otro nodo del sistema y, por lo tanto, no tiene el vídeo completo.

En ambos casos el procedimiento de envío es el mismo, excepto que en el primero se trocea el vídeo en chunks (trozos) y se manda. En el segundo caso, el nodo no tiene que trocearlo porque ya tiene los chunks en disco, los reenvía y, en el caso de que el enlace de recepción sea mas lento que el de envío, espera a completar un chunk para mandarlo. Esta decisión de trocear los vídeos en varios chunks viene dada para evitar varias llamadas al sistema de escritura en disco, en las que se escriban pocos bits. Esto ralentiza significativamente la ejecución. Partiendo el vídeo en chunks se guardan en memoria los bits recibidos hasta completar un chunk, es entonces cuando hace una sola llamada al sistema para la escritura y borra de memoria el chunk. A parte, este proceso nos permite reenviar un vídeo al mismo tiempo que lo estamos recibiendo, ahorrando espacio en disco.

Por último, en recepción, se van recibiendo los chunks mientras se toma la decisión de qué hacer con el vídeo. Como ya hemos visto, se pueden reenviar para procesar el vídeo en remoto, o procesarse en local. Para procesarlo en local el nodo espera a recibir todos los chunks y reescribirlos para formar el archivo original. En la figura 4.1 podemos ver el flujo de la ejecución de este proceso en el caso especifico de que la

decisión sea con CLIPS. Como veremos en la sección 3.2.4, podemos elegir el tipo de decisión al lanzar cada nodo.

3.2.2. Procesado de datos

En principio todas las máquinas pueden hacer uso de la herramienta para procesar vídeo, pero las Raspberries llegan a los requisitos mínimos muy justas, sobre todo en memoria RAM. Esto es un problema, que al añadir el código desarrollado a los hilos de ejecución del propio sistema hace que deje de funcionar todo por falta de memoria. Así pues, aunque las Raspberries sean capaces de procesar vídeo, hemos desactivado esa funcionalidad poniendo a cero los slots disponibles.

Con el fin de procesar un vídeo, el nodo en cuestión debe reconstruir el archivo original antes de lanzar la función de procesado. Reconstruido el archivo procede a actualizar su número ocupación al nodo de medidas, de forma que el resto de nodos puedan calcular el tiempo estimado en cola correctamente. Antes de lanzar el proceso de la herramienta de vídeo comprueba que haya slots disponibles para procesarlo. Si no hay el hilo queda bloqueado en una cola FIFO hasta que le llegue su turno. Cuando la función consigue un slot para procesar hace una llamada al sistema que lanza la herramienta de procesado de vídeo. Al terminar de analizarlo, vuelve a actualizar su valor de ocupación y notifica al nodo de medidas el resultado, donde es apuntado en un registro que como veremos en el capítulo 5 utilizaremos para analizar los resultados. En la figura 3.4 podemos ver un resumen gráfico de la ejecución.

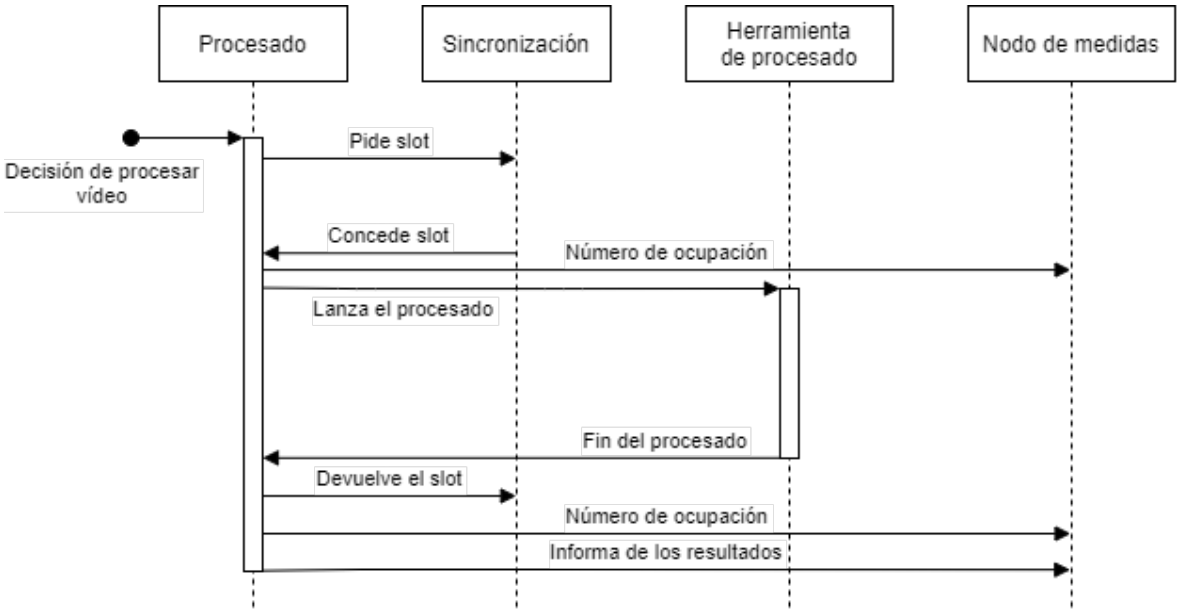

Figura 3.4: Intercambio de mensajes durante la función de procesado.

3.2.3. Estimación de tiempos

La estimación de los tiempos la conseguiremos monitorizando los parámetros del sistema en tiempo real. Para compartir las medidas entre los nodos hay muchas posibles soluciones, nosotros hemos optado por un servidor aparte de los nodos que procesan vídeo, para que se encargue de almacenar las medidas. Si bien existen otras alternativas que pueden ser más eficientes, el objetivo del proyecto no se centraba en el sistema de monitorización y se ha considerado que esta era una solución rápida y viable que permite que todos los nodos utilicen medidas reales de la red.

Tiempo de procesado

Para poder realizar una estimación del tiempo de procesado debemos medir el tiempo que le cuesta a cada tipo de nodo procesar cada tipo de vídeo. La figura 3.5 muestra el conjunto de vídeos, sus características y el tiempo medio medido para cada vídeo en cada tipo de máquina. Vemos que tenemos tres vídeos más ligeros en orden de mayor a menor tamaño: 240p1; alex; trafico. Y luego los otros dos son notablemente más pesados. Veremos luego en el análisis de los resultados la relevancia de estas características.

Vídeos						Server	MacMini	pi200
Tamaño(Kb)	Nombre	Resolución(p)	Fotogramas	fps	duracion (s)	tiempo(s)	tiempo(s)	tiempo(s)
2035	alex.mp4	720x576	619	10,00	61,90	12,36	36,10	62,14
1195	240p1.mp4	426x240	489	24,00	20,38	8,20	20,50	96,34
2781	trafico.mp4	400x300	1499	23,98	62,51	12,14	35,98	142,90
38037	puertaSol.mp4	960x540	9741	25,00	389,64	49,80	210,46	728,70
23143	Interseccion.m4v	960x540	8267	25,00	330,68	45,72	175,40	650,96

Figura 3.5: Características del conjunto de vídeos

Con estas medidas los nodos pueden estimar el tiempo de procesado medio en cada tipo nodo, aunque realmente el proceso que siguen para la estimación no es consultar la medida, pero el resultado al que llega será el mismo que aparece en la tabla. Esto es porque la estimación del tiempo de procesado está preparada para hacer procesados parciales de los vídeos, dividiendo un vídeo en partes más pequeños y procesándolas en varias máquinas. Finalmente el sistema no realiza esta tarea puesto que no hay una decisión implementada que la ejecute. Entonces para estimar el tiempo de procesado el nodo consulta un modelo de regresión lineal introduciendo el tipo de vídeo y el número de fotogramas, llegando igualmente a los valores de la tabla puesto que la longitud del vídeo siempre es la misma. Los detalles sobre cómo se han obtenido los

modelos para cada vídeo están en el capítulo 5.

Tiempo de espera en cola

Los nodos comparten sus datos de ocupación con este servidor, y cuando un nodo necesita estimar el tiempo en cola, le pide el dato del nodo que está evaluando para procesar en remoto. En el caso de que todos los slots del nodo que va a procesar estén ocupados, la petición entrará en cola. El modelado matemático de esta cola no es directo, puesto que desconocemos tanto el proceso de llegada como la distribución del tiempo de servicio. Incluso modelándolo como una M/G/K, siendo K el número de slots disponibles en la máquina, la resolución no es directa. En cualquier caso, como los nodos conocen la ocupación del resto de nodos gracias al mecanismo de monitorización (aunque no el tipo de vídeo que se está procesando), podemos realizar la siguiente aproximación para tener una estimación del tiempo de espera: utilizamos como tiempo de servicio μ la media de los cuatro tipos de vídeo. Así, siendo N el número de peticiones en cola: $T_q = \mu \cdot N$.

Tiempo de transmisión

Este caso es algo más complejo, la herramienta que usamos para estimar las capacidades de los enlaces (IPerf) es muy intrusiva, por lo que si cada nodo decidiese por sí mismo cuándo medir el ancho de banda, podría darse el caso de que coincidieran dos o más nodos midiendo a la vez, estropeando la medida. Por esto, es el servidor de medidas el encargado de avisar a los nodos de cuándo deben medir el ancho de banda, evitando así solapamientos. Con el dato del ancho de banda y el tamaño del vídeo, estimamos el tiempo de transmisión: $T_x = BW \cdot bits$

El servidor de medidas tiene una serie de **timers**, actúan como una alarma periódica. Cuando salta la alarma manda una señal, el servidor de medidas la recoge y le pide al nodo en cuestión que mida el ancho de banda. Solo hay un hilo encargado para esta tarea, de forma que las peticiones para actualizar el ancho de banda serán secuenciales. El nodo que recibe el mensaje hace una llamada al sistema para lanzar IPerf en el enlace indicado por el servidor. La salida estándar queda vinculada a una variable en Go, cuando IPerf termina de ejecutarse el nodo trata el string de salida para obtener el dato del ancho de banda en bits/s. Para finalizar el proceso el nodo notifica al servidor que ha terminado de medir. En la imagen 3.6 podemos ver un resumen gráfico de la ejecución.

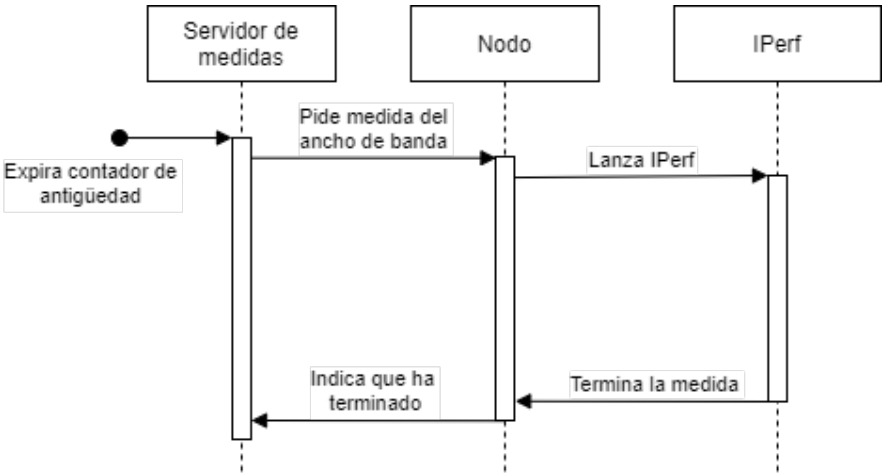

Figura 3.6: Ejecución del proceso de medida del ancho de banda

Si bien se trata de unas estimaciones muy simplificadas, con un importante margen de mejora, nos proporcionan una base para poder desarrollar el objetivo principal del proyecto, que es la toma de decisiones

3.2.4. Decisión

Una vez somos capaces de realizar las estimaciones de los tres tiempos, ya podemos empezar a hablar de la toma de decisiones en base a la fecha limite. Procesar en local tiene ventajas: la primera es que los datos de la ocupación son totalmente fiables porque son propios; segunda es que se elimina el tiempo de transmisión. Por lo que nos queda únicamente el tiempo de espera en cola y el tiempo de procesado. Si la máquina está muy ocupada, o es poco potente y el vídeo a procesar pesado, será cuando esta opción no sea viable.

Por la otra parte, procesar en remoto tiene las desventajas de la incertidumbre de las medidas y la suma del tiempo de transmisión, pero para máquinas poco potentes o vídeos con un deadline más ajustado, puede ser mejor opción que procesar en local. Los detalles se proporcionan en el apartado 3.4 y en el capítulo 4

3.2.5. Ejecución

Después de ver todas las funcionalidades que posee cada nodo es interesante ver como se usan, como transcurre el flujo de ejecución. Como ya hemos dicho, lo primero que hace un nodo es el proceso de auto-configuración en el que lee los archivos correspondientes para conocer a sus vecinos y sus capacidades. Después el hilo principal

del nodo lanza un servicio de escucha en el puerto 9544, donde espera recibir mensajes ya sean archivos de vídeo o relacionados con las mediciones. En el caso de la Raspberrys antes de quedarse escuchando en el puerto, lanzan un hilo encargado de emular una cámara de vídeo. En un directorio tiene el vídeo que va a emular la cámara de seguridad y un archivo de configuración con la tasa de envío, el nombre del vídeo y el número de fotogramas. Se configura un **timer** con la tasa de envío y cada vez que expire lanzara la función de decisión. En la figura 3.7 vemos los tipos de mensajes que le pueden llegar a un nodo y su proceder.

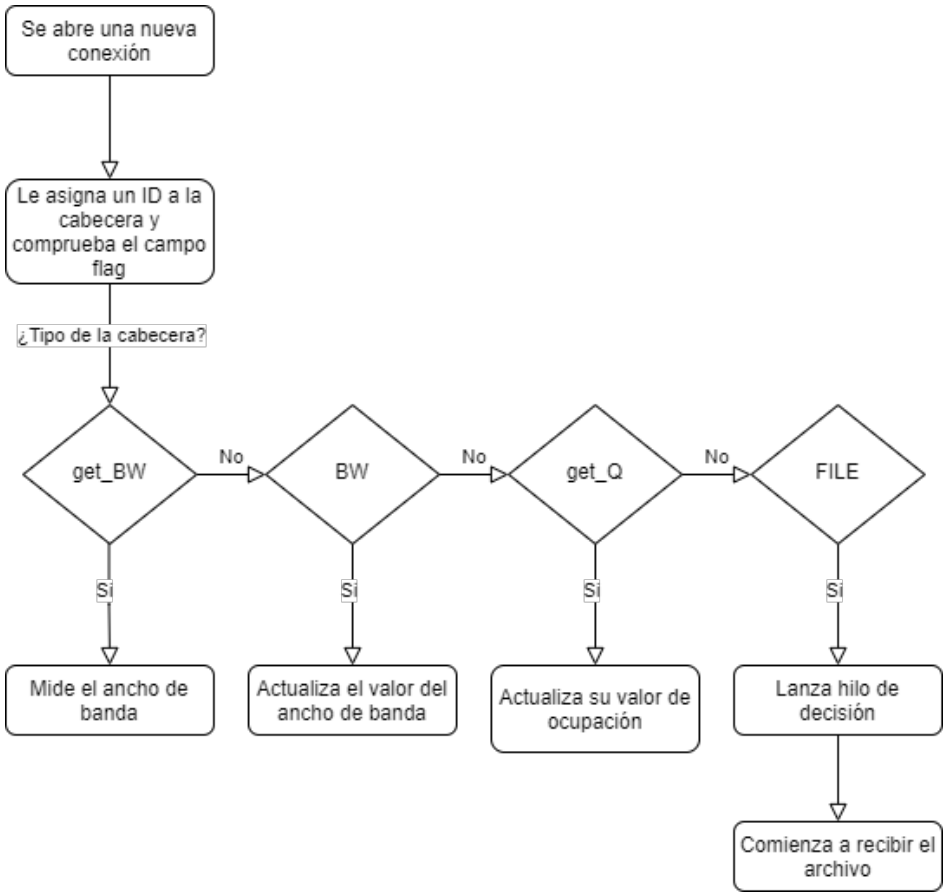

Figura 3.7: Diagrama de estados del hilo de recepción.

- El mensaje "get_BW" ejecuta el procedimiento que describe la figura 3.6.

- El mensaje "BW" simplemente actualiza un valor de ancho de banda en el mapa de vecinos.

- El mensaje "get_Q" lo utiliza el servidor de medidas cuando un nodo lleva un tiempo inactivo. El nodo al recibir este mensaje actualiza su valor de ocupación.

– El mensaje "FILE" lo manda un nodo a otro para enviarle un vídeo. El nodo se prepara para recibir el vídeo, ejecuta el procedimiento descrito en la figura 4.1 para recibir el vídeo y para procesarlo igual que hemos visto en la figura 3.4.

3.3. Estructuras de datos

En esta sección hablaremos de las estructuras de datos implementadas y qué función cumplen. En la figura 3.8 vemos un resumen de las estructuras de datos, las desarrollaremos una por una:

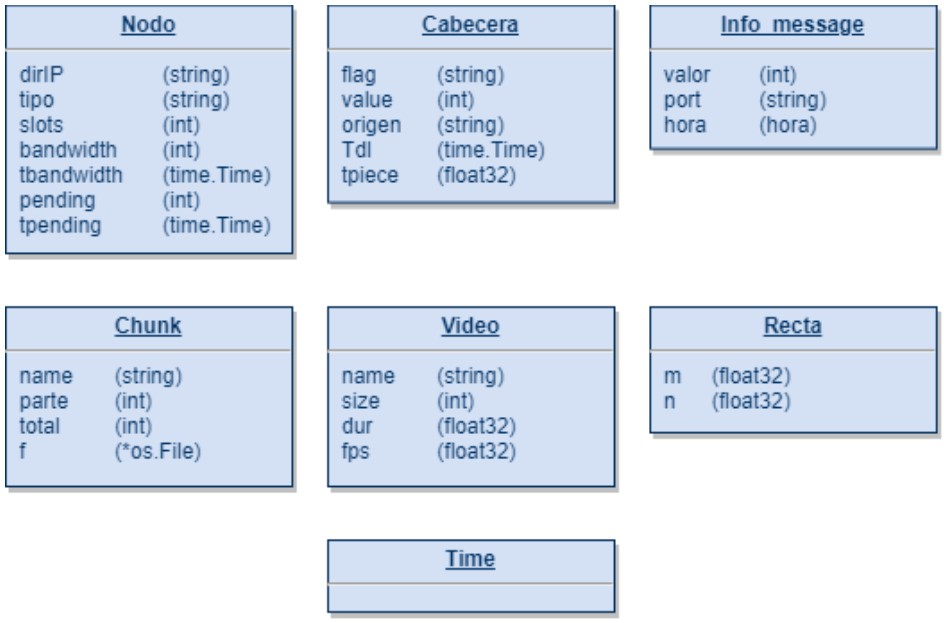

Figura 3.8: Estructuras de datos

– Nodo: Recoge los parámetros necesarios para identificar un nodo y también sus datos de estado. **dirIP**: Dirección IP del nodo. **tipo**: Puede ser uno de los tres tipos. **slots**: Número de instancias de procesado de vídeo que soporta en paralelo. **bandwidth**: Valor del ancho de banda del enlace con ese nodo. **tbandwidth**: Antigüedad del dato de ancho de banda. **pending**: Número de vídeos pendientes por terminar de procesar. **tpending**: Antigüedad del dato de pending.

– Cabecera: Siempre se manda al inicio de una conexión y recoge los parámetros necesarios para identificar la finalidad de la conexión y llevarla a cabo. **flag**: Marca el tipo de conexión. **value**: Campo reservado para números enteros. **origen**: Campo reservado para texto. **Tdl**: Es utilizado para indicar un instante de tiempo, normalmente el deadline de un vídeo. **tpiece**: Vale cero siempre, este campo está pensado en un principio para indicar que el vídeo en recepción esta cortado indicando la duración del corte.

- Info_message: sirve como formato para actualizar el estado de un nodo al servidor de medidas a través de la red. **value**: Valor de pending del nodo. port, sirve como identificador. **hora**: Indica el la antigüedad del dato.

- Chunk: Sirve para identificar las partes que componen un archivo de vídeo. **name**: Nombre del vídeo. **parte**: Identifica que parte del archivo total es. total, indica el total de partes en las que se ha divido el vídeo original. f, es el descriptor de archivo del chunk.

- Vídeo: Recoge información de un vídeo. **name**: Nombre del vídeo. **size**: Tamaño en bits. **dur**: Duración del vídeo en segundos. **fps**: Fotogramas por segundo.

- Recta: Guarda los valores de regresión lineal del coste en tiempo de procesar un vídeo en función de su duración. **m**: Indica la pendiente de la recta. **n**: La ordenada en el origen.

- Time: Esta estructura de datos pertenece a la librería time de Go, la utilizamos para guardar los tiempos y calcular diferencias entre ellos teniendo en cuenta las zonas horarias.

3.4. Reglas estáticas

Una vez vistas todas las funcionalidades del nodo dentro del sistema, podemos hablar de toma de decisiones. Las reglas estáticas se implementan con una sucesión de condicionales IF THEN ELSE, por eso son estáticas: se define una estrategia y esa permanece invariante al funcionamiento del sistema. Tal y como se ha comentado en apartados anteriores, en muchos casos una estrategia deseada es procesar los datos lo más cerca posible de la fuente, evitando en la medida de lo posible que los datos tengan que ser transmitidos a centros de datos. Por esta razón, la estrategia definida consiste en que si el deadline lo permite, el proceso se realiza localmente o en los nodos más cercanos, dejando el servidor de alta capacidad como última opción.

En la figura 3.9 vemos la implementación en Go de las reglas estáticas. Como el sistema desarrollado no es muy complejo, no tenemos muchas posibles acciones y las reglas no quedan muy extensas. Se ejecuta el mismo código en todas las máquinas, por eso en primer lugar se comprueba qué tipo de máquina es. Si es el servidor, decide ejecutar directamente, porque, como hemos dicho, evitamos procesar lejos de la fuente. Si un nodo decide procesar en el servidor es porque no tenia más opción. Luego sigue el caso de que no sea el servidor, comprobando si le da tiempo a procesar en local. En

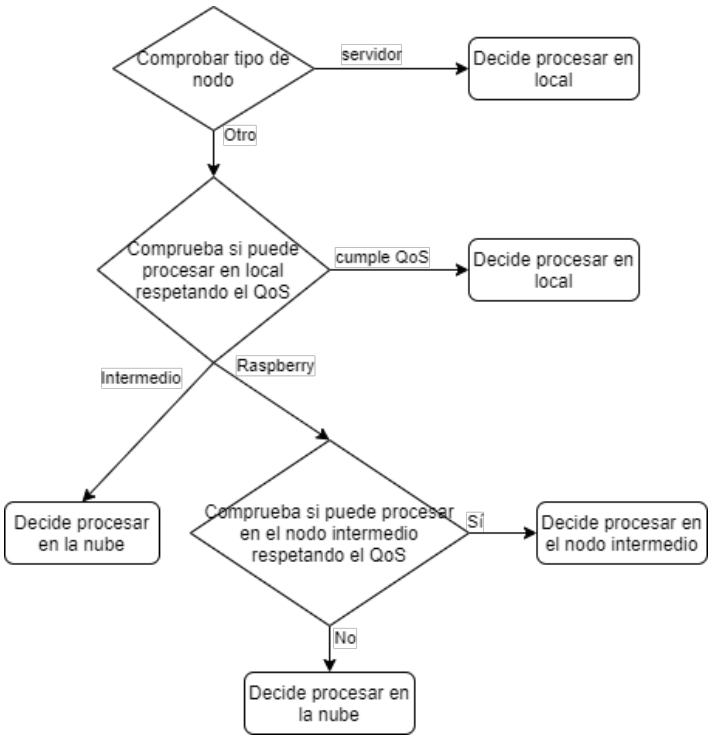

Figura 3.9: Implementación de las reglas estáticas

caso de que no haya tiempo, comprueba qué tipo de nodo es. En este caso ya solo puede ser nodo intermedio o Raspberry. Si es nodo intermedio decide mandar al servidor. Si es Raspberry entonces calcula si da tiempo a procesar en el nodo intermedio antes que en el servidor.

Capítulo 4

CLIPS reglas dinámicas

En este capítulo mostraremos como funciona CLIPS con un ejemplo típico del paradigma imperativo. Luego entraremos a fondo en las reglas implementadas y como hacen evolucionar los hechos.

4.1. Funcionamiento de CLIPS

Como hemos dicho en la sección 2.4, CLIPS estructura el conocimiento en hechos, que van evolucionando según las reglas establecidas. Podríamos decir que los hechos son información del entorno que CLIPS usa para razonar y establecer nuevos hechos o conocimientos más complejos que los de partida. Mientras que las reglas son las encargadas de hacer que el sistema evolucione, extrayendo conocimiento del conjunto de los hechos, normalmente modificando hechos o creando nuevos.

Para a empezar conocer el funcionamiento de CLIPS [11] lo más esencial es añadir y eliminar hechos de la base de hechos, sabiendo que todos los comandos van delimitados por paréntesis, `(assert (hecho))` para añadir, `(retract id)` para borrar. Los hechos pueden tener varios campos, por ejemplo: `(assert (nombre-edad Carlos 32))`. Las reglas en cambio tienen dos partes, se conocen como la parte izquierda (left-hand side LHS) y la parte derecha (right-hand side RHS), están separadas por por una flecha del tipo: si A entonces B (A =>B). En la LHS están las condiciones para que se active la regla y en la RHS están las acciones que se toman cuando se activa, para seguir con el ejemplo anterior, veremos una regla que determina si el sujeto es mayor de edad:

```
   (defrule Adulto
?id <- (nombre-edad ?nombre ?edad)
(test (<= 18 ?edad))
=>
```

```
(assert (adulto Carlos))
(retract ?id))
```

Es un ejemplo muy sencillo pero que nos deja ver el mecanismo de las reglas. La primera condición es que haya un hecho cuyo primer campo sea 'nombre-edad' y además tenga dos campos más cuales sean: empezando una palabra por interrogante indicamos a CLIPS que guarde ese valor para utilizarlo de nuevo más tarde. De esta forma guardamos el identificador de la regla en la variable "?id". La siguiente función `(test)` nos sirve para comprar si la edad es mayor o igual a 18, al cumplirse todas las condiciones la regla se activa y ejecuta las acciones de la RHS. En este caso, cuando la regla se active añadirá un hecho a la base de hechos (adulto Carlos). Por último, borra el hecho que ha activado la regla haciendo un `(retract)`. Para este ejemplo solo tenemos una regla, pero existe un valor de "salience" en cada regla que fija su prioridad, cuando hay más de una regla por activarse lo hacen en orden de mayor valor de "salience" a menor. Si no lo modificamos tiene un valor por defecto de cero, y un rango de [-10000, 10000].

Además de la estructura de hechos y reglas, otra de las principales diferencias de CLIPS con los lenguajes tradicionales como C, Java, python, etc... es el uso del paradigma declarativo. En el paradigma imperativo se indican todos los pasos hasta llegar a la solución del problema, mientras que en el declarativo solo se describe el problema a resolver a nivel de usuario, no se describen los pasos para resolverlo. Esto se realizará mediante mecanismos de inferencia. Podemos decir que en el imperativo el flujo de ejecución del programa esta previamente cableado, mientras que en el declarativo *no hay un flujo concreto y preestablecido de ejecución*. Para entender esto vamos a ver como se resuelve en CLIPS un problema clásico, el cálculo del factorial [14].

Primero hemos de tener en cuenta que las reglas son elementos independientes que necesitan hechos para poder ser ejecutadas. Necesitaremos un hecho que defina el factorial de un número, por ejemplo `(fact 3 6)` representa que el factorial de 3 es 6. La forma mas sencilla de representar hechos en CLIPS es mediantes unordered facts, con ellos estableces relaciones entre elementos de cualquier tipo, la única restricción es que el primer elemento del hecho sea un símbolo. CLIPS maneja diferentes tipos de datos, los símbolos son cualquier secuencia de caracteres. También soporta cadenas de caracteres (strings) entrecomillados, números enteros, en coma flotante, etc. Podríamos hacerlo de dos formas diferentes, con reglas hacia delante y hacia detras. En este caso la regla será hacia delante, por lo que calculará el factorial de un número dado otro.

La regla lo que hará será establecer la recurrencia entre un factorial y el del número siguiente:

```
   (defrule factorial
(fact ?x ?y)
=>
(assert (fact (+ ?x 1) (* ?y (+ ?x 1)))))
```

Ahora la regla simplemente comprueba si hay un hecho `fact` y crea otro que relaciona el siguiente número con su factorial. Para ello tendríamos que introducir un hecho `(fact 1 1)`, pero nuestro programa calcularía el factorial de todos los número y no pararía nunca. A las reglas no podemos introducirles parámetros, por lo que necesitaríamos otro hecho u otro campo en el hecho introducido para indicarle el límite, `(limite 6)` y añadir un par de requerimientos en la LHS.

```
   (defrule factorial
?ef <= (fact ?x ?y)
(limite ?l)
(test (<?x ?l)) =>
(assert (fact (+ ?x 1) (* ?y (+ ?x 1))))
(retract ?ef))
```

Así establecemos un límite a partir del cual la regla dejará de activarse y eliminamos el hecho anterior para no dejar en la memoria los factoriales intermedios.

4.2. Reglas y Hechos implementados

El objetivo del uso de CLIPS en nuestro sistema es la toma de decisiones sobre donde procesar un vídeo dado un nodo y su conocimiento sobre el estado del sistema. igual que en las reglas estáticas seguimos una estrategia, aquí seguiremos una también. Dado que en CLIPS las reglas son elementos independientes, nos permite hacer separación de tareas de una forma muy sencilla. Hacemos una regla para una tarea y no nos podemos olvidar de ella. Esto también nos permite de una forma muy sencilla implementar un mecanismo de fidelidad de hechos, de forma que teniendo en cuenta la antigüedad de un hecho CLIPS pueda evaluar la veracidad de ese hecho. Esto casa muy bien con el paradigma distribuido puesto que los nodos tienen un conocimiento parcial del sistema y posiblemente desactualizado. También nos permite

descartar nodos como opción de procesado remoto, porque hayan visto comprometido su funcionamiento. Este mecanismo lo hemos implementado usando un coeficiente de fidelidad (CF a partir de ahora), siendo un número que como máximo puede valer uno, máxima fiabilidad, y cuanto menor sea menos fiable será la medida. Esta implementado en todos los hechos pero la estimación del tiempo de procesado siempre es la misma, por lo que ahí el valor del CF es estático.

Siendo Δt la diferencia de tiempo entre el instante de la toma de la medida y el instante actual en minutos, los CFs los calculamos tal que:

$$CF_q = \frac{(0{,}2\Delta t)}{\overline{T_s}}$$

$$CF_{bw} = \frac{10 - \Delta t}{10}$$

Siendo $\overline{T_s}$ el tiempo de servicio medio del nodo para el que se está calculando el CF. En el caso de la medida del ancho de banda se han establecido diez minutos puesto que el tiempo entre mediciones está sobre cinco minutos, dependiendo del escenario como comentaremos en el capítulo 5.

Como hemos visto en el apartado anterior, para que CLIPS pueda ejecutar sus reglas son necesarios los hechos. Desde Go cada vez que tomamos una decisión creamos un entorno vacío de CLIPS. Luego cargamos un archivo de configuración en el que van especificadas todas las reglas, es entonces cuando empezamos a introducir los hechos. Dependiendo del tipo de nodo introduciremos unos u otros. En el caso de las Raspberrys introducimos los hechos correspondientes al nodo intermedio y al servidor. En el nodo intermedio se introducen los hechos para procesar en local y en el servidor. Por último en el servidor se introducen los hechos para procesar en local y en el nodo intermedio para el caso de que quiera desviar carga de trabajo. Se introducen las métricas en forma de hecho con el siguiente aspecto:

```
(assert (transfer-to-calc inter server 23143 8000000 20000 0.86))
```

Introducimos las métricas haciendo uso de los **unordered facts** mencionados en el apartado anterior. Con este hecho relacionamos las métricas para la estimación de tiempo de transmisión. Establecemos: el tipo de nodo origen; el tipo en recepción; el tamaño del vídeo en kilobits; la velocidad del enlace en bits/s; tamaño de chunk; CF. Una regla se encarga de transformar este hecho en otro más útil:

```
(defrule calc_transfer
?rule <- (transfer-to-calc ?origen ?destino ?size ?bandwidth ?chunksize
?CF)
```

```
(test (neq ?origen ?destino))
=>
(retract ?rule)
(assert (estimated-transfer (from-to ?origen ?destino) (value
(tx_calculator ?size ?bandwidth ?chunksize)) (CF ?CF))))
```

En la LHS llamamos a la funcion test para comprobar que el origen y el destino no sean el mismo, ese caso seria el de procesado en local y el coste en tiempo de transmisión sería cero. En la RHS llamamos a la función "tc_calculator", CLIPS también soporta el paradigma imperativo permitiéndonos programar pequeñas funciones:

```
(deffunction tx_calculator
(?size ?bandwidth ?chunksize)
(float (/ (+ ?size (* 41 (/ ?size ?chunksize))) ?bandwidth)))
```

Tiene dos partes, la primera son los argumentos que se introducen a la función, y la segunda la acción de la función. La función calcula el tiempo que costará enviar el archivo de vídeo a través de la red.

Para las otras dos estimaciones introducimos los hechos de la misma forma, en el caso de la estimación en el tiempo de procesado introducimos los valores de la recta de regresión y el número de fotogramas:

```
(assert (exec-to-calc server 8267 0.00948 6.45))
```

La regla que se encarga del tiempo estimado de procesado:

```
(defrule calc_exec
?rule <- (exec-to-calc ?destino ?x ?m ?n)
=>
(retract ?rule)
(assert (estimated-exec (at ?destino) (value (lineal-regresion ?x ?m
?n)) (CF 0.8))))
```

En la RHS volvemos a llamar a una función que simplemente calcula el valor de la recta dados los fotogramas. Por último añadimos el hecho para el tiempo de espera en cola:

```
(assert (waiting-to-calc server 1 25.0 0.9))
```

Relacionamos: el tipo del nodo origen; los vídeos en cola; el tiempo medio de servicio; CF. También tiene una regla que se encarga modificar el hecho y añadir uno nuevo:

```
(defrule calc_waiting
?rule <- (waiting-to-calc ?destino ?cola ?tiempo-medio ?CF)
=>
(retract ?rule)
(assert (estimated-waiting (at ?destino) (value (* ?cola
?tiempo-medio)) (CF ?CF))))
```

Aquí al ser una operación directa la introducimos directamente en la regla. Una vez tenemos las estimaciones ya podemos decidir donde mandar el vídeo. Hay una regla por cada posible decisión:

```
(defrule execution-server
    (declare (salience ?*slack-server*))
    (estimated-transfer
        (from-to ?a server)
        (value ?v-txon)
        (CF ?CF-txon))
    (estimated-waiting
        (at server)
        (value ?v-wait)
        (CF ?CF-wait))
    (estimated-exec
        (at server)
        (value ?v-tex)
        (CF ?CF-tex))
    (deadline ?dl)
    (test (neq ?a server))
    (pointer chunk-chan ?pointer ?id)
    ?ef <- (fire executionr&:(bind ?*slack-server*
        (integer (* 100 (min ?CF-txon ?CF-wait ?CF-tex)))))
    (test (<= (+ ?v-txon ?v-wait ?v-tex) ?dl))
    (test (> (min ?CF-txon ?CF-wait ?CF-tex) 0.2))
=>
    (retract ?ef)
    (process-at-server ?pointer ?id))
```

Esta regla decide si procesa en el servidor. La otras dos reglas son prácticamente iguales. En la LHF vemos que además de las estimaciones que acabamos de explicar

hay otros requisitos. El mas evidente es el cumplimiento del deadline `(test (<= (+ ?v-txon ?v-wait ?v-tex) ?dl))`. También la regla ha de tener un mínimo de fidelidad, el menor valor de los CFs ha de ser superior a 0.2 para considerar las medidas fiables: `(test (>(min ?CF-txon ?CF-wait ?CF-tex) 0.2))`. Otro requisito que llama la atención es el hecho: `(fire executionr)`, lleva agregados los caracteres "&:", sirven para ejecutar una acción en la LHS. En este caso estamos actualizando el valor de la variable **?*slack-server***. Si nos fijamos en la primera línea, el valor de **salience** (prioridad) de la regla va ligado a esta variable. Si hay varias reglas de decisión de este tipo que cumplan los requisitos, la que tenga las métricas más fiables se activará antes que el resto.

Por último, cuando la regla se activa, a parte de eliminar el hecho `(fire executionr)` para que el resto de reglas de decisión no se activen, llama a la función "process-at-server". Esta no es como las funciones que hemos visto antes, es una función de call-back, la hemos definido en Go y cargado en el entorno de CLIPS. De forma que podemos ejecutar funciones programadas en Go desde CLIPS (figura 4.1).

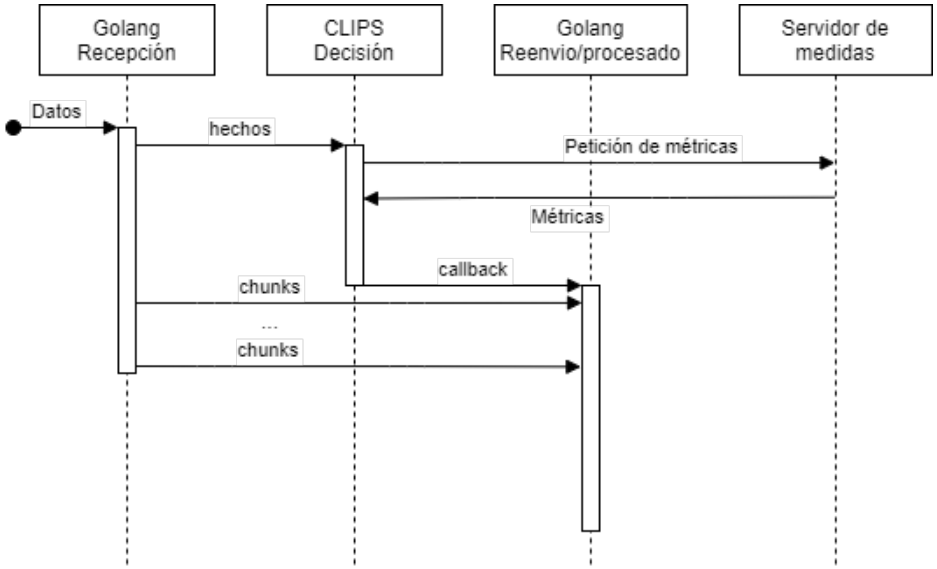

Figura 4.1: Ejecución de la recepción de un vídeo.

Dependiendo de la regla que se haya activado el hilo de ejecución reenviará el vídeo a otro nodo o reconstruirá el archivo de vídeo original y lanzará la herramienta de análisis de vídeo.

Tal y como están, las reglas de decisión tienen un problema, pues en caso de que en ninguna regla se cumplan todos los requerimientos el vídeo a procesar se pierde en el sistema y no se procesa. Para que esto no pase se han añadido otro grupo de reglas,

que infieren cuáles son las medidas más confiables, dentro de que ninguna cumple el mínimo de 0.2 en el CF o no cumple el deadline. Para ello se suman los CF de las tres medidas, y decide procesar en el nodo que tenga la suma más alta, en el que sea más confiable. Para modificar los hechos de las métricas a la suma de CFs se encarga la regla:

```
(defrule total_CF
(estimated-transfer
    (from-to ? ?local)
    (value ?v-txon)
    (CF ?CF-txon))
(estimated-waiting
    (at ?local)
    (value ?v-wait)
    (CF ?CF-wait))
(estimated-exec
    (at ?local)
    (value ?v-tex)
    (CF ?CF-tex))
=>
    (assert (CF-at ?local (+ ?CF-txon ?CF-wait ?CF-tex))))
```

Vemos que por cada trío de métricas añade un hecho sumando sus CFs. Luego hemos de saber cual de todos estos es el más alto, para eso esta la regla:

```
(defrule min_value
    (declare (salience 19))
    ?ef <-(CF-at ?a ?t)
    (CF-at ?b ?t2)
    (test (neq ?a ?b))
    (test (>= ?t2 ?t))
=>
    (retract ?ef))
```

Compara pares de hechos **CF-at** y elimina el menor hasta que solo queda uno, entonces se activa la siguiente regla:

```
(defrule no_execution
    (declare (salience 18))
    ?ef <- (fire executionr)
```

```
    (CF-at ?place ?)
    (pointer chunk-chan ?pointer ?id)
=>
    (retract ?ef)
    (no-action ?pointer ?id ?place))
```

Al activarse ejecuta una función de call-back igual que en el caso anterior solo que añade el lugar donde decide que va a procesar. Esto es porque en el caso anterior teníamos tres reglas, una por cada decisión mientras que aquí solo aquí solo tenemos una. Así quedarían cubiertas todas las posibilidades, sin dejar a ningún vídeo sin decisión.

Capítulo 5

Análisis y Resultados

En este capítulo veremos un estudio del coste en tiempo de procesar los distintos tipos de vídeo y también estudiaremos los efectos de la interferencia que se causan una instancia de la herramienta a otra. Luego, se muestran los resultados obtenidos del funcionamiento del sistema durante largos periodos de tiempo, comparando la reglas estáticas con el sistema experto.

5.1. Análisis

Para poder estimar el tiempo de procesado es necesario hacer una serie de medidas y encontrar un modelo que se ajuste bien a ellas. Como hemos nombrado varias veces en capítulos anteriores, podríamos combinar varias decisiones de en qué máquina procesar, realizando un procesado parcial de un vídeo en varias máquinas. Por eso las medidas que hemos tomado para este análisis consisten en: partir un vídeo en trozos de distintas duraciones y medir el tiempo que cuesta procesarlos. Esto se ha realizado para todos los vídeos en los tres tipos de nodos, obteniendo gráficas como las de la figura 5.1. En ellas vemos el conjunto de medidas y, el modelo de regresión lineal que se ajusta perfectamente, afirmando que el tiempo de procesado es directamente proporcional al número de fotogramas analizados, sin importar el contenido del vídeo.

La pendiente depende de otros factores como: la resolución; el formato de vídeo; la velocidad de bits. Sin embargo no hemos estudiado como afectan estos valores al tiempo de procesado. Ya que el contenido de los vídeos no influye en el tiempo de procesado y el sistema procesa los mismos formatos de vídeo todo el rato, no es necesario conocer su efecto. En caso de haber querido que el sistema procese cualquier vídeo, sí hubiera sido necesario estudiar el efecto de estos parámetros para poder estimar el tiempo de procesado y de espera en cola correctamente.

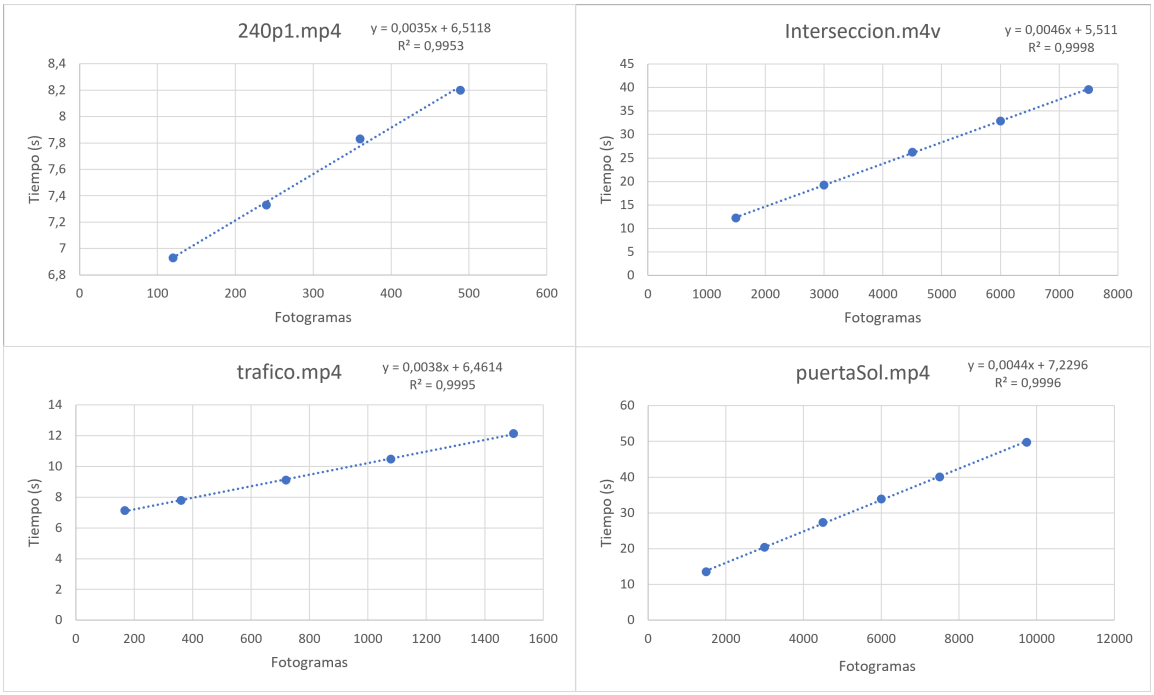

Figura 5.1: Análisis de los vídeos por fotogramas procesados en el servidor

La ordenada en el origen representa el tiempo de despliegue de la herramienta, varía según el tipo de nodo y es para tener en cuenta. El caso mas extremo sería procesar un solo fotograma, este tiempo de despliegue hace que no merezca la pena fraccionar los datos a tal extremo.

Análogo a este estudio hemos hecho otro modificando la frecuencia de muestreo de la herramienta de vídeo, es decir, cambiando el valor del módulo que indica que fotograma se procesa y cual no. Realmente estamos haciendo lo mismo que en el estudio anterior, reducir el número de fotogramas analizados con el objetivo de reducir el tiempo de procesado, por lo que el modelo que podemos extraer de analizar como cambia el tiempo de procesado al modificar la frecuencia de muestreo será el mismo que el del estudio anterior. Porque como hemos concluido el contenido del vídeo no es relevante, solo el número de fotogramas analizados. La reducción del tiempo de procesado sería especialmente interesante en los nodos de bajos recursos, pero claro reducir la frecuencia de muestreo implica que se puedan dejar de detectar objetos, implica un error y por lo tanto sería un parámetro más que medir para el QoS.

Por otra parte, también hemos estudiado como afecta la interferencia de un proceso a otro. Así hemos establecido los slots que soporta cada nodo sin llegar a degradar demasiado las prestaciones. Para ello hemos lanzado varias instancias de la herramienta y medido sus tiempos mientras estuviesen compartiendo recursos con

el resto de instancias. También hemos comprobado que el tipo de vídeo que se esté procesando en cada slot no cambia la interferencia causada. La figura 5.2 muestra como interfiere el reparto de recursos entre las instancias de la herramienta de procesado de vídeo en el servidor. El servidor está configurado para soportar cuatro vídeos al mismo tiempo.

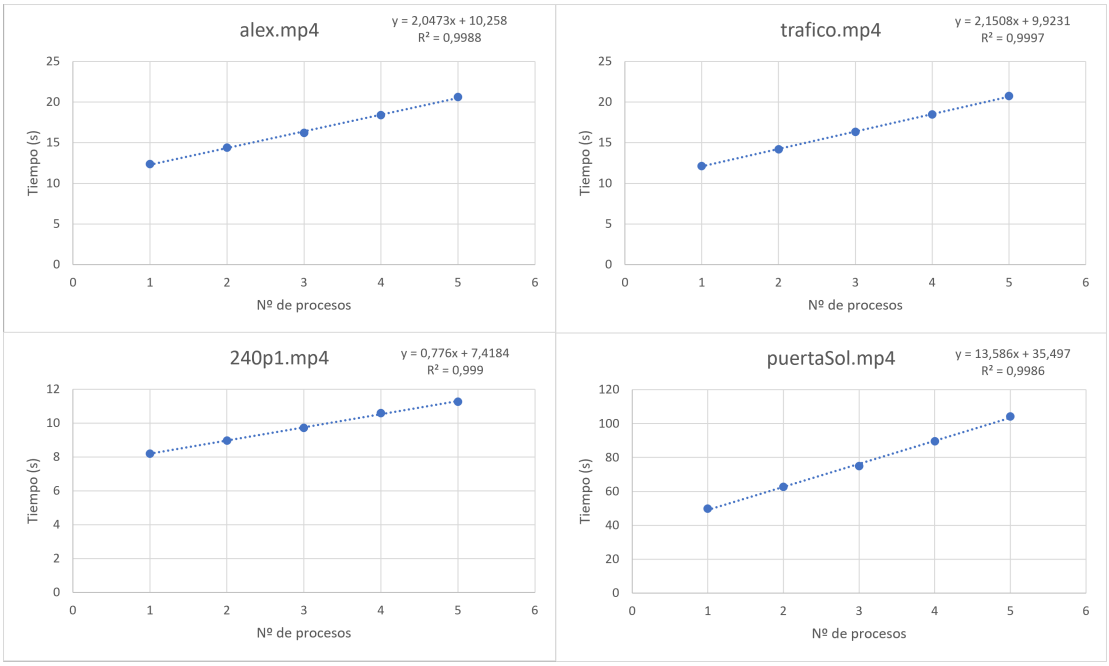

Figura 5.2: Interferencia causada en el servidor para cada tipo de vídeo

La interferencia no se queda solo en el ámbito de la propia aplicación, también afectan los procesos que se estén ejecutando en la máquina real. Estos son más difíciles de controlar, porque, en el caso de las Raspberrys o el nodo intermedio se pueden dedicar en exclusiva a esta tarea. Pero el servidor concentra muchos recursos y sería muy ineficiente que en algún momento dado no haya vídeos que procesar y se desaproveche ese nodo. Por eso sería normal que en el servidor hubiese más de un proceso externo a nuestro sistema causando una interferencia que no podemos predecir.

5.2. Resultados

En este apartado mostramos los resultados obtenidos de analizar ambos tipos de decisiones. Cuando un nodo termina de procesar un vídeo se lo notifica al nodo de medidas, donde se centralizan los resultados de todos los procesados en un archivo de texto. Allí guardamos la fecha de finalización del procesado y la diferencia de esta fecha con el deadline, dando como resultado un número en segundos. Si es positivo

indica que el vídeo se ha procesado antes del deadline y si no es que se ha procesado tarde, violando el QoS. También guardamos la ruta que ha seguido para saber cuantos saltos ha dado el vídeo. Mínimo da un salto, de una Raspberry al nodo intermedio o al servidor. Dos saltos en caso de que el nodo intermedio valore el vídeo por segunda vez y decida enviarlo al servidor.

Para tomar los datos hemos preparado dos escenarios que pongan a prueba los sistemas de decisión y los hemos dejado en ejecución durante más de **24 horas** cada uno. En el primer escenario no hay limitación de ningún tipo, todas las máquinas y enlaces funcionan sin problemas. Las máquinas decidirán en condiciones idóneas donde procesar. En el segundo escenario sin embargo, la función de medida del ancho de banda no funcionará correctamente. El servidor de medidas solo mandará actualizar el valor del ancho de banda tres veces desde que se lance el sistema, el resto del tiempo no se actualizará el valor de ancho de banda. Tras estas tres medidas reducimos el ancho de banda en el enlace de las Raspberrys al servidor usando el tc, visto en la sección 2.6. El tc limita el ancho de banda en el enlace ascendente, por lo que limitaremos el ancho de banda en las cuatro Raspberrys. Las reglas que aplica el tc funcionan por interfaz de red, por lo tanto, para evitar limitar el enlace de las Raspberrys al nodo intermedio, hemos creado otra red overlay desviando este tráfico por otra interfaz. Aunque pueda ser obvio, veamos primero los resultados y que conclusiones podemos sacar de ellos.

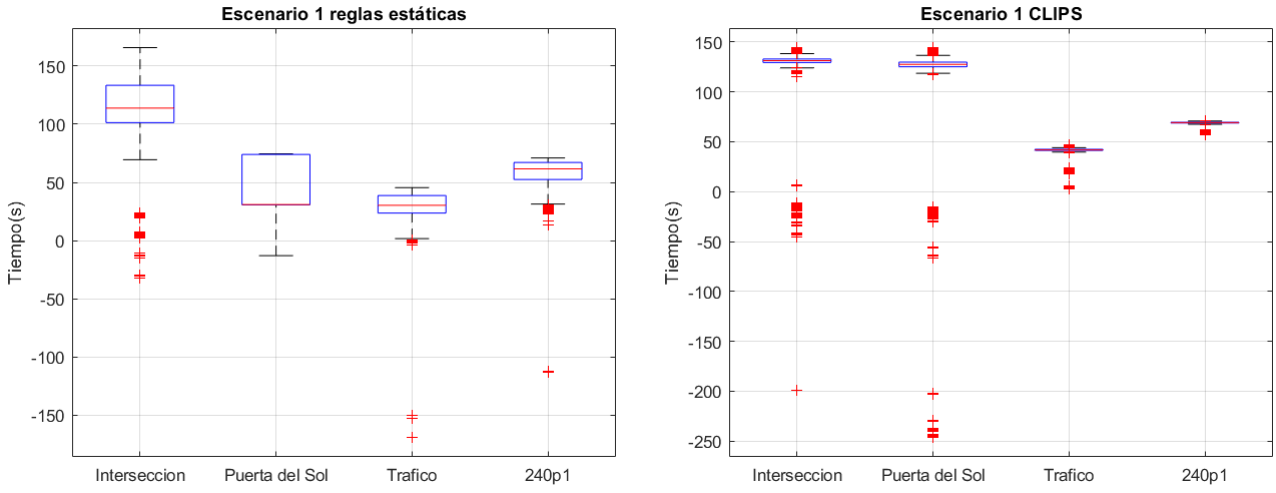

Figura 5.3: Resultados del primer escenario.

Tenemos cuatro conjuntos de datos distintos, uno por cada tipo de vídeo. En el eje de ordenadas está la diferencia de entre los instantes de tiempo actual y el deadline, en segundos. En este primer escenario (figura 5.3) tanto las reglas estáticas como CLIPS hacen bien su trabajo de decisión, lo que verifica que ambas estrategias son

válidas cuando el sistema funciona sin imprevistos. En el caso de las reglas estáticas la mayoría de los datos son positivos o cercanos a cero, lo que nos indica que se esta cumpliendo el QoS. Y en el caso de CLIPS los datos están notablemente más concentrados y elevados, parece que consigue mejor su objetivo de procesar a tiempo los datos, pero también hay bastantes más datos atípicos muy negativos, esto puede deberse a diversos motivos: el primero es que la decisión de los nodos no es siempre la misma, depende de la antigüedad de las medidas y si hay alguna medida muy desactualizada puede variar; el segundo motivo es que las reglas estáticas permiten un bucle de tráfico entre el servidor y el nodo intermedio, mientras que las reglas estáticas no contemplan desviar carga del servidor. Dando pie a más error con situaciones en las que se decide procesar en un nodo teniendo información desactualizada de los vídeos en cola de ese nodo. En cualquier caso ambos consiguen su objetivo, veamos el segundo escenario.

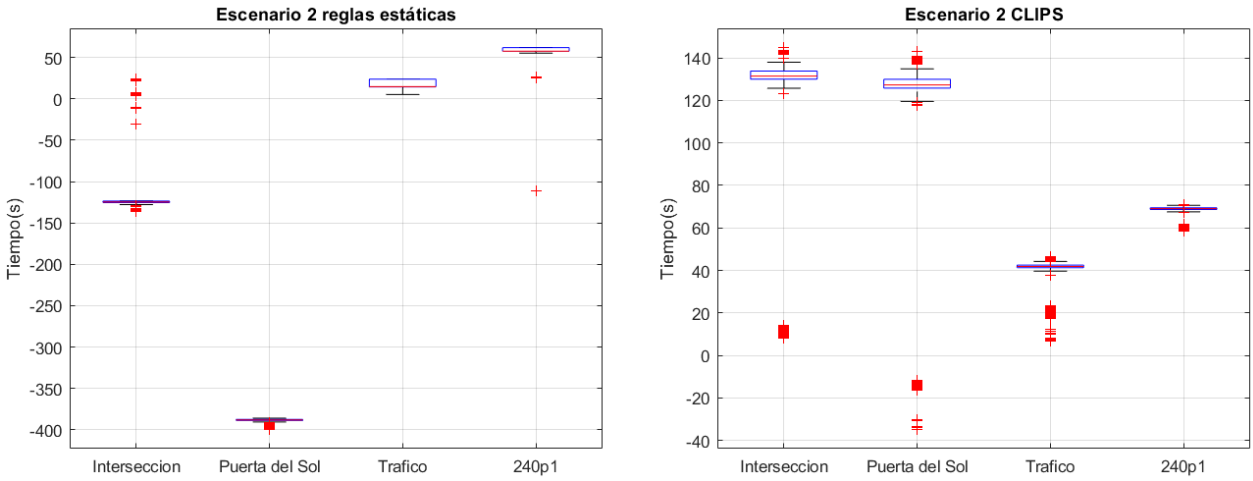

Figura 5.4: Resultados del segundo escenario.

En este segundo escenario con el ancho de banda limitado entre las Raspberries y el servidor apreciamos un cambio. Las reglas estáticas comienzan a fallar, esto es porque decide enviar al servidor directamente, calculando erróneamente el tiempo de transmisión. Si nos fijamos en cada tipo de vídeo por separado, veremos que los dos vídeos más pesados son los que llegan tarde, mientras que los otros dos vídeos se mantienen cumpliendo el deadline. El error que se comete con los vídeos más ligeros es menor. En cambio, CLIPS aunque realiza las mismas estimaciones del tiempo de transmisión, sigue procesando los vídeos sin violar el QoS. Esto es porque detecta que la última medida del ancho de banda no cumple con el CF mínimo (mecanismo desarrollado en la sección 4.2) y por lo tanto no se puede fiar de ella, dando por falso ese hecho.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Podemos decir que hemos cumplido los dos objetivos de este proyecto. El primero diseñar e implementar un pequeño sistema distribuido, el cual hemos realizado y llevado a cabo, ya que el sistema funciona y hemos podido obtener datos a partir de él. El segundo era comparar dos formas diferentes de toma de decisiones, mostrando como los sistemas expertos pueden ayudar a solucionar los problemas que plantea una arquitectura descentralizada en un sistema de computación distribuido.

En cuanto a esto podemos extraer algunas conclusiones de los últimos resultados: está claro que el problema de las reglas estáticas es que no tienen en cuenta el posible fallo al tomar las medidas. Lo podríamos corregir sin demasiada complejidad porque el sistema en sí es muy sencillo, y añadir el mismo sistema de coeficientes de fidelidad en las reglas estáticas sería viable. Pero esto mismo lo podemos extrapolar a un sistema mas grande y complejo, donde programar las reglas con "IF THEN" se convierte en una tarea muy compleja y tediosa por la gran cantidad de opciones y parámetros a tener en cuenta, en lugar de las pocas lineas de nuestro sistema. Comparado, CLIPS está diseñado para el desarrollo de sistemas expertos, y está preparado para implementar este tipo de mecanismos sea cual sea la magnitud del sistema. Cuando la decisión se toma desde un paradigma declarativo, el flujo de la ejecución ya está preprogramado, si se enfrenta a una situación que no se ha tenido en cuenta para su diseño, fallará. En un sistema experto hay herramientas para evitarlo, partiendo de la base del paradigma declarativo la ejecución de la decisión no está programada para seguir un flujo determinado, lo que le permite adaptarse a situaciones excepcionales. Además de contar con otras herramientas como el sistema de mantenimiento de la verdad o el aprendizaje automático que no hemos llegado a implementar.

6.2. Trabajo futuro

El proyecto en sí marca el inicio para estudiar el efecto de los sistemas expertos resolviendo problemas mas concretos del paradigma distribuido. Las posibilidades son muchas, a continuación vamos a enumerar algunas:

- **Aprendizaje del sistema experto:** En este proyecto no hemos llegar a utilizar el motor de inferencia de CLIPS. Dotando al sistema de realimentación sobre las decisiones tomadas sería capaz no solo de perfeccionar su funcionamiento conforme aprende, sino que podría detectar comportamientos extraños que se escapen a las reglas establecidas y actuar en consecuencia en un momento concreto. En el caso de un sistema distribuido más complejo con muchos más parámetros que influyan en la decisión sería clave.

- **Medidas:** Cómo se comparten las métricas en un sistema distribuido es una de sus características principales. En este proyecto hemos implementado una solución que centraliza las mismas. Investigar e implementar una forma óptima de compartir las medidas es una tarea pendiente para desplegarlo a mayor escala que la manejada en este proyecto.

- **Decisiones:** Incrementar el abanico de decisiones aumentaría la complejidad del sistema y la flexibilidad en la planificación. Los nodos podrían concatenar distintas decisiones, aumentando la eficiencia.

Bibliografía

[1] . «How to use overlay networks».
https://docs.docker.com/network/network-tutorial-overlay/.

[2] . «Matlab - boxplot».
https://es.mathworks.com/help/stats/boxplot.html.

[3] . «Networking with overlay networks».
https://docs.docker.com/network/overlay/.

[4] (April 25, 2017). «Deep dive into Dcoker Overlay networks».
https://blog.revolve.team/2017/04/25/deep-dive-into-docker-overlay-networks-par

[5] (Aril 2, 2011). «Yaz repository by Sommers J.».
https://github.com/jsommers/yaz.

[6] (December 13, 2001). *tc-tbf(8) - Linux manual page*.
https://man7.org/linux/man-pages/man8/tc-tbf.8.html.

[7] (December 16, 2001). *tc(8) - Linux manual page*.
https://man7.org/linux/man-pages/man8/tc.8.html.

[8] (February 14, 2018). «Herramienta de procesado de vídeo por Kasukurthi N.».
https://github.com/Nikhil-Kasukurthi/object-detection-videos.

[9] (January 13, 2020). «Sistema experto — Wikipedia».
https://es.wikipedia.org/wiki/Sistema_experto.

[10] (January 13, 2020). «Usos de los sistemas expertos».
https://www.dail.es/sistemas-expertos-usos-comerciales/.

[11] (July 1, 2015). *CLIPS User's Guide Version 6.30*. Joseph C., Giarratano Ph.D..

[12] (November 25, 2021). «Repositorio del proyecto».
https://github.com/RLogrono/Decentralised-decision-making-in-distributed-system

[13] (November 29, 2017). «How to Use the Linux Traffic Control». https://netbeez.net/blog/how-to-use-the-linux-traffic-control/.

[14] (September , 2010). *CLIPS Code Snippets Versión 0.8*. Béjar J.. https://drive.google.com/file/d/10_VRu6WJx0t-n_bXlAAoZK0GOxRL7Q51/view.

[15] (September 16, 2020). «Integración bidireccional CLIPS-Go por Keysight Technologies». https://github.com/Keysight/clipsgo.

[16] J., Pasquale (September 1988). «Using Expert Systems to Manage Distributed Computer Systems». *IEEE network*, pp. 22–28.

[17] Mahalingam M., Duda K. Agarwal P. Kreeger L. Sridhar T. Bursell M. Wright C., Dutt D. (August 2014). «RFC 7348 - Virtual eXtensible Local Area Network (VXLAN):A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks». *Informe técnico*, Independent Submission. https://datatracker.ietf.org/doc/rfc7348/.

Lista de Figuras