

Trabajo Fin de Grado

Sistema de medida de la calidad del audio para
entornos masivos de telefonía IP

Audio quality measurement system for massive IP
telephony environments

Autor

Jorge Burgos Rodríguez

Director

Joaquín Lázaro Romero

Ponente

Julián Fernández Navajas

Escuela de Ingeniería y Arquitectura

2023

Sistema de medida de la calidad del audio para entornos masivos de telefonía IP

Resumen

En los sistemas telefónicos basados en telefonía IP (Donde el intercambio de información entre los extremos de una comunicación telefónica se realiza a través de internet), en ocasiones (principalmente en llamadas internacionales) las conexiones telefónicas dependen de varias operadoras diferentes. Éstas operadoras delegan en proveedores para hacer llegar la llamada hasta el destinatario final. Durante este proceso, la calidad de la voz percibida en una llamada puede verse afectada por motivos como el retardo o las pérdidas de información, lo que supone una disminución en la satisfacción de los usuarios.

Durante este Trabajo Fin de Grado se ha desarrollado un sistema que permite medir la calidad del audio de llamadas telefónicas en un entorno de telefonía IP preparado para trabajar con una muy elevada cantidad de tráfico. Con él se pretende complementar la plataforma ofrecida por la empresa System One Noc & Development Solutions, S.A. (SONOC), añadiendo una medida de la calidad perceptual de las llamadas telefónicas, aportando a sus clientes más posibilidades a la hora de evaluar los proveedores a los que “traspasar” las llamadas.

Tras el análisis previo de las herramientas y protocolos que se van a utilizar, se ha desarrollado un sistema capaz de separar los flujos de audio frente al resto de flujos que atraviesan los servidores de la empresa. Para ello se ha utilizado una tarjeta de red capaz de implementar diferentes filtros programables. Posteriormente se han calculado las estadísticas indicativas de la calidad del audio a partir de los flujos filtrados, mediante el uso de un software específico. Finalmente se ha desarrollado un programa que permite añadir estas estadísticas a los registros de llamadas generados anteriormente por la empresa, que únicamente contaban con datos relativos a la señalización de la llamada, guardando los datos en una plataforma de Big Data a los cuales puede acceder cualquier aplicación que necesite hacer uso de ellos.

Finalmente, se recogen las conclusiones obtenidas mediante el desarrollo del Trabajo Fin de Grado, así como las posibles líneas futuras del proyecto.

Audio quality measurement system for massive IP telephony environments

Abstract

In telephone systems based on IP telephony (where the exchange of information between the ends of a telephone communication is done over the internet), sometimes (mainly in international calls) telephone connections depend on several different operators. These operators delegate providers to get the call to the final recipient. During this process, the perceived voice quality of a call can be affected by reasons such as delay or loss of information, which leads to a decrease in user satisfaction.

During this Final Degree Project, a system has been developed to measure the audio quality of telephone calls in an IP telephony environment prepared to work with an extremely high amount of traffic. The aim is to complement the platform offered by the company System One Noc & Development Solutions, S.A. (SONOC), adding a measurement of the perceptual quality of telephone calls, providing its customers with more possibilities when evaluating the providers to which to "transfer" the calls.

After a preliminary analysis of the tools and protocols to be used, a system has been developed capable of separating audio flows from the rest of the flows that pass through the company's servers. For this purpose, a network card capable of implementing different programmable filters was used. Subsequently, statistics indicating the quality of the audio from the filtered flows have been calculated using specific software. Finally, a program has been developed that allows these statistics to be added to the call records previously generated by the company, which only had data relating to the signaling of the call, saving the data in a Big Data platform that can be accessed by any application that needs to make use of them.

Finally, the conclusions obtained through the development of the Final Degree Project are included, as well as the potential future lines of the project.

ÍNDICE

Capítulo 1: Introducción	1
1.1. Introducción	1
1.2. Objetivos y motivación	2
Capítulo 2: Fundamentos teóricos del sistema	4
2.1. Herramientas	4
2.2. Protocolos.....	11
2.3. Parámetros relacionados con la calidad del audio.....	13
Capítulo 3: Escenario previo y planteamiento del problema.....	15
Capítulo 4: Descripción del sistema	19
4.1. Planteamiento general	19
4.2. Filtrado de paquetes RTP	20
4.3. Cálculo de métricas a partir de flujos RTP.....	23
4.4. Almacenamiento de las métricas calculadas. Kafka	25
4.4.1. Configuración de un topic.	25
4.4.2. nProbe como productor Kafka.....	27
4.5. Procesamiento de datos para el enriquecimiento de CDR. Apache Flink	31
Capítulo 5: Pruebas de funcionamiento.....	39
Capítulo 6: Conclusión y líneas futuras.....	45
6.1. Conclusión	45
6.2. Líneas futuras.....	46
Anexos.....	47
Anexo 1: Guía de instalación y configuración del software Napatech.....	47
Anexo 2: Implementación del filtro mediante programación en NTPL.....	49
Anexo 3: Comunicación entre un productor y Kafka.....	50
Anexo 4: Comunicación entre un consumidor y Kafka	52
Anexo 5: Fichero de configuración de nProbe	53
Anexo 6: Schema de los CDR	55
Anexo 7: Generar una clase de java a partir de un schema	57
Anexo 8: Código que implementa la lógica de procesamiento para el enriquecimiento de los CDR	58
Anexo 9: Comandos básicos para Apache Flink	62
Anexo 10: Resultado programa de procesamiento de Flink.....	63
Bibliografía.....	66

Capítulo 1: Introducción

1.1. Introducción

Las comunicaciones telefónicas representan en la actualidad una de las formas de comunicación entre personas más utilizada en el día a día, a pesar del avance en los últimos años de otros tipos de comunicación como las redes sociales o la mensajería instantánea. Su popularidad reside en su inmediatez, interacción dinámica y capacidad de establecer un contacto humano directo, lo que hace que las llamadas telefónicas sigan siendo una forma confiable y efectiva de comunicación interpersonal y comercial.

Actualmente la telefonía IP (ToIP, basada en la transmisión de Voz IP, VoIP) es la opción más aceptada entre las empresas de telecomunicaciones frente a la telefonía tradicional. Entre sus ventajas, destaca una menor necesidad de infraestructura, dado que se utiliza la infraestructura de internet para transmitir las llamadas, no siendo necesaria una red de líneas telefónicas físicas. Esto supone una reducción en su coste, además de suponer una solución mucho más escalable y flexible.

Como principal inconveniente, la telefonía IP carece de una calidad de voz tan estable como la telefonía tradicional, ya que depende de factores como la congestión de la red o la calidad de la conexión a internet. Esto puede afectar negativamente a la calidad de experiencia del usuario final durante el transcurso de una llamada telefónica.

En ocasiones, para que una llamada pueda ser recibida por el destinatario es necesario que los datos deban pasar por diversos proveedores de telecomunicaciones en su ruta al receptor, cuyo camino depende de los acuerdos existente entre ellos. La elección del proveedor al que redirigir una llamada también puede disminuir la calidad de la voz en la comunicación final entre los usuarios. Es por ello por lo que muchas operadoras cuentan con cálculos que permiten estimar la calidad percibida por los usuarios para cada una de las comunicaciones establecidas.

En definitiva, la calidad de una llamada sobre VoIP puede ser muy variable dependiendo del estado de la red y de los proveedores intermediarios en la telecomunicación y una mala comunicación supone una reducción importante de la calidad de experiencia de los usuarios. Por ello es conveniente que las empresas del sector de las telecomunicaciones cuenten con mecanismos que permitan evaluar la calidad de aquellas comunicaciones a las que dan servicio.

1.2. Objetivos y motivación

Tomando como base el problema de la calidad de las llamadas VoIP, este Trabajo de Fin de Grado surge para satisfacer las necesidades del análisis de la calidad por parte de la empresa SONOC (SYSTEM ONE NOC & DEVELOPMENT SOLUTIONS SA).

La actividad principal de SONOC es el desarrollo de una plataforma que es ofrecida a las operadoras de telefonía para poder controlar el tráfico internacional, con el objetivo de proveerlas de las herramientas necesarias para mejorar sus operaciones de voz y mensajería, proporcionando una serie de estadísticas que caracterizan las llamadas.

Como se ha mencionado en la introducción, en ocasiones, para poder establecer la comunicación de una llamada, las operadoras necesitan enrutar el tráfico a través de otros proveedores, pudiendo suponer una reducción de la calidad de la llamada en función de la elección de proveedores intermedios. Esta decisión puede ser asumida en ciertas circunstancias que deben ser adecuadamente evaluadas.

El sistema desarrollado en este Trabajo Fin de Grado se incorporará a la plataforma de SONOC a petición de algunos de sus clientes, ya que éste puede ser de ayuda a la hora de elegir por qué proveedor enrutar la llamada. Así pues, la llamada se redirigirá a un proveedor u otro según el precio que éstos cobren, las estadísticas que ofrezca y la cantidad de dinero pagada por el cliente. Actualmente la plataforma de SONOC proporciona estadísticas de la tasa de llamadas contestadas y de la duración media de éstas. A las medidas anteriores, se pretende añadir mediante este trabajo la estadística de la calidad del audio percibida por los usuarios en la llamada, lo cual supone una mejor caracterización de la actividad telefónica de los clientes.

Además, tener estadísticas sobre la calidad de la comunicación va a suponer para los clientes una herramienta con la cual poder detectar problemas, pudiendo identificar aquellas conexiones con baja calidad, y darle una solución analizando aquellos parámetros que provoquen las bajas prestaciones.

Por ello, el objetivo de este trabajo de fin de grado es implementar un sistema que calcule la calidad perceptual de las comunicaciones telefónicas en entornos masivos con el fin de mejorar las prestaciones de la plataforma de SONOC y aumentar la satisfacción de sus clientes.

1.3. Organización de la memoria

La memoria se compone de los siguientes capítulos:

- Capítulo 1: Se realiza una introducción y se muestran los objetivos que motivan la realización del proyecto.
- Capítulo 2: Se explican las herramientas utilizadas, así como los fundamentos teóricos necesarios para la comprensión del sistema implementado.
- Capítulo 3: Descripción del escenario de partida del proyecto y planteamiento general del problema.
- Capítulo 4: Descripción del sistema donde se muestra detalladamente el funcionamiento y la implementación de este.
- Capítulo 5: Prueba del funcionamiento final del sistema y análisis de trazas que facilitan su comprensión.
- Capítulo 6: Conclusiones del proyecto y líneas futuras a seguir.

Además, al final de la memoria se incluyen a modo de anexos apartados que aportan información complementaria sobre el sistema desarrollado

Capítulo 2: Fundamentos teóricos del sistema

2.1. Herramientas

A continuación, se muestran las herramientas, aplicaciones y lenguajes de programación que se han utilizado en el desarrollo del Trabajo de Fin de Grado.

Java

Java [1] es un lenguaje de programación multiplataforma orientado a objetos y de alto nivel. Para su ejecución necesita ser compilado, es decir generar un binario que se ejecuta en la JVM (Java Virtual Machine), máquina virtual que proporciona un entorno de ejecución en la plataforma Java.

NT200A02 SmartNIC

[2] Es una tarjeta de red fabricada por Napatech la cual es diseñada para aplicaciones de monitoreo y análisis de red. Permite la captura masiva de paquetes de red que llegan a la tarjeta y el acceso a su contenido para su análisis y monitoreo. Soporta tráfico medio de entrada de 100Gbps y ráfagas de hasta 200Gbps con pérdidas de 0 paquetes. También es capaz de eliminar paquetes duplicados o filtrar paquetes para reducir la cantidad de datos.

NTPL

NTPL (NapaTech Programming Language) [3] es un es una sencilla pero potente interfaz de configuración de adaptadores que permite el filtrado y *hashing*, es decir, la asignación de una clave que identifica los paquetes en función de alguna de sus características, como por ejemplo su dirección IP fuente. También permite la división de buffers multi-CPU y anotación de paquetes en las tarjetas Napatech, tarjetas que permiten la recepción y transmisión de tráfico. Se define como un lenguaje de *scripting*, que a diferencia de otros lenguajes de programación como java no necesita ser compilado para su ejecución.

nProbe

[4] Aplicación software desarrollada por Ntop para el monitoreo y análisis de tráfico en tiempo real. Para ello captura paquetes de red y los analiza, siendo capaz de mostrar datos y estadísticas para múltiples protocolos de red, como IPv4, TCP, UDP, HTTP, SIP o RTP. Por ejemplo, es capaz de mostrar el número de paquetes capturados de un flujo de audio, el número de paquetes perdidos y la calidad media percibida en la comunicación. Además, permite exportar datos a herramientas de *big data* como Kafka.

Apache Kafka

Kafka [5] es una plataforma distribuida utilizada para la transmisión de mensajes en tiempo real, altamente escalable cuya arquitectura sigue un modelo de publicación-suscripción. Sus componentes principales son los productores, consumidores y temas (más conocidos por su nombre en inglés *topic*). De esta manera, su funcionamiento al más básico nivel consiste en productores que publican mensajes o *record* en los *topic*, y consumidores que se subscriben a esos *topic* y reciben los mensajes publicados en ellos. Esto se muestra en la “Figura 1: Diagrama conceptual del funcionamiento básico de Kafka”.

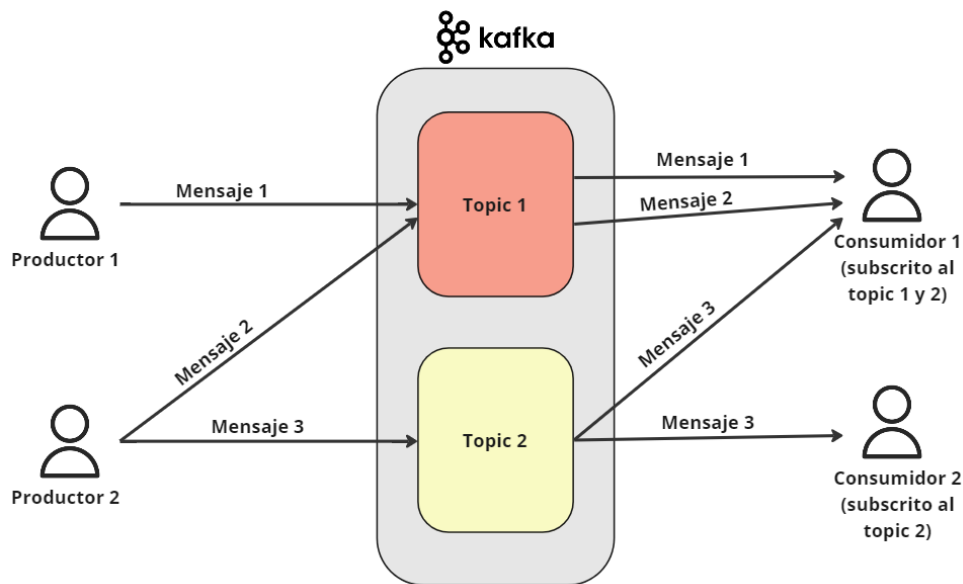


Figura 1: Diagrama conceptual del funcionamiento básico de Kafka

Otro concepto relevante acerca de Kafka son los *broker*, que son cada uno de los servidores que forman su topología. El conjunto de todos los *broker* se denomina clúster. Cada uno de los servidores que lo componen reciben un identificador para distinguirlo de los demás. Además, dentro de un clúster se define un *broker* líder.

A continuación, se muestran con más detalle cada uno de los componentes principales de Kafka:

- **Topic:**

Un *topic* se define como la unidad fundamental de almacenamiento en Kafka. Representa un *stream* de datos particular, y su funcionamiento se asemeja al de una tabla de una base de datos tradicional.

Para distribuir la carga que soporta el *topic* entre más de un nodo disponible en el clúster se definen un número determinado de particiones en él. Una partición es cada uno de los segmentos en los que se divide un *topic*, permitiendo que los datos dirigidos al *topic* se almacenen en distintos *broker*.

El número de particiones que le asignamos al *topic* en el momento de crearlo, se reparte de manera equitativa entre los *broker* del clúster. Es decir, si mi clúster está formado por

3 servidores y creo un *topic* con una única partición, ésta será asignada a tan sólo uno de los *broker*. En cambio, si lo configuro con 6 particiones, a cada *broker* se le asignan 2 de ellas. La motivación que lleva a particionar un *topic* es permitir el paralelismo y que el sistema sea escalable horizontalmente. El responsable del reparto de particiones entre los nodos es el *broker* líder del clúster. Cuando un *broker* se añade o se quita del clúster se lleva a cabo una reasignación de las particiones para garantizar el reparto equitativo en todos los nodos.

El *broker* que controla una partición se dice que es el líder de esta. Cada partición tan sólo puede estar liderada por uno de los *broker*.

Dentro de cada partición, cada vez que un productor escribe un mensaje en ella, a éste se le asigna un identificador llamado *offset*, que crece en una unidad cada vez que se escribe en la partición. Gracias a este *offset* se garantiza que, dentro de una partición, todos los mensajes están ordenados. Un mensaje con un determinado *offset* siempre será anterior que los que tengan un *offset* superior. Sin embargo, el concepto del *offset* sólo tiene sentido a nivel de partición y no de *topic*, es decir, entre mensajes de distintas particiones, un *offset* mayor no significa que el mensaje se haya recibido más tarde.

Otro parámetro relevante a la hora de configurar el *topic* es el factor de replicación. Este valor representa el número de réplicas que existen en el clúster del contenido de cada partición. Es decir, llevado a un ejemplo, si se da un caso donde tenemos un clúster de 3 *broker*, con 3 particiones, cada *broker* es líder de una de las particiones y todas las operaciones que deban hacerse con los mensajes contenidas en ellas se hacen conectándose con el líder de cada una. Si a eso le añado un factor de replicación 3, los mensajes de cada partición estarán replicados en los otros dos *broker*. A pesar de tener el contenido de cada partición replicado en todos los nodos, cuando un cliente escribe o lee mensajes en una partición, éste siempre se conectará y enviará los datos al *broker* líder de esta. Posteriormente es el *broker* líder el encargado de mandar esos datos a las réplicas mediante mensajes ISR (In-Sync Réplicas) por medio de una comunicación interna entre ellos. Este ejemplo se detalla en la “Figura 2: Diagrama conceptual de ejemplo de replicación en Kafka”.

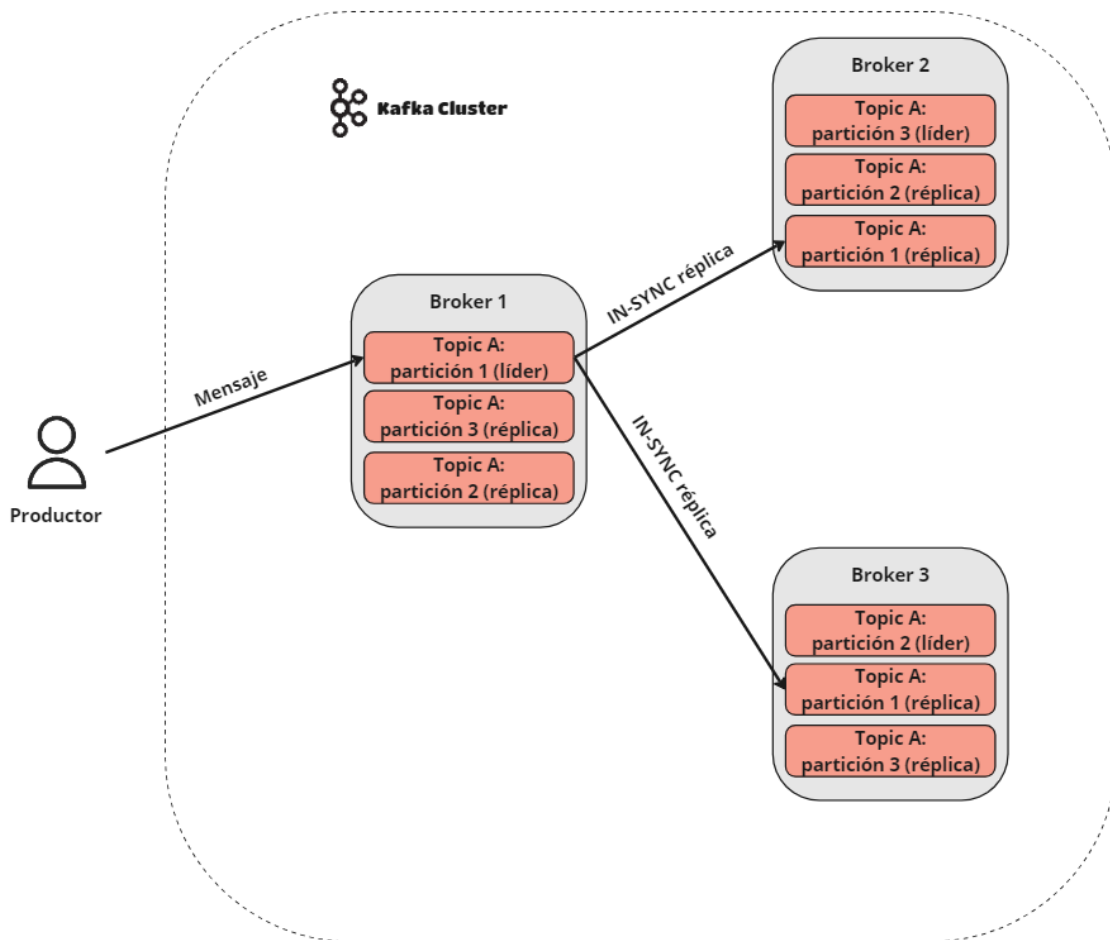


Figura 2: Diagrama conceptual ejemplo de replicación en Kafka

La utilidad de la replicación es hacer la arquitectura más robusta, previniendo la pérdida de información en caso de caída de uno de los servidores. Así pues, si un *broker* deja de funcionar, las particiones que lideraba pasarán a ser lideradas por alguno de los *broker* que las tenían como réplicas.

El valor del factor de replicación puede variar entre 1 y el número total de servidores del clúster. Con un factor de replicación N , se consigue que el sistema soporte la caída de $N-1$ *broker* sin perder información.

El tiempo que se almacenan los mensajes en un *topic* también es un parámetro que se puede configurar. Cada *topic* se identifica mediante un nombre, que se asigna en el momento de crearlo. Una vez creado un *topic* los parámetros que lo caracterizan no se pueden modificar.

- **Productor:**

Un productor Kafka es todo proceso que escribe mensajes en un *topic* de un clúster de Kafka. La comunicación entre los productores y Kafka se lleva a cabo mediante el protocolo propio de Kafka llamado *Kafka Protocol*.

Los productores para enviar mensajes necesitan que el usuario les indique una lista con uno o más *topic* en los cuales escribir mensajes y al menos el nombre de host o IP y

puerto de uno de los *broker* que componen el clúster. Las conexiones con cada uno de los nodos de Kafka se hacen por defecto en el puerto 9092.

El productor, una vez se arranca se conecta con alguno de los *broker* del clúster y solicita toda la información acerca de cómo está distribuido el *topic* al que quiere mandar mensajes. Una vez obtiene esta información comienza a escribir sobre el *topic* enviando mensajes a los nodos del clúster. Por defecto cuando un productor escribe datos en un *topic* los mensajes se reparten entre las diferentes particiones en modo *Round Robin*. Este proceso se muestra más detallado en el “Anexo 3: Comunicación entre un productor y Kafka”.

Una particularidad de Kafka es que la información se almacena en forma de bits, por lo que los mensajes deben ser serializados antes de enviarlos al *broker* correspondiente. La serialización consiste en convertir en bits datos que son de tipo objeto (*int*, *String*, *float*...).

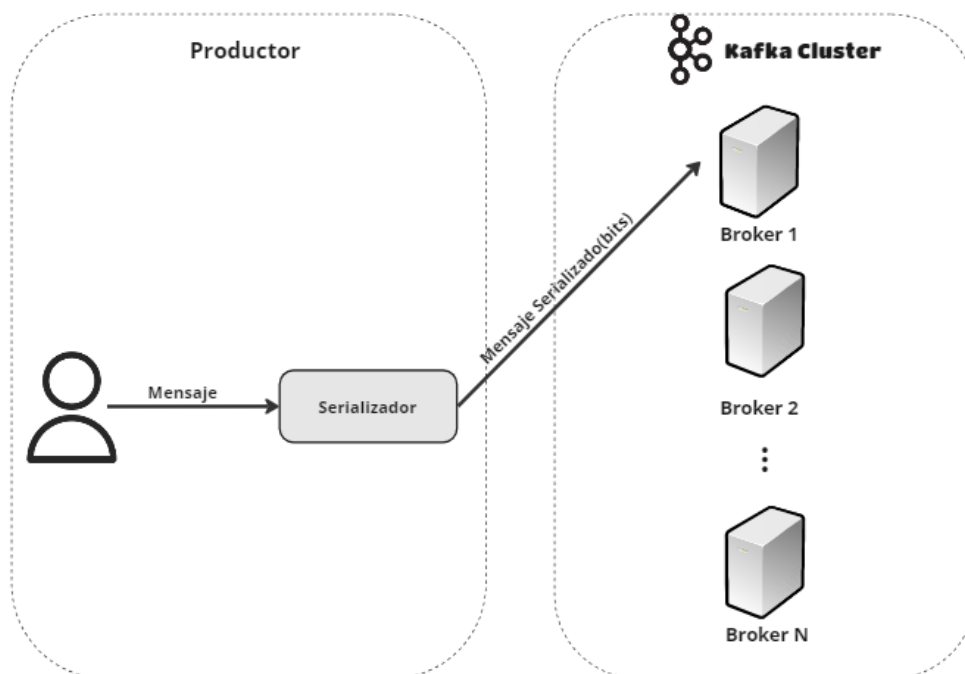


Figura 3: Diagrama conceptual de la serialización de mensajes

Debido a que el protocolo usa TCP, los mensajes tendrán una confirmación por parte de los *broker* en forma de ACK. Un punto importante de la configuración del productor es la gestión que hace de los ACK, dado que hay 3 posibles modos de funcionamiento:

- El productor publica mensajes en los *broker* e ignora completamente los ACK que éstos le envían. En caso de que un mensaje se pierda no se producirá ningún reenvío por parte del productor y simplemente se pierde la información. Es el que produce una menor congestión en la red y retardo, pero no es habitual utilizarlo debido a la alta probabilidad de pérdidas.

- El productor manda un mensaje al líder de una determinada partición, y si éste lo recibe, inmediatamente envía el ACK confirmando su recepción. Si el productor no recibe la confirmación del mensaje, lo reenvía hasta que éste llegue correctamente. En este caso el servidor de Kafka envía el ACK sin saber si el resto de *broker* han recibido correctamente el mensaje ISR.
- El productor manda un mensaje a un broker y éste espera confirmación del resto de *broker* de que han recibido el mensaje ISR para mandar el ACK al productor.

Con el objetivo de tener una menor congestión de red, los productores pueden, en vez de enviar los mensajes a los *broker* de uno en uno, mandarlos por lotes, es decir, esperar a reunir un número determinado de mensajes, y enviarlos todos simultáneamente. Además, estos lotes pueden ser comprimidos, lo que reduce la latencia y el tamaño de los mensajes al almacenarse en los *broker*.

- **Consumidor:**

Se llama consumidor Kafka a todas aquellas aplicaciones que se encargan de leer los mensajes publicados en aquellos *topic* a los que está suscrito. De la misma forma que los productores, los consumidores se comunican con los nodos de un clúster de Kafka mediante el Kafka *protocol*.

El consumidor necesita una lista de aquellos *topic* de los que se desean leer mensajes y al menos la dirección y puerto asociado de uno de los *broker* del clúster. Al arrancar un consumidor, éste solicita a uno de los nodos del clúster toda la información relativa a un *topic* y posteriormente empieza a solicitar mensajes.

Un concepto relevante es el de grupo de consumidores. Este grupo consiste en la unión de múltiples consumidores para leer la información de un *topic* de manera colectiva. Cuando varios consumidores pertenecientes al mismo grupo establecen conexión con el clúster de Kafka para leer mensajes de un determinado *topic*, las particiones en las que esté dividido el tema se reparten entre todos los consumidores, de tal forma que nunca dos consumidores pueden leer mensajes de una misma partición y entre todos los consumidores se lee la totalidad del contenido del *topic*. Gracias a ello se permite el paralelismo y se favorece la escalabilidad a la hora de leer mensajes.

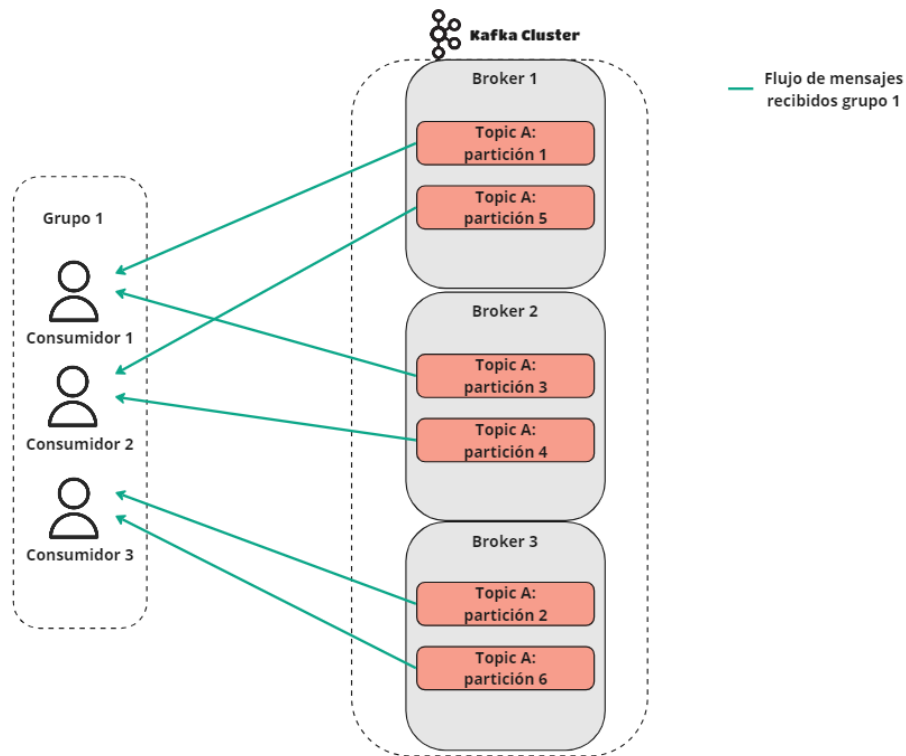


Figura 4: Diagrama conceptual grupo de consumidores

En el caso de tener dos grupos de consumidores distintos, en cada uno de los grupos los consumidores estarán leyendo de la totalidad de particiones de un *topic*, es decir dos consumidores que pertenecen a distintos grupos pueden leer de la misma partición, pero dos que pertenecen al mismo grupo no.

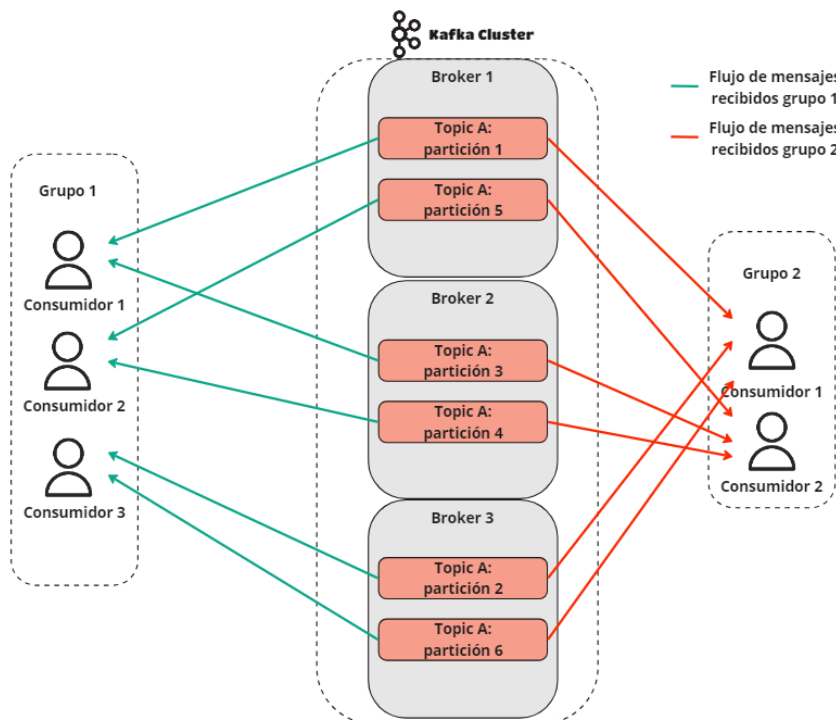


Figura 5: Diagrama conceptual múltiples grupos de consumidores

Según cómo configuramos el consumidor hay 3 posibilidades a la hora de comenzar a recibir información:

- El consumidor solicita recibir todos los mensajes almacenados en los *broker*.
- El consumidor solicita recibir únicamente mensajes que se escriban en Kafka a partir del momento en el que se inicia.
- El consumidor solicita recibir mensajes a partir del último mensaje leído.

Dado que Kafka envía la información en forma de bytes, el consumidor deberá implementar un deserializador que permita transformar los bytes en un objeto.

Para ampliar la información relativa al consumidor se debe consultar el “Anexo 4: Comunicación entre un consumidor y Kafka”.

Apache Flink

[6] Plataforma de procesamiento de datos en tiempo real y por lotes que permite, de una manera eficiente y escalable, el análisis y transformación de grandes volúmenes de tráfico. Para la transformación de datos, Flink ofrece un modelo de programación flexible, pudiendo escribir código en lenguajes como Java.

Wireshark

Wireshark es un software de código abierto que permite analizar los protocolos de red a partir de tráfico capturado, proporcionando una información detallada acerca de los paquetes que viajan por la red. Cuenta con una interfaz gráfica que permite visualizar toda la información de una manera sencilla.

Tcpdump

Herramienta de captura y análisis que se ejecuta en línea de comandos y permite almacenar la captura para que Wireshark (o cualquier otra herramienta) pueda realizar un análisis más detallado.

2.2. Protocolos

A continuación, se explican brevemente los protocolos de red relacionados con este proyecto.

IP

IP (Internet Protocol) [7] es un protocolo cuya función es proporcionar las funciones necesarias para enviar un paquete de bits desde una fuente a un destino a través de un sistema de redes interconectado.

UDP

UDP (User Datagram Protocol) [8] es un protocolo definido para hacer posible una comunicación en un entorno de redes interconectadas en modo conmutación de paquetes. Este protocolo utiliza el protocolo IP como protocolo de nivel inferior. Este protocolo no realiza ningún tipo de control sobre el tráfico obteniendo un retardo menor en transmisión.

TCP

El protocolo TCP (Transmission Control Protocol) [9] se define como un protocolo que proporciona una conexión fiable entre host en un entorno de conmutación de paquetes. Para ello hace uso de mecanismos como el control de congestión y control de flujo. Al igual que UDP utiliza el protocolo IP como protocolo de nivel inferior.

RTP

El protocolo RTP (Real-time Transport Protocol) [10] proporciona servicios de transporte para datos con características de tiempo real, como por ejemplo audio y video interactivo. Generalmente las aplicaciones utilizan el protocolo RTP sobre el protocolo UDP. Esto se debe a que en una transmisión en tiempo real se prioriza tener un menor retardo asumiendo que puede suponer la pérdida de información.

A pesar de no tomar medidas frente a posibles problemas, el protocolo RTP usa números de secuencia, que se incrementan en una unidad con cada paquete enviado, así como un *Timestamp* que indica cuándo se emite el datagrama. Gracias a esto se pueden detectar errores e implementar una solución a los mismos de manera ajena al protocolo.

RTCP

RTCP (Real-time Transport Control Protocol) es un protocolo que trabaja en conjunto con RTP, basado en el envío periódico de paquetes a todos los participantes de una sesión multimedia, con el objetivo principal de proveer *feedback* acerca de la calidad de los datos de la sesión.

SIP

SIP (Session Initiation Protocol) [11] es un protocolo de control a nivel de aplicación que permite establecer, modificar y terminar sesiones multimedia, tales como llamadas VoIP. SIP hace uso de elementos llamados proxy servers que ayudan a direccionar las peticiones a la ubicación del usuario, autorizar y autenticar usuarios para determinados servicios y proporcionar características a los usuarios. Por normal general, los mensajes SIP se envían al puerto 5060.

Kafka Protocol

Protocolo propio de Kafka utilizado para la comunicación entre un servidor Kafka y sus clientes. El protocolo se utiliza sobre una conexión TCP.

2.3. Parámetros relacionados con la calidad del audio

A continuación, se muestran los parámetros que influyen en la calidad de una comunicación telefónica [12] y cómo a partir de ellos podemos obtener una estimación de la calidad percibida por los usuarios.

MOS

Para estimar la calidad percibida por los usuarios en las llamadas telefónicas se definió la métrica del MOS (mean opinion score), la cual consiste en asignar una puntuación de entre 1 y 5 según la calidad de la comunicación.

Inicialmente era una medida subjetiva, donde las personas puntuaban la calidad de la llamada en función de su percepción. Sin embargo, para las llamadas VoIP, es posible hacer una estimación del MOS gracias a la correlación de estas puntuaciones subjetivas con las de una medida objetiva como el Factor R.

OWD

OWD (one way delay) se define como el tiempo que tardan los datagramas en transmitirse del emisor al destinatario. Este valor depende de la distancia y la velocidad de transmisión.

RTT

RTT (Round Trip Time) indica el tiempo medio que los paquetes tardan en propagarse al destinatario y volver al emisor. Es más sencillo de medir que el OWD porque no requiere sincronización de relojes. Además, es un parámetro que influye en la percepción subjetiva.

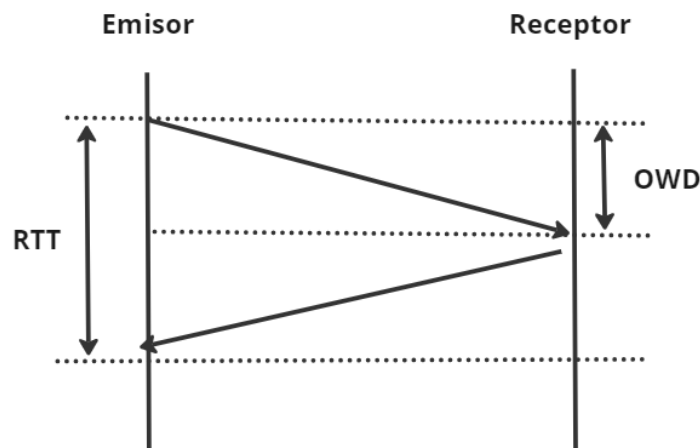


Figura 6: Diagrama conceptual OWD y RTT

Pérdidas

Cuando se transmiten paquetes a través de internet cabe la posibilidad de que éstos nunca lleguen a su destinatario. Dado que en las comunicaciones VoIP el audio se transmite sobre el protocolo UDP, que no realiza ningún tipo de control sobre los paquetes enviados. Estos paquetes no se reenvían y por lo tanto simplemente se pierde dicha información.

Jitter

El *jitter* se define como la fluctuación media en el retardo que tienen los datagramas de una comunicación.

En las comunicaciones VoIP los datagramas enviados pueden verse afectados por factores como la congestión de la red, lo que hace que el retardo no sea el mismo para cada uno de los paquetes que se transmiten en una conexión telefónica. A la diferencia máxima de retardo entre dos paquetes consecutivos se le denomina delta máxima.

Este parámetro puede llegar a provocar pérdidas si los paquetes no se reciben antes de ser reproducidos en el receptor.

De-jitter buffer

Con el objetivo de minimizar las pérdidas provocadas por el *jitter*, en los sistemas donde se transmiten datos en tiempo real se implementa un *de-jitter* buffer, que sirve para almacenar temporalmente los paquetes que van llegando antes de ser reproducidos en el receptor. Allí se reordenan si es necesario y se descartan aquellos que llegan demasiado tarde.

Factor R

El factor R se define como una medida para estimar la calidad del audio en telefonía IP a partir de los parámetros definidos anteriormente. Se calcula a partir de la siguiente fórmula:

$$\text{➤ } R = 93,2 - \frac{\text{Retardo efectivo}}{40} - 2,5 * \text{porcentaje de pérdidas}$$

Siendo el retardo efectivo:

$$\text{➤ } \text{Retardo efectivo} = OWD + 2 * \text{jitter} + 10$$

Para estimar el MOS en una comunicación se hace uso de la siguiente fórmula:

$$\text{➤ } MOS = 1 + 0,035 * R + 0,000007 * R * (R - 60) * (100 - R)$$

Capítulo 3: Escenario previo y planteamiento del problema

Para el desarrollo del Trabajo de Fin de Grado se tiene como escenario de partida la infraestructura de la empresa en la sucursal ubicada en Madrid, donde se va a implementar el sistema desarrollado en una primera instancia.

Tal y cómo se muestra en la “Figura 7: Escenario Previo”, la topología de red consiste en dos Core LAN Switches que dan acceso a internet, con los cuales se pretende tener los enlaces duplicados, asegurando el funcionamiento en caso de la caída de alguno de ellos. Un nivel por debajo encontramos la capa de acceso, compuesta por una serie de switches que conectan con el resto de los elementos de la red.

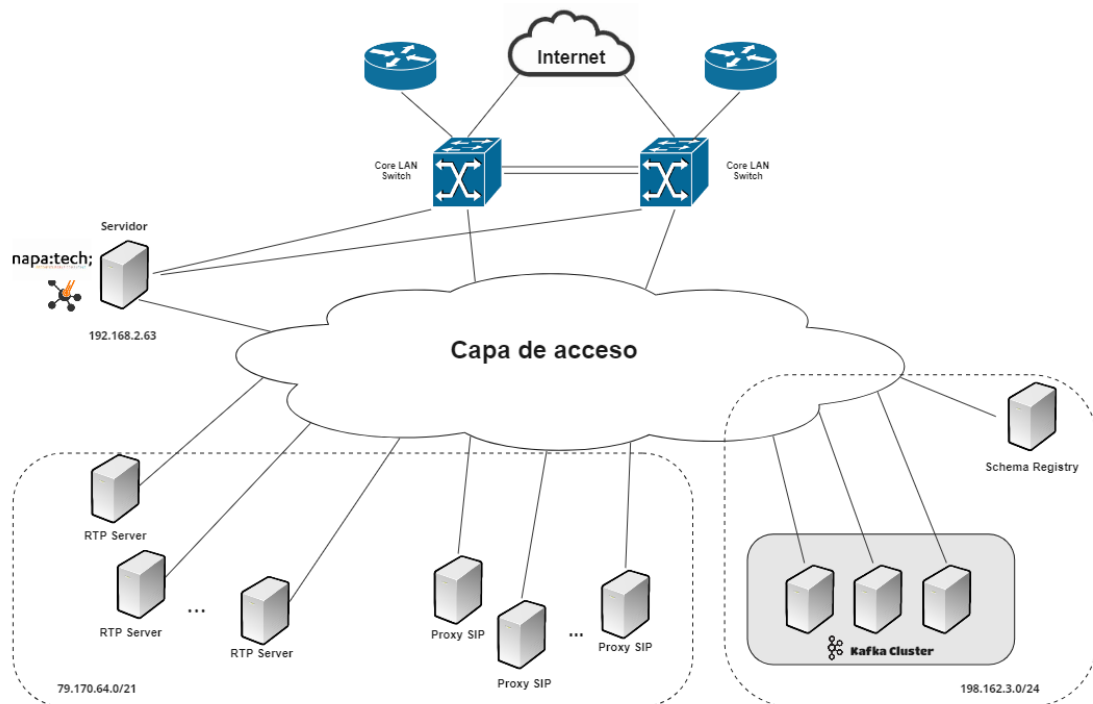


Figura 7: Escenario Previo

Durante el transcurso de una llamada telefónica podemos diferenciar claramente dos tipos distintos de datos. En primer lugar, tenemos la información de señalización, la cual utiliza el protocolo SIP y comprende las fases de establecimiento y terminación de la llamada. Cuando entra una llamada de un cliente a través de internet, los datos de señalización llegan a un Servidor proxy IN, y éstos se enrutan desde un servidor proxy OUT de nuevo hacia internet.

Durante el proceso descrito con anterioridad, los servidores Proxy generan un CDR (Call Detail Record) de la llamada a partir de la señalización. La función de los CDR comprende desde la tarificación de la llamada hasta el control de calidad pudiendo registrar el motivo de la caída de la conexión. Éstos CDR son enviados a Kafka como se puede observar en la “Figura 8: Datos de Señalización”, con el objetivo de poder ser leídos por otras aplicaciones que pueden a su vez generar métricas y reportes. De esta

manera se obtienen las estadísticas de llamadas contestadas y duración media de éstas, mencionadas en el apartado “1.2 Objetivos y Motivación”.

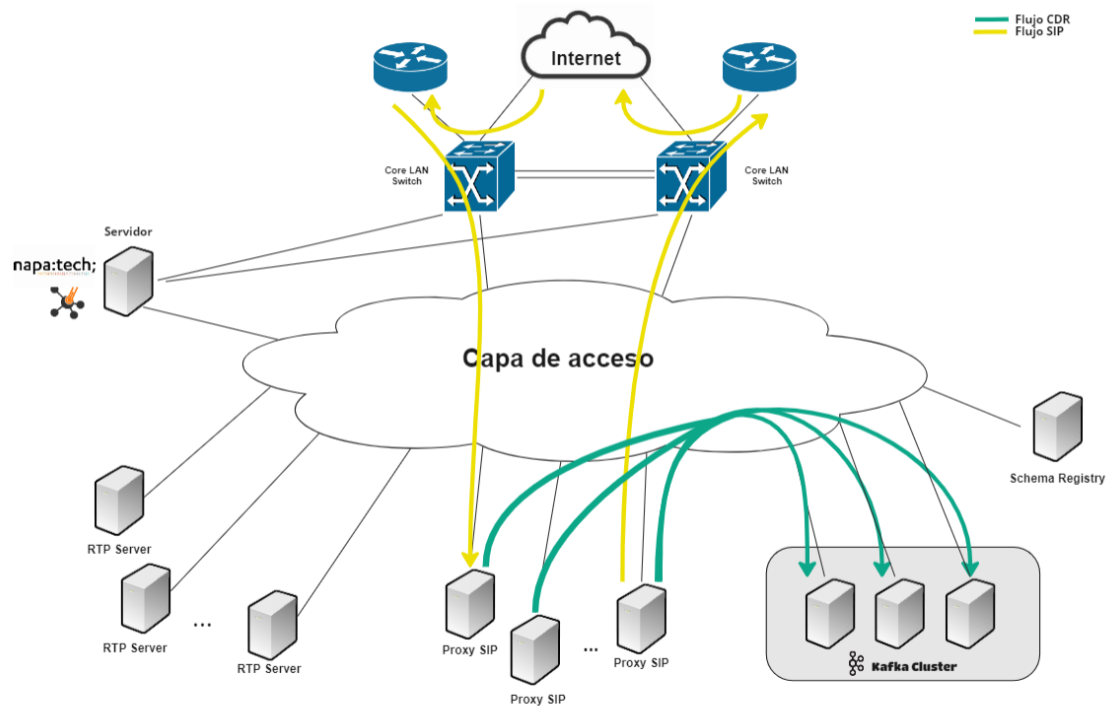


Figura 8: Datos de señalización

Una vez establecida la conexión mediante SIP, los datos de voz se transmiten por medio del protocolo RTP. En esta comunicación tendremos 4 flujos que transcurren por los servidores. Para la información que envía el dispositivo que origina la llamada hasta el receptor habrá 2 flujos. Uno dirigido hacia los servidores RTP de la empresa por parte del cliente, y el segundo por parte de los servidores RTP hacia el proveedor. Los otros dos flujos corresponden con el tráfico que genera el usuario que recibe la llamada, que comprende los mensajes dirigidos del proveedor al servidor RTP y del servidor RTP hacia el cliente.

La combinación de los dos flujos establecidos entre el cliente y los servidores RTP (tanto el flujo dirección fuente -> destino como el flujo dirección destino -> fuente) se denomina flujo *inbound*. De igual manera la combinación de los flujos de entrada y salida entre el proveedor y los servidores RTP se denomina flujo *outbound*.

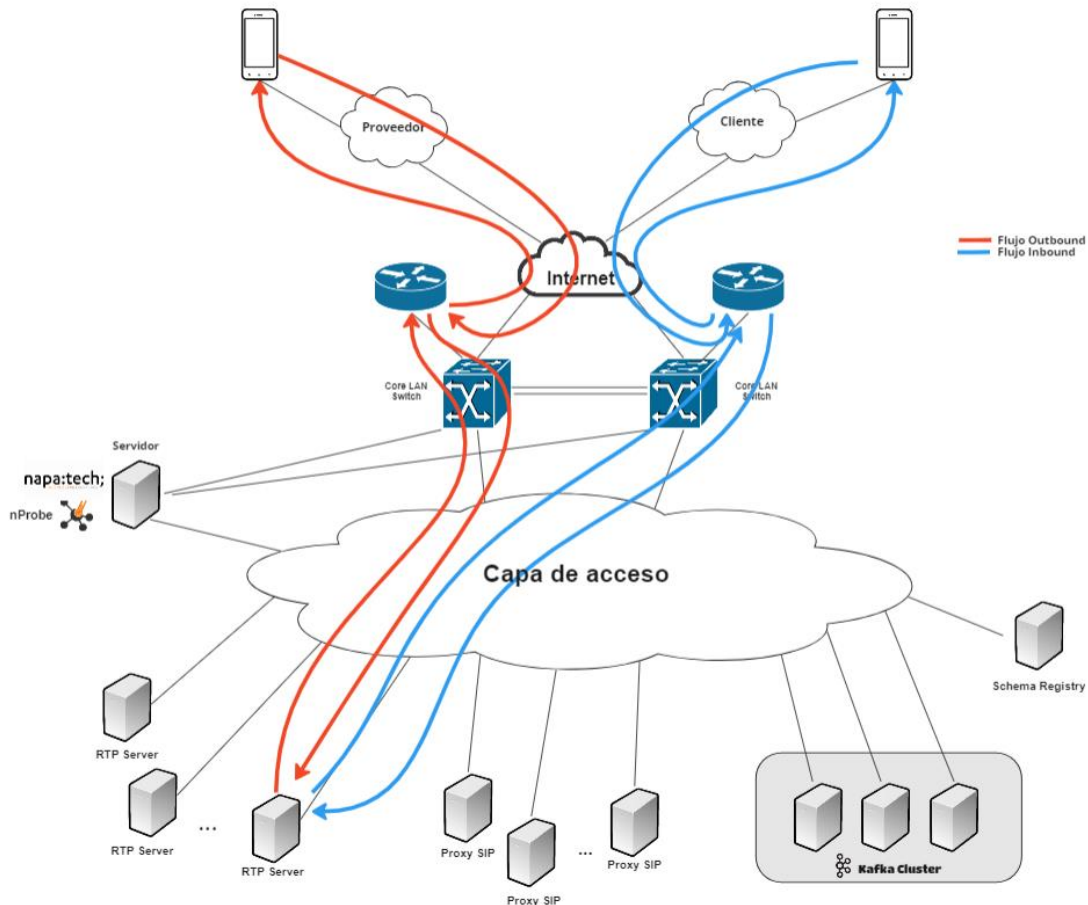


Figura 9: Tráfico RTP

En el escenario con el que trabajamos de partida las CDR tan sólo tienen en cuenta la información de señalización, pero como se ha mencionado en la introducción, las métricas que nos permiten caracterizar la calidad del audio de una llamada se obtienen a partir del flujo RTP. Por lo tanto, la problemática del proyecto reside en elaborar un sistema que permita enriquecer las CDR existentes añadiendo estadísticas obtenidas durante el intercambio de datos de voz.

Para el análisis de los flujos RTP se cuenta con un servidor cuyos puertos de monitoreo están conectados mediante interfaces *mirror* (que replican todo el tráfico, incluyendo RTP) a los Core LAN switches que dan acceso a internet por los cuales entra todo el tráfico proveniente o dirigido a internet. Dicho servidor cuenta fundamentalmente con una tarjeta NT200A02 SmartNIC, que permitirá filtrar tráfico y un software nProbe para la obtención de métricas de los tráficos filtrados. La “Figura 10: Tráfico resultante del port mirroring” muestra lo explicado anteriormente.

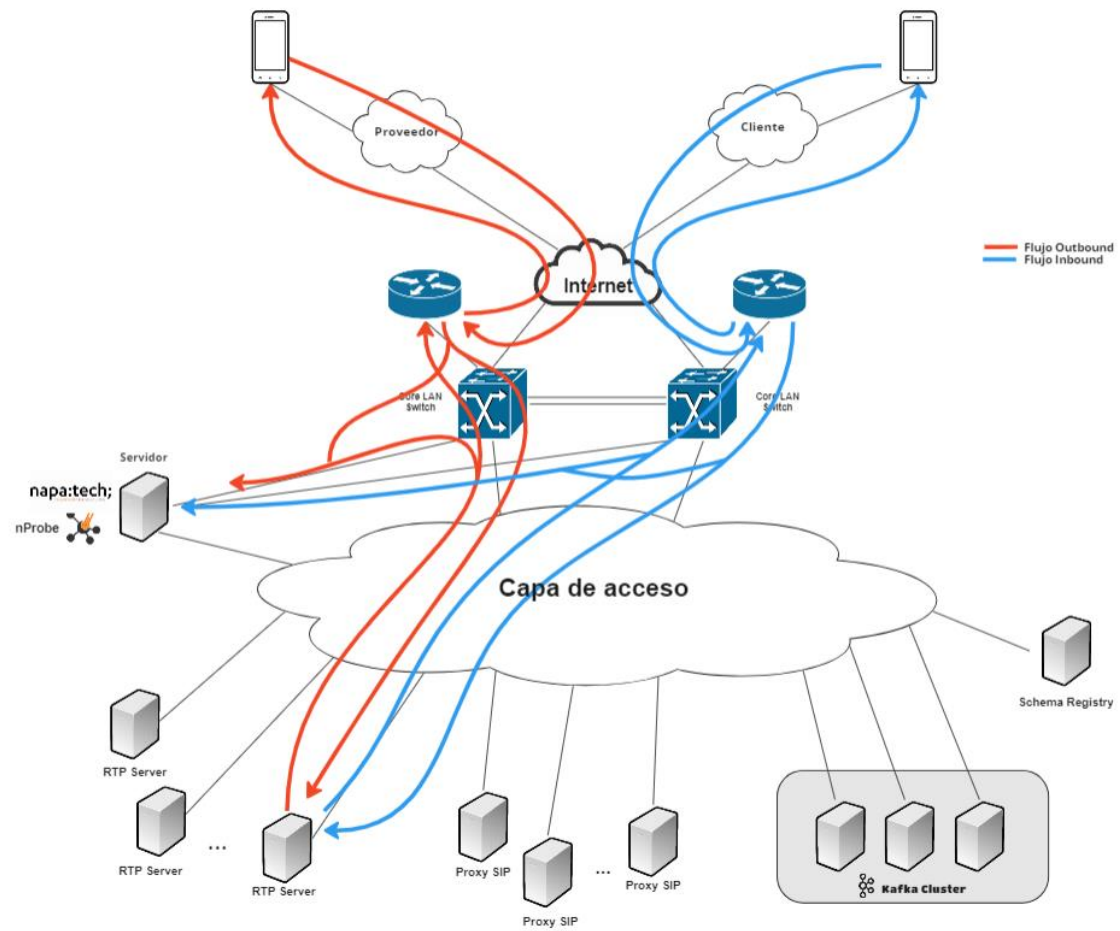


Figura 10: Diagrama del tráfico resultante del port mirroring

Capítulo 4: Descripción del sistema

4.1. Planteamiento general

Para solventar la problemática descrita en el capítulo anterior, la solución consiste en una primera instancia en la captura de los paquetes correspondientes al audio de una comunicación telefónica, los cuales son transportados por la red utilizando el protocolo RTP.

Como se menciona en el “Capítulo 3: Escenario previo y planteamiento del problema”, para el desarrollo del sistema, se cuenta con un servidor el cual está recibiendo todo el tráfico que transcurre entre los Core LAN switches e Internet. Este servidor cuenta con un sistema operativo Linux, concretamente CentOS.

El tráfico redireccionado al servidor por medio de *port mirroring* en los *switch* va a ser capturado por una tarjeta de red de Napatech. La tarjeta que se tiene instalada en el servidor es la NT200A02 SmartNIC en su versión 8x10G la cual cuenta con 8 puertos de entrada, donde cada uno de ellos soporta un tráfico de hasta 10Gb/s. De estos 8 puertos se van a utilizar 2 de ellos, conectados a cada Core LAN switch. Del mismo modo, cada uno de los dos enlaces que conectan los Core LAN Switch a internet tiene una capacidad de 10Gb/s, lo cual se ajusta a la limitación de capacidad soportada por los puertos de la tarjeta Napatech.

Para poder analizar la calidad del audio, de todo el tráfico que está capturando la SmartNIC tan solo nos interesan aquellos paquetes que viajan bajo el protocolo RTP. Por tanto, se deberá distinguir este tipo de tráfico del resto mediante un filtrado de paquetes. Dicho filtrado se va a realizar gracias al software de captura de la tarjeta, que permite cargar filtros programables utilizando un lenguaje de programación específico de Napatech llamado NTPL. En el “Anexo 1: Guía de instalación y configuración del software Napatech” se explica cómo instalar y configurar el software que utiliza la tarjeta.

Posteriormente, los paquetes filtrados, correspondientes a flujos RTP, van a ser capturados por el software nProbe, que permite el monitoreo y análisis de tráfico en tiempo real, y que se encuentra instalado en el servidor. Este software permite la obtención de métricas de los flujos que captura, por lo que lo usaremos para calcular estadísticas del tráfico RTP de cada llamada que pasa por los servidores de la empresa, tales como el *jitter*, MOS o pérdida de paquetes.

La necesidad de almacenar una gran cantidad de datos transmitidos en tiempo real hace que una vez obtenidas estas métricas se envíen a un clúster Kafka. Esta aplicación es una herramienta de big data que va a permitir albergar todos los registros por un tiempo limitado para que puedan ser leídos y procesados por otras aplicaciones. En este mismo clúster también se encuentran los CDR producidos con métricas de la señalización como se detalla en el “Capítulo 3: Escenario previo y planteamiento del problema”.

En última instancia se procederá a enriquecer las CDR añadiendo las métricas relacionadas con la calidad del audio. Para ello será necesario el uso de una aplicación que permita procesar transmisiones continuadas de datos, denominadas *stream* con una

gran cantidad de información. La plataforma elegida para el procesamiento de datos es Apache Flink, la cual ejecutará un programa de procesamiento de los datos que correlará las métricas del RTP con sus correspondientes métricas obtenidas de la parte de señalización.

El programa que define la lógica del procesamiento de los datos deberá leer tanto de los *stream* con las estadísticas calculadas a partir del audio de una llamada telefónica como de las CDR calculadas a partir del SIP. Conforme se leen los datos de Kafka se deberán transformar los mensajes y agruparlos en función de las direcciones IP y puertos utilizados en la comunicación, dado que estos parámetros nos permitirán relacionar ambos *stream*. Por último, se creará un nuevo *stream* cuyos mensajes serán la unión de aquellos cuyas direcciones IP y puertos son coincidentes, creando así una versión mejorada de las CDR añadiendo estadísticas que permitirán analizar la calidad del audio de una comunicación telefónica. De nuevo se almacenarán en Kafka las CDR enriquecidas para que otras aplicaciones puedan leer los datos para analizarlos u obtener estadísticas a partir de ellos.

4.2. Filtrado de paquetes RTP

Nuestra tarjeta Napatech, conectada a los Core LAN *switch* va a estar recibiendo por sus interfaces una cantidad inmensa de tráfico, donde se mezclan todo tipo de flujos sin distinción alguna. Para ello se ha elaborado un filtro que, de todo ese tráfico, seleccione únicamente aquellos paquetes pertenecientes a una determinada comunicación telefónica.

Para ello ha sido necesario la programación de un filtro en lenguaje NTPL que al cargarse en la tarjeta de red cumpla con la funcionalidad descrita anteriormente. Este lenguaje nos permite filtrar los paquetes entrantes en función de sus cabeceras, por lo tanto, haremos uso de las cabeceras tanto a nivel IP como a nivel de transporte para cumplimentar el filtro.

En la “Figura 9: Tráfico RTP” se muestran precisamente los flujos con los que nos deseamos quedar tras el filtro de entre todo el tráfico entrante y saliente. Por lo tanto, nuestro programa ha de ser capaz de identificar estos flujos de entrada y de salida RTP mediante el análisis de las cabeceras de los paquetes.

Centrándonos en los flujos entrantes éstos podremos identificarlos ya que serán paquetes UDP cuya dirección IP destino estará englobada en la red 79.170.64.0/21 y el puerto destino estará englobado en el rango 10.000-65.535, rango de puertos en el cual están escuchando los servidores RTP. Además, debemos distinguir aquellos paquetes RTP de una comunicación de su correspondiente flujo RTCP, protocolo asociado que proporciona información de control de una sesión multimedia en tiempo real. La distinción se hace según si el puerto del que hace uso el servidor RTP es par o impar, pues las conexiones RTP se realizan utilizando puertos pares, mientras que el RTCP asociado irá dirigido al puerto impar inmediatamente superior.

El filtro es capaz de analizar la cabecera IP, de ahí extrae la dirección IP destino, la cual se puede leer de los bytes 12, 13, 14 y 15 de la cabecera (“Figura 11: Cabecera IP”). Una vez obtenida, la compara con el rango de IPs que engloba la red de la empresa.

4	8	16	31
VERS	LON	Tipo	Longitud
Identificación		Flags	Offset
TTL	Protocolo	Comprobación de cabecera	
Dirección IP origen			
Dirección ip destino			
Opciones		Padding	
Datos ...			

Figura 11: Cabecera IP

De la misma manera, el filtro accede a la cabecera UDP del datagrama, para extraer el puerto destino, que corresponde a los bytes 2 y 3 de la cabecera. El valor obtenido se compara con el rango de puertos en el que escuchan los servidores RTP y se comprueba si es par o impar.

4	8	16	31
Puerto origen		Puerto destino	
Longitud		Checksum	
Datos			
...			

Figura 12: Cabecera UDP

Con estos valores, el filtro determina que un paquete pertenece a un flujo RTP cuando cumple todas las condiciones descritas anteriormente, y todos los que no cumplan con ellas se descartan.

De una manera análoga sucede para los flujos RTP que son dirigidos hacia internet. En ese caso nuestro filtro se queda con aquellos paquetes UDP cuya dirección IP fuente pertenece a la red 79.170.64.0/21, el puerto fuente se engloba en el rango 10.000-65.535 y además es par. Los datagramas que cumplen estas características son identificados como RTP, y el resto son descartados.

Así pues, el filtro descarta todo el tráfico que no nos es de interés para el desarrollo del sistema, y se queda únicamente con los flujos RTP.

El *output* del filtro se distribuye entre 15 interfaces lógicas de salida. El número de interfaces lógicas disponibles es un parámetro configurable en el fichero de configuración de la tarjeta Napatech. En nuestro caso, el valor elegido es de 16, donde 15 de ellas se encuentran a disposición de utilizarse en este proyecto, y la última se utiliza para otros fines. La elección de este valor se debe a las características del servidor en el que estamos trabajando, ya que éste cuenta con un total de 16 CPU. La funcionalidad del filtro descrita se detalla en la “Figura 13: Esquema del filtro”.

La finalidad de haber elaborado el filtro es que el software nProbe lea la información de las interfaces por las que saldrán los flujos RTP para luego calcular las estadísticas oportunas. Cada vez que se lanza un proceso de nProbe, éste utiliza una CPU distinta. Nuestro sistema va a trabajar con una cantidad muy elevada de tráfico, ya que aproximadamente pueden llegar a establecerse unas 500 llamadas cada segundo a través de nuestros servidores. Por ello, el distribuir el tráfico filtrado entre las 15 interfaces mencionadas anteriormente que están a nuestra disposición va a permitir lanzar 15 procesos nProbe, cada uno haciendo uso de una CPU distinta, aprovechando al máximo la cantidad de recursos que tenemos disponibles.

De no haber distribuido la carga computacional entre todas las CPU disponibles, podría darse la situación de que una CPU estuviese al 100% de utilización debido a la alta carga que soportaría, y debido a ello se perdieran paquetes posteriores al filtrado, haciendo que nProbe no pudiese calcular las métricas de los flujos de manera correcta.

La posibilidad de que a la salida del filtro los paquetes puedan ser distribuidos por diferentes interfaces nos ha llevado a necesitar establecer una estrategia de cómo se distribuyen los paquetes a su salida por las diferentes interfaces. Por defecto los paquetes se distribuyen en formato Round Robin, es decir, el primer paquete saldrá por la interfaz 0, el siguiente por la 1 y así sucesivamente. Esto crea otra nueva problemática, en la cual los paquetes pertenecientes a un mismo flujo acabarían saliendo por interfaces diferentes, y siendo leídas por distintos procesos nProbe.

Para que una instancia de nProbe pueda calcular correctamente los parámetros que miden una comunicación éste necesita tener el *stream* completo perteneciente al audio de una llamada, por lo que todos los paquetes de un mismo *stream* deben ser leídos por un mismo nProbe, es decir deben ser enviados por la misma interfaz de salida.

Para ello se ha implementado en el filtro una función de *hash*, que consiste en un identificador que se le aplica a cada paquete para poder relacionarlo con paquetes cuya naturaleza sea la misma. En nuestro filtro se ha aplicado una función de *hashing* a nivel IP y a nivel de transporte, por lo que, en su ejecución, aquellos paquetes en cuya cabecera se identifiquen coincidencias a esos niveles, como el protocolo utilizado,

direcciones IP o puertos fuente y destino siempre se les asignará la misma interfaz de salida. De esta manera aseguramos que todos los paquetes de una misma comunicación telefónica acaben siendo analizados por un único proceso nProbe, que podrá reconstruir el flujo de una manera precisa.

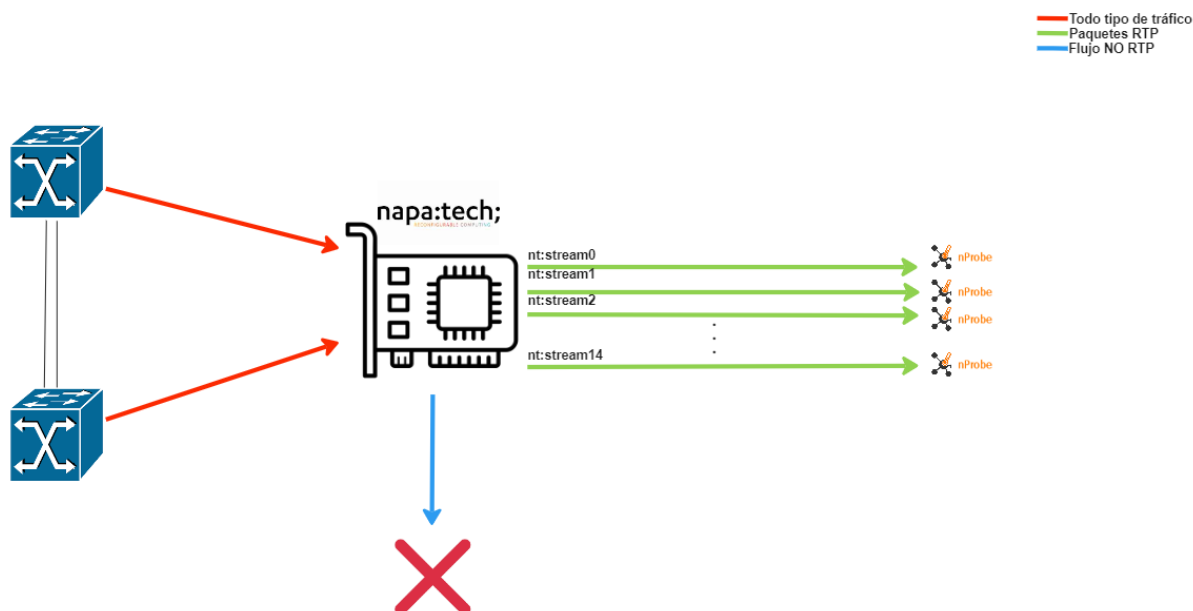


Figura 13: Esquema del filtro

Una vez que hemos diseñado el filtro, tan solo hay que cargarlo en la tarjeta. Esta operación es sencilla y se puede realizar en caliente, es decir, puedo cambiar mi filtro en cualquier momento sin tener que parar el sistema. Para ello basta con ejecutar la siguiente instrucción situándonos en la carpeta bin generada en la instalación del software de la Napatech:

```
./ntpl -f <directorio del filtro>/<nombre del filtro>.ntpl
```

Para conocer detalles acerca de la programación que tiene el filtro consultar el “Anexo 2: Implementación del filtro mediante programación en NTPL”.

4.3. Cálculo de métricas a partir de flujos RTP

A lo largo de la explicación del diseño del sistema desarrollado hasta este momento se ha descrito como se ha llevado a cabo un filtrado y una organización de tráfico para que el software nProbe sea capaz de calcular medidas que ayudarán a determinar la calidad del audio de una llamada VoIP.

Como se detalla en el apartado anterior, el *output* del filtro se reparte por 15 interfaces distintas, por lo tanto, se tienen que ejecutar 15 instancias diferentes de nProbe, cada una de ellas con su correspondiente fichero de configuración. Dicha configuración es igual para cada uno de ellos a excepción de la instrucción que indica de qué interfaz leen el tráfico.

También se asegura que todos los paquetes que pertenecen a un mismo flujo salen por la misma interfaz para que nProbe pueda reconstruir correctamente dicho flujo. Sin embargo, una de las problemáticas que han tenido que ser resueltas es que, al estar trabajando únicamente con RTP, solamente analizando este tipo de paquetes no podemos determinar cuándo un flujo ha terminado o, por el contrario, sigue activo. Esto es debido a que RTP va sobre el protocolo UDP en vez de TCP, que es un protocolo no orientado a conexión, en el que no hay fase de establecimiento y cierre y tampoco tenemos la correspondiente señalización SIP, asociada a la comunicación, que lo podría indicar.

Para que a partir de paquetes RTP, nProbe sea capaz de saber cuando debe cerrar un flujo se definen dos parámetros temporales los cuales pueden alterarse en la configuración dependiendo de las necesidades del sistema:

- **Lifetime-timeout:** Este parámetro determina el tiempo tras la recepción del primer paquete de un flujo tras el cual nProbe considerará cerrado un flujo, independientemente de si este permanece o no activo.
- **Idle-timeout:** Tiempo que ha de pasar desde la recepción del último paquete recibido de un flujo para que éste se considere inactivo, y por tanto se asuma que dicha comunicación ha terminado.

Una vez se da por cerrado un flujo, nProbe calcula aquellas métricas que le sean indicadas en la configuración de este, analizando todos los paquetes capturados de dicha comunicación. Cabe destacar que al referirnos a los flujos a partir de los cuales nProbe calcula estadísticas nos referimos a los flujos *inbound* y *outbound* definidos en el “Capítulo 3: Escenario previo y planteamiento del problema”, que comprenden tanto los flujos de entrada y salida entre los servidores y el cliente o entre los servidores y el proveedor respectivamente.

La solución que se ha seguido en el desarrollo del sistema, en el presente proyecto, respecto a esta problemática es la de asignar un valor al parámetro *lifetime-timeout* muy elevado, con el objetivo de que nunca se cierre un flujo que siga activo y un valor *idle-timeout* de 10 segundos. Por lo tanto, en nuestro sistema nProbe va a identificar cuándo un flujo ha terminado si transcurren 10 segundos desde el último paquete recibido.

Una vez transcurridos esos 10 segundos, nProbe calculará una serie de métricas de utilidad para caracterizar la calidad de cada comunicación telefónica. Los parámetros que deseamos mostrar han de indicarse en el pertinente fichero de configuración.

En el sistema se ha configurado cada instancia nProbe para que calcule el RTT, el *jitter*, el número de paquetes capturados, el número de paquetes perdidos, el número de paquetes descartados por el *jitter buffer*, el delta máximo, el factor R y el MOS en los dos sentidos de la comunicación. Para ayudar con su análisis y posterior procesamiento, estas estadísticas se muestran acompañadas de las direcciones IP fuente y destino, el protocolo usado a nivel de transporte, los puertos fuente y destino y los números de secuencia del primer y último paquete capturados.

La configuración desarrollada para el sistema se detalla en el “Anexo 5: Fichero de configuración de nProbe”.

4.4. Almacenamiento de las métricas calculadas. Kafka

La necesidad de que la gran cantidad de métricas calculadas a partir de flujos en tiempo real por nProbe, deban ser leídas por otras aplicaciones, hace imprescindible la utilización de una plataforma capaz de almacenar toda esta información para que pueda ser utilizada si se desea. Para ello se hace uso de Apache Kafka

Para el desarrollo del proyecto hemos contado con un clúster Kafka compuesto por 3 servidores distintos. En la “Figura 7: Escenario Previo” se ubica el clúster dentro de la red interna de la empresa. El nombre de host de estos *broker* es, mad-kafka-1.sonoc.io, mad-kafka-2.sonoc.io y mad-kafka-3.sonoc.io, y sus respectivas direcciones IP son 192.168.3.69, 192.168.3.70 y 192.168.3.71.

4.4.1. Configuración de un topic.

El primer paso a seguir ha sido la creación de un *topic* dónde almacenar toda la información calculada previamente a partir de los flujos RTP. El topic creado se ha llamado RTP_stats y se ha configurado con un número de particiones igual a 12 y un factor de replicación de 3. Es decir, cada uno de los tres *broker* es líder de 4 particiones y la información de cada una de ellas se replica en todos los nodos.

Con esta configuración conseguimos maximizar la utilización de recursos distribuyendo la carga entre los 3 nodos del clúster que tenemos a nuestra disposición. Con vistas a que en un futuro pueda ampliarse el tamaño del clúster también se garantiza la escalabilidad, pudiendo repartir el almacenamiento de las métricas entre todos los nodos del clúster hasta un número máximo de 12. Además, el sistema es robusto frente a la caída de alguno de los nodos, pudiendo soportar la caída simultánea de 2 de los *broker* sin perder información gracias a la replicación.

En la “Figura 14: Diagrama de la distribución de las particiones en el topic RTP_stats” se detalla cómo se reparte la información entre los distintos nodos del clúster. Cada *broker* es líder de las particiones coloreadas en amarillo siempre que esté activo. Las particiones de color naranja pasarán a ser liderada por el *broker* solamente si el servidor líder original se encuentra inactivo. Por último, aquellas coloreadas de rojo sólo son controladas por ese *broker* si los otros dos están caídos.

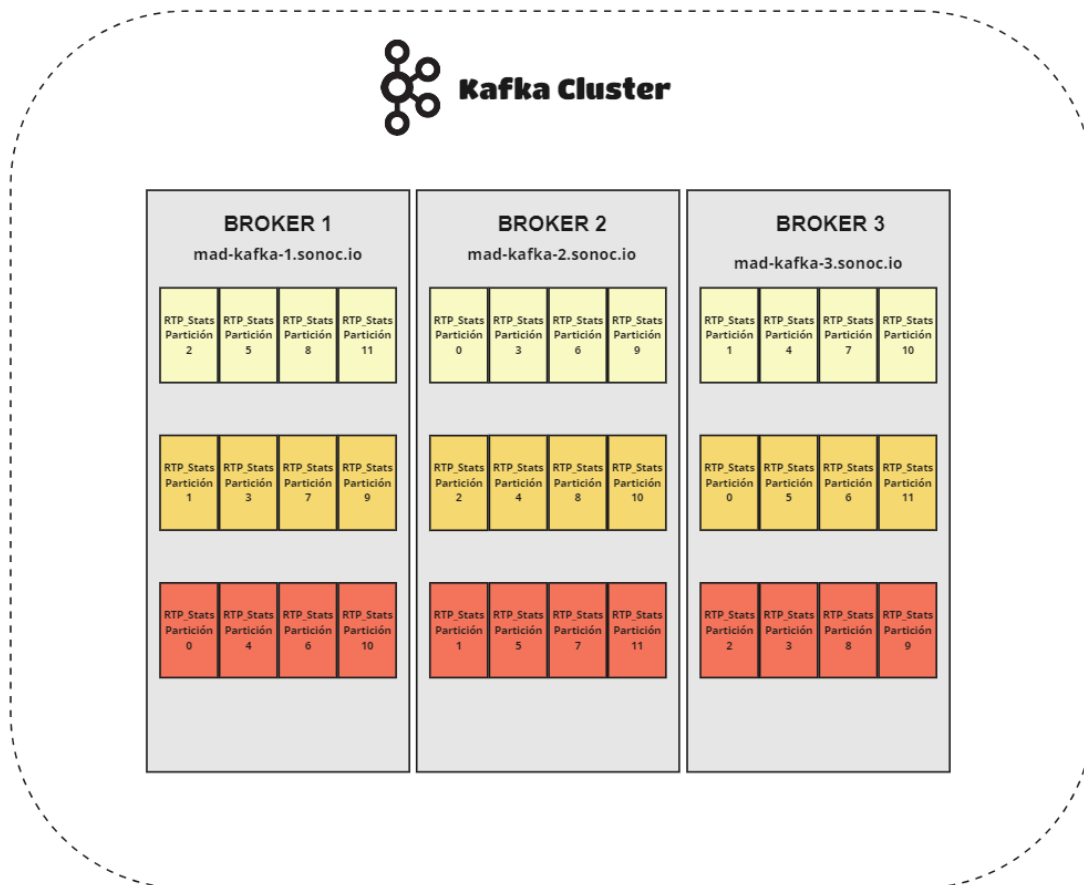


Figura 14: Diagrama de la distribución de las particiones en el topic RTP_stats

Para poder ver de una manera sencilla e intuitiva toda la información relativa a los *broker*, sus *topic*, particiones y los mensajes que se guardan en ellas se cuenta con una interfaz de usuario de Kafka.

UI for Apache Kafka b0c367c v0.6.2

Dashboard
iris-dev
MAD-SNC
Brokers
Topics
Consumers
Schema Registry
LAX-SNC
MUM-TAT
PAR-SNC
MIA-SNC

Topics / RTP_stats

Overview Messages Consumers Settings Statistics

Produce Message

Partitions	Replication Factor	URP	In Sync Replicas	Type	Segment Size	Segment Count	Clean Up Policy
12	3	0	36 of 36	External	19 GB	36	DELETE

Message Count
11474672

Partition ID	Replicas	First Offset	Next Offset	Message Count
0	2, 3, 1	1564813467	1565768415	954948
1	3, 1, 2	1564547696	1565501623	953927
2	1, 2, 3	1564706609	1565665970	959361
3	2, 1, 3	1564472946	1565431015	958069
4	3, 2, 1	1564381759	1565338512	956753
5	1, 3, 2	1564196196	1565148706	952510
6	2, 3, 1	1564599011	1565555484	956473
7	3, 1, 2	1564430077	1565388243	958166
8	1, 2, 3	1564555733	1565515203	959470

Figura 15: Interfaz de usuario de Kafka

4.4.2. nProbe como productor Kafka

Una vez creado el *topic* RTP_stats que permitirá el correcto almacenamiento y distribución de la información entre los tres nodos que constituyen el clúster podemos empezar a mandar los datos de las métricas calculadas para cada llamada telefónica.

En el “Capítulo 4.3: Cálculo de métricas a partir de flujo RTP” se menciona la necesidad de ejecutar 15 procesos nProbe para distribuir la carga de trabajo entre las diferentes CPU con las que cuenta el servidor en el que estamos trabajando. Debido a ello cada instancia de nProbe actuará como un productor de Kafka independiente, por lo que tendremos un total de 15 productores escribiendo todos sobre el *topic* creado anteriormente, como se muestra en la “Figura 16: Diagrama conceptual de los productores nProbe”.

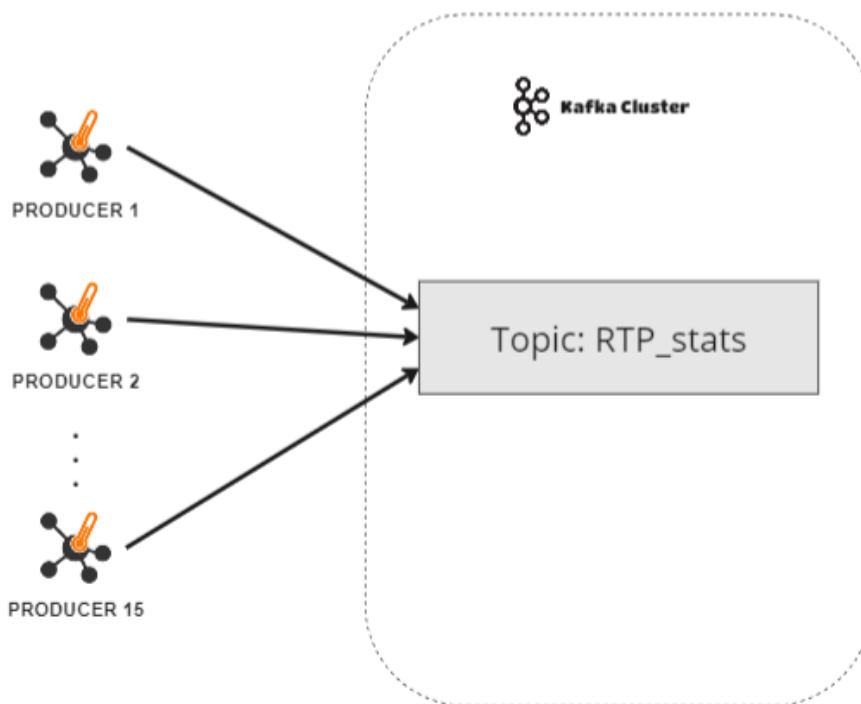


Figura 16: Diagrama conceptual de los productores nProbe.

La configuración que indica el comportamiento fundamental de cada proceso nProbe como productor de Kafka se indica en el fichero de configuración de este a través del comando - kafka. Los parámetros que hemos aportado para la configuración son:

- Listado de dirección o nombre de host y puerto de al menos uno de los *broker* del clúster: En nuestro caso hemos indicado el nombre de todos los nodos, para garantizar la robustez del sistema frente a la caída de alguno de ellos.
- Nombre del *topic* al que van dirigidos los mensajes: En nuestro caso RTP_stats.

- Topic adicional (En caso de querer escribir en más de un topic): En nuestro caso *none*.
- Gestión de los ACK: En el “Capítulo 2.1: Herramientas” se nombran las diferentes opciones disponibles a la hora de cómo un productor de Kafka gestiona los ACK. En este caso se ha elegido la opción de esperar confirmación únicamente por el *broker* líder de la partición donde se escribe, que garantiza una baja probabilidad de pérdidas con una menor congestión en la red.
- Formato de compresión: Para reducir el tamaño de los mensajes almacenados, se ha elegido utilizar la compresión *snappy*, ya que es la que ofrece una mejor relación entre el uso de CPU y compresión de entre los formatos de compresión que soporta Kafka.

Además, se pueden añadir opciones adicionales de configuración mediante el comando `-kafka-conf`. En el sistema desarrollado se ha configurado el productor para enviar los mensajes por lotes, con el objetivo de reducir la congestión de la red. Cada instancia de *nProbe* esperará a acumular métricas de 300 llamadas para enviarlas a Kafka de una manera agrupada. Si transcurrido un segundo no se cuenta con 300 métricas a enviar, se envía un lote con las que se hayan acumulado hasta ese momento.

La configuración que define el comportamiento detallado es la siguiente:

```
--kafka="mad-kafka-1.sonoc.io:9092,mad-kafka-2.sonoc.io:9092,mad-kafka-3.sonoc.io:9092;RTP_stats;none;1;snappy"
```

```
--kafka-conf="queue.buffering.max.ms=1000"
```

```
--kafka-conf="batch.num.messages=300"
```

A continuación, en la “Figura 17: Intercambio de mensajes de un productor con los broker de Kafka en el sistema desarrollado” se muestra un ejemplo del funcionamiento del sistema para la configuración elegida.

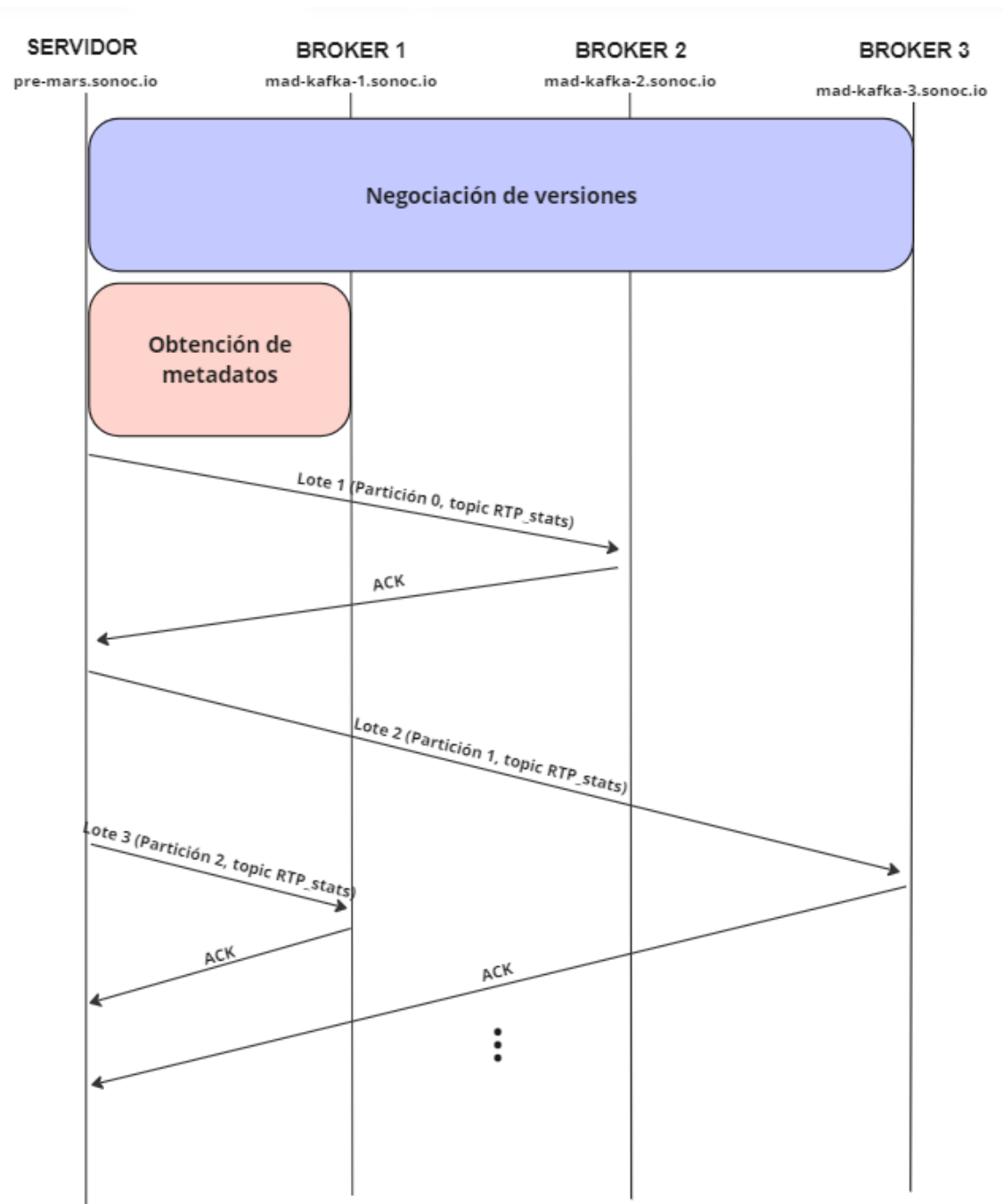


Figura 17: Intercambio de mensajes de un productor con los broker de Kafka en el sistema desarrollado

Ciñéndonos al esquema inicial de la topología de la red de la empresa, esta comunicación se lleva a cabo a través de los *switch* que conforman la capa de acceso de la red. En la “Figura 18: Producción de mensajes con las métricas calculadas” se muestra cómo se transmiten estos lotes de mensajes con las métricas calculadas de las llamadas telefónicas en la red.

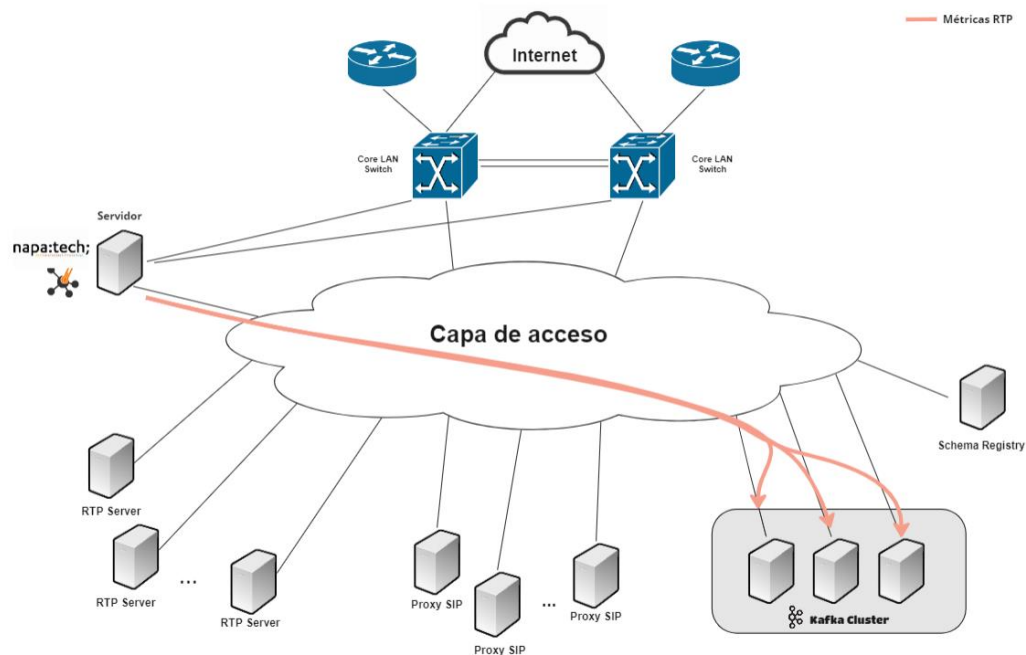


Figura 18: Producción de mensajes con las métricas calculadas

En la interfaz de usuario de Kafka podemos visualizar cada uno de los mensajes que son almacenados por los *broker*. En la “Figura 19: Ejemplo de métricas almacenadas en Kafka” se muestra un ejemplo de cómo se guarda la información. Ésta se guarda en formato JSON, es decir siguiendo una dupla clave-valor, siendo la clave el nombre de la métrica que se está calculando.

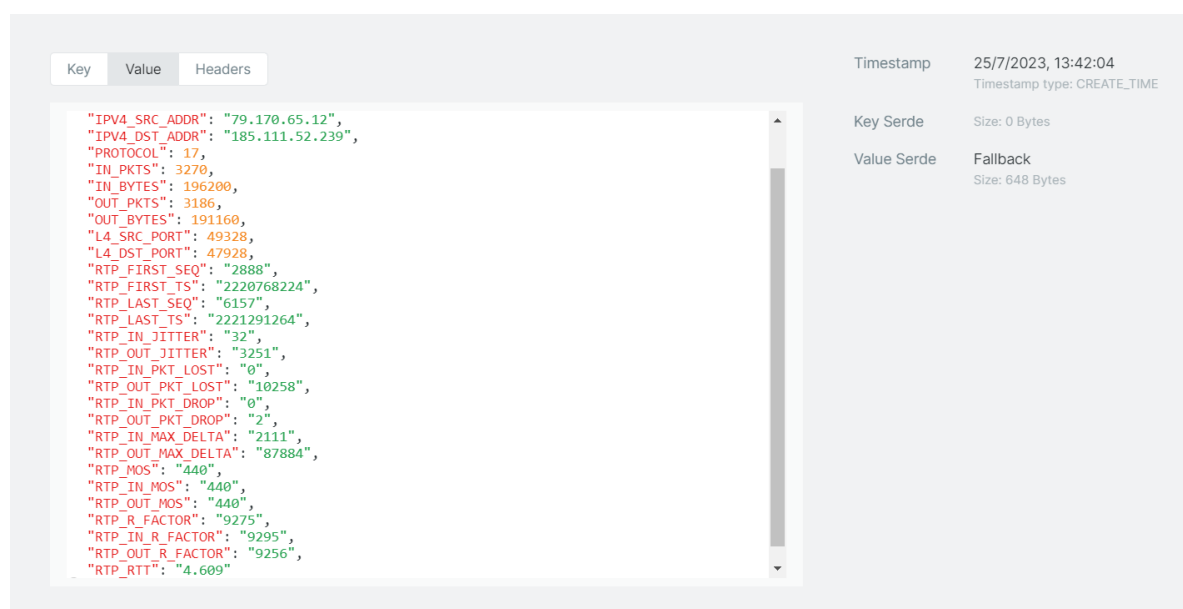


Figura 19: Ejemplo de métricas almacenadas en Kafka para el tópico RTP_stats

4.5. Procesamiento de datos para el enriquecimiento de CDR. Apache Flink

Si nos remontamos al “Capítulo 3: Escenario previo y planteamiento del problema”, se menciona que la empresa ya cuenta con un sistema implementado que genera unas métricas a partir de la información de señalización de las llamadas, a las que llamamos CDR. Éstos CDR también son enviados y almacenados en los nodos del clúster de Kafka en cuatro *topic* distintos, cuyos nombres son `mad_bts_dedalus_cdr_111`, `mad_bts_dedalus_cdr_112`, `mad_bts_dedalus_cdr_113` y `mad_bts_dedalus_cdr_114`.

El objetivo final del proyecto es obtener unos CDR enriquecidos, que combinen las métricas obtenidas de la parte de señalización con las métricas obtenidas a partir del flujo RTP de una misma llamada telefónica. Para llevar a cabo el procesamiento de una cantidad tan alta de información hemos hecho uso de Apache Flink, plataforma especializada en el procesamiento de *streaming* de eventos.

Al igual que Kafka, Flink está diseñado para trabajar de una manera distribuida formando un clúster entre las distintas máquinas. La arquitectura del clúster de Flink está formada por uno o más *task manager*, máquinas encargadas de ejecutar los procesos que conllevan la manipulación de los flujos de datos. Además, es imprescindible la presencia de un *job manager*, que se encarga de coordinar los distintos *task manager* y distribuir los procesos entre ellos.

Para poder llevar a cabo el enriquecimiento de los CDR se ha elaborado un programa de Java que, utilizando las librerías específicas de Flink implementa la lógica de procesamiento necesaria para realizar la tarea. Dicho programa, al ser ejecutado por los *task manager* de cualquier clúster de Flink debe ser capaz de configurar un consumidor de Kafka que permita leer los mensajes de los *topic* `mad_bts_dedalus_cdr_` y `RTP_stats`. Posteriormente se deberán transformar los datos para, de entre todas las métricas extraer tanto las direcciones IP como los puertos asociados a la conexión. Dicha información está presente tanto en los mensajes de señalización como en los de RTP, por lo tanto, nos permite relacionar aquellos mensajes que pertenezcan a la misma llamada. Una vez extraída la información que permite relacionar los mensajes de distintos flujos, se deberá crear un nuevo flujo con la información combinada de aquellos que pertenezcan a la misma comunicación, y configurar un productor que envíe los datos de dicho flujo a Kafka. Para almacenar este nuevo tipo de flujo con las métricas combinadas se ha creado un nuevo *topic* en Kafka llamado `CDR_stats`, cuyas propiedades son las mismas que las del *topic* `RTP_stats`. En la “Figura 20: Diagrama conceptual del enriquecimiento de los CDR” se muestra un esquema del proceso mencionado anteriormente.

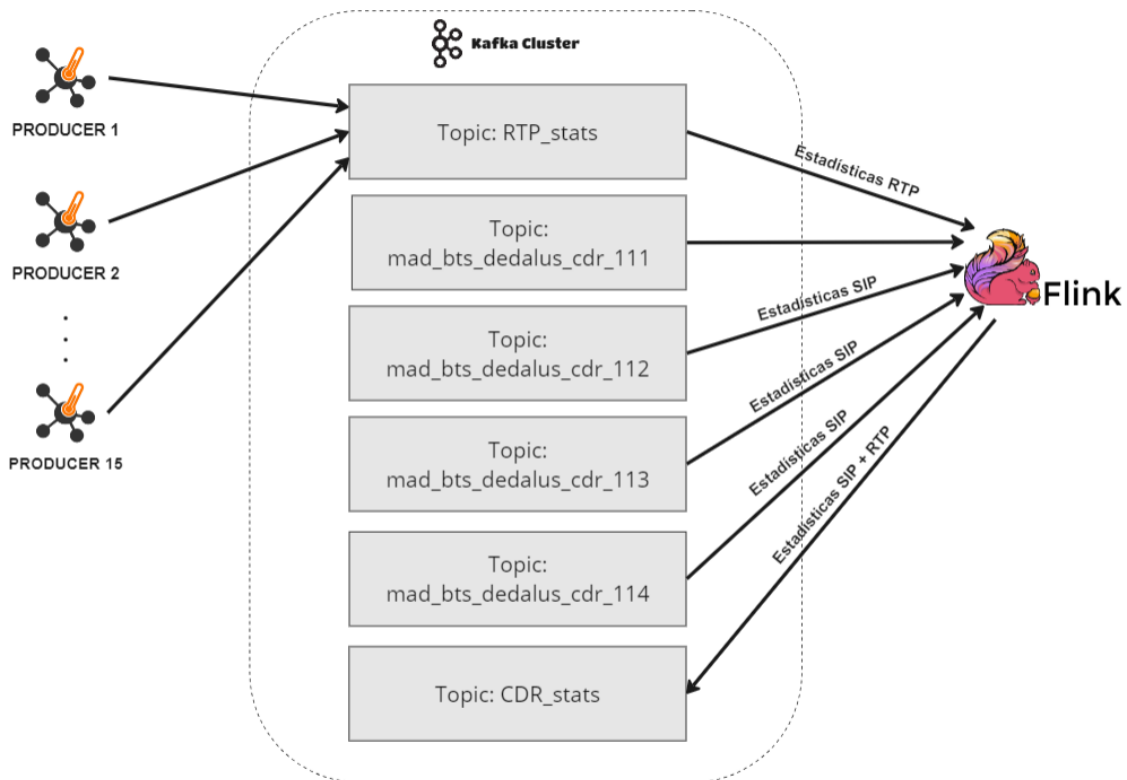


Figura 20: Diagrama conceptual del enriquecimiento de los CDR

La información almacenada en los *topic* `mad_bts_dedalus_cdr` está escrita en un formato distinto al de las métricas del *topic* `RTP_stats`. En este caso tenemos el valor de cada una de las métricas separadas por comas, tal y cómo se ve en la “Figura 21: Ejemplo de métricas almacenadas en Kafka para los *topic* `mad_bts_dedalus_cdr`”.

Key	Value	Headers
	<pre>"inbound", "SST12FRA_eWQGVmAEVjFeC31kD15uBCFXVxd3ZhheZABPXw--" , "B2B.10446.6560154.1690386857", "ANSWER", "+491636162367" , "4917601151397", "79.170.64.155", 5060, "2023-07-26 15:54:18" , "2023-07-26 15:54:23", "2023-07-26 15:54:42", 24, 19 , NORMAL_CLEARING, 5131, 6679, "4917601151397", 4, "491636162367", 4 , "4917601151397", 4, 4, "2023-07-26 15:54:20", 8, "79.170.64.151" , 5060, "internal", "149.81.144.34", 57900, "79.170.65.22", 45544 , "", "", "send_bye", 105994, "T", 0, "1.2.3.4:5060", "", " +4917601151397:84.246.115.72:200;", 14545, s, 1, "84.246.115.94" , "R", "8-0-18-96", "false", "", "", "", "", "26344", 19206, "", "", 0 , 1, "", "", "", "2", "", "", "", "", "pai", "", "", "1", "", "", "" , "", "", "", "", "149.81.144.34", "57900"</pre>	

Timestamp

26/7/2023, 17:54:48

Timestamp type: CREATE_TIME

Key Serde

Size: 0 Bytes

Value Serde

Fallback

Size: 672 Bytes

Figura 21: Ejemplo de métricas almacenadas en Kafka para los *topic* `mad_bts_dedalus_cdr`

Debido a que vamos a tener dos tipos de flujos claramente diferenciados, que van a necesitar un procesamiento diferente de su información, en el programa desarrollado se

hacen dos configuraciones distintas para el consumidor de Kafka, con el objetivo de que cada tipo de flujo se guarde en una variable diferente para su posterior procesamiento.

La información con la que se ha configurado el consumidor de Kafka es la siguiente:

- Listado de los *broker* del clúster de Kafka con sus puertos asociados.
- Listado de *topic* de los que extraer la información
- Estrategia de lectura a la hora de arrancar el programa: De las 3 posibilidades que se presentan en el Capítulo “2.1: Herramientas” se ha elegido que al ejecutarse el programa se empiece a leer a partir del último mensaje contenido en los *topic*. Con esta estrategia es posible que si se para el clúster de Flink, se pierda la información de las llamadas que llegan hasta el momento que se vuelve a arrancar, pero es necesario seguir esta estrategia ya que buscamos que los mensajes con métricas de RTP y de señalización sean recibidos aproximadamente de manera simultánea. De seguir otra estrategia para comenzar a leer mensajes, podría darse el caso de que se lean mensajes de los *topic* de una manera desincronizada, por lo que no podríamos relacionar los mensajes de distintos flujos.
- Nombre del grupo de consumidor: Pensando en una posible ejecución del programa de manera simultánea por varios *task manager*, se define un grupo de consumidores para que las diferentes máquinas se repartan las particiones de los *topic* a la hora de recibir los mensajes con las métricas y poder trabajar de manera concurrente.
- Deserializador: Para poder trabajar con los mensajes éstos se deben deserializar para pasar de bytes a una variable de tipo objeto. En este caso se indica que los mensajes se deserialicen como un *String*, es decir, cada mensaje se convierte en un *String* con todos los datos contenidos en él.

La configuración de ambos consumidores es idéntica a diferencia del listado de *topic*, donde en un caso indicamos únicamente RTP_stats, y en el otro indicamos los 4 *topic* mad_bts_dedalus_cdr. Posteriormente cada tipo de flujo se guarda en una variable de tipo *DataStream* diferente, específica para trabajar flujos continuados de datos.

A continuación, debemos transformar los datos para extraer de ellos las direcciones IP fuente y destino, y el puerto fuente de cada mensaje que se recibe para relacionar aquellos coincidentes. El puerto destino no se utiliza ya que las métricas de señalización calculadas por la empresa no lo tienen en cuenta, ya que estas se extraen antes de que sea conocido por las máquinas de su red. A pesar de ello, tan solo con las direcciones IP y el puerto fuente de la comunicación es suficiente para que si tenemos dos mensajes cuyos valores coinciden, asegurar que pertenecen a la misma comunicación, ya que la probabilidad de que en una franja de tiempo limitada se repitan estos valores es ínfima.

La transformación se lleva a cabo con el operador de Flink *flatMap*, que recibe como entrada un flujo cuyos datos son de un tipo determinado (*String*, entero...) y da como resultado otro flujo pudiendo modificar tanto el valor de los datos recibidos como el tipo de estos. Para el caso del flujo de mensajes recibidos del *topic* RTP_stats, el operador *flatMap* implementa una lógica que transforma cada mensaje entrante, que cuenta con un determinado valor de tipo *String*. Dicho valor son todas las métricas calculadas, que se muestran en la “Figura 19: Ejemplo de métricas almacenadas en

Kafka para el topic RTP_stats” convertidas en un *string*. A partir de la entrada, el operador implementa una función que convierte el flujo entrante en un flujo de tipo tupla, es decir cada mensaje estará formado por una clave y un valor. Como clave se ponen las dos direcciones IP y el puerto fuente, extrayéndolas del valor que tenemos como entrada. Como el objetivo es juntar el valor de los dos tipos de flujos (SIP y RTP) cuando sus claves coincidan debemos unificar el formato de dicho valor. Por lo tanto, para este nuevo flujo resultante, el nuevo valor de los mensajes con métricas de RTP será únicamente la medida calculada para cada métrica, separada por comas, es decir, se elimina el nombre de la métrica que aparecía en los mensajes del *topic* RTP_stats. Este proceso se muestra en la “Figura 22: Operación flatMap para el flujo de mensajes del topic RTP_stats”.

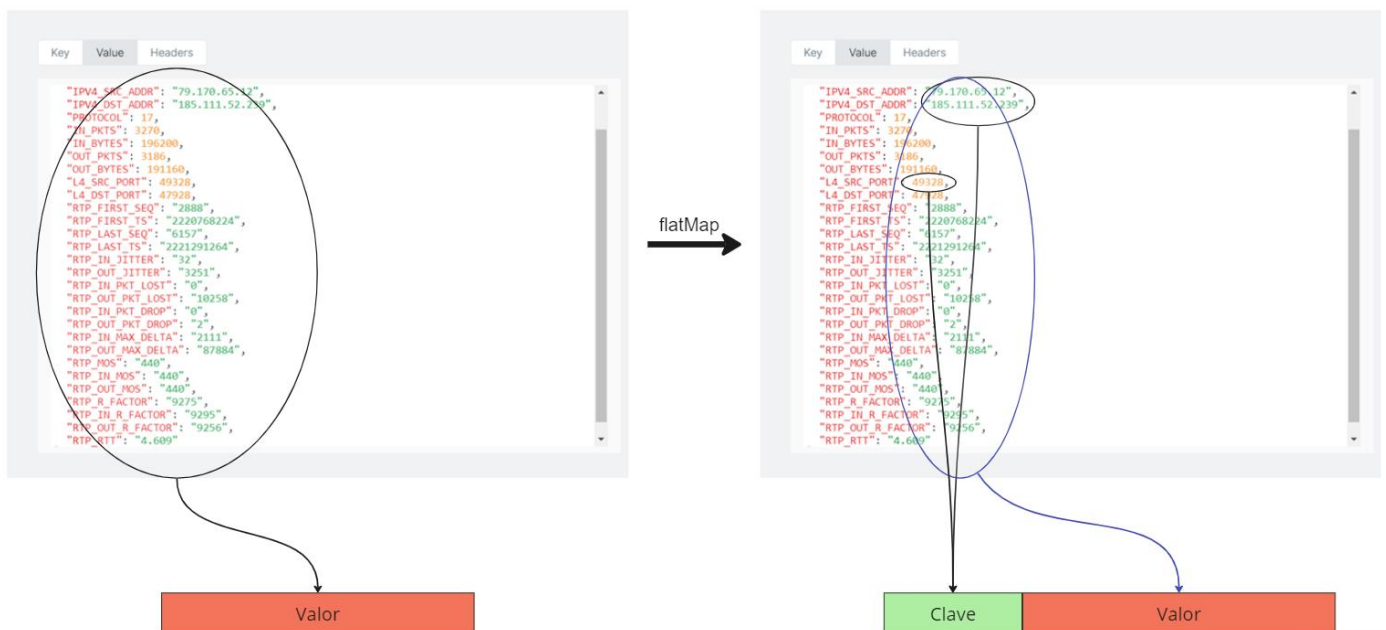


Figura 22: Operación flatMap para el flujo de mensajes del topic RTP_stats

Para el flujo de datos leídos de los *topic* `mad_bts_dedalus_cdr`, la lógica que implementa el `flatMap` convierte los mensajes del flujo entrante en una tupla clave-valor, dónde la clave está formada por las direcciones IP y el puerto fuente y el valor es el valor íntegro de los mensajes de entrada como se muestra en la “Figura 23: Operación flatMap para el flujo de mensajes de los topic `mad_bts_dedalus_cdr`”.

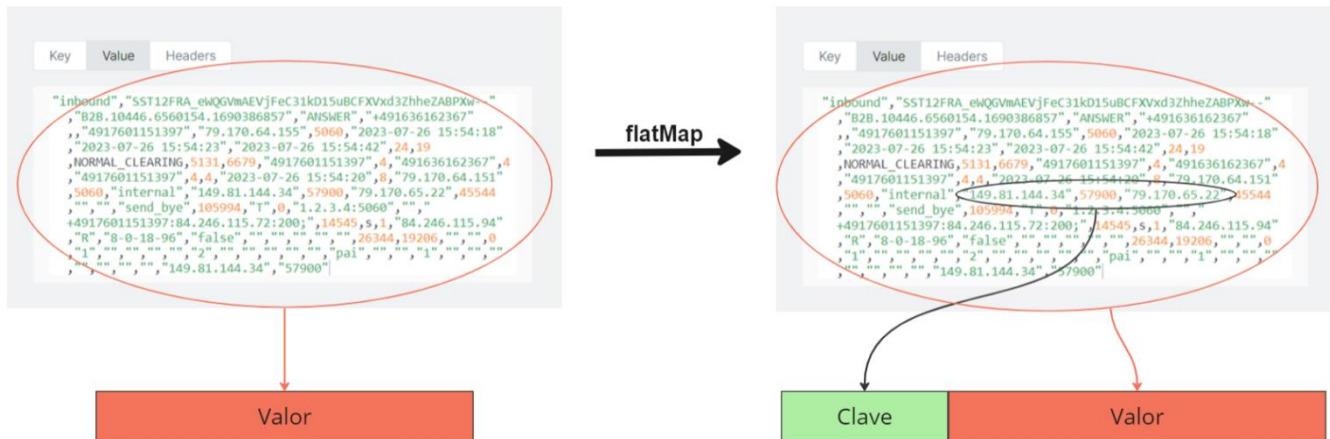


Figura 23: Operación flatMap para el flujo de mensajes de los topic mad_bts_dedalus_cdr

En el caso de que esta implementación se ejecutase en un sistema con más de *un task manager*, donde cada uno de ellos reciba una parte del total de mensajes que se escriben sobre los *topic*, podría darse la situación de que un *task manager* tuviera las métricas obtenidas del tráfico RTP de una llamada telefónica, y las métricas de señalización de esa misma llamada telefónica hayan sido leídas por otro *task manager* diferente. Ante esa situación, no podríamos generar un mensaje conjunto con ambas métricas, ya que esa operación se lleva a cabo independientemente en cada uno de los *task manager*. Puesto que la implementación está pensada para poder ser ejecutada en paralelo, garantizando así la escalabilidad, es necesario aplicarles a los dos flujos el operador `keyBy`. Al aplicar este operador, se produce un reordenamiento interno de la información entre los distintos *task manager* del clúster, en función de la clave de los mensajes. De esta forma se garantiza que todos los mensajes cuya clave coincida vayan a parar al mismo *task manager*. La “Figura 24: Diagrama conceptual del operador `keyBy`” muestra el funcionamiento descrito.

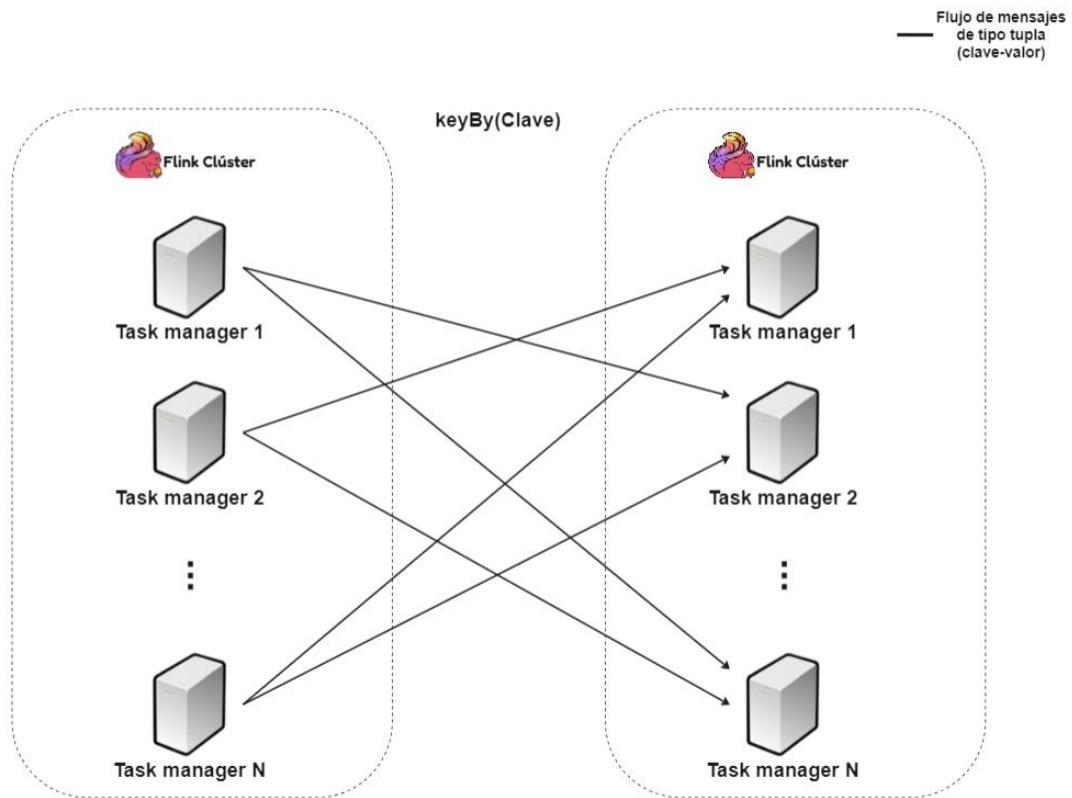


Figura 24: Diagrama conceptual del operador keyBy

El siguiente paso consiste en juntar los dos flujos de métricas en uno solo cuando sus claves coincidan. Para ello se usa el operador join, que toma como entrada dos flujos diferentes, define una ventana de tiempo, en este caso de 30 segundos, y compara las claves de todos los mensajes que llegan en esos 30 segundos. Cuando dos mensajes tienen la misma clave se ejecuta una función que genera un nuevo flujo cuyos mensajes tan sólo tendrán un valor que resultará de la concatenación de los valores de los mensajes entrantes, como se muestra en la “Figura 25: Diagrama conceptual del operador join”.



Figura 25: Diagrama conceptual del operador join

El último paso consiste en enviar este nuevo flujo de mensajes al clúster de Kafka, para que puedan ser leídos siempre que otras aplicaciones lo necesiten. Las aplicaciones que van a utilizar este tipo de datos en el entorno de la empresa están preparadas para trabajar con mensajes serializados en formato AVRO. Este formato es el más utilizado para la serialización de *streaming* de datos. AVRO hace uso de *schemas* y la definición de sus datos está en formato JSON, lo cual facilita su lectura.

Para poder serializar los mensajes en formato AVRO ha sido necesario la elaboración de un *schema* donde se ha definido cada uno de los campos que se exponen en los mensajes de los CDR, así como el tipo de variable correspondiente a dicho campo. El *schema* desarrollado se muestra detalladamente en el “Anexo 6: Schema de los CDR”.

Una vez creado el *schema*, a partir de él se genera una nueva clase de java, que tendrá el nombre CDR. Para saber cómo generar una clase de java a partir de un Schema AVRO consultar el “Anexo 7: Generar una clase de java a partir de un schema”.

A continuación, se vuelve a hacer uso del operador de Flink flatMap, con el objetivo de preparar nuestros datos para ser serializados siguiendo el *schema* desarrollado. Para ello se elabora una función que convierte el flujo de mensajes obtenido tras la operación join, que era de tipo String, con el valor de cada una de las métricas separados por comas, en un nuevo flujo de tipo CDR donde asignamos el valor de cada una de estas métricas a su campo asociado de la clase creada anteriormente.

Una vez hecho el cambio se ha configurado un productor de Kafka que envía los CDR a Kafka serializados en formato AVRO. Para ello debemos indicar:

- Lista de broker y puertos asociados del clúster de Kafka.
- Listado de topic al que enviar los datos: En este caso “CDR_stats”
- Serializador: Serializador AVRO
- URL del Registro de Esquemas: Para esta última parte también se ha utilizado la funcionalidad de Kafka del *Schema Registry*, que pasará a explicarse a continuación. En este caso la URL es “https://schema-registry.mad-k8s.sonoc.io”

El registro de esquemas o *schema registry* consiste en un proceso que se lleva a cabo fuera del clúster de Kafka, y consiste en un servidor dónde se almacenan los esquemas asociados a aquellos *topic* del clúster de Kafka donde la información se guarda en formato AVRO. Su función es la de validar el formato de los datos que se envían al clúster, así como poder trabajar con esquemas que varíen con el tiempo.

El funcionamiento básico consiste en que el productor, antes de enviar los datos a Kafka manda el esquema que siguen esos datos al registro de esquemas. Éste comprueba en su base de datos si tiene ese mismo esquema asociado al *topic* al que se quiere mandar la información, y si es así, devuelve al productor una confirmación para que éste mande los datos a Kafka. En la “Figura 26: Diagrama de funcionamiento del Schema Registry” se muestra este proceso.

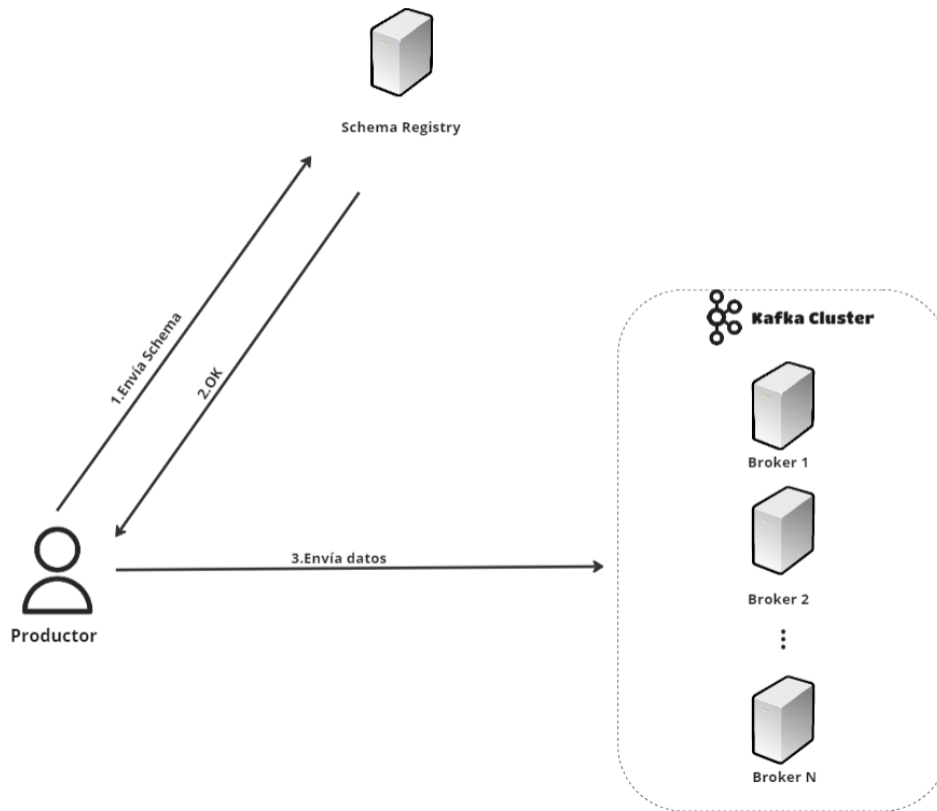


Figura 26: Diagrama de funcionamiento del Schema Registry

Para más información acerca del código que implementa la lógica de procesamiento descrita ver el “Anexo 8: Código que implementa la lógica de procesamiento para el enriquecimiento de los CDR”.

Capítulo 5: Pruebas de funcionamiento.

Durante el desarrollo del sistema y una vez terminado el mismo, se han llevado a cabo una serie de pruebas y análisis con el objetivo de comprobar y demostrar que el funcionamiento del sistema es tal y cómo se describe en el “Capítulo 4: Descripción del sistema”.

Para comprobar lo relativo al productor nProbe de Kafka se ha capturado con Wireshark en el enlace que une el servidor disponible con la tarjeta de red y el software nProbe con la capa de acceso de la topología que se muestra en la “Figura 7: Escenario previo”. La “Figura 27: Captura de un mensaje enviado a Kafka” nos muestra cada una de las etapas en la comunicación entre el servidor (productor) y el clúster de Kafka tal y cómo muestra el esquema representado en la “Figura 17: Intercambio de mensajes de un productor con los broker de Kafka en el sistema desarrollado”. Éstas corresponden a la negociación de versiones, la obtención de metadatos y posteriormente el envío de información a los *broker*, así como la recepción de las confirmaciones.

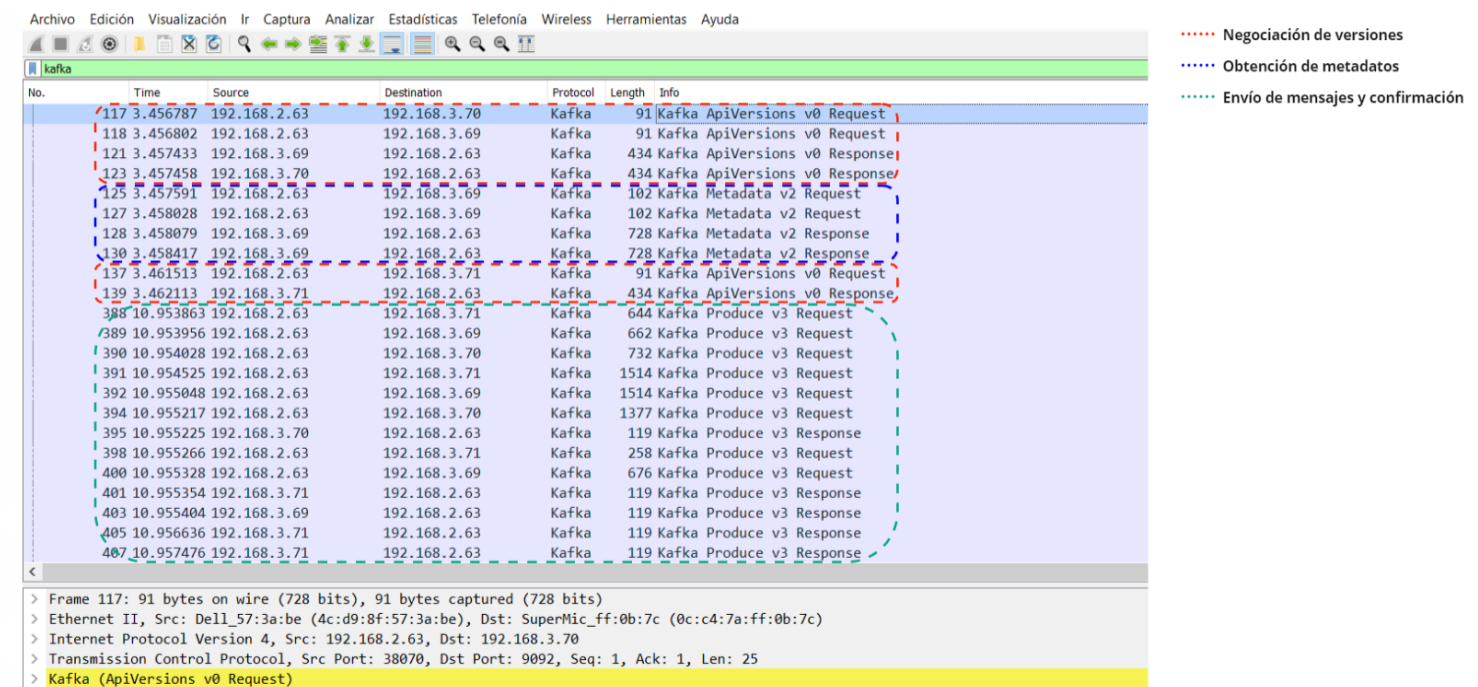


Figura 27: Captura de un mensaje enviado a Kafka

Además, si desplegamos uno de los mensajes que manda el servidor a uno de los *broker* para así analizar su contenido, vemos cierta información importante para que el *broker* que lo reciba sepa qué hacer con el mensaje. Podemos ver que se indica la estrategia de gestión de los ACK, el nombre del *topic* y la partición sobre la que queremos escribir. Por último, se manda un lote con varios registros que contienen las métricas que se desean almacenar. La “Figura 28: Información contenida en un mensaje enviado por el productor” muestra toda esta información.

kafka						
No.	Time	Source	Destination	Protocol	Length	Info
453	11.954833	192.168.2.63	192.168.3.71	Kafka	788	Kafka Produce v3 Request
454	11.954845	192.168.2.63	192.168.3.70	Kafka	1147	Kafka Produce v3 Request
455	11.955129	192.168.2.63	192.168.3.70	Kafka	1514	Kafka Produce v3 Request
456	11.955199	192.168.2.63	192.168.3.70	Kafka	1514	Kafka Produce v3 Request
457	11.955545	192.168.2.63	192.168.3.69	Kafka	1514	Kafka Produce v3 Request

> Frame 455: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)
 > Ethernet II, Src: Dell_57:3a:be (4c:d9:8f:57:3a:be), Dst: SuperMic_ff:0b:7c (0c:c4:7a:ff:0b:7c)
 > Internet Protocol Version 4, Src: 192.168.2.63, Dst: 192.168.3.70
 > Transmission Control Protocol, Src Port: 38070, Dst Port: 9092, Seq: 3084, Ack: 528, Len: 1448
 > Kafka (Produce v3 Request)
 Length: 1211
 API Key: Produce (0)
 API Version: 3
 Correlation ID: 6
 Client ID: rdkafka
 Transactional ID: [Null]
 Required Acks: Leader (1)
 Timeout: 5000
 Topic (Name=RTP_stats)
 Topic Name: RTP_stats
 Partition (ID=9)
 Partition ID: 9
 Message Set
 Record Batch

Figura 28: Información contenida en un mensaje enviado por el productor

Si analizamos la información contenida en el lote de mensajes, podemos ver que aparece información referente al tipo de compresión, el número de mensajes o *record* que contiene el lote y el ratio de compresión. Además, en cada uno de los *record* está contenida la información de las métricas calculadas para una llamada telefónica.

> Message Set
 > Record Batch
 Offset: 0
 Message Size: 1147
 Leader Epoch: 0
 Magic Byte: 2
 CRC32: 0x0252be9c
 = Compression Codec: Snappy (2)
 = Timestamp Type: CreateTime (0)
 = Transactional: Non-transactional (0)
 = Control Batch: Data batch (0)
 Last Offset Delta: 9
 First Timestamp: Jan 1, 1970 00:06:32.-664311326 UTC
 Last Timestamp: Jan 1, 1970 00:06:32.-664310326 UTC
 Producer ID: -1
 Producer Epoch: 0
 Base Sequence: -1
 Size: 10
 Compression Reduction (compressed/uncompressed): 0.192930
 Record
 Record Attributes (reserved): 0
 Timestamp: Jun 20, 2023 10:43:56.002000000 UTC
 Offset: 0
 Key: <NULL>
 Value [truncated]: "{"IPV4_SRC_ADDR":"116.202.168.53","IPV4_DST_ADDR":"79.170.65.29","IN_PKTS":32,"IN_BYTES":6400,"
 Headers
 > Record
 > Record
 > Record
 > Record
 > Record
 > Record
 > Record
 > Record

Figura 29: Información contenida en el lote

Con el objetivo de comprobar el funcionamiento del programa elaborado para ser ejecutado en un entorno de apache Flink, éste se ha ejecutado en un servidor de Flink *standalone* utilizado para hacer pruebas. El servidor de pruebas, mediante el comando `start-cluster` genera un task manager y un job manager. Éste se ubica en la topología según la “Figura 30: Topología de red con el servidor de pruebas añadido”.

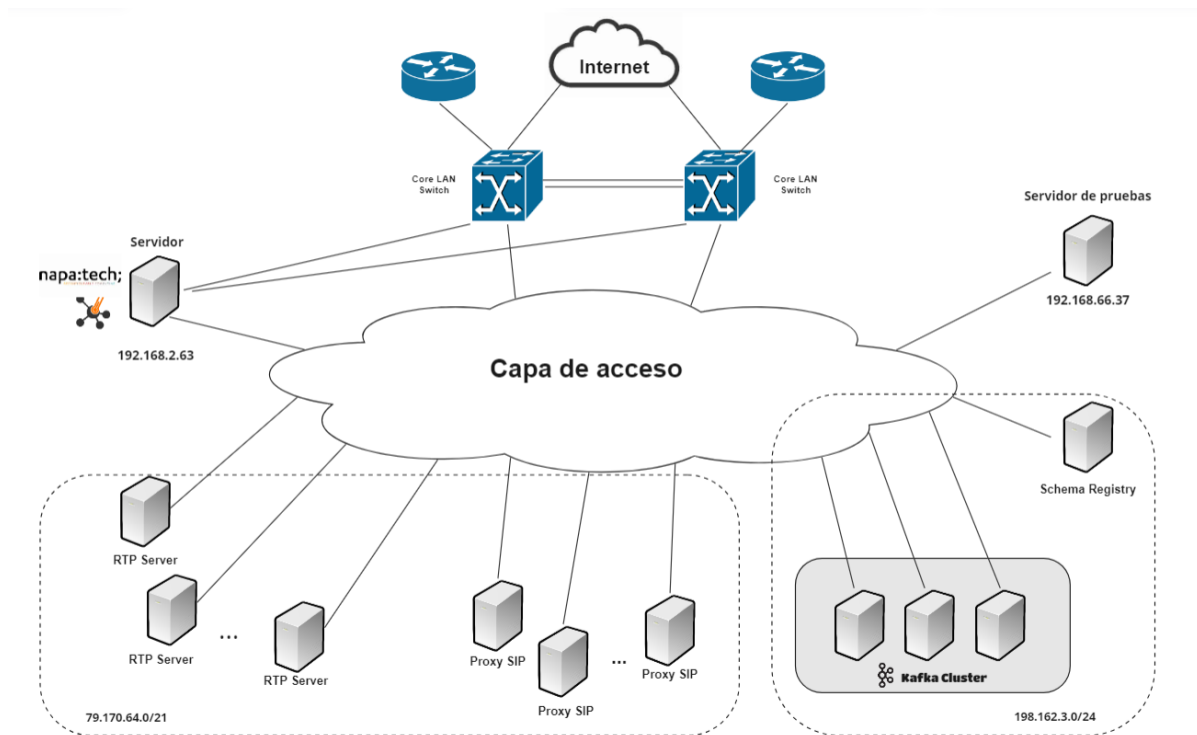


Figura 30: Topología de red con el servidor de pruebas añadido

Para la puesta en ejecución del programa hemos tenido que convertirlo en un archivo ejecutable .jar, y posteriormente transferir dicho archivo al servidor de pruebas para ser ejecutado por el *task manager*. Ver el “Anexo 9: Comandos básicos para Apache Flink” para ampliar la información al respecto.

Una vez puesto en ejecución el programa podemos ver en la interfaz gráfica de Kafka, en el *topic* CDR_stats que se están generando los mensajes con los CDR enriquecidos, juntando la información SIP con la información RTP, que se muestran en el “Anexo 10: Resultado programa de procesamiento de Flink”.

También se puede analizar el tráfico que transcurre por el enlace que une el servidor de pruebas con la capa de acceso de la red. Para ello se ha realizado una captura mediante `tcpdump` en dicho enlace y se ha analizado con Wireshark.

A nivel esquemático, la “Figura 31: Diagrama conceptual del enriquecimiento de CDR” muestra los pasos que siguen los datos durante su procesamiento. Constantemente el servidor de pruebas solicita a los *broker* de Kafka que le envíen los mensajes con métricas contenidas. Conforme van llegando, éstos se van procesando, y cada 30 segundos (periodo que dura la ventana definida para encontrar dos mensajes cuyas claves coincidan), el servidor envía de nuevo a los *broker* de Kafka los nuevos CDR

generados, previamente comprobando si su formato es correcto consultando al *schema registry*.

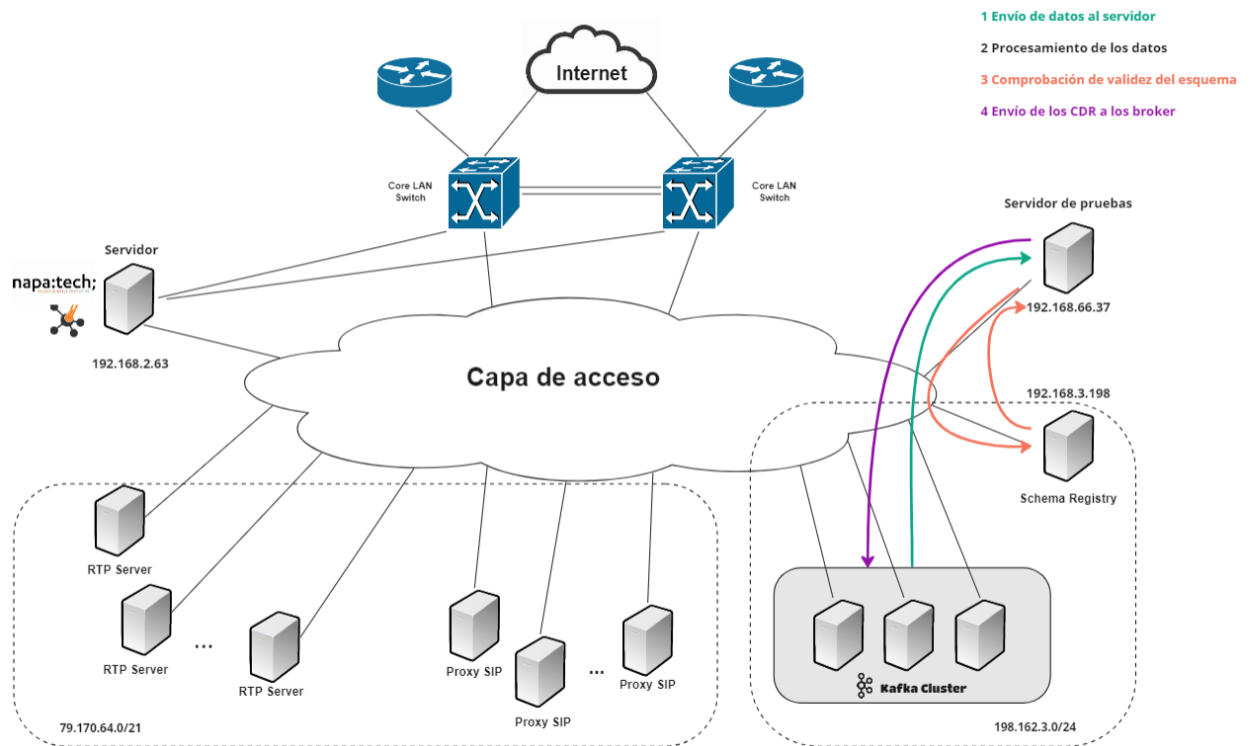


Figura 31: Diagrama conceptual del enriquecimiento de CDR

De esta manera en la captura de Wireshark, una vez se realizan las fases de configuración del productor y consumidor de Kafka (negociación de versiones, obtención de metadatos...), se empiezan a ver los mensajes dónde el servidor de prueba solicita los datos al clúster de Kafka (“Figura 32: Captura de la solicitud de mensajes por parte del servidor de Flink”):

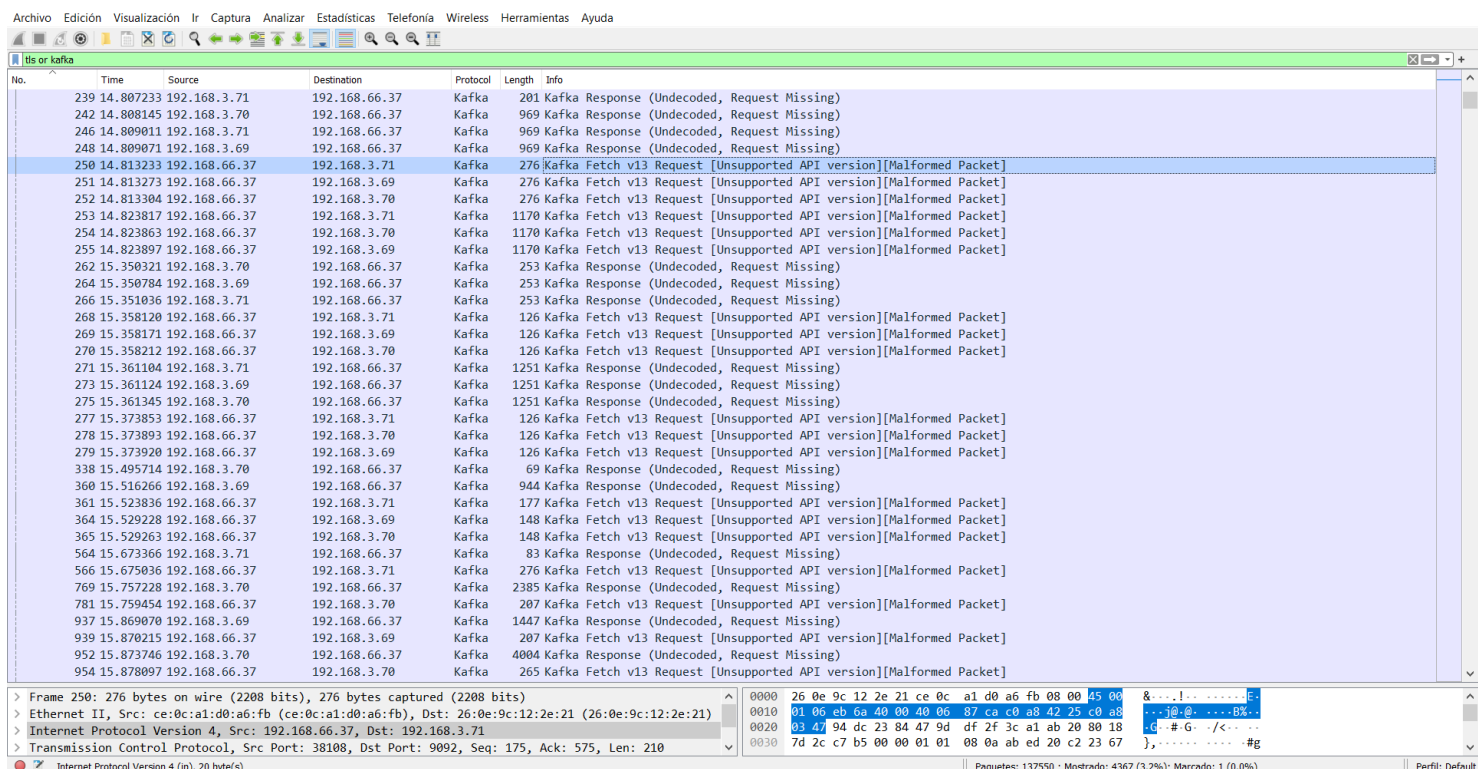


Figura 32: Captura de la solicitud de mensajes por parte del servidor de Flink

En la “Figura 33: Captura del envío de los CDR por parte del servidor de Flink” se ve como transcurridos aproximadamente 30 segundos del inicio de la recepción de datos se realiza un intercambio de mensajes (Cifrados) entre el servidor de pruebas y el registro de esquemas, consultando la validez del esquema de los datos. Una vez recibe la confirmación, envía los datos serializados en AVRO al clúster de Kafka.

Archivo Edición Visualización Ir Captura Analizar Estadísticas Telefonía Wireless Herramientas Ayuda						
tis or kafka						
No.	Time	Source	Destination	Protocol	Length	Info
61504	57.653350	192.168.3.71	192.168.66.37	Kafka	1072	Kafka Response (Undecoded, Request Missing)
61507	57.657208	192.168.66.37	192.168.3.71	Kafka	148	Kafka Fetch v13 Request [Unsupported API version]
61752	57.778852	192.168.3.70	192.168.66.37	Kafka	3227	Kafka Response (Undecoded, Request Missing)
61777	57.772023	192.168.66.37	192.168.3.70	Kafka	207	Kafka Fetch v13 Request [Unsupported API version][Malformed Packet]
61784	57.785522	192.168.66.37	192.168.3.198	TLSv1	442	Client Hello
61886	57.826794	192.168.3.198	192.168.66.37	TLSv1	598	[TCP Previous segment not captured] , Ignored Unknown Record
61894	57.833620	192.168.66.37	192.168.3.198	TLSv1	72	Change Cipher Spec
61982	57.847043	192.168.3.70	192.168.66.37	Kafka	5390	Kafka Response (Undecoded, Request Missing)
61990	57.848598	192.168.66.37	192.168.3.70	Kafka	265	Kafka Fetch v13 Request [Unsupported API version][Malformed Packet]
61992	57.871901	192.168.66.37	192.168.3.198	TLSv1	140	Application Data
61993	57.873620	192.168.66.37	192.168.3.198	TLSv1	437	Application Data
61996	57.874755	192.168.66.37	192.168.3.198	TLSv1	1738	Application Data
61997	57.874934	192.168.66.2	192.168.66.37	ICMP	500	Destination unreachable (Fragmentation needed)
62002	57.874985	192.168.66.2	192.168.66.37	ICMP	500	Destination unreachable (Fragmentation needed)
62229	57.947004	192.168.3.198	192.168.66.37	TLSv1	263	Application Data
62371	57.991126	192.168.3.69	192.168.66.37	Kafka	873	Kafka Response (Undecoded, Request Missing)
62378	57.995749	192.168.3.71	192.168.66.37	Kafka	123	Kafka Fetch v13 Response [Unsupported API version] [Unknown 12818] [Malformed Packet]
62399	58.012697	192.168.66.37	192.168.3.71	Kafka	116	Kafka ApiVersions v3 Request
62400	58.024050	192.168.3.71	192.168.66.37	Kafka	87	Kafka Response (Undecoded, Request Missing)
62401	58.024332	192.168.66.37	192.168.3.71	Kafka	126	Kafka Fetch v13 Request [Unsupported API version][Malformed Packet]
62402	58.024925	192.168.3.69	192.168.66.37	Kafka	87	Kafka Response (Undecoded, Request Missing)
62403	58.025021	192.168.66.37	192.168.3.69	Kafka	126	Kafka Fetch v13 Request [Unsupported API version][Malformed Packet]
62404	58.025500	192.168.3.70	192.168.66.37	Kafka	87	Kafka Response (Undecoded, Request Missing)
62405	58.025577	192.168.66.37	192.168.3.70	Kafka	126	Kafka Fetch v13 Request [Unsupported API version][Malformed Packet]
62410	58.049902	192.168.3.71	192.168.66.37	Kafka	505	Kafka ApiVersions v3 Response
62412	58.050283	192.168.66.37	192.168.3.71	Kafka	122	Kafka Metadata v12 Request [Unsupported API version]
62416	58.077219	192.168.3.70	192.168.66.37	Kafka	317	Kafka Response (Undecoded, Request Missing)
62418	58.087467	192.168.3.71	192.168.66.37	Kafka	741	Kafka Metadata v12 Response [Unsupported API version] [Unknown 26441] [Unknown 256] [Unknown 256] [Invalid Message] [Invalid Message]
62425	58.106268	192.168.66.37	192.168.3.71	Kafka	836	Kafka Produce v9 Request [Unsupported API version][Malformed Packet]
62427	58.106480	192.168.66.37	192.168.3.71	Kafka	529	Kafka Produce v9 Request [Unsupported API version][Malformed Packet]
62428	58.106626	192.168.66.37	192.168.3.71	Kafka	800	Kafka Produce v9 Request [Unsupported API version][Malformed Packet]
62429	58.106826	192.168.66.37	192.168.3.71	Kafka	798	Kafka Produce v9 Request [Unsupported API version][Malformed Packet]
62440	58.141210	192.168.66.37	192.168.3.70	Kafka	116	Kafka ApiVersions v3 Request
62445	58.143906	192.168.66.37	192.168.3.71	Kafka	573	Kafka Produce v9 Request [Unsupported API version][Malformed Packet]
62448	58.149849	192.168.3.71	192.168.66.37	Kafka	126	Kafka Response (Undecoded, Request Missing)
> Frame 250: 276 bytes on wire (2208 bits), 276 bytes captured (2208 bits) on interface 0 > Ethernet II, Src: ce:0c:a1:d0:a6:fb (ce:0c:a1:d0:a6:fb), Dst: 26:0e:9c:12:2e:21 (26:0e:9c:12:2e:21) > Internet Protocol Version 4, Src: 192.168.66.37, Dst: 192.168.3.71 > Transmission Control Protocol, Src Port: 38108, Dst Port: 9092, Seq: 175, Ack: 575, Len: 210						
0000 26 0e 9c 12 2e 21 ce 0c a1 d0 a6 fb 08 00 45 00 8...l...E 0010 01 06 eb 6a 40 00 00 05 87 ca c0 a8 42 25 c0 a8 ...j@...E%.. 0020 03 47 94 dc 23 84 47 9d df 2f 3c a1 ab 20 80 18 -G-#G- /<... 0030 7d 2c c7 b5 00 00 01 01 08 0a ab ed 20 c2 23 67 },.....-#g						
Paquetes: 137550 • Mostrado: 4367 (3.2%) • Marcado: 1 (0.0%) Perfil: Default						

Figura 33: Captura del envío de los CDR por parte del servidor de Flink

Gracias a las pruebas realizadas se ha conseguido comprobar el funcionamiento de la lógica de procesamiento que genera las CDR actualizadas, así como dotar de un punto de vista práctico del funcionamiento del sistema explicado anteriormente.

Capítulo 6: Conclusión y líneas futuras

6.1. Conclusión

En el presente Trabajo de Fin de Grado se ha diseñado un sistema capaz de calcular los parámetros más relevantes respecto a la calidad de la voz en una llamada telefónica en un entorno masivo de telefonía IP. El objetivo consiste en dotar a la plataforma desarrollada por la empresa SONOC de una mayor caracterización de las llamadas, permitiendo así a sus clientes tener una información más completa a la hora de elegir a sus proveedores.

Para desarrollar el sistema se ha redirigido todo el tráfico que transcurre entre la red de la empresa de la sede de Madrid e internet, con el objetivo de distinguir el tráfico RTP asociado a las comunicaciones de voz de la llamada telefónica. Para ello se ha hecho uso de la tarjeta de red NT200A02 SmartNIC, que recibe todo ese tráfico por dos de sus puertos de entrada. En esta primera fase se ha llevado a cabo:

- La configuración del software de la tarjeta con el objetivo de optimizar al máximo los recursos disponibles.
- La elaboración de un filtro que descarte todo el tráfico no necesario para lo que sigue del proyecto, quedándonos únicamente con el tráfico RTP.

Posteriormente se ha utilizado el software nProbe, que se ha configurado para ser capaz de:

- Leer los paquetes RTP filtrados en cada una de las interfaces de salida del filtro.
- Ser capaz de identificar la secuencia completa de paquetes que pertenecen a una comunicación, desde que ésta se inicia, hasta que termina.
- Obtener las métricas que se han considerado más relevantes en términos de calidad de voz a partir del conjunto de paquetes RTP de una llamada.
- Configurar un productor que envía los datos obtenidos a un clúster de Kafka.

Con el objetivo de agregar estos nuevos datos calculados a los que ya formaban parte de la plataforma (obtenidos a partir de la señalización) se ha realizado un programa pensado para ejecutarse en un entorno de la plataforma Apache Flink que:

- Lee los mensajes que tienen las métricas extraídas de la señalización.
- Lee los mensajes que tienen las métricas extraídas del audio (RTP).
- Procesa los mensajes, siendo capaz de relacionar aquellos que pertenecen a la misma llamada gracias a sus parámetros comunes.
- Crea nuevos mensajes con la información combinada de aquellos que pertenecen a la misma llamada.
- Serializa los mensajes en formato AVRO y los envía a Kafka haciendo uso del *Schema Registry*

El funcionamiento del programa ha sido puesto a prueba con su ejecución en un servidor de Flink, donde se ha comprobado su correcto funcionamiento, y, en definitiva, que se han alcanzado los objetivos planteados previamente.

Durante el desarrollo del Trabajo Fin de Grado se han adquirido conocimientos acerca de los lenguajes de programación NTPL y Java, así como de herramientas como nProbe, Kafka y Apache Flink, las cuales gozan de una gran popularidad actualmente, siendo utilizadas por multitud de empresas.

Con este proyecto se ha logrado cumplir los objetivos, que satisfacen las necesidades planteadas por SONOC en la actualidad, pero que también garantiza la escalabilidad frente a un crecimiento futuro. Se ha conseguido desarrollar un sistema que será de gran utilidad a sus clientes para analizar las opciones disponibles en el mercado, pudiendo así elegir los proveedores adecuados. También es posible detectar y analizar posibles problemas en sus comunicaciones, para poder darles una solución.

6.2. Líneas futuras

Respecto a las posibles ampliaciones del proyecto, el primer paso a seguir debe ser la puesta en producción del sistema. Para ello se plantea la construcción de un clúster de Flink en Kubernetes, donde a diferencia del servidor de pruebas que hemos utilizado, éste podría contar con más de un *task manager* que trabaje de forma paralela. Una vez construido, el programa desarrollado puede trabajar de igual manera al ser ejecutado por esas máquinas. Con esto también se garantizaría la alta disponibilidad del sistema, además del auto escalado horizontal, es decir, se puede comenzar con un *task manager* y en el momento que se supere cierto nivel de uso de CPU, se levante otro *task manager*.

Otra posible mejora es el envío de las estadísticas generadas a Prometheus, una base de datos especializada en el almacenamiento de métricas. También permitiría instrumentar la aplicación de enriquecimiento de CDRs elaborada, mostrando estadísticas del rendimiento de la aplicación, como puede ser la latencia con Kafka o cuánta memoria utiliza la JVM. Esto es posible modificando el código desarrollado utilizando la librería de Java específica de Prometheus. Todas las estadísticas almacenadas en Prometheus pueden posteriormente mostrarse de manera gráfica en otras aplicaciones como Grafana.

Por último, como se ha comentado, el proyecto está diseñado para utilizarse en primera instancia por el operador de telecomunicaciones BTS (Principal cliente y empresa matriz de SONOC) utilizando los servidores de Madrid, por lo que habría que hacer ligeras adaptaciones en caso de ser utilizado por otro cliente o en otra ubicación.

Anexos

Anexo 1: Guía de instalación y configuración del software Napatech

En este anexo se muestra cómo se debe realizar la instalación (o actualización) y configuración del software que utiliza la tarjeta de red NT200A02 SmartNIC, que comprende tanto los *driver* (software que conecta el sistema operativo con el hardware de la tarjeta) y la imagen (programación de la tarjeta en su interfaz de configuración).

Para descargar el software necesitamos descargar la carpeta correspondiente a la versión que se desea desde el siguiente enlace:

<https://supportportal.napatech.com/index.php?/selfhelp/categories/Link%E2%84%A2%20Capture%20Software%20Linux/139>

En la carpeta descargada podemos encontrar un archivo ejecutable para la instalación de los *driver*, así como una carpeta con distintas versiones de imágenes, además de información adicional.

Para la instalación de los *driver* hay que situarse en el directorio donde se encuentra el archivo descargado y ejecutar el comando:

```
./package_install_3gd.sh
```

Tras la ejecución se creará una nueva carpeta llamada *napatech3*, en el directorio */opt*. En dicha carpeta se encuentran directorios donde podemos encontrar archivos de configuración, archivos binarios o documentación. Por último, se deben cargar los *driver*. Para ello hay que ejecutar el siguiente comando:

```
/opt/napatech3/bin/ntload.sh
```

Si ya hay instalada alguna versión anterior del software en nuestro sistema, al ejecutar el comando de instalación se nos preguntará si se desea actualizar los *driver*. Si seleccionamos que sí, el programa modificará únicamente los archivos que cambian de una versión a otra en el directorio *napatech3*, mientras que, si elegimos que no, se eliminará completamente el directorio y se creará uno nuevo.

En caso de actualizar los *driver*, hay que asegurarse de que éstos estén descargados antes de ejecutar el comando de instalación. Para descargar los *driver* se utiliza el comando:

```
/opt/napatech3/bin/ntunload.sh
```

Una vez instalados los *driver* el siguiente paso es la instalar la imagen de la configuración. La tarjeta cuenta con dos bancos donde podemos cargar dos imágenes distintas, pudiendo cambiar de una a otra de una manera rápida. Para instalar una imagen en uno de los dos bancos debe escribirse el comando:

```
/opt/napatech3/bin/imgctrl -p -a 0 --ntimg=<nombre del  
archivo de la imagen>.ntimg --ini sonoc
```

Para cargar una nueva imagen en un banco, los *driver* deben estar descargados, y hay que verificar que la imagen elegida de entre todas las disponibles sea compatible con la versión de los *driver* instalada y con el modelo de tarjeta de red utilizada.

Si disponemos de una imagen en cada uno de los bancos podemos cambiar de una a otra mediante los comandos:

```
/opt/napatech3/bin/imgctrl -s -a 0
```

```
/opt/napatech3/bin/imgctrl -s -a 0 -confirm
```

Además, con el siguiente comando podemos ver toda la información relativa a las imágenes:

```
/opt/napatech3/bin/imgctrl -q
```

La configuración del software se encuentra en el fichero `/opt/napatech3/config/ntservice.ini`. En dicho fichero se puede modificar el número de salidas lógicas del filtro de paquetes RTP modificando las variables `NumberWorkerThreads` y `HostBufferRx`, como se muestra en la “Figura 34: Fichero de configuración de la tarjeta de red”.

```
[System]
HostBufferRefreshIntervalAll = default      # default* - 1 - 5 - 10 - 50 - 100 - 250 - 500
LinkPropagationPortPairs =                  # [portA, portB], ...
NtplFileName =                             # String
NumberWorkerThreads = 16                   # 1 .. 100
SDRAMFillLevelWarning = 0                  # X1, X2, X3, X4
TimeSyncOsTimeReference = None             # None* - adapter-0 - adapter-1 - adapter-2 - adapter-3 -
TimestampFormat = NATIVE_UNIX              # NATIVE - NATIVE_NDIS - NATIVE_UNIX* - UNIX_NS - PCAP -
TimestampFormat = PCAP                     # NATIVE - NATIVE_NDIS - NATIVE_UNIX* - UNIX_NS - PCAP -
TimestampMethod = EOF                      # SOF - EOF*

[Logging]
LogBufferWrap = wrap                       # wrap* - nowrap
LogFileName = /var/log/napatech/Log3G_%.log # String
LogMask = 7                               # See ini-file help for information about possible values
LogToFile = true                           # true/false
LogToSystem = true                         # true/false

[Adapter0]
AdapterType = NT200A02_8X10                # NT40A01_4X1 - NT20E3_2_PTP - NT40E3_4_PTP - NT80E3_2_PTP
BondingType = Separate                     # Separate*
CancelTxOnCloseMask = 0                    # See ini-file help for information about possible values
DeduplicationWindow = 100                  # 10 .. 2000000
DisableFec = 0                             # 1 - 0* - true - false*
DisableTxRemoteFault = 0                   # 1 - 0* - true - false*
DiscardSize = 16                           # 16 .. 63
HighFrequencySampling = DISABLE             # DISABLE* - ENABLE
HostBufferHandlerAffinity = -2              # -2 .. 15
HostBufferPollInterval = default            # default* - 10 - 50 - 100 - 250 - 500 - 1000 - 10000 - 2
HostBufferRefreshIntervalRx = default       # default* - 1 - 5 - 10 - 50 - 100 - 250 - 500
HostBufferRefreshIntervalTx = default       # default* - 1 - 5 - 10 - 50 - 100 - 250 - 500
HostBufferSegmentAlignmentRx = default      #
HostBufferSegmentSizeRx = default           # default* - 1 - 2 - 4 - 1M - 2M - 4M
HostBufferSegmentSizeTx = default           # default* - 1 - 2 - 4 - 1M - 2M - 4M
HostBufferSegmentTimeOut = default          # default* - 10 - 50 - 100 - 250 - 500 - 1000 - 10000 - 2
HostBuffersRx = [16,16,-1]                 # [x1, x2, x3], ...
HostBuffersTx = [1,1024,-1]                # [x1, x2, x3], ...
IfgMode = NS                               # NS* - BYTE
KmTcamConfig = [2,4,0],[4,1,0]             # [cnt, len, dualLookup], ...
MaxFrameSize = 9018                        # 1518 .. 10000
NumaNode = -1                              # -1 .. 16
OnBoardMemorySplit = Even                  # Even* - Proportional
PacketDescriptor = NT                      # PCAP - NT*
PacketDescriptor = PCAP                     # PCAP - NT*
```

Figura 34: Fichero de configuración de la tarjeta de red

Anexo 2: Implementación del filtro mediante programación en NTPL

En este anexo se muestra el código desarrollado para el filtrado de paquetes RTP. El Código NTPL que implementa la lógica que utiliza el filtro para descartar todos aquellos paquetes que no son RTP es el siguiente:

```
Delete=All
```

```
Define SRCPort_RTP =
```

```
Filter(Data[DynOffset=DynOffUDPFrame;Offset=0;DataType=ByteStr2] ==  
(10000..65535))
```

```
Define Delete_not_RTP_SRCports = Filter
```

```
(Data[DynOffset=DynOffUDPFrame;Offset=0;DataType=ByteStr2] != 22222 AND  
Data[DynOffset=DynOffUDPFrame;Offset=0;DataType=ByteStr2] != 7772 AND  
Data[DynOffset=DynOffUDPFrame;Offset=0;DataType=ByteStr2] != 5000)
```

```
Define Delete_not_RTP_DSTports = Filter
```

```
(Data[DynOffset=DynOffUDPFrame;Offset=2;DataType=ByteStr2] != 22222 AND  
Data[DynOffset=DynOffUDPFrame;Offset=2;DataType=ByteStr2] != 7772 AND  
Data[DynOffset=DynOffUDPFrame;Offset=2;DataType=ByteStr2] != 5000)
```

```
Define EvenSrcPort =
```

```
Filter(Data[DynOffset=DynOffUDPFrame;Offset=0;DataType=ByteStr2;DataMask=[0:  
0]]==0)Define DstPort_RTP =
```

```
Filter(Data[DynOffset=DynOffUDPFrame;Offset=2;DataType=ByteStr2] ==  
(10000..65535))Define EvenDstPort =
```

```
Filter(Data[DynOffset=DynOffUDPFrame;Offset=2;DataType=ByteStr2;DataMask=[0:  
0]]==0)Define SrcAddr =
```

```
Filter(Data[DynOffset=DynOffIPv4frame;Offset=12;DataType=IPv4Addr]=={[FF.FF.F  
8.00]:[79.170.64.0]})
```

```
Define DstAddr =
```

```
Filter(Data[DynOffset=DynOffIPv4frame;Offset=16;DataType=IPv4Addr]=={[FF.FF.F  
8.00]:[79.170.64.0]})
```

```
Define RTP= Filter(((SRCPort_RTP AND SrcAddr) OR (DstPort_RTP AND DstAddr)  
OR (SRCPort_RTP AND SrcAddr AND DstPort_RTP AND DstAddr)) AND  
EvenSrcPort AND EvenDstPort AND Delete_not_RTP_DSTports AND  
Delete_not_RTP_SRCports)
```

```
HashMode[priority=0; InnerLayer4Type=TCP,UDP,SCTP] = HashInner5TupleSorted
```

```
HashMode[priority=1; InnerLayer3Type=IP] = HashInner2TupleSorted
```

```
HashMode[priority=2; Layer4Type=TCP,UDP,SCTP] = Hash5TupleSorted
```

```
HashMode[priority=3; Layer3Type=IP] = Hash2TupleSorted
```

Assign[StreamID=15] = Layer3Protocol == IP

Assign[StreamID=(0..14)] = (Layer4Protocol == UDP) AND RTP AND (CrcError == FALSE)

Anexo 3: Comunicación entre un productor y Kafka

En el presente anexo se muestra con una mayor profundidad cómo es la comunicación entre un productor y un clúster de Kafka.

Lo primero que sucede cuando se pone en marcha un productor es que éste manda un mensaje *Kafka ApiVersions request* a cada uno de los servidores de la lista. Posteriormente, los *broker* le envían un mensaje *Kafka ApiVersions reply*, donde le indican al productor las versiones de Kafka que soporta. Es decir, esta fase consiste en una negociación de las versiones soportadas por los servidores.

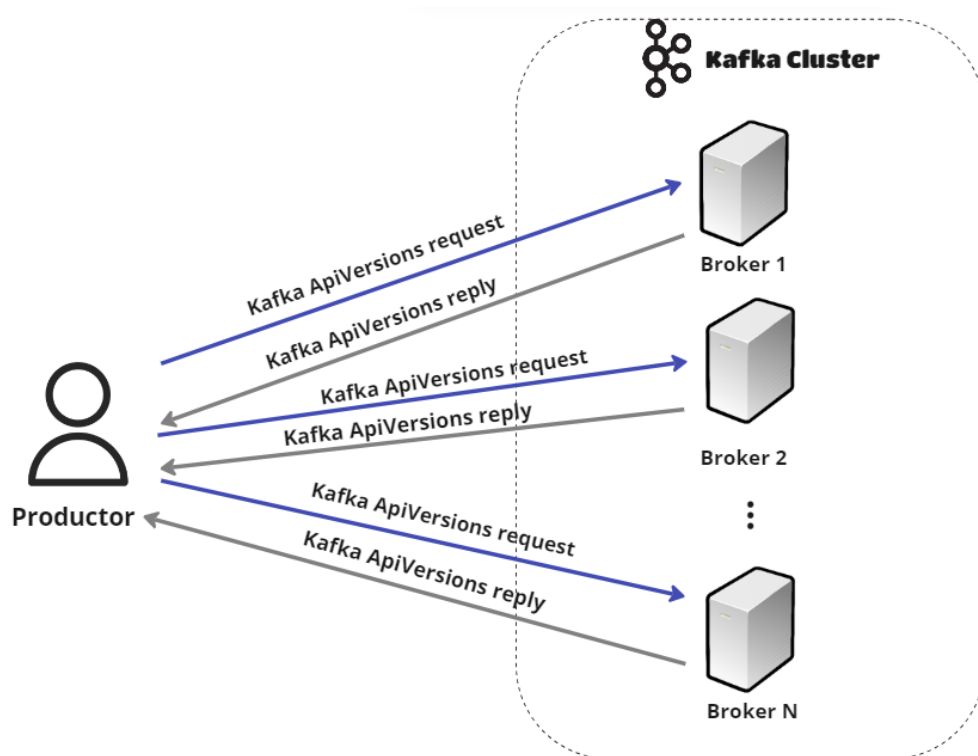


Figura 35: Diagrama conceptual negociación de versiones productor-broker

Una vez terminada la fase de negociación el productor manda un mensaje *Kafka Metadata Request* al primer *broker* de la lista que posee. El objetivo de esto es obtener toda la información relativa a un *topic*. El productor indica en el mensaje *request* el nombre de aquel *topic* del que desea obtener la información.

Los metadatos o *metadata* contienen toda la información relativa a un *topic* determinado. En esa información se incluye el número de particiones que tiene, la dirección IP del *broker* líder de cada partición, su ID o qué *broker* pasará a ser líder de una partición si el servidor que hace de líder originalmente falla.

El *broker* que recibe la solicitud de los metadatos se los manda al productor mediante un mensaje *Kafka Metadata Response*.

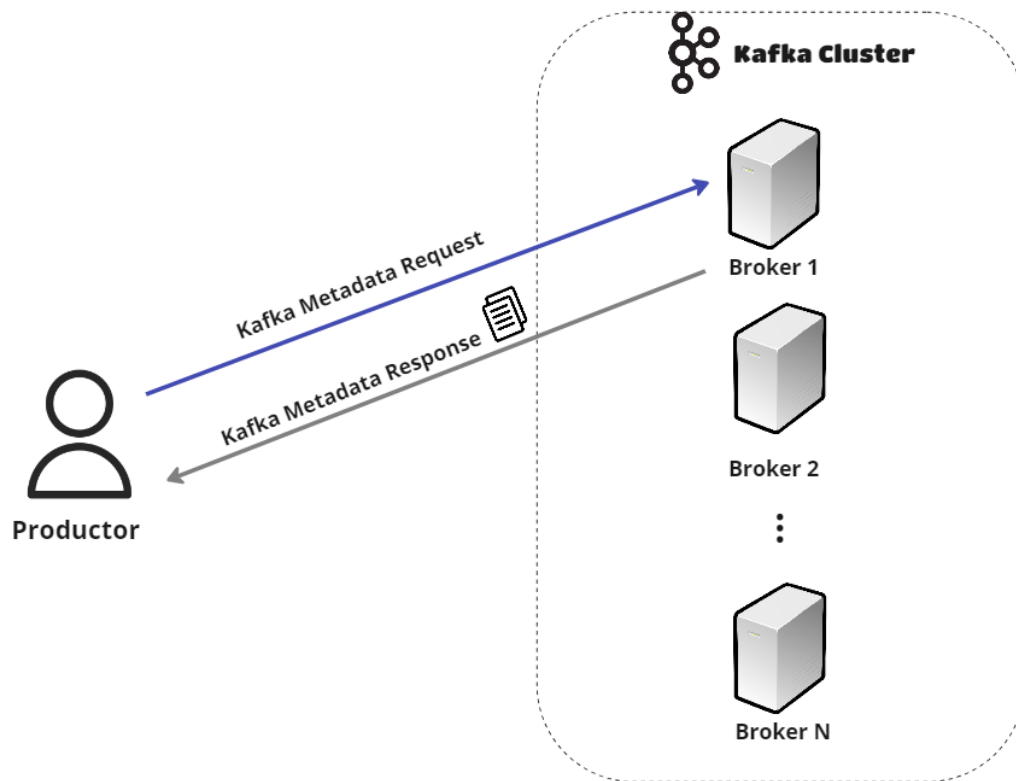


Figura 36: Diagrama conceptual de la obtención de los metadatos

Una vez que el productor recibe los metadatos, empieza a producir mensajes. Cada vez que envía datos a un *topic* de Kafka consultará en los metadatos la dirección IP del servidor líder de la partición en la que desea escribir y los mensajes los enviará directamente a dicha IP.

Cuando el productor solicita los metadatos al *broker*, si detecta que éste está caído al no recibir respuesta, le envía el *request* al siguiente *broker* de la lista. El nodo del clúster al que se le hace la solicitud es indiferente, ya que todos ellos tienen en todo momento la información de todos los componentes del clúster. Los datos se envían mediante un mensaje *Kafka Produce Request*. Además de los datos debe indicarse el *topic* y la partición en la que se quiere guardar el mensaje, debido a que un mismo *broker* puede ser líder de múltiples particiones de distintos *topic*.

Por defecto cuando un productor envía datos en un *topic* los mensajes se reparten entre las diferentes particiones en modo *Round Robin*, de esta forma el productor sabe en todo momento a qué partición desea escribir y gracias a los metadatos la dirección del servidor donde debe mandar la información. Una vez los datos llegan al *broker*, éste analiza su contenido, extrae el *topic* y partición a la que van dirigida los datos y los almacena donde corresponda asignándoles el *offset* correspondiente.

Anexo 4: Comunicación entre un consumidor y Kafka

En este anexo se cuenta más en detalle cómo se comunican los consumidores con el clúster de Kafka.

Internamente Kafka designa a uno de sus *broker* como controlador de grupo con el objetivo de gestionar todo lo relativo al grupo de consumidor al que pertenezca.

La información de qué *broker* hace de coordinador se le indica a cada consumidor en los metadatos. Una vez los consumidores tienen esta información envían un mensaje *JoinGroup Request* al coordinador de grupo para darse a conocer entre los nodos del clúster. Seguidamente el controlador de grupo le devuelve un *JoinGroup Reply* donde indica de qué particiones debe leer datos.

En última instancia, antes de empezar a recibir mensajes, el consumidor manda a Kafka un *Offsets Request* para conocer el offset de una determinada partición a partir del cual empezar a leer datos.

Según cómo configuramos el consumidor hay 3 posibilidades:

- El consumidor solicita el offset del mensaje más antiguo almacenado en la partición por la que pregunta. De esta forma se leerán todos los datos ya almacenados además de los nuevos que lleguen.
- El consumidor solicita el offset del último mensaje almacenado en el *topic*. En este caso sólo se leerán mensajes almacenados a partir de la solicitud.
- El consumidor solicita el último mensaje leído. Si se configura el consumidor de esta forma, mientras éste consume mensajes, cada cierto tiempo enviará a los *broker* líderes de las particiones de las que lee un *Commit Offset* con el objetivo de hacer conocer al servidor de Kafka líder de una partición qué mensajes se han leído. De esta manera si se apaga un consumidor, al volverse a arrancar, continúa leyendo desde el punto en el que se encontraba. En este caso se asegura que el consumidor reciba la totalidad de los mensajes que se escriben en el *topic*.

El *broker* correspondiente enviará a la dirección IP del consumidor un mensaje *Offsets Reply* notificando el *offset* a partir del que debe leer.

En este punto el consumidor ya puede empezar a recibir datos. Para ello envía constantemente mensajes *Fetch Request* al *broker* líder de la partición de la que desea recibir información indicando el *topic*, el número de partición, el número de offset y el tamaño máximo de respuesta. Posteriormente el *broker* le envía los datos al consumidor en un *Fetch Reply* con un número de mensajes inferior al tamaño indicado en el *request*.

Anexo 5: Fichero de configuración de nProbe

En este anexo se muestran las instrucciones necesarias para que el software nProbe cumpla con el funcionamiento explicado en el desarrollo del sistema. Un ejemplo de configuración (la utilizada para leer de la interfaz de salida 1) que utiliza nProbe para obtener métricas y enviarlas a Kafka es la siguiente:

```
--interface=nt:stream1
--pid-file=/var/run/nprobe-zc1-01.pid
--dump-stats=/var/log/nprobe/zc1-01_flows_stats.txt
--dump-path=/media/data1/nprobe/flows01

--kafka="mad-kafka-1.sonoc.io:9092,mad-kafka-2.sonoc.io:9092,mad-kafka-3.sonoc.io:9092;RTP_stats;none;1;snappy"
--kafka-conf="queue.buffering.max.ms=1000"
--kafka-conf="batch.num.messages=300"
#--cpu-affinity=36

--collector=none
#--debug
--lifetime-timeout=7200
--idle-timeout=10
--snaplen=1500
--aggregation=0/1/1/1/0/0/0
--all-collectors=0
--verbose=1
--dump-format=t
--vlanid-as-iface-idx=none
#--hash-size=128000
--hash-size=1024000
--flow-delay=0
--count-delay=10
--min-flow-size=0
--netflow-engine=0:0
```

```

--sample-rate=1:1:1

--as-list=/usr/share/ntopng/httpdocs/geoip/GeoIPASNum.dat

--city-list=/usr/share/ntopng/httpdocs/geoip/GeoLiteCity.dat

--flow-templ="%IPV4_SRC_ADDR %IPV4_DST_ADDR %PROTOCOL %IN_PKTS
%IN_BYTES %OUT_PKTS %OUT_BYTES %L4_SRC_PORT %L4_DST_PORT
%RTP_FIRST_SEQ %RTP_FIRST_TS %RTP_LAST_SEQ %RTP_LAST_TS
%RTP_IN_JITTER %RTP_OUT_JITTER %RTP_IN_PKT_LOST
%RTP_OUT_PKT_LOST %RTP_IN_PKT_DROP %RTP_OUT_PKT_DROP
%RTP_IN_MAX_DELTA %RTP_OUT_MAX_DELTA %RTP_MOS %RTP_IN_MOS
%RTP_OUT_MOS %RTP_R_FACTOR %RTP_IN_R_FACTOR
%RTP_OUT_R_FACTOR %RTP_RTT"

--flow-version=9

--smart-udp-frags=

--ndpi-proto-ports=/etc/nprobe/ndpi-protos.conf

--dump-bad-packets=/var/tmp/nprobe_zc1-01.pcap

```

Las estadísticas que se calculan, indicadas en la configuración, son las siguientes:

- **RTP_FIRST_SEQ:** Número de secuencia del primer paquete recibido de un flujo.
- **RTP_FIRST_TS:** Timestamp del primer paquete recibido de un flujo.
- **RTP_LAST_SEQ:** Número de secuencia del último paquete recibido de un flujo.
- **RTP_LAST_TS:** Timestamp del último paquete recibido de un flujo.
- **RTP_IN_JITTER:** *Jitter* medio medido en paquetes dirección src-> dst representado en ms*1000
- **RTP_OUT_JITTER:** *Jitter* medio medido en paquetes dirección dst-> src representado en ms*1000
- **RTP_IN_PKT_LOST:** Número de paquetes perdidos del flujo en dirección src-> dst
- **RTP_OUT_PKT_LOST:** Número de paquetes perdidos del flujo en dirección dst-> src
- **RTP_IN_PKT_DROP:** Número de paquetes descartados por el *jitter buffer* en dirección src-> dst
- **RTP_OUT_PKT_DROP:** Número de paquetes descartados por el *jitter buffer* en dirección dst-> src
- **RTP_IN_MAX_DELTA:** Diferencia máxima de tiempo entre dos paquetes consecutivos del flujo en dirección src-> dst medida en ms*100
- **RTP_OUT_MAX_DELTA:** Diferencia máxima de tiempo entre dos paquetes consecutivos del flujo en dirección dst-> src medida en ms*100

- **RTP_IN_MOS:** Valor del MOS en dirección src-> dst
- **RTP_OUT_MOS:** Valor del MOS en dirección dst-> src
- **RTP_MOS:** Valor medio del MOS en la comunicación (en ambas direcciones)
- **RTP_IN_R_FACTOR:** Valor del factor R en dirección src-> dst
- **RTP_OUT_R_FACTOR:** Valor del factor R en dirección dst-> src
- **RTP_R_FACTOR:** Valor medio del factor R en la comunicación (en ambas direcciones)
- **RTP_RTT:** Valor del *Round Trip Time* (ms)

Anexo 6: Schema de los CDR

En este anexo se muestra el esquema desarrollado, el cual indica el formato de los datos que tendrán los CDR obtenidos tras la ejecución del programa de procesamiento de datos. El esquema elaborado es el siguiente:

```
{ "namespace": "cdr.avro",
  "type": "record",
  "name": "flink",
  "fields": [
    { "name": "direction", "type": "string", "default": "" },
    { "name": "uuid", "type": "string", "default": "" },
    { "name": "otherleg", "type": "string", "default": "" },
    { "name": "endpoint_disposition", "type": "string", "default": "" },
    { "name": "orig_calling_number", "type": "string", "default": "" },
    { "name": "info_digits", "type": "string", "default": "" },
    { "name": "cld_number_std", "type": "string", "default": "" },
    { "name": "sip_network_ip", "type": "string", "default": "" },
    { "name": "sip_network_port", "type": [ "null", "int" ], "default": null },
    { "name": "start_stamp", "type": "string", "default": "" },
    { "name": "answer_stamp", "type": "string", "default": "" },
    { "name": "end_stamp", "type": "string", "default": "" },
    { "name": "duration", "type": [ "null", "int" ], "default": null },
    { "name": "billsec", "type": [ "null", "int" ], "default": null },
    { "name": "hangup_cause", "type": "string", "default": "" },
    { "name": "rg_in", "type": [ "null", "int" ], "default": null },
    { "name": "rg_out", "type": [ "null", "int" ], "default": null },
    { "name": "called_number", "type": "string", "default": "" },
    { "name": "orig_clg_noa", "type": [ "null", "int" ], "default": null },
    { "name": "ani", "type": "string", "default": "" },
    { "name": "orig_clg_noas", "type": [ "null", "int" ], "default": null },
    { "name": "orig_called_number", "type": "string", "default": "" },
    { "name": "orig_cld_noa", "type": [ "null", "int" ], "default": null },
    { "name": "orig_cld_noas", "type": [ "null", "int" ], "default": null },
    { "name": "progress_stamp", "type": "string", "default": "" },
    { "name": "codec", "type": [ "null", "int" ], "default": null },
    { "name": "sip_local_network_addr", "type": "string", "default": "" },
```

```

{"name": "sip_received_port", "type": ["null","int"], "default": null },
{"name": "sofia_profile_name", "type": "string", "default": "" },
{"name": "remote_rtp_ip", "type": "string", "default": "" },
{"name": "remote_rtp_port", "type": ["null","int"], "default": null },
{"name": "local_rtp_ip", "type": "string", "default": "" },
{"name": "local_rtp_port", "type": ["null","int"], "default": null },
{"name": "rtp_audio_in_packet_count", "type": "string", "default": "" },
{"name": "rtp_audio_out_packet_count", "type": "string", "default": "" },
{"name": "sip_hangup_disposition", "type": "string", "default": "" },
{"name": "route_list", "type": ["null","int"], "default": null },
{"name": "release_initiator", "type": "string", "default": "" },
{"name": "release_code", "type": ["null","int"], "default": null },
{"name": "ariadna", "type": "string", "default": "" },
{"name": "theseus", "type": "string", "default": "" },
{"name": "sip_rh_X_Gw_Chain", "type": "string", "default": "" },
{"name": "call_obj", "type": ["null","int"], "default": null },
{"name": "route_source", "type": "string", "default": "" },
{"name": "index_in_route", "type": ["null","int"], "default": null },
{"name": "sip_rh_X_Provider_RTP", "type": "string", "default": "" },
{"name": "route_type", "type": "string", "default": "" },
{"name": "offer_codecs", "type": "string", "default": "" },
{"name": "bridge_generate_comfort_noise", "type": "string", "default": "" },
{"name": "span", "type": "string", "default": "" },
{"name": "channel", "type": "string", "default": "" },
{"name": "rtp_in_skip_packet_count", "type": "string", "default": "" },
{"name": "rtp_out_skip_packet_count", "type": "string", "default": "" },
{"name": "codecs_by_name", "type": "string", "default": "" },
{"name": "sip_rh_X_Provider_RTP_Port", "type": "string", "default": "" },
{"name": "billmsec", "type": "string", "default": "" },
{"name": "ported_number", "type": "string", "default": "" },
{"name": "ported_number_ANI", "type": "string", "default": "" },
{"name": "rtt_global", "type": "string", "default": "" },
{"name": "reroute_counter", "type": "string", "default": "" },
{"name": "real_bye_time", "type": "string", "default": "" },
{"name": "privacy_id", "type": "string", "default": "" },
{"name": "sip_h_X_Reducing_ANI", "type": "string", "default": "" },
{"name": "ported_number_nrn", "type": "string", "default": "" },
{"name": "custom_pdd", "type": "string", "default": "" },
{"name": "areacode", "type": "string", "default": "" },
{"name": "rate", "type": "string", "default": "" },
{"name": "rate_out", "type": "string", "default": "" },
{"name": "eff_rate", "type": "string", "default": "" },
{"name": "product_id", "type": "string", "default": "" },
{"name": "sip_h_X_Ani_Source", "type": "string", "default": "" },
{"name": "sip_h_X_Reducing_Source", "type": "string", "default": "" },
{"name": "diversion_privacy", "type": "string", "default": "" },
{"name": "presentation_privacy", "type": "string", "default": "" },

```

```

{"name": "transparent_privacy", "type": "string", "default": "" },
{"name": "undo_ttransparent_privacy", "type": "string", "default": "" },
{"name": "sip_rh_X_Diversion_PAI_Rejection", "type": "string", "default": "" },
{"name": "sip_h_X_Identity_Cdr", "type": "string", "default": "" },
{"name": "sip_h_X_Sign_Cdr", "type": "string", "default": "" },
{"name": "sip_h_X_Obr_Spoof_Surcharge", "type": "string", "default": "" },
{"name": "RTP_PROTOCOL", "type": ["null","int"], "default": null },
{"name": "RTP_IN_PKTS", "type": ["null","int"], "default": null },
{"name": "RTP_IN_BYTES", "type": ["null","int"], "default": null },
{"name": "RTP_OUT_PKTS", "type": ["null","int"], "default": null },
{"name": "RTP_OUT_BYTES", "type": ["null","int"], "default": null },
{"name": "RTP_FIRST_SEQ", "type": "string", "default": "" },
{"name": "RTP_FIRST_TS", "type": "string", "default": "" },
{"name": "RTP_LAST_SEQ", "type": "string", "default": "" },
{"name": "RTP_LAST_TS", "type": "string", "default": "" },
{"name": "RTP_IN_JITTER", "type": "string", "default": "" },
{"name": "RTP_OUT_JITTER", "type": "string", "default": "" },
{"name": "RTP_IN_PKT_LOST", "type": "string", "default": "" },
{"name": "RTP_OUT_PKT_LOST", "type": "string", "default": "" },
{"name": "RTP_IN_PKT_DROP", "type": "string", "default": "" },
{"name": "RTP_OUT_PKT_DROP", "type": "string", "default": "" },
{"name": "RTP_IN_MAX_DELTA", "type": "string", "default": "" },
{"name": "RTP_OUT_MAX_DELTA", "type": "string", "default": "" },
{"name": "RTP_MOS", "type": "string", "default": "" },
{"name": "RTP_IN_MOS", "type": "string", "default": "" },
{"name": "RTP_OUT_MOS", "type": "string", "default": "" },
{"name": "RTP_R_FACTOR", "type": "string", "default": "" },
{"name": "RTP_IN_R_FACTOR", "type": "string", "default": "" },
{"name": "RTP_OUT_R_FACTOR", "type": "string", "default": "" },
{"name": "RTP_RTT", "type": "string", "default": "" }]
}

```

Anexo 7: Generar una clase de java a partir de un schema

Para generar una clase de Java a partir de un archivo .asvc correspondiente a un *Schema* se deben seguir los siguientes pasos:

En primer lugar, hay que descargar el archivo ejecutable avro-tools-<versión>.jar. A continuación, se deja el enlace del repositorio donde se puede descargar la versión utilizada para este proyecto, la 1.8.1:

<https://repo1.maven.org/maven2/org/apache/avro/avro-tools/1.8.1/>

Una vez descargado se debe ejecutar el siguiente comando desde la línea de comandos, que creará un archivo .java con la nueva clase generada:

```
java -jar <ubicación del archivo descargado>/avro-tools-1.8.1.jar compile schema <ubicación del schema>/<nombre>.asvc <ubicación archivo generado>
```

Anexo 8: Código que implementa la lógica de procesamiento para el enriquecimiento de los CDR

El código de java desarrollado en el presente Trabajo Fin de Grado que implementa la lógica de procesamiento para la obtención de los CDR es el siguiente:

```
package flink;

import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.JoinFunction;

import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.tuple.Tuple2;

import org.apache.flink.connector.base.DeliveryGuarantee;
import org.apache.flink.connector.kafka.sink.KafkaRecordSerializationSchema;
import org.apache.flink.connector.kafka.sink.KafkaSink;
import org.apache.flink.connector.kafka.source.KafkaSource;
import org.apache.flink.connector.kafka.source.enumerator.initializer.OffsetsInitializer;
import org.apache.flink.formats.avro.registry.confluent.ConfluentRegistryAvroSerializationSchema;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;
import org.apache.log4j.BasicConfigurator;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.time.Duration;

public class CDR_enrichment {

    //Función que implementa selector de clave para el stream de RTP
    public static class Key_selector_RTP implements FlatMapFunction<String, Tuple2<String,String>> {
        @Override
        public void flatMap(String value, Collector<Tuple2<String,String>> out) {
            // Analizar el mensaje y extraer las direcciones IP y puertos
        }
    }
}
```

```

String[] fields = value.split(";");
String Key = fields[0].split(":")[1] + ";" + fields[1].split(":")[1] + ";" +
fields[8].split(":")[1];
String final_value = "";
for(int i=2; i<fields.length-2;i++){
    if (i!=7 & i!=8) {
        final_value = final_value + fields[i].split(":")[1] + ";";
    }
}

Tuple2<String,String> Keyed_msg= new Tuple2<>(Key,final_value); //Creo la
tupla con la clave y valor
out.collect(Keyed_msg); // Emitir como resultado una tupla con la clave y el
valor

}
}

```

```

// Función que selecciona la clave para los stream dedalus
public static class Key_selector_dedalus implements FlatMapFunction<String,
Tuple2<String,String>> {
    @Override
    public void flatMap(String value, Collector<Tuple2<String,String>> out) {
        // Analizar el mensaje y extraer las direcciones IP y puertos
        String[] fields = value.split(";");
        String Key = fields[31] + ";" + fields[29] + ";" + fields[30];
        String final_value = "";
        for(int i=0; i<80; i++){

            final_value = final_value + fields[i] + ";";
        }

        Tuple2<String,String> Keyed_msg= new Tuple2<>(Key,final_value);
        out.collect(Keyed_msg); // Emitir como resultado una tupla con la clave y el
valor

    }
}

```

```

//Función que ejecuta el join para concatenar dos DataStream
public static class join_streams implements JoinFunction<Tuple2<String,String>,
Tuple2<String,String>,String> {
    @Override
    public String join(Tuple2<String,String> Stream_RTP, Tuple2<String,String>
Stream_CDR) {
        return Stream_CDR.f1 + Stream_RTP.f1;
    }
}

```

//Función que tiene como entrada un DataStream de tipo string con todos los campos separados por comas y lo convierte en un DataStream de tipo CDR

```
public static class change_type implements FlatMapFunction<String, cdr> {
    @Override
    public void flatMap(String value, Collector<cdr> out) {
        // Analizar el mensaje y extraer las direcciones IP y puertos
        String[] fields = value.split(",");

        cdr call_records = new cdr();
        //Field[] fields_name = call_records.getClass().getDeclaredFields();
        for(int i=0; i<fields.length;i++){
            //String setter= "set"+fields_name[i].getName();
            if (i==8 | i==12 | i==13 | i==15 | i==16 | i==18 | i==20 | i==22 | i==23 | i==25 |
i==27 | i==30 | i==32 | i==36 | i==38 | i==42 | i==44 | i==80 | i==81 | i==82 | i==83 |
i==84){

                call_records.put(i,emptystring(fields[i]));
            }
            else{
                call_records.put(i,fields[i].replace("\\\"", "\""));
            }

        }
        out.collect(call_records); // Emitir como resultado una tupla con la clave y el
valor

    }
}
```

```
public static final Logger log =
LogManager.getLogger(CDR_enrichment.class.getName());
```

//Función que convierte en enteros aquellos campos que lo requieran

```
static Integer emptystring(String field){
    if( field.equals("") ){
        return null;
    }
    else{
        return Integer.parseInt(field);
    }
}
```

```
public static void main(String[] args) throws Exception{
```

// Configuración log4j para ver los logs

```
BasicConfigurator.configure();
```

// Creo un entorno de ejecución

```

StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

//Conecto flink con el cluster para leer del topic RTP stats
KafkaSource<String> RTP_stats = KafkaSource.<String>builder()

    .setBootstrapServers("mad-kafka-1.sonoc.io:9092,mad-kafka-
2.sonoc.io:9092,mad-kafka-3.sonoc.io:9092")
    .setTopics("RTP_stats")
    .setStartingOffsets(OffsetsInitializer.latest())
    .setGroupId("flink-consumer")
    //.setProperty("enable.auto.commit", "true")
    .setValueOnlyDeserializer(new SimpleStringSchema())
    .build();

//Creo una variable igual que antes para CDR-avro-test
KafkaSource<String> CDR_dedalus = KafkaSource.<String>builder()

    .setBootstrapServers("mad-kafka-1.sonoc.io:9092,mad-kafka-
2.sonoc.io:9092,mad-kafka-3.sonoc.io:9092")

    .setTopics("mad_bts_dedalus_cdr_111","mad_bts_dedalus_cdr_112","mad_bts_dedalus
_cdr_113")
    .setStartingOffsets(OffsetsInitializer.latest())
    .setGroupId("flink-consumer")
    .setValueOnlyDeserializer(new SimpleStringSchema())
    .build();

//Creo las variables DataStreamSource para leer los mensajes de los topic
DataStreamSource<String> Stream_RTP_stats = env.fromSource(RTP_stats,
WatermarkStrategy
    .forBoundedOutOfOrderness(Duration.ofSeconds(20)), "RTP_stats");

DataStreamSource<String> Stream_CDR_dedalus =
env.fromSource(CDR_dedalus, WatermarkStrategy
    .forBoundedOutOfOrderness(Duration.ofSeconds(20)), "CDR_dedalus");

// Asigno como key ip puerto sacándolos de los datos del stream
DataStream<Tuple2<String,String>> keyed_Stream_RTP_stats=
Stream_RTP_stats.flatMap(new Key_selector_RTP())
    .keyBy(0);

DataStream<Tuple2<String,String>> keyed_Stream_CDR_dedalus=
Stream_CDR_dedalus.flatMap(new Key_selector_dedalus())
    .keyBy(0);

// Con la función join junto los mensajes de ambos streams que tengan esa la

```

misma clave

```
DataStream<String> joined_stream=
keyed_Stream_RTP_stats.join(keyed_Stream_CDR_dedalus)
    .where(value -> value.f0)
    .equalTo(value -> value.f0)
    .window(TumblingEventTimeWindows.of(Time.seconds(30)))
    .apply (new join_streams());

// Modifico el tipo del dataStream para poderlo serializar siguiendo el schema
DataStream<cdr> cdr_joined_stream = joined_stream.flatMap(new
change_type());

// Creo una variable KafkaSink para enviar los datos.
KafkaSink<cdr> CDR_stats = KafkaSink.<cdr>builder()
    .setBootstrapServers("mad-kafka-1.sonoc.io:9092,mad-kafka-
2.sonoc.io:9092,mad-kafka-3.sonoc.io:9092,mad-kafka-4.sonoc.io:9092")
    .setRecordSerializer(KafkaRecordSerializationSchema.builder()
        .setTopic("CDR_stats")

    .setValueSerializationSchema(ConfluentRegistryAvroSerializationSchema.forSpecific(c
dr.class,"CDR_stats","https://schema-registry.mad-k8s.sonoc.io"))
        .build()
    )
    .setDeliveryGuarantee(DeliveryGuarantee.AT_LEAST_ONCE)
    .build();

cdr_joined_stream.sinkTo(CDR_stats);
env.execute();
}
```

Anexo 9: Comandos básicos para Apache Flink

Para poder utilizar apache Flink se debe descargar la versión deseada, disponible en el siguiente enlace:

<https://flink.apache.org/downloads/>

Una vez elegida la versión se descargará en nuestro sistema un archivo comprimido .tgz, que al descomprimir dará lugar a un directorio donde podemos encontrar archivos binarios, ejemplos, archivos de configuración de la aplicación, etc.

Para iniciar un clúster virtual de Flink debemos situarnos en la carpeta bin y ejecutar el siguiente comando:

```
./start-cluster.sh
```

Una vez iniciado el clúster virtual podemos poner a ejecutar trabajos que correrán en el *task manager*. Para ello debemos ejecutar el siguiente comando:

```
./bin/flink run --detached ./<directorio>/<nombre>.jar
```

Al ejecutar el comando se le asignará un ID al proceso, el cual se mostrará por pantalla. Para terminar de correr el proceso debe ejecutarse el siguiente comando:

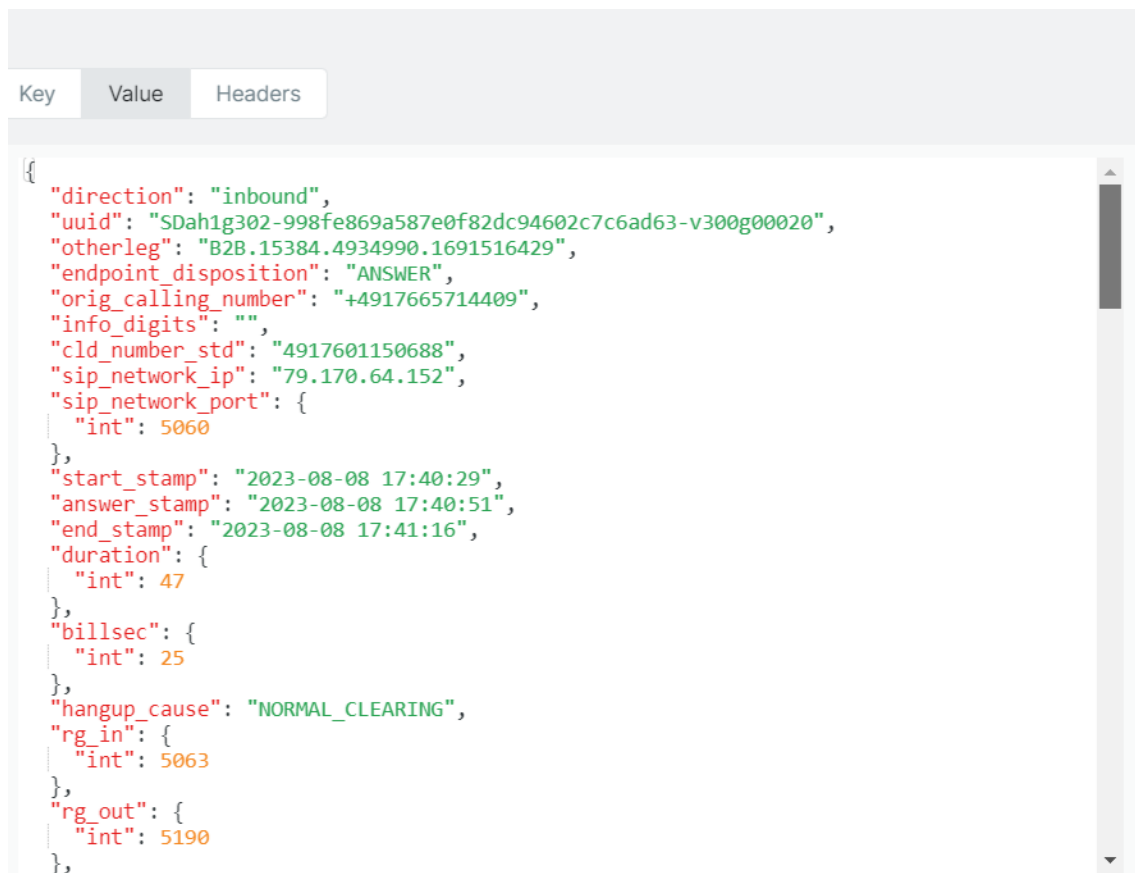
```
./bin/flink stop $ID
```

Por último, Podemos listar todos los trabajos en ejecución ejecutando:

```
./bin/flink list
```

Anexo 10: Resultado programa de procesamiento de Flink

Una vez puesto en ejecución el programa elaborado para la plataforma de Apache Flink en el servidor de pruebas, se generan registros de llamadas telefónicas que combinan tanto las estadísticas de señalización como las de audio calculadas en este proyecto. Si miramos en el *topic* CDR_stats en la interfaz gráfica disponible para Kafka, podemos ver de una manera visual el resultado del proceso de enriquecimiento. A continuación se muestra un ejemplo del CDR de una llamada telefónica obtenido tras el procesamiento de Flink:



```

"called_number": "4917601150688",
"orig_clg_noa": {
  "int": 4
},
"ani": "4917665714409",
"orig_clg_noas": {
  "int": 4
},
"orig_called_number": "832234917601150688",
"orig_cld_noa": {
  "int": 4
},
"orig_cld_noas": {
  "int": 4
},
"progress_stamp": "2023-08-08 17:40:32",
"codec": {
  "int": 8
},
"sip_local_network_addr": "79.170.64.157",
"sip_received_port": {
  "int": 5060
},
"sip_profile_name": "internal",
"remote_rtp_ip": "84.16.20.109",
"remote_rtp_port": {
  "int": 13632
},
"local_rtp_ip": "79.170.65.29",
"local_rtp_port": {
  "int": 14142
},
"rtp_audio_in_packet_count": "",
"rtp_audio_out_packet_count": "",
"sip_hangup_disposition": "recv_bye",
"route_list": {
  "int": 103707
},
"release_initiator": "0",
"release_code": {
  "int": 0
},
"ariadna": "1.2.3.4:5060",
"theseus": "",
"sip_rh_X_Gw_Chain": "4917601150688:62.52.128.72:200;",
"call_obj": {
  "int": 11780
},
"route_source": "n",
"index_in_route": {
  "int": 1
},
"sip_rh_X_Provider_RTP": "62.52.127.214",
"route_type": "R",
"offer_codecs": "8-101",
"bridge_generate_comfort_noise": "false",

```

```

"span": "",
"channel": "",
"rtp_in_skip_packet_count": "",
"rtp_out_skip_packet_count": "",
"codecs_by_name": "",
"sip_rh_X_Provider_RTP_Port": "43880",
"billmsec": "25573",
"ported_number": "",
"ported_number_ANI": "",
"rtt_global": "0",
"reroute_counter": "1",
"real_bye_time": "",
"privacy_id": "",
"sip_h_X_Redirecting_ANI": "",
"ported_number_nrn": "",
"custom_pdd": "3",
"areacode": "",
"rate": "",
"rate_out": "",
"eff_rate": "",
"product_id": "",
"sip_h_X_Ani_Source": "pai",
"sip_h_X_Redirecting_Source": "",
"diversion_privacy": "",
"presentation_privacy": "1",
"transparent_privacy": "",
"undo_ttransparent_privacy": "",
"sip_rh_X_Diversion_PAIR_Rejection": "",

```

```

"sip_h_X_Identity_Cdr": "",
"sip_h_X_Sign_Cdr": "",
"sip_h_X_Obr_Spoof_Surcharge": "",
"RTP_PROTOCOL": {
  "int": 17
},
"RTP_IN_PKTS": {
  "int": 1285
},
"RTP_IN_BYTES": {
  "int": 257000
},
"RTP_OUT_PKTS": {
  "int": 1096
},
"RTP_OUT_BYTES": {
  "int": 219200
},
"RTP_FIRST_SEQ": "924",
"RTP_FIRST_TS": "147840",
"RTP_LAST_SEQ": "2208",
"RTP_LAST_TS": "353280",
"RTP_IN_JITTER": "62",
"RTP_OUT_JITTER": "15",
"RTP_IN_PKT_LOST": "0",
"RTP_OUT_PKT_LOST": "0",
"RTP_IN_PKT_DROP": "0",
"RTP_OUT_PKT_DROP": "0",
"RTP_IN_MAX_DELTA": "2029",
"RTP_OUT_MAX_DELTA": "171999",
"RTP_MOS": "440",
"RTP_IN_MOS": "440",
"RTP_OUT_MOS": "440",
"RTP_R_FACTOR": "9294",
"RTP_IN_R_FACTOR": "9294",
"RTP_OUT_R_FACTOR": "9295",
"RTP_RTT": "0.146"

```

Bibliografía

[1] Documentación de Java

- *¿Qué es Java? - Explicación del lenguaje de programación Java - AWS*. (s. f.). Amazon Web Services, Inc. <https://aws.amazon.com/es/what-is/java/>

[2] Documentación de la tarjeta NT200A02 SmartNIC

- Napatech. (2023, 13 febrero). *NT200A02 SmartNIC* - NapaTech. <https://www.napatech.com/products/nt200a02-smartnic-capture/>

[3] Documentación de NTPL

- *Documentation Portal*. (s. f.). <https://docs.napatech.com/r/Reference-Documentation/NTPL>

[4] Documentación de nProbe

- *NProbe*. (2023, 14 junio). ntop. <https://www.ntop.org/products/netflow/nprobe/>
- *Introduction — NProbe 10.1 Documentation*. (s. f.). <https://www.ntop.org/guides/nprobe/introduction.html>

[5] Documentación de Apache Kafka

- *Apache Kafka*. (s. f.). Apache Kafka. <https://kafka.apache.org/>

[6] Documentación de Apache Flink

- *Apache Flink® — Stateful computations over data streams*. (s. f.). Apache Flink. <https://flink.apache.org/>

[7] Documentación del protocolo IP

- *RFC 791: Internet Protocol*. (1981, 1 septiembre). IETF Datatracker. <https://datatracker.ietf.org/doc/html/rfc791>

[8] Documentación del protocolo UDP

- Postel, J. (1980b). *User Datagram Protocol*. <https://doi.org/10.17487/rfc0768>

[9] Documentación del protocolo TCP

- Eddy, W. M. (2022, 1 agosto). *RFC 9293: Transmission Control Protocol (TCP)*. <https://www.rfc-editor.org/rfc/rfc9293.html>

[10] Documentación del protocolo RTP

- Schulzrinne, H., Casner, S., Frederick, R., & Jacobson, V. (1996b). *RTP: a transport protocol for Real-Time applications*. <https://doi.org/10.17487/rfc1889>

[11] Documentación del protocolo SIP

- Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., & Schooler, E. M. (2002). *SIP: Session Initiation Protocol*. <https://doi.org/10.17487/rfc3261>

[12] Documentación de parámetros de la calidad del audio

- Altanai. (2022, 10 abril). *VOIP call metric Monitoring and MOS (mean Opinion Score)*. Telecom R & D. <https://telecom.altanai.com/2018/04/17/voip-call-metric-monitoring/#mos-rfactor>
- *Call quality defining using R-Factor and MOS* | VoiceHost. (s. f.). <https://www.voicehost.co.uk/help/call-quality-defining-using-r-factor-and-mos>
- *Improving Real-time communication (RTC) client experience with AWS Global Accelerator* | Amazon Web Services. (2021, 18 octubre). Amazon Web Services. <https://aws.amazon.com/es/blogs/networking-and-content-delivery/improving-real-time-communication-rtc-client-experience-with-aws-global-accelerator/>
- Matango, F., & Matango, F. (2019). Jitter - Fluctuaciones de velocidad | VOIP. VoIP. <http://www.servervoip.com/blog/jitter-fluctuaciones-de-velocidad/#:~:text=La%20idea%20b%C3%A1sica%20del%20E2%80%9Cjitter,producen%20a%20una%20velocidad%20constante>