Patrones de captura de datos dinámicos

PID_00276694

Sergi Grau López

Universitat Oberta de Catalunya

Sergi Grau López

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Albert Solé Ribalta

Primera edición: marzo 2021

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Sergi Grau López
Producción: FUOC

Índice

Introducción

En este módulo introductorio primero vamos a describir y analizar qué entendemos por datos en *streaming*. Seguidamente, describiremos los métodos básicos de captura de datos dinámicos y pondremos en contexto las arquitecturas y patrones empleados en dicha captura. Se va a poner especial énfasis en los entornos en tiempo real o próximos al mismo donde la escalabilidad horizontal y los sistemas distribuidos son esenciales para el procesamiento de datos de alta demanda.

Se analizarán las diversas arquitecturas existentes para esta adquisición de datos y la necesidad de una capa intermedia necesaria para que los clientes que consumen los datos dispongan de tiempo suficiente para poder procesar los datos que se van adquiriendo, garantizando que no se pierda información, en caso que así sea necesario. Esta capa intermedia se conoce como *capa broker de mensajería*.

Adicionalmente, en la parte final del módulo se van a describir dos casos de uso que ejemplifican y contextualizan dos de los patrones fundamentales y más usados. Entender y saber trabajar estos casos de uso va a ser fundamental para la comprensión de los siguientes módulos.

1. ¿Qué son los datos en *streaming*?

1.1. Definiciones

La National Communications System (NCS) de los Estados Unidos definió a mediados de los noventa los datos en *streaming* o flujos de datos (*data stream*) como:

> «A sequence of digitally encoded signals used to represent information in transmission».

Es decir, cualquier señal que represente datos y que se emita de forma secuencial. Aun así, a partir de la introducción del concepto *big data* (en 2005 por Roger Mougalas en O'Reilly Media) la definición de *data stream* ha ido evolucionando y, actualmente, podemos encontrar definiciones más adaptadas a nuestros tiempos como, por ejemplo, la dada por Amazon, que aconsejamos revisar en profundidad:

> «Los datos de *streaming* son datos que se generan constantemente a partir de miles de fuentes de datos, que normalmente envían los registros de datos simultáneamente en conjuntos de tamaño pequeño (varios kilobytes)».

Como podemos ver, aunque no lo hacen explícitamente, estas definiciones ya van orientadas a incorporar las tres V del *big data*: variedad, volumen y velocidad.

Como iremos viendo a lo largo de los siguientes módulos, los datos en *streaming* pueden provenir de muchas fuentes y tener estructuras muy distintas en contenido y velocidad de generación, como también pueden ser muy distintos los dispositivos que los generan. Dada esta gran variedad de circunstancias, existen infinidad de clasificaciones posibles sobre flujos de datos: biométricos/no biométricos, periódicos/racheados, numéricos, etc. Aquí dejamos a cargo del lector los diversos tipos de clasificaciones que pueden ser más o menos convenientes según el tipo de aplicación que tengamos entre manos y pedimos que no se entiendan como estrictas las clasificaciones que damos en los siguientes ejemplos.

Ejemplos

- **Internet de las cosas (IoT) y monitorización del entorno**: en este campo podríamos encontrar desde datos generados por ciudades (tráfico, polución, meteorología) hasta la *industrial IoT* (IIoT). También podríamos considerar en esta tipología datos de viviendas monitorizadas y *smart homes*.
- **Biométricos y fisiológicos**: datos para la monitorización de organismos biológicos. Datos de monitores personales como relojes *smartwatch*, sensores cardiovasculares, dispositivos médicos de monitorización de cualquier tipo, sensores de humedad/ra-

diación solar para plantas y, en general, sensores de monitorización para la agricultura / ganadería.

- **Datos en entornos económicos**: mercado de valores, transacciones entre países, movimientos bancarios, etc.
- **Redes sociales**: Twitter, Facebook, WhatsApp, etc.

1.2. Sistemas de *streaming* y sus diferencias con los sistemas en tiempo real

Desde hace tiempo disponemos de sistemas en tiempo real, pero no debemos confundirlos con los sistemas de *streaming*. Estos últimos (*streaming*) se concentran en suministrar datos a los clientes en el momento que estos son necesitados, como por ejemplo el acceso a información de redes sociales (Twitter, MeetUp, etc.), la consulta de información de tráfico en tiempo real, los estados de vuelos, el acceso a las cotizaciones bursátiles, etc. Muchos de estos datos se generan de manera continua en el tiempo y los clientes que hagan uso del *streaming* de datos consumen estos datos en función de sus necesidades.

> «En este contexto consideraremos un sistema de *streaming* como un sistema en tiempo real que transmite los datos para que estén disponibles en el momento en que una aplicación cliente los necesita». (Psaltis, 2017. *Streaming Data: Understanding the Real-Time Pipeline*. Nueva York: Manning Publications).

A diferencia de los sistemas en *streaming*, los sistemas en tiempo real necesitan ejecutarse sin demora, con tiempos de respuesta en general inferiores al milisegundo, como los frenos de un vehículo o un marcapasos. En general, estos sistemas requieren del uso de sistemas operativos de tiempo real (RTOS).

Lectura recomendada

Para profundizar en estas diferencias revisar el capítulo 1 de esta obra:

F. Julbe, J. Conesa, J. Casas, M. E. Rodríguez (2019). *Captura, pre-processament i emmagatzematge de dades massives*. Barcelona: Universitat Oberta de Catalunya.

1.3. Modelos de datos en *streaming*

Como hemos visto en el apartado anterior, podemos encontrar definiciones más o menos técnicas sobre qué son los datos en *streaming*, pero seguramente la analogía que nos ayudará más a entender qué son los flujos de datos es la comparación con un río sin origen ni final, dónde el agua fluye ininterrumpidamente a un ritmo determinado y con cierta ordenación temporal. El agua que salió a tiempo *t* siempre precede a la que saldrá a tiempo *t + 1*. Esta propiedad temporal es deseable y la mayor parte de sistemas de streaming, aunque no todos, dan opciones de configuración para garantizar este aspecto. A partir de esta definición cualitativa podemos introducir una clasificación de datos en *streaming* bastante más formal, la que habitualmente se usa (Muthukrishnan, 2005).

Lectura recomendada

S. Muthukrishnan (2005). *Data streams: Algorithms and applications*. Hannover: Now Publishers Inc.

Nuestra **entrada de datos**, a la que nos referimos como ***flujo de datos***, llega secuencialmente, elemento por elemento, a_i, y describe una señal subyacente, A, que es una función unidimensional, en un dominio discreto y ordenado.

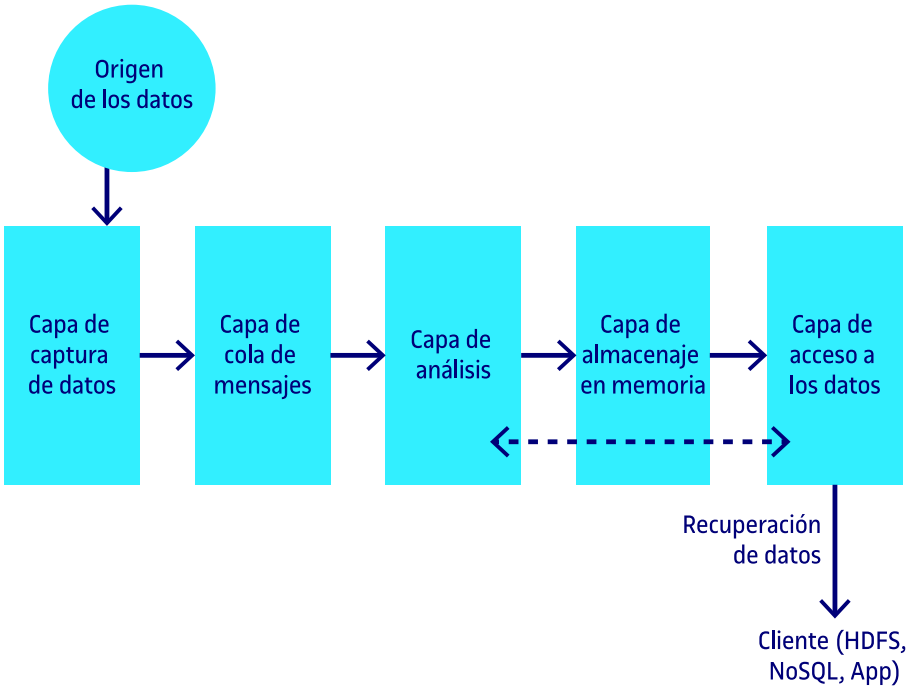
El flujo puede describir la señal subyacente de muchas maneras, produciendo diferentes modelos de datos como resultado: A:[0…N] -> Z^+ y los valores se pueden representar como un continuo de tuplas a_i = (j, Ui).

- **Modelo serie temporal** (*time series model*): cada A[i] es igual a a_i y aparecen en orden creciente de *i*. Este es un modelo adecuado para datos de series temporales en los que, por ejemplo, se está monitorizando el tráfico IP (p. ej. paquetes por minuto) en un dispositivo de red IP cada cierto tiempo, o cuando se analizan las transacciones de bolsa (p. ej. valor actual de un determinado valor).

- **Modelo de caja registradora** (*cash register model*): este es quizás el modelo de flujo de datos más popular. En el modelo de la cinta de la caja registradora, los elementos a_i corresponden a incrementos de la señal. Por tanto, el valor en tiempo *i*, $A_i[j]$, se construye mediante el valor llegado a tiempo i y sus anteriores: $A_i[j] = A_{(i-1)}[j] + U_i$. De forma similar a una caja registradora, varios valores de a_i's pueden incrementar el valor de $A_i[j]$ a lo largo del tiempo. Por ejemplo, si representamos el monitoreo de direcciones IP que acceden a un servidor web, número de paquetes que pasan por un dispositivo, etc. Nótese que las mismas direcciones IP (valor *j*) pueden acceder al servidor web varias veces o enviar múltiples paquetes en el enlace a lo largo del tiempo. Por ejemplo, `(80.70.10.10, /home),(121.60.70.11, /productos),(80.70.10.10, /productos).`

- **Modelo torniquete** (*turnstile model*): aquí a_i son actualizaciones de $A_i[j]$. Si recibimos la tupla $a_i = (j,U_i)$, significa que $A_i[j] = A_{(i-1)}[j] + U_i$, donde $A_i[j]$ es el valor de la señal después de ver el elemento i-ésimo en la secuencia, y U_i puede ser positivo o negativo. Este es el modelo más general. Está ligeramente inspirado en una concurrida estación de metro donde el torniquete de acceso realiza un seguimiento de las personas que llegan y salen continuamente. Démonos cuenta que este modelo generaliza en sí mismo el de caja registradora.

Figura 1. Detalle de un sistema de análisis de datos



Fuente: elaboración propia.

1.4. Arquitectura de un sistema de tratamiento de datos

En general, para todos los sistemas de análisis de datos se pueden identificar una serie de etapas diferenciadas, tal y como se ilustra en la figura 1.

1) captura de datos,

2) análisis de los datos,

3) almacenamiento de los datos y, opcionalmente,

4) visualización de resultados.

En este caso, dependiendo de los sistemas, las capas 2 y 3 se pueden intercambiar en orden.

En la figura 1 también aparece una capa adicional identificada como *cola de mensajes*. Como veremos posteriormente, esta capa no es imprescindible pero sí aporta valor al sistema. A lo largo de los módulos de la asignatura vamos a detallar cada una de estas capas y vamos a contextualizarlas en el entorno *big data* centrándonos principalmente en las aplicaciones de *streaming*. A continuación comenzaremos con el estadio de captura de datos.

Lectura recomendada

Para extender más en los conceptos básicos de una arquitectura de tratamiento de datos se recomienda revisar los capítulos 7, 8, 10 y 11 de la referencia:

J. Nin Guerrero, F. Julbe López, J. Casas Roma (2019). «Big data: análisis de datos en entornos masivos». *Big data*, págs. 1-287.

2. Introducción a la captura de datos masivos en *stream*

Cualquier sistema que pretenda analizar un conjunto de datos tiene que, en primera instancia, capturar o leer estos datos. Esta etapa es forzosamente previa al almacenaje y procesamiento de los datos.

En los entornos *big data* podemos diferenciar entre la captura en sistemas de procesos de lotes (*batch processing*) o en sistemas de análisis de datos en flujo (*streams data*).

En los sistemas *batch*, el proceso es muy básico ya que los datos se asumen almacenados y fijos. En estos casos, independientemente de la complejidad de los datos o acceso a ellos, simplemente basta con leerlos, interpretarlos y hacer los cálculos necesarios. En cambio, cuando nos fijamos en sistemas de flujo, el proceso se vuelve mucho más complejo y requiere de estructuras y arquitecturas especializadas. En la mayoría de casos identificamos tres agentes con tareas diferenciadas:

- los productores
- los consumidores
- los agentes de mensajería o *brokers*

Ved también

Veremos los sistemas *batch* en más profundidad en el módulo «Perspectiva histórica al procesado por lotes».

Se denominan ***productores*** a los agentes que generan información, en forma de mensajes o datos. Como hemos visto, los orígenes de datos pueden ser muy diversos: redes sociales, sensores IoT, *smart cities*, meteo, cotizaciones bursátiles, información en servidores[1].

(1)Las informaciones que se recogen desde servidores acostumbran a ser ficheros *log*, UNIX *syslog* o *network sockets*.

Los **consumidores** son aquellos elementos del sistema que van a obtener la información que generan los productores, a veces para almacenarla en una base de datos (en la mayoría de las ocasiones NoSQL dada la gran cantidad de información) o para procesarla de manera distribuida con y sin necesidad de un almacenaje previo. En muchas ocasiones, estas bases de datos son privadas (*data silos*), y en otras ocasiones los datos están accesibles de manera pública[2].

(2)Se denomina *open data*, o datos abiertos, la práctica que persigue que determinados tipos de datos estén disponibles de forma libre para todo el mundo, sin licencias, patentes u otros mecanismos de control.

Los **agentes de mensajería** (*brokers*) tienen como finalidad controlar y gestionar el proceso de conexión entre productores y consumidores, y permiten garantizar determinadas políticas de acceso al dato por parte de los consumidores:

- A lo sumo una vez (*at most once*): el consumidor recibirá el mensaje como mucho una sola vez. El mensaje puede perderse y no ser nunca recibido.
- Al menos una vez (*at least once*): el consumidor recibirá el mensaje como mínimo una vez. El mensaje nunca se perderá, pero un consumidor puede recibirlo varias veces.
- Exactamente una vez (*exactly once*): el mensaje nunca se pierde y el consumidor solo lo recibe una vez.

A lo largo de los módulos veremos que garantizar la política *exactly once* es el objetivo último de todo sistema de *streaming*, ya que garantiza que todos los datos se van a tratar y que solamente se van a tratar una sola vez. Cuando parte del sistema no pueda garantizar la política *exactly once* veremos que tenemos que buscar alternativas para garantizar la fiabilidad de los análisis resultantes.

Ejemplo

Como veremos, la capa de *streaming* en la arquitectura Lambda no garantiza el procesamiento *exactly once* y, como consecuencia, tenemos que añadir un procesado paralelo mediante una capa *batch*.

Siguiendo con el tema que nos concierne en estos momentos, además de asegurar ciertas políticas de procesamiento, los *brokers* gestionan las colas de mensajes pudiendo aplicar procesos de normalización, cálculo o agrupación de la información previa a la etapa de almacenaje de la información y garantizando en general el buen funcionamiento del sistema de captura.

Ved también

Como veremos en el módulo «Arquitecturas Big Data para el procesado de datos en flujo», cuando hay que garantizar que los datos generados por el productor no se pierdan, el papel del agente de mensajería es fundamental. En estos casos es habitual que el agente se incluya de manera distribuida en la arquitectura, para garantizar que si el agente cae, existan otros de respaldo para su suplantación.

3. La captura de datos (*data ingestion*) y los patrones de adquisición de datos

La etapa de adquisición de datos es la encargada de recoger los datos e introducirlos en el *pipeline* descrito en la figura 1 para que estos sean o bien consumidos por el cliente, o tratados por el agente de intermediación. En este apartado vamos a ver cómo los diferentes elementos en un sistema de información pueden comunicarse y después vamos a ver los patrones de comunicación más usados.

3.1. Comunicación entre sistemas de información distribuidos

Durante la evolución de los sistemas de información, la comunicación entre diversos agentes software siempre ha sido una parte fundamental y, consecuentemente, se han diseñado, desarrollado y usado infinidad de:

- medios (aire, coaxial, RJ45, fibra, líneas eléctricas),
- protocolos de red (TCP, UDP, Bluetooth),
- mecanismos que proporciona un sistema operativo para permitir que los procesos administren datos compartidos como IPC y
- librerías de software (de bajo nivel como *sockets* o de más alto nivel de abstracción, como los de envío de mensajes, como CORBA, RPC, RMI, POSIX, SOAP, REST o *websockets*) para comunicar los diversos agentes que componen los sistemas.

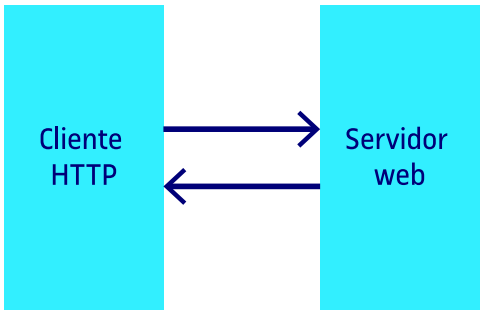
Actualmente, lo más habitual es que los sistemas funcionen mediante protocolos TCP/IP, independientemente de la capa física sobre la que estemos trabajando. Se recomienda revisar el modelo OSI para ampliar los conocimientos sobre las arquitecturas de comunicación más habituales.

Cuando estamos hablando de la adquisición de datos, y en paralelo al esquema de la figura 1, vemos que el flujo de interacción entre el sistema de origen de los datos y el agente de captura es siempre el mismo: un elemento del sistema genera y otro lee.

Para que esta interacción tenga sentido, estos dos elementos tienen que establecer un protocolo de comunicación en común para poder entenderse en su comunicación (capa de aplicación en el modelo OSI).

Básicamente, este protocolo de comunicación establece cómo va a ser el flujo de información y la presentación de los datos para cada una de las funcionalidades que el elemento productor de datos ofrezca. Un ejemplo muy básico lo podemos encontrar en las peticiones mediante el método GET del protocolo HTTP. En este caso podemos experimentar directamente con cualquier navegador web, ya que los parámetros de la llamada van codificados en la URL a la que se accede. Vamos a ver el protocolo de comunicación con el servicio de Accuweather con un ejemplo para determinar la meteorología para Barcelona.

Figura 2. Petición con método GET y respuesta HTTP/1.1



Fuente: elaboración propia.

En este caso los parámetros de las llamadas al servicio son una petición HTTP:

```
http://dataservice.accuweather.com/currentconditions/v1/307297?apikey={your_key}&details=false
```

y la respuesta HTTP contiene un JSON:

```
[
   {
    "LocalObservationDateTime":"2020-08-13T23:36:00+02:00",
    "EpochTime":1597354560,
    "WeatherText":"Partly cloudy",
    "WeatherIcon":35,
    "HasPrecipitation":false,
    "PrecipitationType":null,
    "IsDayTime":false,
    "Temperature":{
         "Metric":
   {"Value":26.1, "Unit":"C", "UnitType":17},
         "Imperial":
   {"Value":79.0, "Unit":"F", "UnitType":18}
    }
   }
]
```

Como se puede ver en la llamada, la api de AccuWeather requiere una clave para poder hacer uso de las llamadas. Se recomienda revisar y experimentar un poco con el api, para entender mejor el funcionamiento de este tipo de protocolos.

Existe una infinidad de protocolos, entornos y medios para comunicar aplicaciones, lo que se traduce en una gran diversidad de patrones de captura de datos. A continuación, vamos a ver una clasificación de los más comunes.

3.2. Patrón petición/respuesta

El patrón **petición/respuesta (*request/response*)** se usa cuando el cliente espera una respuesta realizada por un servicio en el momento que se ha solicitado. Se trata del patrón más simple y es muy conocido por su empleo por el protocolo HTTP, el estándar que se utiliza en la navegación web clásica.

Este patrón, ilustrado en la figura 3, consta de dos fases:

1) En la primera fase (*request*) el agente que inicia la comunicación envía un mensaje al segundo agente solicitando cierto tipo de información.

2) Una vez recopilada (o calculada la información solicitada), se devuelve la respuesta al primer agente (*response*).

Cuando dos agentes se comunican mediante este patrón, hasta que el servicio no realiza la acción solicitada, el cliente queda esperando de manera síncrona la obtención del resultado. Tenemos **sincronía**, tanto por parte del cliente como del servicio. Actualmente existen variaciones a este comportamiento añadiendo asincronía tanto en el cliente como en el servicio.

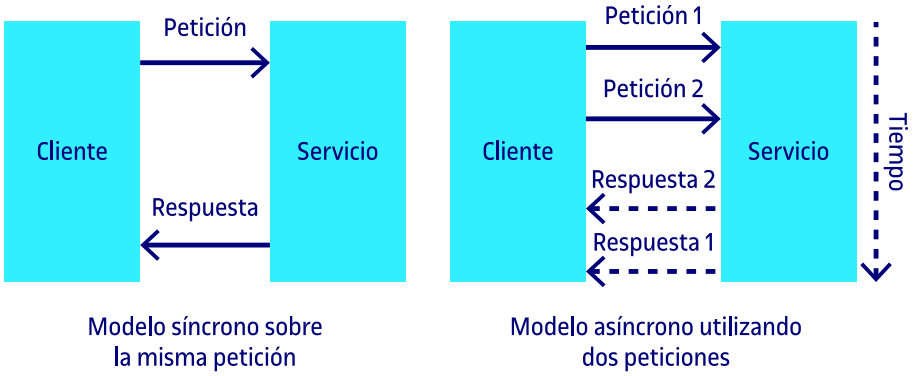
Muchas veces este tipo de patrón de petición/respuesta también se denomina comunicación mediante *pull*. En este caso debemos entender que el productor es el que «empuja» los mensajes de información al consumidor.

En el uso habitual de este patrón (en la comunicación web), la frecuencia de interacción es muy baja de media, del orden de pocas interacciones por minuto. Por lo tanto, podríamos decir que es una **comunicación de baja frecuencia**. Cuando estamos pensando en usar este patrón para la captura de datos en *streaming*, debemos pensar en frecuencias de interacción mucho más altas, del orden de miles o decenas de miles por minuto, como mínimo. En estas circunstancias este tipo de comunicación comparte las ventajas y problemas de la comunicación síncrona. Los descubrimos a continuación.

La ventaja de este tipo de patrón es que es sencillo de implementar y, por tanto, se encuentra presente en multitud de protocolos, especialmente en el protocolo HTTP. Su principal inconveniente es la escalabilidad y la eficiencia. Pensemos en un sistema donde la comunicación entre el consumidor y el productor sea muy intensiva y genere respuestas innecesarias y redundantes, o donde la inexistencia de un gestor intermedio (un *broker*, que modele, garantice y reajuste las comunicaciones) provoque una saturación y lentitud tanto de la red de comunicaciones como del proceso de adquisición de datos. La principal fuente de problemas es la sincronía del patrón entre petición y respuesta. El cliente queda esperando la obtención de la respuesta (la mayoría de los sistemas solucionan la sincronía del lado del servicio haciendo uso de la programación concurrente). Este inconveniente se ha solucionado mediante bibliotecas y técnicas de programación que permiten que las peticiones del cliente se realicen de manera asíncrona, de tal manera que la ejecución del código en el cliente continúe, y en el momento que el servicio envíe la información de respuesta, esta quede recogida por la función que realizó la petición (*callback*[3]). La figura 3 muestra el patrón en sus dos modos, síncrono y asíncrono.

[3]Se denomina *callback* a un código ejecutable (o un puntero a este código) que se pasa como argumento dentro de una función. Si la ejecución de este código es inmediata se llama *callback* síncrono y si la ejecución se puede realizar más tarde entonces llama *callback* asíncrono.

Figura 3. Patrón petición/respuesta



Fuente: elaboración propia.

3.3. Aspectos prácticos y tecnologías de implementación del patrón de petición/respuesta

Como ya se ha comentado a lo largo de este apartado, el protocolo HTTP es quizás el protocolo más conocido que incorpora este mecanismo de comunicación.

A nivel práctico debemos saber que prácticamente todos los lenguajes de programación incorporan librerías que permiten usar este protocolo de manera muy sencilla mediante llamadas HTTP, algunas de manera síncrona, otras de manera asíncrona.

Podemos apreciarlo en el uso de XHR o la API fetch en JavaScript, o con el módulo `http.client` de Python y la biblioteca *requests* para llamadas síncronas, o AIOHTTP para peticiones HTTP asíncronas con Python. Para que se entien-

da la sencillez del uso de estas librerías, los siguientes códigos muestran tres ejemplos de cómo realizar una petición HTTP con Python, uno con la biblioteca HTTP nativa, otro con la biblioteca *requests* para peticiones síncronas y el último utiliza la biblioteca AIOHTTP para llamadas asíncronas al servidor.

1) Uso de la biblioteca HTTP nativa

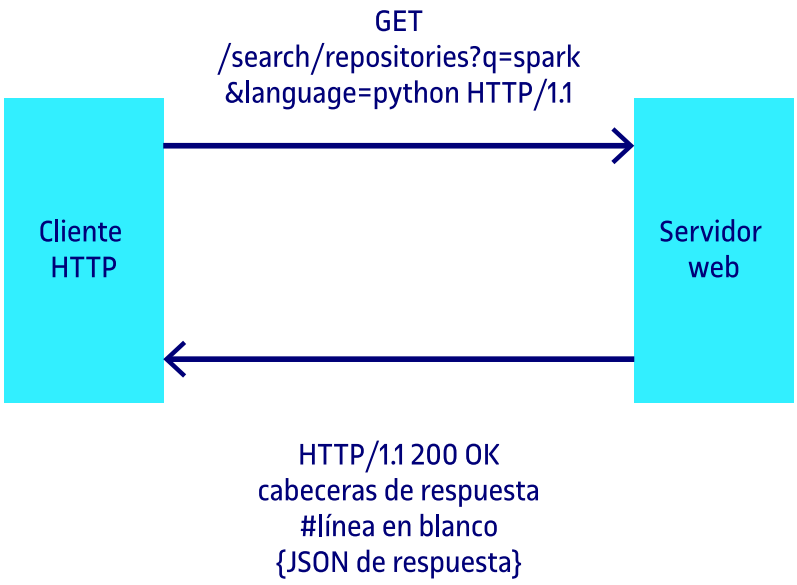
```
#uso de la biblioteca http nativa de Python. Http es un paquete que recoge varios módulos para
#trabajar con el protocolo HTTP. client, es un módulo define clases que implementan la parte de
#cliente de los protocolos HTTP y HTTPS
import http.client
#definimos el host al que vamos a conectar mediante HTTPS
conn = http.client.HTTPSConnection("www.uoc.edu")
#realizamos una petición con el método GET
conn.request("GET", "/")
#recuperamos la respuesta de la petición HTTPS
r1 = conn.getresponse()
print(r1.status, r1.reason)
```

Salida:

```
200 OK
```

2) Uso de la biblioteca *requests*

Figura 4. Petición y respuesta HTTP realizada con la biblioteca *requests* de Python



Fuente: elaboración propia.

```
#uso de la biblioteca requests
import requests
respuestaHttp = requests.get(
    'https://api.github.com/search/repositories',
    params={'q': 'spark+language:python'},
```

```
)

# analizamos algunas propiedades del json recuperado
json_response = respuestaHttp.json()
for repositorio in json_response['items']:
    print(f'Nombre repositorio: {repositorio["name"]}')  #requiere Python 3.6+
    print(f'Descripción repositorio: {repositorio["description"]}')
```

Salida:

```
Nombre repositorio: spark-deep-learning
Descripción repositorio: Deep Learning Pipelines for Apache Spark
Nombre repositorio: TensorFlowOnSpark
Descripción repositorio: TensorFlowOnSpark brings TensorFlow programs to Apache Spark clusters.
Nombre repositorio: koalas
Descripción repositorio: Koalas: pandas API on Apache Spark
Nombre repositorio: elephas
Descripción repositorio: Distributed Deep learning with Keras & Spark
Nombre repositorio: dpark
     ...
```

3) Uso de la biblioteca AIOHTTP para llamadas asíncronas

```
#uso de la biblioteca aiohttp para llamadas asíncronas, requiere Python 3.7+
import aiohttp
import asyncio
#Las corutinas declaradas con la sintaxis async/await definen funciones asíncronas

async def peticion_asincrona(session, url):
    async with session.get(url) as response:
 #await permite al bucle de eventos gestionar otra corutina
#no se queda esperando obtener la respuesta y continúa la ejecución
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await peticion_asincrona(session, 'https://www.uoc.edu')
        print(html)

if __name__ == '__main__':
     #el método run obtiene el bucle de eventos y pasa un primera corutina
loop = asyncio.run(main())
```

Salida:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="es">
<head>
        <meta http-equiv="content-type" content="text/html; charset=UTF-8">
        <meta name="robots"  content="all" />
        <meta name="author" content="Universitat Oberta de Catalunya" />


     ...
```

Para un uso más simple de este protocolo, si cabe, muchos sistemas operativos ofrecen comandos como `curl`, `wget` o `netcat` que permiten realizar peticiones HTTP desde el mismo intérprete de comandos. Usando la herramienta `curl`, el código equivalente al ejemplo anterior sería:

```
$ curl --raw https://www.uoc.edu/
```

Salida:

```
1f41


<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="ca">
<head>
     <meta http-equiv="content-type" content="text/html; charset=UTF-8">
…
```

Este tipo de llamada nos puede ser muy útil en el caso de querer recopilar datos en formato texto o comprimidos. Vemos dos ejemplos para recolectar un fichero de texto:

```
$ curl -O http://dominio.com/fichero.txt
$ wget http://dominio.com/fichero.txt
```

3.4. Patrón petición/acuse

El patrón **petición/acuse** (***request/acknowledge***) es similar al de petición/respuesta pero sin la necesidad de una respuesta por parte del servicio, siendo solo necesario el conocimiento que la petición ha sido recibida (se recibe un mensaje ACK).

Normalmente se utilizan en contextos de seguimiento (*tracking*) de comportamientos de usuarios, como plataformas de compras, de video, etc. En estos casos simplemente se envía una información del comportamiento del usuario, sin esperar una respuesta con contenido del mismo.

3.5. Patrón publicación/subscripción

Sin duda el **patrón publicación/subscripción (*publish/subscribe*)** es el más ampliamente utilizado, y consiste en que el productor (*publisher*) o productores envían mensajes a uno o más consumidores (*subscriber*) que pueden clasificarse o agruparse en relación a un asunto (*topic*).

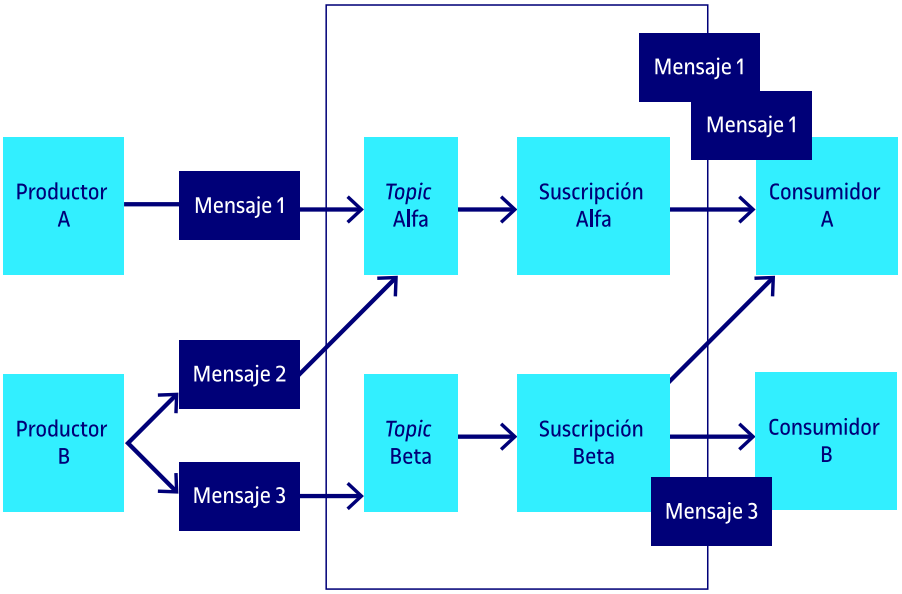
Normalmente llamaremos a este asunto *tópico de interés*. Existen distintas variantes para enlazar productor y consumidor. Las iremos viendo a lo largo de los módulos, junto con varios ejemplos, pero en este punto es mejor quedarse solamente con la idea.

Una vez los mensajes se encuentran en el agente de intermediación, estos serán enviados a los consumidores (*push*). En algunos casos pueden ser los consumidores los que solicitan la información (*pull*), definiendo el «ritmo» de adquisición de los datos. Debemos ver que en la variante *pull* de este patrón es bastante similar al patrón *request/response* que hemos visto anteriormente.

Este patrón difícilmente se entiende sin un agente de intermediación (observemos la figura 5), ya que se requiere cierto procesado previo inicial y almacenamiento temporal de los datos. El principal papel de este agente o *broker* es el desacoplado entre la producción y el consumo de los datos, actuando como *buffer* para adaptar las velocidades del productor y consumidor. Adicionalmente, y no menos importante, los agentes de intermediación permiten que varios consumidores usen los mismos datos de forma muy eficiente.

En el siguiente esquema podemos ver el rol del *broker* como gestor de colas de mensajes, permitiendo modular los mensajes que son consumidos por los consumidores y siendo capaz de, no solo gestionar el *buffer* con los mensajes que se van generando, sino también realizar agrupaciones y clasificaciones de los mensajes que se van suscribiendo.

Figura 5. Patrón publicación/subscripción. El *broker*



Fuente: elaboración propia.

Existen una gran variedad de protocolos que implementan el patrón de publicación/subscripción: MQTT, AMQP, XMPP o STOMP. Cada uno de estos patrones está especialmente diseñado para ser eficiente en un campo de utilización. Por ejemplo, el protocolo MQTT se utiliza en entornos IoT, mientras que AMQP es ampliamente utilizado en entornos de mercados financieros.

MQTT

Message Queue Telemetry Transport (también conocido como *MQ Telemetry Transport*).

AMQP

Advanced Message Queuing Protocol, es un protocolo de transmisión binario.

Para que entendamos un poco hasta qué punto son importantes los factores que contextualizan el campo de utilización en relación a la eficiencia de esta comunicación, podemos ver que la implementación de **MQTT** (protocolo *Open Source* creado por IBM para el sector industrial) solo ofrece a los suscriptores una interfaz de comunicación muy básica: *connect*, *disconnect* y *publish*. Al estar orientado a IoT, en la mayor parte de los casos no se requiere mayor funcionalidad que enviar/recibir datos básicos como medidas de temperatura, presión, polución, etc. Adicionalmente, al asumir que los mensajes son muy ligeros en contenido, soporta una gran cantidad de suscripciones al servicio de forma simultánea.

AMQP es un protocolo que se centra en la fiabilidad y la interoperabilidad. Utiliza una cabecera con propiedades y un cuerpo con el mensaje. Permite enrutamiento flexible, colas confiables, transacciones y seguridad. AMQP se puede utilizar en casos de uso más amplios y complejos que MQTT donde se requiera transaccionalidad (en el mismo sentido que las transacciones y *rollbacks* en bases de datos) y una mayor granularidad en el acceso a las colas y la gestión de su profundidad.

De manera diametralmente opuesta encontramos **STOMP** (*Simple Text Oriented Message Protocol*), que está orientado al intercambio de mensajes en formato texto (texto plano, XML o JSON), a diferencia de MQTT y AMQP, que son protocolos binarios. Dado que está pensado para aplicaciones bastante más

complejas que MQTT, el conjunto de operaciones que ofrece es también mucho más rico, y tiene como ventajas la simplicidad, interoperabilidad y compatibilidad con HTTP. Al igual que AMQP, STOMP proporciona un encabezado de mensaje (o marco) con propiedades y un cuerpo de marco. Aquí, los principios de diseño fueron crear un protocolo simple y ampliamente interoperable.

Ejemplo

Es posible conectarse a un agente STOMP utilizando software tan simple como un cliente telnet (es el nombre de un protocolo de red que nos permite acceder a otra máquina para manejarla remotamente como si estuviéramos sentados delante de ella).

El protocolo STOMP es ampliamente similar a HTTP y funciona a través de TCP utilizando los siguientes comandos:

- Connect
- Send
- Subscribe
- Unsubscribe
- Begin
- Commit
- Abort
- Ack
- Nack
- Disconnect

Vemos como las operaciones de *commit* y *abort* permiten validar transacciones y hacer *rollbacks* de forma parecida a la que se usa en SQL.

STOMP no se ocupa de la gestión de colas de mensajes ni de los *topics*: utiliza una semántica de enviar (*send*) con una cadena de «destino». El *broker* debe mapear datos que entienda internamente, como un asunto, una cola o un intercambio. Los consumidores se suscriben (*subscribe*) a esos destinos.

3.6. Patrón unidireccional

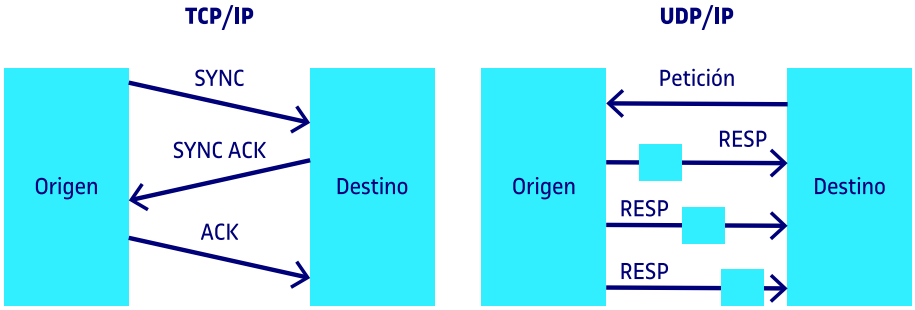
El **patrón unidireccional (*one-way*)** se utiliza en aquellos sistemas donde la petición del cliente no necesita una respuesta del servicio. En este caso no existirá una garantía de que el mensaje ha sido recibido por el servicio.

En este modelo se usa habitualmente el protocolo UDP. Mayoritariamente su uso se centra en determinados servicios de red como, por ejemplo, el servicio NTP (*Network Time Protocol*), y en algunas aplicaciones de IoT. Centrándonos en el **protocolo NTP**, que se utiliza para sincronizar relojes en entornos de sistemas informáticos, podemos asumir que una vez el sistema está sincronizado, no hay problema si se pierde una fracción pequeña de paquetes, ya que en general los relojes continuarán sincronizados por un periodo relativamente

largo de tiempo. Cuando este tipo de patrón está orientado a la captura de datos, raramente se usan en entornos con mensajes complejos y sus usos están relegados a aplicaciones IoT en las que la pérdida de paquetes no sea crítica.

La mayoría de los patrones unidireccionales utilizan como protocolo de la capa de transporte UDP en vez de TCP. Es importante recordar que UDP se trata de un protocolo que está basado en el intercambio de datagramas a través de la red sin que se haya establecido una negociación de conexión entre dispositivos IP, siendo el propio datagrama el que incorpora suficiente información de direccionamiento en su cabecera. Tampoco tiene confirmación ni control de flujo, por lo que los paquetes pueden adelantarse unos a otros; y tampoco se sabe si ha llegado correctamente, ya que no hay confirmación de entrega o recepción. Últimamente está adquiriendo mucha importancia por ser el patrón elegido por HTTP/3 y QUIC, los cuales utilizan tramas UDP.

Figura 6. TCP *vs*. UDP, este último corresponde al patrón unidireccional



Fuente: elaboración propia.

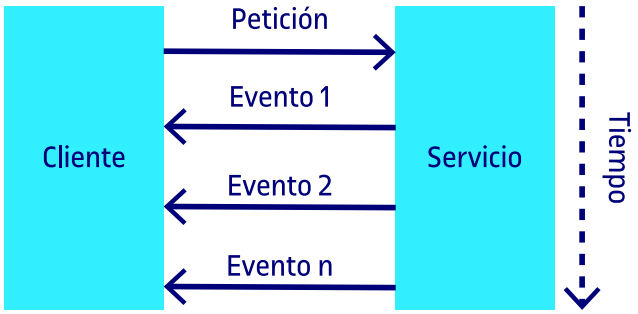
3.7. Patrón de flujo

En el **patrón de flujo** (***stream***) el servicio envía los datos al cliente de manera inversa al funcionamiento del patrón petición/respuesta.

Normalmente el proceso se inicia con una petición/respuesta clásica para que después el servicio empiece a enviar datos en *streaming*. A partir de ese momento el cliente va adquiriendo datos definiendo el ritmo de producción de los mensajes en el productor, que va generando datos continuamente. Es decir, hasta que el cliente no solicita datos, no los recibe. Se utiliza cuando es necesario que los clientes consuman los datos cuando los necesitan, no cuando no los necesitan, bien porque no los quieren o porque aún están procesando los anteriores. El principal objetivo de este patrón de flujo es evitar una saturación al intentar procesar todos los datos que pueden llegar a enviar los productores y solucionar problemas de latencia[4]. Ejemplos clásicos de este patrón los encontramos en seguimiento de sensores IoT, o *logs* de acceso en servidores web.

[4]Suma de retardos temporales dentro de una red producidos por la demora en la propagación y transmisión de paquetes dentro de la red.

Figura 7. Patrón de flujo



Fuente: elaboración propia.

En este terreno podemos encontrar a:

- Apache Spark Streaming (que permite procesamiento de lotes y de *stream* en una arquitectura Lambda).

- Apache Storm (orientado a procesar y dispensar los mensajes que llegan tan pronto como sea posible).

- Spring Cloud Data Flow (diseñado para la ingestión de datos, análisis en tiempo real, procesamiento por lotes y exportación de los mismos).

También podemos catalogar en esta categoría al protocolo CoAP (un protocolo de transferencia web especializado que se utiliza en redes restringidas y nodos para aplicaciones de máquina a máquina). CoAP tiene varias extensiones, entre la que encontramos el observador de recursos, que permite hacer un *push* de las representaciones de recursos de los servidores a los clientes interesados.

En este momento no nos extenderemos más en describir el patrón de *streaming* ni sus utilidades, ya que vamos a verlo en detalle en los siguientes módulos.

4. La importancia de la capa de cola de mensajes

En aquellos sistemas que implementan el patrón de publicación/subscripción, es muy importante considerar aquellos aspectos relacionados con la seguridad y la tolerancia de fallos. En las arquitecturas de la nube modernas, las aplicaciones se desacoplan en bloques pequeños e independientes que son más fáciles de desarrollar, implementar y mantener. Las colas de mensajes proporcionan la comunicación y la coordinación para estas aplicaciones distribuidas.

Las colas de mensajes pueden simplificar de forma significativa la escritura de código para aplicaciones desacopladas y, a la vez, mejorar el rendimiento, la fiabilidad y la escalabilidad.

También se pueden combinar colas de mensajes con mensajería de publicación-suscripción en un patrón de diseño de distribución ramificada.

4.1. Software que implementan infraestructuras MOM (*Message-oriented middleware*)

- **Apache Kafka**: es un proyecto de intermediación de mensajes de código abierto desarrollado por LinkedIn y donado a la Apache Software Foundation. Kafka está escrito en Java y Scala. Está basado en el patrón publicación/subscripción y proporciona una plataforma unificada, de alto rendimiento y de baja latencia para la manipulación en tiempo real de orígenes de datos. Vamos a estudiar Kafka en detalle en módulos posteriores.

- **RabbitMQ**: es un software de negociación de mensajes (MOM) de código abierto que utiliza un sistema de colas de mensajes (MQ) que actúa de *middleware* entre productores y consumidores.

- **Amazon Kinesis**: homólogo de Kafka para la infraestructura Amazon Web Services.

- **Microsoft Azure Event Hubs**: homólogo de Kafka para la infraestructura Microsoft Azure.

- **Google Pub/Sub**: homólogo de Kafka para la infraestructura Google Cloud.

5. Almacenaje de datos adquiridos por *streaming*

El almacenaje de los datos adquiridos puede ser en dos contextos:

1) Por una parte, y para facilitar el trabajo de la cola de mensajes, hay sistemas que realizan un **almacenaje de estos datos en memoria**. Este es el caso de Redis, una base de datos clave-valor que trabaja en memoria (aunque puede tener persistencia). Algunos sistemas de gestión de colas de mensaje, como el caso de Kafka, también incorporan sistemas de almacenaje en memoria de los mensajes.

2) Por otro lado, los sistemas finales de persistencia de datos pueden ser **sistemas de ficheros distribuidos** como Hadoop HDFS, Amazon S3, o bien en **bases de datos NoSQL**, las cuales, por sus características de escalabilidad y tolerancia a fallos, son idóneas para el tratamiento de datos masivos. Algunas de las bases de datos más utilizadas son: HBase, Apache Cassandra o MongoDB.

6. Caso de uso en la adquisición de datos en *streaming* basado en el patrón petición/respuesta

6.1. Contexto

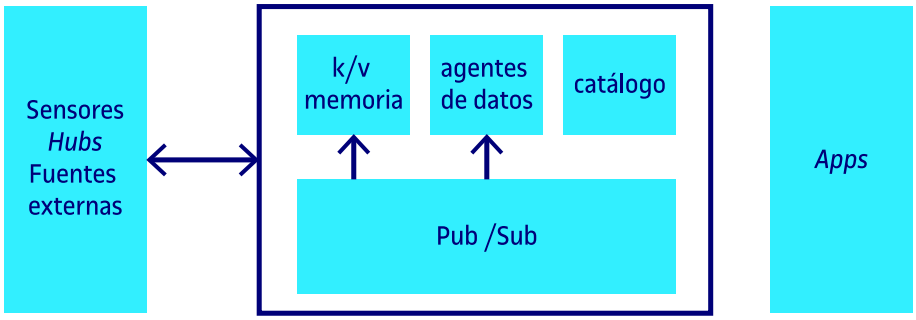
A continuación, vamos a analizar una aplicación IoT que utiliza el modelo de captura basado en el patrón de petición/respuesta para la adquisición de datos. La plataforma **Sentilo** es un software libre multiplataforma, orientado a la gestión de entornos IoT, diseñado con el objetivo de compartir información entre sistemas heterogéneos e integrar fácilmente aplicaciones heredadas. Sentilo se inició en noviembre de 2012 por parte del Ayuntamiento de Barcelona, a través del Instituto Municipal de Informática (IMI), para su proyecto de ciudad inteligente llamado City OS. Se trata de una plataforma que hace uso del patrón petición/respuesta, utilizando una API REST, en el envío de información de los varios sensores y actuadores distribuidos por la ciudad.

6.2. Detalles técnicos

Esta plataforma dispone de un *front-end* ejecutado sobre Apache Tomcat para el procesamiento de mensajes, con una API REST simple. La información se organiza en un catálogo con la jerarquía proveedor, componente y sensor, y mediante una consola de administración se puede configurar el sistema y administrar el catálogo.

Se utiliza una base de datos NoSQL Redis (una base de datos en tiempo real en memoria, haciendo uso de un mecanismo de publicación/subscripción), para la gestión de datos de memoria, destinada a lograr altas tasas de rendimiento, y MongoDB (otra base de datos NoSQL), para el almacenaje de los datos de los sensores de manera persistente.

Figura 8. Diseño arquitectura Sentilo



Fuente: elaboración propia.

Si nos centramos en los aspectos de captura de datos, la publicación de cada dato se realiza mediante HTTP/1.1.

Enlace recomendado

Para completar los conocimientos en el uso del protocolo HTTP podéis revisar: https://bit.ly/34ZlTYx

```
$ curl -X PUT -H "IDENTITY_KEY: <your provider's token>" http://<your sentilo url>/data/<your provider>/<your sensor>/42.0
```

obteniendo una respuesta `HTTP/1.1 200 OK`.

Si lo que se desea es leer una de las observaciones registradas en Sentilo, el formato de la petición HTTP debe ser:

```
$ curl -X GET -H "IDENTITY_KEY: <YOUR_KEY>" http://<your sentilo url>/data/<your provider>/<your sensor>
```

y en este caso la respuesta es:

```
{
  "observations":
 [{"value": "42.0", "timestamp": "22/11/2016T11:52:28", "location": ""}]
}
```

6.3. Ventajas e inconvenientes

Se trata de una arquitectura de componentes extensible, que permite ampliar la funcionalidad de la plataforma sin modificar el sistema central. Sentilo comienza con un conjunto inicial de agentes: uno para exportar datos a bases de datos relacionales y otro para procesar alarmas internas basadas en reglas básicas. El sistema no necesita configurar ningún agente ni otros componentes opcionales como Elasticsearch u OpenTSDB. Dado que utiliza para el registro de información el patrón de petición/respuesta, es muy sencillo de implementar en dispositivos de pocos recursos como son los de IoT, o realizar pruebas o envíos directamente con `curl` o `postman`, como hemos visto anteriormente. Otra ventaja importante es el hecho de utilizar una base de datos como MongoDB, dado que permite un repositorio de datos flexible y escalable horizontalmente porque los campos de los sensores pueden ser muy variables. Finalmente, indicar que como todo el sistema está desarrollado con Java, se puede desplegar en múltiples entornos y sistemas operativos, ofreciendo un buen rendimiento en términos de concurrencia y estabilidad, pero no tanto en consumo de memoria.

Aunque el sistema está basado en componentes, no se basa en soluciones distribuidas como Apache Hadoop, Spark o Storm. En consecuencia, en entornos muy exigentes en cuanto a número de sensores, peticiones o proveedores, va a requerir un trabajo adicional para descomponer el sistema en diversas ins-

talaciones con diversos servicios o en disponer de potentes servidores. Si estos sensores enviasen una cantidad más elevada de información en tiempo o cantidad, sería necesario pensar en realizar extensiones y modificaciones a la arquitectura principal para poder reutilizarla. Sin embargo, Sentilo es un producto robusto que permite su integración con Kafka y otras soluciones.

7. Caso de uso en la adquisición de datos en *streaming* basado en el patrón *stream*

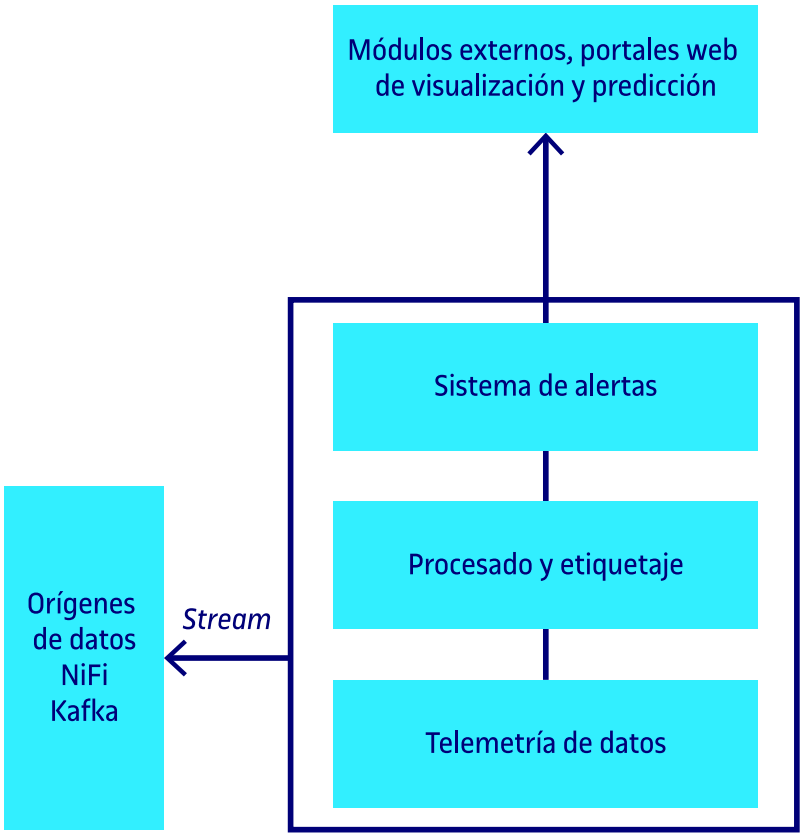
7.1. Contexto

En este caso de uso vamos a analizar el modelo de captura basado en el patrón *stream* contextualizado a los entornos de ciberseguridad.

Apache Metron proporciona un marco de análisis de seguridad avanzado y escalable creado en colaboración con la Comunidad Hadoop que evoluciona del Proyecto Cisco OpenSOC. Apache Metron ofrece un marco de aplicación de seguridad cibernética que brinda a las organizaciones la capacidad de detectar anomalías cibernéticas y permite a las organizaciones responder rápidamente a las anomalías identificadas. Permite una visión unificada de diversos datos de seguridad en *streaming* y ofrece alta escalabilidad para ayudar a los centros de operaciones de seguridad a detectar y responder rápidamente a las amenazas. Utiliza Apache Storm como elemento central que se caracteriza por hacer uso del patrón *stream* en la adquisición de datos.

En la figura 9 podemos ver el flujo de datos y su tratamiento en la plataforma.

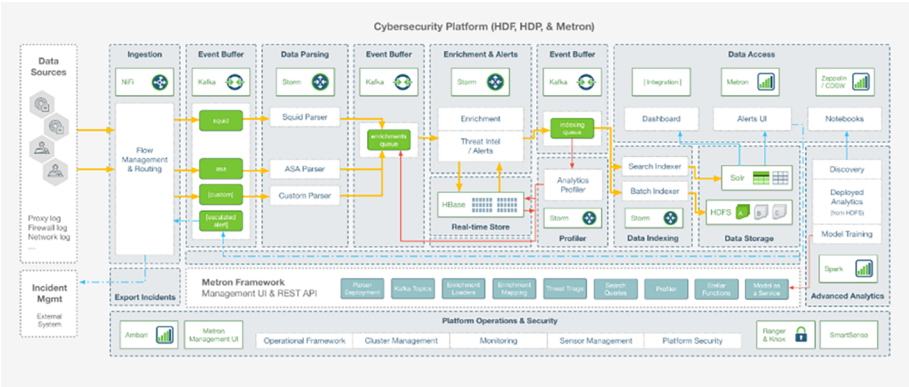
Figura 9. Diseño arquitectura Apache Metron



Fuente: elaboración propia.

Así mismo, en el siguiente diagrama (figura 10) se muestran los componentes lógicos de Apache Metron en una plataforma de ciberseguridad.

Figura 10. Componentes lógicos Apache Metron



Fuente: https://medium.com/@laurencedaluz/anatomy-of-a-streaming-data-engineering-pipeline-c13c0211067e

7.2. Detalles técnicos

Metron es un *framework* desarrollado en Java que proporciona capacidades para la agregación de registros, indexación de captura de paquetes completos, almacenamiento, análisis de comportamiento avanzado y enriquecimiento de datos, al tiempo que aplica la información de inteligencia de amenazas más actualizada a la telemetría de seguridad dentro de una sola plataforma.

El funcionamiento, en el caso de adquirir una nueva telemetría, pasaría por las etapas de configurar un nuevo sensor, capturar los registros del mismo, canalizarlos a Kafka, y recoger los registros con una topología de análisis de Metron. Posteriormente, estos datos podrían ser analizados a través de la tubería de procesamiento de flujo de Metron.

Como se trata de un sistema orientado a la seguridad informática un sensor podría ser Squid Proxy. En este caso, en el directorio `/var/log/squid` encontramos el fichero access.log, que contiene un listado con todas las entradas que han sido configuradas previamente. Después debemos indicar a Metron que utilizará Kafka para ingestar determinados *topics* desde Squid. Ahora estamos listos para abordar la configuración de topología de análisis de Metron. Para ello, debemos decidir si usaremos el analizador basado en Grok para la nueva telemetría. El analizador Grok es perfecto para registros estructurados o semiestructurados. Cabe indicar que estos ficheros normalmente se encuentran en el sistema de ficheros distribuidos como HDFS (Hadoop), dado el gran volumen de datos Metron lee los datos de HDFS, estos datos se transfieren mediante patrón *stream*. Finalmente, los datos parseados por Grok quedan organizados en HDFS y pueden ser visualizados por el panel de control de Metron.

Squid

Es un *proxy* de almacenamiento en caché para la web que admite diversos protocolos.

Apache Metron consta de cuatro funciones clave:

1) *Data lake* de seguridad para almacenar una amplia variedad de datos empresariales junto con datos de seguridad, telemetría mejorada y datos PCAP durante periodos de tiempo prolongados.

2) Marco modular, que ofrece un completo conjunto de analizadores para fuentes de datos de seguridad comunes (`pcap`, `netflow`, `bro`, `snort`, `fireye` y `sourcefire`), pero también proporciona un marco modular para añadir nuevos analizadores personalizados. También se pueden añadir modelos de aprendizaje automático y de otros tipos a la adquisición de datos en tiempo real. Para ello utiliza Apache NiFi, un proyecto de software de Apache Software Foundation diseñado para automatizar el flujo de datos entre sistemas de software. Se basa en el software NiagaraFiles desarrollado previamente por la NSA. También pueden crearse orígenes personalizados e integrarlos con Kafka.

3) Plataforma de detección de amenazas, basada en algoritmos de aprendizaje automático y detección de anomalías que se pueden aplicar en tiempo real a medida que se van recibiendo las transmisiones de eventos. Este procesado se construye con la tecnología STORM que veremos a lo largo de los módulos.

4) Aplicación de respuesta a incidencias, que es una evolución de las funciones SIEM (alerta, marco de inteligencia de amenazas, agentes para la incorporación de fuentes de datos...).

Como hemos indicado, Metron incorpora Apache Hadoop para el almacenaje de datos distribuidos y Apache Storm para la adquisición y procesado de datos por *stream*.

7.3. Ventajas e inconvenientes

Utilizar arquitecturas distribuidas como Hadoop y Storm permite a Metron un funcionamiento excelente en la adquisición de datos en entornos *big data*, y haciendo uso del patrón *stream* permite garantizar que el sistema sea robusto y capaz de adquirir los datos necesarios para identificar amenazas de ciberseguridad. Su inconveniente principal es la complejidad de su instalación, su exigencia en requerimientos de infraestructura y su dependencia de otras soluciones *big data* de Apache.

Analizada esta arquitectura en relación a lo visto en Sentilo, vemos que en la mayor parte de ocasiones debemos encontrar un equilibrio entre fiabilidad y robustez con la complejidad y requerimientos computacionales necesarios para ejecutar el sistema.

Resumen

A modo de sumario, en este módulo hemos visto en qué consiste el proceso de captura de datos en *streaming* y dónde está ubicado en una arquitectura *big data*. Se han explicado y desarrollado los diferentes patrones que permiten realizar la ingestión de datos y se han presentado dos casos de uso para los más ampliamente utilizados:

- El patrón petición/respuesta (*request/response*) se usa cuando el cliente espera una respuesta realizada por un servicio en el momento que se ha solicitado. Se trata del patrón más simple y que utiliza dos fases: en la primera fase (*request*) el agente que inicia la comunicación envía un mensaje al segundo agente solicitando cierto tipo de información. Posteriormente, en la fase *response*, el segundo agente devuelve la respuesta al primero. Este patrón se puede implementar de manera síncrona o asíncrona.

- El patrón petición/acuse (*request/acknowledge*) es similar al de petición/respuesta pero sin la necesidad de una respuesta con contenido por parte del servicio, siendo solo necesario el conocimiento que la petición ha sido recibida (se recibe un mensaje ACK), y el primer agente continúa operando sin esperar una respuesta con contenido del mismo.

- El patrón publicación/subscripción (*publish/subscribe*) es el más ampliamente utilizado en entornos de captura de datos y consiste en que un productor o productores (*publisher*) envían mensajes a un/unos consumidores (*subscriber*) que pueden clasificarse o agruparse en relación a un asunto. Normalmente esta comunicación se realiza mediante un agente de intermediación (*broker*) para mejorar el rendimiento y la consistencia en el proceso de captura de datos.

- El patrón unidireccional (*one-way*) se utiliza en aquellos sistemas donde la petición del cliente no necesita una respuesta del servicio. En este caso no existirá una garantía de que el mensaje ha sido recibido por el servicio. En este modelo generalmente se usa el protocolo UDP y se implementa en determinados servicios de red como, por ejemplo, el servicio NTP (*Network Time Protocol*), y en algunas aplicaciones de IoT.

- Finalmente, el patrón de flujo (*stream*) consiste en el envío de los datos al cliente de manera inversa al funcionamiento del patrón petición/respuesta. Normalmente, el proceso se inicia con una petición/respuesta clásica para que después el servicio empiece a enviar datos en *streaming*. A partir de ese momento, el cliente va adquiriendo datos definiendo el ritmo de producción de los mensajes en el productor, que va generando datos continuamente. Se utiliza cuando es necesario que los clientes consuman los

datos cuando los necesitan, no cuando no los necesitan, bien porque no los quieren o porque aún están procesando los anteriores.

Bibliografía

Ajuntament de Barcelona. Sentilo, documentación técnica. https://sentilo.readthedocs.io/en/latest/

Atlassian Confluence Open Source Project. https://cwiki.apache.org/confluence/display/METRON/Metron+Architecture

Gilbert, A.; Kotidis, Y.; Muthukrishnan, S.; Strauss, M. (2001). «Surfing wavelets on streams: One pass summaries for approximate aggregate queries». *VLDB Journal*, núm. 79–88.

Google. «¿Qué es Pub/Sub?». Data Analytics Products, Cloud Pub/Sub. https://cloud.google.com/pubsub/docs/overview

Hoff, T. (2020). http://highscalability.com/

Julbe, F.; Conesa, J.; Casas, J.; Rodríguez, M. E. (2019). *Captura, pre-processament i emmagatzematge de dades massives*. Barcelona: UOC.

Muthukrishnan, S. (2005). *Data streams: Algorithms and applications*. Hannover: Now Publishers Inc.

NSA/CSS Communications Officer. *Niagarafiles Keep Data Flowing*. www.nsa.gov/News-Features/News-Stories/Article-View/Article/1830203/niagarafiles-keep-data-flowing/

Psaltis, A. G. (2017). *Streaming Data: Understanding the Real-Time Pipeline*. Nueva York: Manning Publications.