
**Treball de Fi de Grau**

**GRAU D'ENGINYERIA INFORMÀTICA**

**Facultat de Matemàtiques**
**Universitat de Barcelona**

---

# PROPOSTA PER A UNA ARQUITECTURA ESCALABLE D'INGESTA DE MISSATGES

---

**Oriol López Sánchez**

Director: Lluís Garrido Ostermann
Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB

Barcelona, Juny de 2016

# Abstract

Our world is becoming more interconnected with everything. As time passes, more electronic components emit information to the Internet. Mobile devices, intelligent wearables, beacons, any kind of machine with enough electricity to connect to the Internet may send information of its surroundings.

At Beabloo, we strive to achieve a higher understanding of the environment these devices describe us by listening them. As the amount of these devices increases, harnessing such high amount of information becomes difficult, and we believe it is necessary to study new ways that allow us scale up horizontally, and allows us to bring such information as fast as possible to our data processing pipelines, which then are being used to calculate Key Performance Indicators that will be used for decision making at higher level.

This paper describes a design of a data ingression architecture, keeping in mind previous design flaws that slowed down ingestion capacity. Its objective is, by keeping a small and minimalistic architecture design, to allow attaining an almost real-time data processing performance.

The project uses Apache Kafka, and Apache Storm to achieve a new real-time execution environment, and to be able to evaluate its performance, it uses Prometheus to store performance metrics and Grafana to visualize them with ease.

# Resum

El nostre món cada vegada està més interconnectat. A mesura que el temps avança, més components electrònics emeten informació cap a Internet. Dispositius mòbils, roba intel·ligent, «beacons», qualsevol màquina amb prou electricitat per a poder connectar-se a Internet enviarà informació del seu voltant.

A Beabloo, ens esforcem per aconseguir un major enteniment de l'entorn que aquests dispositius descriuen escolten-los. Així com la quantitat d'aquests dispositius augmenta, l'aprofitament de tanta informació s'esdevé complicat, i creiem que és necessari estudiar noves maneres que ens permetin escalar horitzontalment, i permetre'ns transportar tota aquesta informació als nostres «pipelines» de processament de dades, que seran usats per a calcular Indicadors Clau de Rendiment, o «KPI», per aplicar decisions a alt nivell.

Aquest paper descriu un disseny d'una arquitectura d'ingestió de dades, tenint en compte els errors de disseny previs que alentien la capacitat d'ingesta. El seu objectiu és, mentre que manté un disseny arquitectònic petit i minimalista, permetre aconseguir un rendiment proper a temps real a l'hora de processar dades.

Aquest projecte utilitza Apache Kafka, i Apache Storm per aconseguir un entorn d'execució proper a temps real, i per avaluar el seu rendiment, s'utilitza Prometheus per a desar mètriques de rendiment i Grafana per a visualitzar-les.

# Resumen

Nuestro mundo cada vez está más interconectado. A medida que el tiempo avanza, más componentes electrónicos emiten información hacia Internet. Dispositivos móviles, ropa inteligente, «beacons», cualquier máquina con suficiente electricidad para poder conectarse a Internet enviará información de su alrededor.

En Beabloo, nos esforzamos para conseguir una mejor comprensión del entorno que estos dispositivos describen escuchándolos. Así como la cantidad de estos dispositivos aumenta, el aprovechamiento de tanta información se torna complicado, i creemos que és necesario estudiar nuevas maneras que nos permitan escalar horizontalmente, i permitirnos transportar toda esta información a nuestros «pipelines» de procesamiento de datos, que serán usados para calcular Indicadores Clave de Rendimiento, o «KPI», para aplicar decisiones a alto nivel.

Este documento describe un diseño de una arquitectura de ingestión de datos, teniendo en cuenta los errores de diseños previos ralentizaban la capacidad de ingesta. Su objetivo es, mientras que mantiene un diseño arquitectónico pequeño y minimalista, permitir conseguir un rendimiento cercano a tiempo real a la hora de procesar datos.

Este proyecto utiliza Apache Kafka, i Apache Storm para conseguir un entorno de ejecución cercano a tiempo real, i para evaluar su rendimiento, se utiliza Prometheus para guardar métricas de rendimiento y Grafana para visualizarlas.

# Taula de Continguts

# Taula de Continguts

# 1. Motivació

## 1.1 Context del projecte

Inicialment es va fer un estudi de l'estat actual de l'arquitectura d'ingesta de dades a la plataforma de Big Data que s'estava utilitzant, i observar quines millores es poden oferir, i quines mancances s'havien d'arreglar.

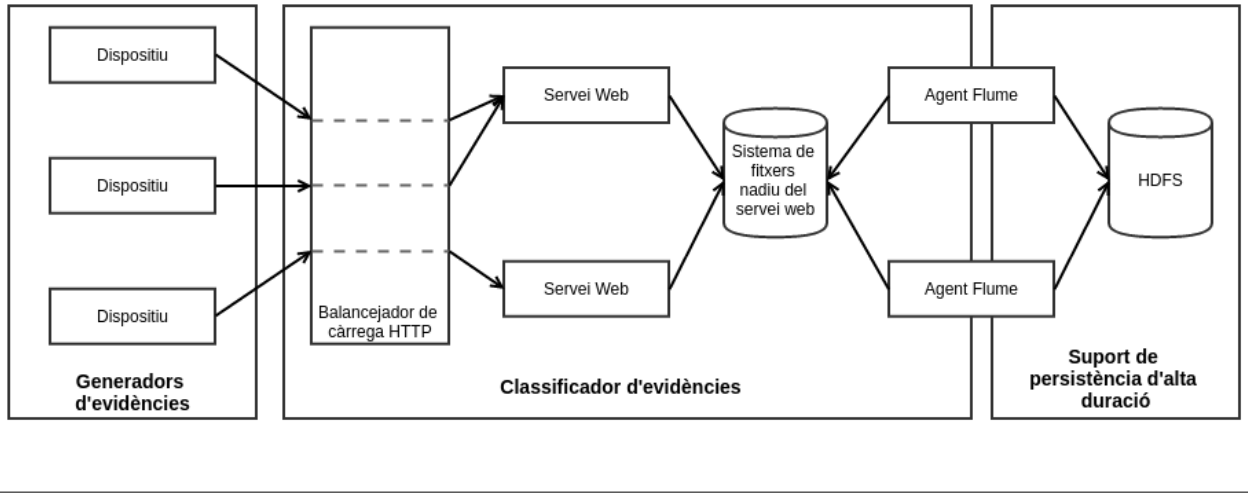


Figura 1.2.a. Esquema de la implementació actual sent usada

En la figura es pot veure de manera resumida el cicle de vida d'una evidència:

- En primer lloc, aquesta evidència és creada per un dispositiu, i aquest dispositiu acumula les evidències fins a un límit i les envia empaquetades.
- Un balancejador de càrrega decideix a quins dels servidors web serà processada el paquet d'evidències.
- Quan aquest paquet arriba al servei web, és processada, és a dir, es valida i es transforma l'evidència en una dada semi-estructurada, acceptable per al cluster, i la desa en el disc nadiu.
- Un agent de **Flume** monitora la zona de disc a on s'estan desant les dades, i les acaba enviant a l'HDFS del cluster de Hadoop.

Tot i ser una infraestructura bastant simple per a tractar el processament d'evidències, en compondre's de pocs elements, hi té algunes mancances que pot arribar a provocar greus

problemes de manteniment i de rendiment:

- La validació feta pel servidor web és extremadament costosa. S'han de fer un nombre gens petit de connexions a bases de dades per assegurar-se que l'evidència és correcte.

- És possible que, a causa de problemes puntuals de comunicació entre els agents de Flume i l'HDFS, que un paquet d'evidències sencer el dupliqui. Flume només assegura que la dada s'ha enviat, però no t'assegura quantes vegades la pot arribar a enviar.

- No hi ha cap component per a monitorar activament el processament dels paquets d'evidències. Si per alguna raó, un dispositiu deixa d'enviar evidències, actualment es triga un temps a saber que ha deixat de fer-ho.

- Les evidències no arriben prou aviat. Tot i que Flume es pot configurar exhaustivament per a definir com persistir les dades a l'HDFS, actualment, no és possible tenir les evidències amb prou rapidesa.

## 1.3 Objectius del projecte

Tenint en compte tots els factors descrits anteriorment, i atès que es vol donar una solució que resolgui els problemes mencionats, el que es proposa és el següent:

- Crear una nova infraestructura per a definir el processament d'evidències d'una manera escalable.

- Permetre que les evidències es puguin processar a un ritme ràpid, i que hi siguin disponible el més abans possible a l'HDFS.

- Monitorar els diferents entorns per a preveure potencials problemes, i permetre avaluar el rendiment dels diferents components de la infraestructura.

- Evitar duplicitat de processament d'evidències.

- Provar-ho tot en un entorn distribuït, sense el requeriment d'un «cluster» de Hadoop, evitant possibles problemes de hardware, xarxa o configuració dels diferents serveis de Hadoop.

Per això, el projecte s'enfocarà en donar una solució al tractament d'evidències de potències WiFi detectades en una zona. El projecte haurà de definir els conceptes i presentar les eines necessàries per a transformar aquestes potències detectades per dispositius a dades semi-estructurades desades a un HDFS.

# 2. Disseny i implementació

## 2.1 Terminologia

Per a poder entendre el context del problema a resoldre, hi ha una sèrie de conceptes que s'han d'explicar:

- **Dispositiu**. Un dispositiu és un emissor d'evidències. Aquest element envia dades que es volen guardar, per a ser processades posteriorment per extreure més informació d'aquestes evidències. Un dispositiu es pot comunicar en diferents protocols per a enviar les evidències que captura.

- **Evidència**. Una evidència és tota aquella informació que un dispositiu capta d'un esdeveniment del seu entorn en un instant del temps, i que és prou rellevant per a ser guardada per al seu posterior processament.

- **Protocol de comunicació**. És el conjunt de tipus de canal de comunicació, el tipus d'informació que es vol transmetre i la manera en què es transmet que defineix un protocol de comunicació.

- **«Logger»**. És un servei que s'encarrega d'escoltar un canal de comunicació pel qual un dispositiu emetrà dades. El logger s'encarrega d'introduir cada evidència o conjunt d'evidències al suport d'alta duració basat en Hadoop.

- **«Data Pipeline»**. És un conjunt de serveis que s'encarreguen de validar, i transformar les evidències enviades pels loggers per a ser desades finalment al format òptim per a ser usat per posteriors processaments d'aquestes dades.

- **«Data Lake»**. És el conjunt heterogeni de dades finalment processades pels diferents «data pipelines», contingudes en diferents suports, que poden ser explotats per diferents eines analítiques per a extreure mètriques agregades.

## 2.2 Estudi de tecnologies i arquitectura proposada

Abans de poder proposar una infraestructura adequada per atacar el problema, s'ha d'entendre que el problema té diferents variants que ens acabarà definint el programari necessari.

Es requereix un sistema de processament en temps real, a ser possible que sigui distribuït, per a permetre capacitat d'escalar la capacitat de processament en cas que hi hagi més demanda de recursos.

A més, es requereix un sistema per a poder integrar les dades entre diferents serveis existents, que permeti una integració ràpida i sense excessiva complexitat.

Un altre requeriment important és la capacitat de monitorar els serveis que s'usaran. Aquest component ha de ser autocontingut, i a més, fàcilment explotable per una eina de visualització de l'estat dels serveis.

### *2.2.1 Sistema de processament en temps real*

A l'ecosistema de Hadoop existeix una gran quantitat de solucions pel processament de dades en temps real. Tots aquests sistemes presenten el seu context d'execució distribuït per a processar les dades, i abstreuen la gran majoria de problemes d'alta concurrència i distribució de tasques, per a permetre definir la implementació de negoci que es requereix.

S'ha de tenir en compte que aquest problema és completament antagònic al tractament de dades en lot, com fa «MapReduce», o l'abstracció d'alt nivell basada en SQL, «Hive».

Els candidats que es proposen per a processar les dades en temps reals són els següents:

- «**Apache Spark**». A l'igual que «MapReduce», «Spark» és un «framework» per a dissenyar processaments de dades distribuïts. Però a on marca la principal diferència és l'ús intensiu de memòria per a desar les dades, i reutilitzar-les, per accelerar el càlcul que es vulgui fer sobre un set de dades. Suporta processament de dades en temps real, basat en el processament de petits lots de dades.

- «**Apache Storm**». A diferència de «Spark» i «MapReduce», «Storm» és un sistema complex processador d'esdeveniments distribuït. Està enfocat a tractar corrents de dades, a on s'envien grans quantitats de missatges amb molt baixa latència, i en principi, no està pensat per a tractar dades en lots.

- «**Apache Flink**». Si «Storm» està pensat per a processar esdeveniments en un corrent en temps real, i «Spark» està pensat per a processar les dades de manera distribuïda, però en lots, «Flink» és el framework que intenta unificar tots dos paradigmes, i tractar d'una manera genèrica tots dos problemes.

Finalment, es decideix utilitzar «**Storm**», atès que de les solucions proposades, sembla la més natural pel problema que es vol tractar, ja que no és en si un problema de gestió de memòria, sinó de distribució de processament de missatges.

És una solució que planteja una manera d'escalar bastant transparent, i es pot avaluar ràpidament si el rendiment degenera. I garanteix el tractament d'errors, i la possibilitat de reintentar les tasques en cas d'error.

### *2.2.2 Storm*

«Apache Storm» és un «framework» dedicat al processament d'esdeveniments en temps real. La idea es basa en definir una «topologia» de nodes que generen les dades, els «spouts», que seràn consumits per altres nodes, els «bolts», que generaran els resultats del processament de dades.

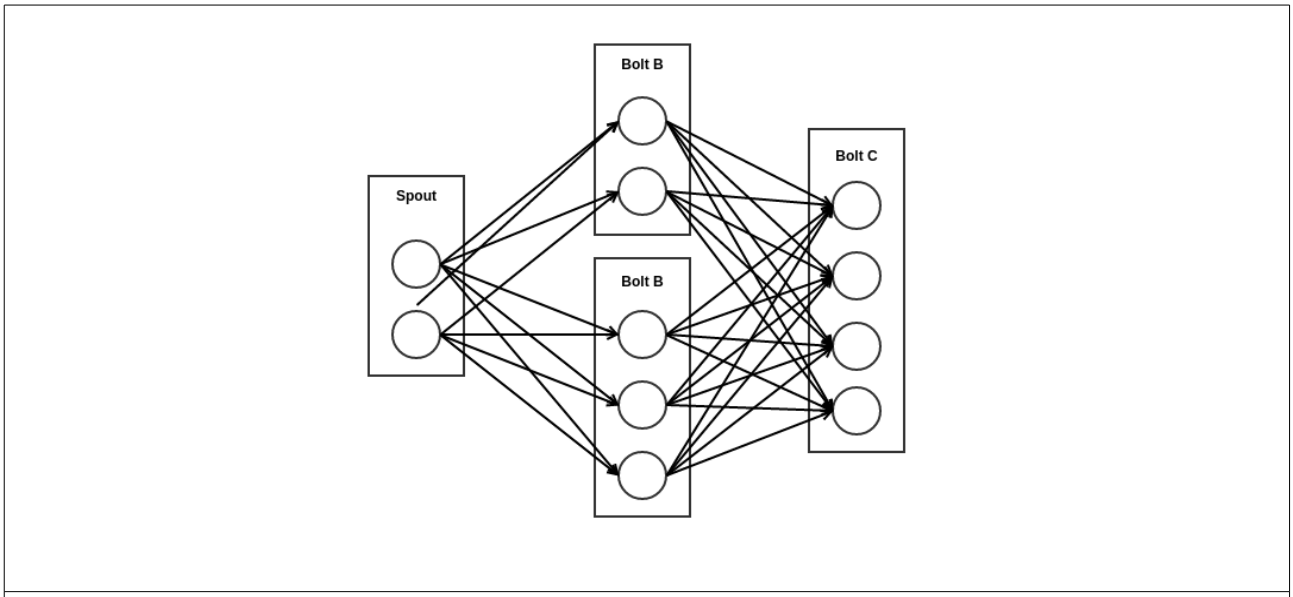


Figura 2.2.2.a Esquema d'una topologia

Per a entendre «Storm», s'ha d'introduir una sèrie de conceptes:

- Una «**topologia**» és un graf dirigit i acíclic a on cada node defineix una o diverses operacions de transformació sobre les tuples que travessen aquest graf.

- Un «**corrent**», o «**stream**», és una aresta en el graf definit en la topologia. És una col·lecció seqüencial de «tuples», creades en paral·lel, i processades de manera distribuïda.

- Una «**tupla**» és l'estructura de clau-valor, que conté la informació que es vol processar.

- El «**spout**» és un productor de «tuples». Envia tuples als diferents «corrents» als que estigui connectat.

- El «**bolt**» és el component processador, o consumidor, de «tuples», que acabarà generant noves «tuples» que les enviarà a altres «bolts».

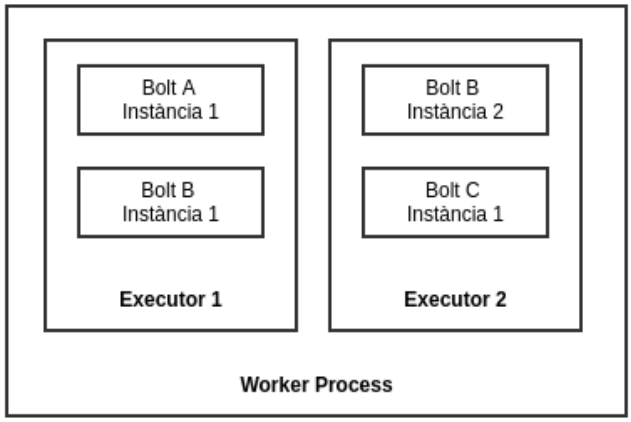


Figura 2.2.2.b Esquema dels «worker processes», «executors» i «tasks» de «Storm»

Amb la figura **2.2.2.b** es pot veure que tots els nodes d'una «**topologia**» són distribuïts com a tasques que seran executades per diferents «**executors**», o fils executors, en funció del grau de paral·lelisme que s'hagi configurat sobre aquell «spout» o «bolt», i aquests executors es distribueixen entre diferents «**worker processes**», o màquines físiques.

Per publicar una topologia a Storm, existeix un servei, anomenat «**Nimbus**», que s'encarrega de processar aquestes publicacions, i reparteix les tasques que s'hauran d'executar a cadascun dels «**supervisors**», les màquines que instanciaran «worker processes», disponibles al «cluster».

### *2.2.3 Sistema d'integració entre serveis: «Apache Kafka»*

A diferència dels sistemes de processament distribuït, pels sistemes de missatgeria per

comunicar diferents serveis entre si, no he vist alternatives clarament definides, en l'ecosistema de Hadoop, i que s'estigui utilitzant en entorns de producció, només he vist «Apache Kafka» com la possibilitat més sòlida per a comunicar serveis entre si mitjançant cues de missatges.

«**Apache Kafka**» és un sistema de publicació-subscripció de missatges distribuït per a processar grans quantitats de «corrents» de dades. És una implementació de cua de missatges que persisteix el seu estat en disc de manera molt optimitzada, i a més, està replicada entre els diferents nodes treballadors d'un «cluster» de «Kafka».

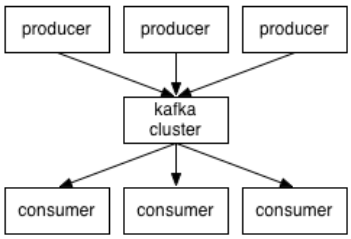


Figura 2.2.3.a Esquema de productor-consumidor de Kafka
(Extret de la documentació de Kafka)

«Kafka» manté una sèrie de missatges d'una categoria, que s'anomena «**topic**». Aquests missatges són produïts per un «**productor**», i aquests missatges són escoltats pels «**consumidors**», que es connecten a «Kafka» per a consumir-los. Aquestes cues poden estar a diferents servidors, anomenats «**broker**».

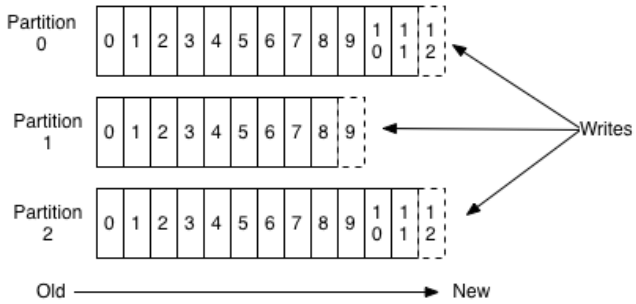


Figura 2.2.3.b Estructura bàsica d'un «topic»
(Extret de la documentació de Kafka)

Un «**topic**» és una col·lecció de missatges ordenats per ordre de recepció, tal com es pot veure a la figura **2.2.3.b.** Aquesta col·lecció està guardada en diferents particions que són mantingudes per diferents «**brokers**». Però, només un «**broker**» actua com a «**líder**», és a dir, el servidor al qual se li preguntarà les dades que té sobre un «**topic**». Tots els altres «**brokers**» poden actuar com a «**followers**», i contindran rèpliques de la partició que té el

«**líder**».

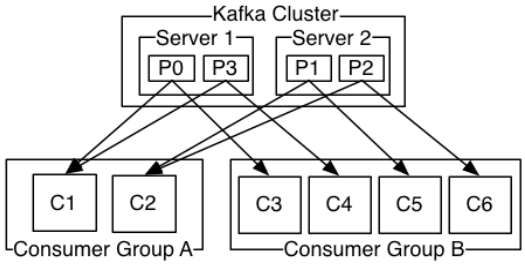


Figura 2.2.3.c Diagrama dels «consumer groups» d'un «topic»
(Extret de la documentació de Kafka)

Com es pot veure a la figura **2.2.3.c**, els «**consumidors**» s'agrupen en «**consumer groups**» que se'ls hi assigna una sola partició que poden consumir. D'aquesta manera s'assegura l'escalabilitat de consumició de missatges, a la vegada que es controla l'ordre de consumició.

### *2.2.4 Arquitectura proposada*

En l'arquitectura que es presenta en aquest projecte es proposen dos aplicacions per a assegurar que es pugui desar les evidències captades pels dispositius a un cluster de Hadoop:

- «**Log Gateway**». És la implementació d'un «logger» minimalista.
- «**Log Pipeline**». És la implementació d'un conjunt de «data pipelines» que s'encarreguen de desar les dades enviades pel «Log Gateway»

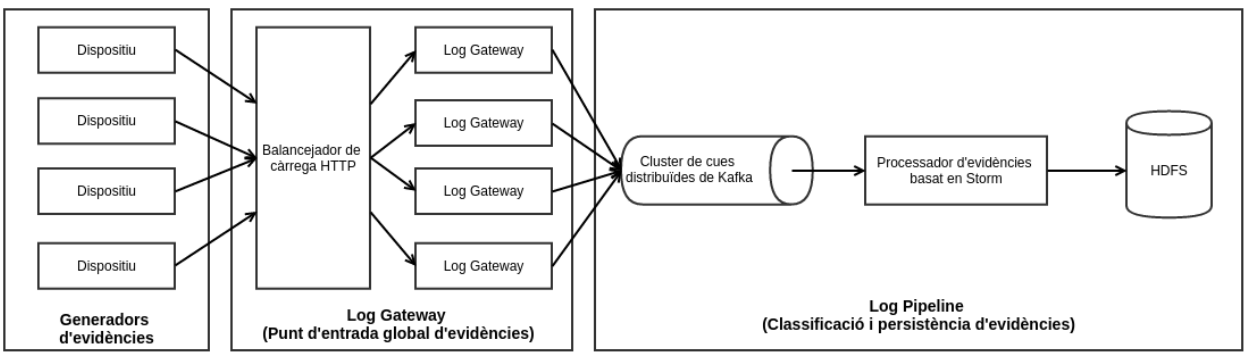


Figura 2.2.4.a. Esquema general de l'arquitectura presentada

Essencialment, el que mostra la figura **2.2.4.a** són les instàncies de «**log gateway**» són serveis web que reben les evidències enviades pels dispositius. Aquestes evidències són enviades a un «cluster» de cues distribuïdes de «**Kafka**», que són consumides per una instància de «log pipeline», que és una topologia de «**Storm**» que acaba validant i transformant les evidències en el tipus de dada que finalment volem desar en el nostre suport de persistència d'alta duració, en aquest cas **HDFS**.

### *2.2.5 Descripció estructural de la proposta presentada*

Aquesta proposta es compon d'una sèrie de projectes basats en Java, que implementen les diferents parts descrites anteriorment, gestionat amb Gradle, i generen una sèrie d'artefactes desplegables a un «cluster» de Hadoop:

- «**hadoop-common**». Aquesta llibreria conté models i implementacions transversals que s'usen tant a «**log-gateway**» com a «**log-pipeline**».

- «**log-gateway**». Aquest projecte genera un artefacte tipus «war» que es pot publicar en qualsevol contenidor de servlets com Tomcat, o Jetty per a processar les evidències enviades pels dispositius, i introduir-les al «log pipeline».

- «**log-pipeline**». Aquest projecte genera un artefacte tipus «jar» que conté la implementació de la topologia de Storm que processa les evidències enviades per un «Log Gateway».

- «**storm-hdfs**». Aquesta llibreria és utilitzada pel projecte «**log-pipeline**», i és oferida per Apache Storm per a implementar integracions d'HDFS dins Storm.

- «**scripts**». Aquesta carpeta conté una sèrie de petits programes fets en Python per a simular els dispositius que enviaràn dades als «**log gateways**».

A més de la implementació presentada, s'inclou el projecte **Structor** de Hortonworks, amb diverses modificacions per a poder crear configuracions de cluster virtualitzats i fer proves més exhaustives.

Al capítol sobre Experiments i Resultats s'explicarà amb més detall el seu us.

## 2.3 Protocol de comunicació YAELP

**YAELP** és l'acrònim de «**Yet Another Event Logging Protocol**», o «**Un altre protocol d'enregistrament d'esdeveniments**», és un protocol molt simple per a comunicar esdeveniments de qualsevol tipus que succeeixin al llarg del temps, i té com a propòsit demostrar la integració d'un protocol de comunicació en la infraestructura per a persistir evidències captades pels dispositius.

Aquest protocol només indica com un dispositiu ha d'enviar una evidència captada a un «Log Gateway».

La idea és que tota evidència es pot descompondre en dos components que defineixen la informació captada d'aquell esdeveniment en aquell moment del temps:

- L'**entorn**, o «**environment**». El context on es genera aquella evidència.
- El **disparador**, o «**trigger**». Aquella acció que genera l'evidència, i aporta la informació necessària per a ser desada.

La combinació d'aquests dos elements defineix un llistat de propietats, anomenat «definició d'evidència», cadascun d'aquests contenint un valor numèric, o una cadena de caràcters, descrivint quantitativament o textualment l'evidència.

Aquestes propietats és tota aquella informació mínima rellevant que descriu completament l'evidència, i que és necessari que existeixi. Per contra, pot arribar a existir informació menys rellevant, o que possiblement no existeix en aquell moment. Aquest tipus d'informació s'anomena «**metadades**».

### *2.3.1 Format d'un missatge en protocl YAELP*

S'utilitza el format «**JSON**» per a representar la informació captada d'una evidència, i segueix l'estructura que es pot veure a la figura **2.3.1.a**.

```
{
  "events": [
    {
      "meta": {
      },
      "data": {
        "trigger": ""
      }
    }
  ]
}
```

Figura 2.3.1.a. Diagrama de seqüència de crides de YAELP

El node «**event**» conté una llista d'un o més objectes amb dues propietats que descriuen un esdeveniment:

- El node «**data**» conté totes les propietats necessàries per descriure un esdeveniment. Dins ha de contenir necessàriament la propietat «**trigger**» per assegurar de quin tipus de disparador ha generat l'evidència.

- El node «**meta**» permet afegir més propietats, no obligatòries.

Donat que pot contenir un conjunt de diferents tipus d'evidències per a un mateix context, segons l'acció que hagi provocat l'evidència, només es pot enviar en un mateix paquet d'evidències totes aquelles accions que siguin del mateix entorn d'on s'han originat les evidències.

Per a l'enviament d'aquests contenidors d'evidències, s'utilitza el protocol «**HTTP**». Quan un dispositiu vol enviar-les, pot fer-ho mitjançant una petició «**GET**» o una petició «**POST**» a un «**log gateway**», i aquest li enviarà una resposta «**HTTP**» amb codi 200, si s'ha pogut desar l'evidència correctament.

### *2.3.2 Exemple de protocol: Captació d'evidències Wifi*

En el problema presentat, existeix una sèrie de dispositius receptors que capten les potències de transmissió a la xarxa Wifi de les antenes dels dispositius mòbils que passen al voltant dels receptors. Aquests receptors, quan capten les potències, les guarden i eventualment seran enviades per a ser desades.

La figura **2.3.2.a** ens mostra un exemple d'una evidència utilitzant el protocol YAELP.

```
{
  "events": [
    {
      "meta": {
      },
      "data": {
        "trigger": "3",
        "organization": 39159,
        "startEvent": 1451606400,
        "hotspot": 57649,
        "sensor": 5169,
        "device": "6393cfc4ab6d416690b21278957aeab2",
        "oui": "AA:BB:CC",
        "power": -72
      }
    }
  ]
}
```

Figura 2.3.2.a. Exemple de l'estructura d'una evidència d'una presència d'un dispositiu mòbil detectat per un receptor.

Aquesta estructura descriu l'evidència captada indicant:

- **Qui ha escoltat l'evidència**. Els camps «**organization**», «**hotspot**» i «**sensor**» indiquen de quina organització, i en quin grup de sensors, i quin sensor ha detectat l'evidència.

- **Qui ha sigut escoltat**. Els camps «**device**» i «**oui**» descriuen el dispositiu mòbil, i el fabricant que ha sigut detectat.

- **Quan s'ha escoltat l'evidència**. El camp «**startEvent**» és una marca de temps en format Unix que indica el moment en que s'ha generat l'evidència.

- **Què s'ha escoltat**. El camp «**power**» indica la potència de transmissió amb la qual el sensor del receptor capta el dispositiu mòbil.

## 2.4 Log Gateway

El «**Log Gateway**» és un servei dedicat a escoltar les evidències enviades pels dispositius. Se li anomena «**gateway**», o porta d'entrada, pel fet que aquest component es troba a la frontera, o l'inici del cicle de vida d'una evidència que serà desada al «**data lake**».

Per tant, els objectius d'aquest component són els següents:

- **Oferir una implementació simple**, i una interfície que permeti el desenvolupament ràpid i sense complicacions d'integracions amb serveis externs.

- **Assegurar que totes les evidències puguin arribar als «data pipelines»**, i evitar que es puguin perdre, independentment de la correctesa de l'evidència.

- Permetre una **alta capacitat de consumició d'evidències**.

A diferència de la implementació actual, la nova presentada intenta simplificar tot el seu disseny, centrant-se en les úniques responsabilitats que té. Aquest disseny està basat en el microserveis, on cadascuna de les peces encarregades de canalitzar les dades pel cau adequat és un servei, o implementació monolítica, que permet la seva interacció a través d'una comunicació **HTTP** o mitjançant algun tipus de **RPC**.

### *2.4.1 Cicle de vida d'una evidència en el «Log Gateway»*

Quan es processa les evidències d'un dispositiu, la primera prioritat, i la més important, es assegurar que l'evidència queda desada en un suport de persistència de curta vida.

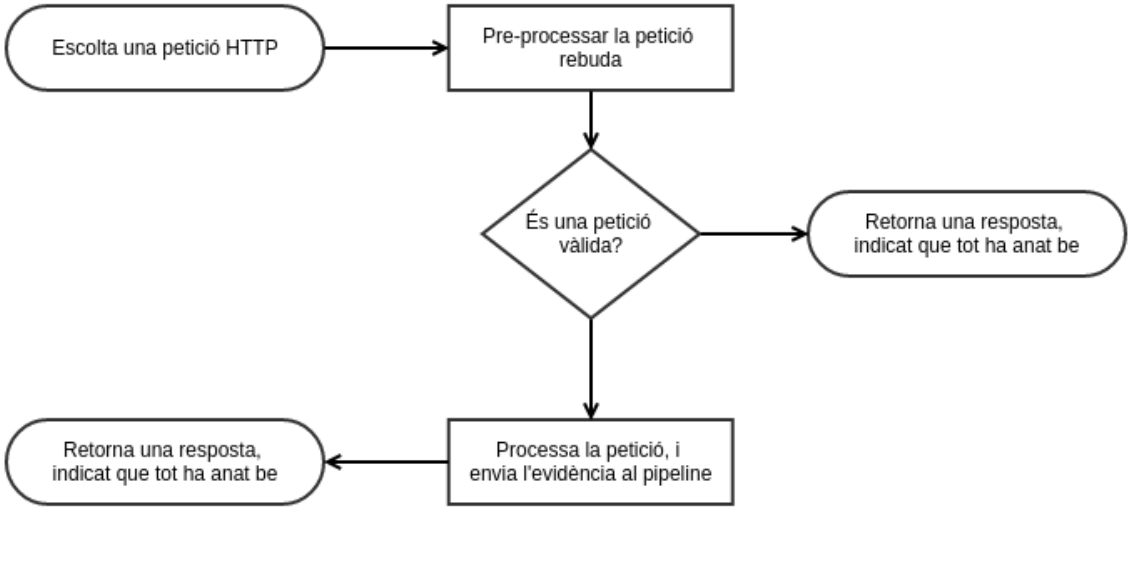


Figura 2.4.1.a. Cicle de vida de la introducció d'una evidència des del «log pipeline» cap al «data pipeline»

Tot i que en aquesta proposta només es tracta dispositius que envien dades amb el protocol **HTTP**, s'ha de tenir en compte que el que acaba processant és la unitat mínima del protocol de comunicació que suporta.

A més, el preprocessament no és en si necessari, però permet certa flexibilitat a l'hora de tractar una evidència abans de fer cap validació. Però és important que no s'ha d'aplicar cap mena de transformació de la informació de l'evidència.

La validació ha de ser molt lleugera, i no hauria d'usar cap mena de recurs complex, com interacció amb bases de dades relacionals o interacció amb el disc. Això és necessari per què, en el fons, no és responsabilitat del «**log gateway**» decidir si és correcta o no l'evidència que s'envia.

Totes aquestes consideracions estàn resumides en la figura **2.4.1.a**, on es pot veure clarament definit el cicle de vida de l'entrada d'una evidència al «log pipeline».

### *2.4.2 Infraestructura del «Log Gateway»*

Com s'ha mencionat anteriorment, l'estructura ha de ser bastant minimalista, i fàcilment extensible per a adaptar nous protocols de comunicació, evitant complexitat innecessària a l'hora d'implementar qualsevol integració amb un servei extern.

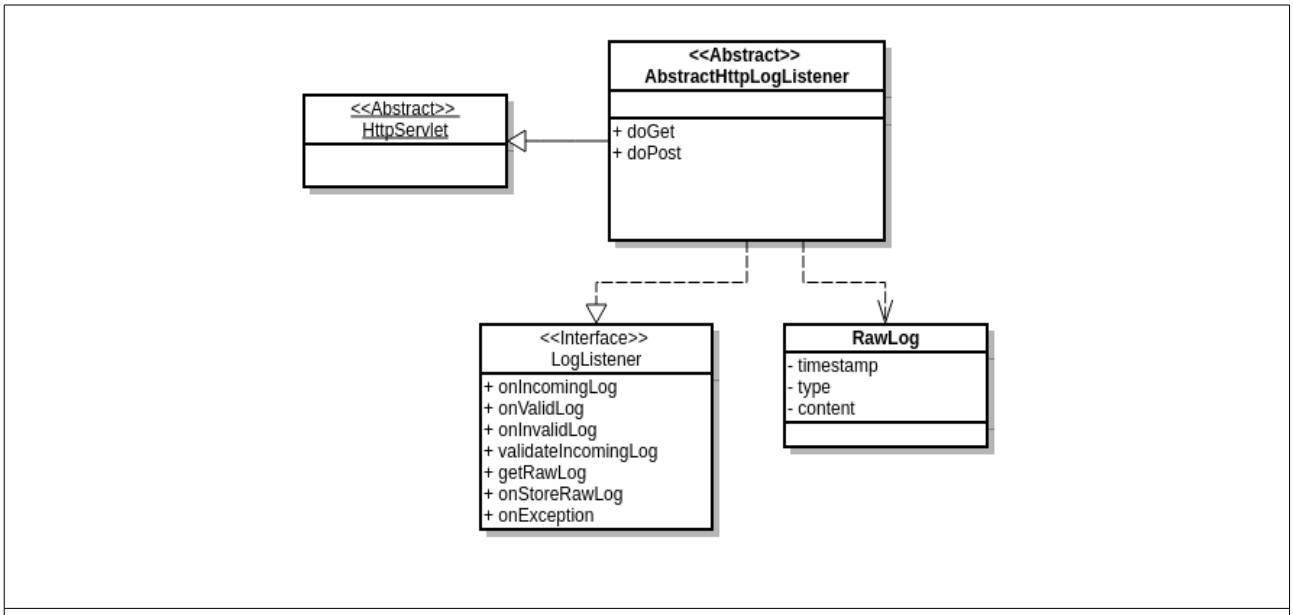


Figura 2.4.2.a. Diagrama simplificat de la infraestructura base del «Log Gateway»

La figura 2.4.2.a descriu breument les classes essencials per a poder aplicar el disseny explicat al punt 2.4.1.

### *2.4.3 Log Listener*

Aquesta interfície està definida al voltant del cicle de vida del tractament d'una evidència enviada per un dispositiu per a ser introduïda al conjunt de «data pipelines».

- El mètode «**onIncomingLog**» s'ha d'utilitzar per a fer qualsevol tipus d'acció abans de fer una validació sobre l'evidència rebuda. D'aquesta manera es pot tenir accés a la dada crua abans de ser tractada de qualsevol manera.

- El mètode «**validateIncomingLog**» s'utilitza per fer una validació l'evidencia rebuda, o el conjunt d'evidències rebudes. Ha de ser la més lleugera i el menys obstructiva possible, per a evitar potencials colls d'ampolla.

- El mètode «**getRawLog**» s'encarrega d'extreure tota aquella informació rellevant del paquet en el canal de comunicació en el que s'ha enviat l'evidència en un objecte de classe «**RawLog**».

- El mètode «**onStoreRawLog**» ha de definir els conjunts d'accions necessaris per a introduir l'evidència prèviament encapsulada en un «**RawLog**» al «**data pipeline**».

- El mètode «**onValidLog**» és el mètode que s'executarà en el cas que una evidència s'ha pogut convertir amb èxit en un objecte de tipus «**RawLog**».

- El mètode «**onInvalidLog**» és el mètode que s'executarà en el cas que hagi hagut algun problema a l'hora de tractar una evidència rebuda.

### *2.4.4 El «Raw Log»*

Un «**Raw Log**» és l'envoltori que encapsula una evidència, preparada per a ser processada per un «**data pipeline**». És la representació intermedi del que en un futur pot arribar a ser una evidència, o un conjunt d'evidències en el «**data lake**», i que pugui ser explotat per altres processos d'anàlisi.

Aquesta estructura conté:

- La data de recepció de l'evidència al «log gateway».
- El tipus d'evidència rebuda.
- La informació crua rellevant de l'evidència, amb el menor o cap transformació sobre el seu contingut.

### *2.4.5 Detalls de la implementació*

Els dispositius que s'escoltaran envien les seves dades amb el protocol HTTP. La implementació presentada és un servidor web basat en Java, sense cap dependència de gestió de transaccions HTTP que l'API Servlet 3.0 de Java.

S'ha decidit no utilitzar cap mena de «framework» per a evitar qualsevol tipus d'«overhead» que pugui provocar un coll d'ampolla innecessari a l'hora de tractar una transacció HTTP. Per tant, queda a criteri de l'implementador de com fer totes les accions necessàries per a transformar una evidència en format HTTP a un objecte «**RawLog**».

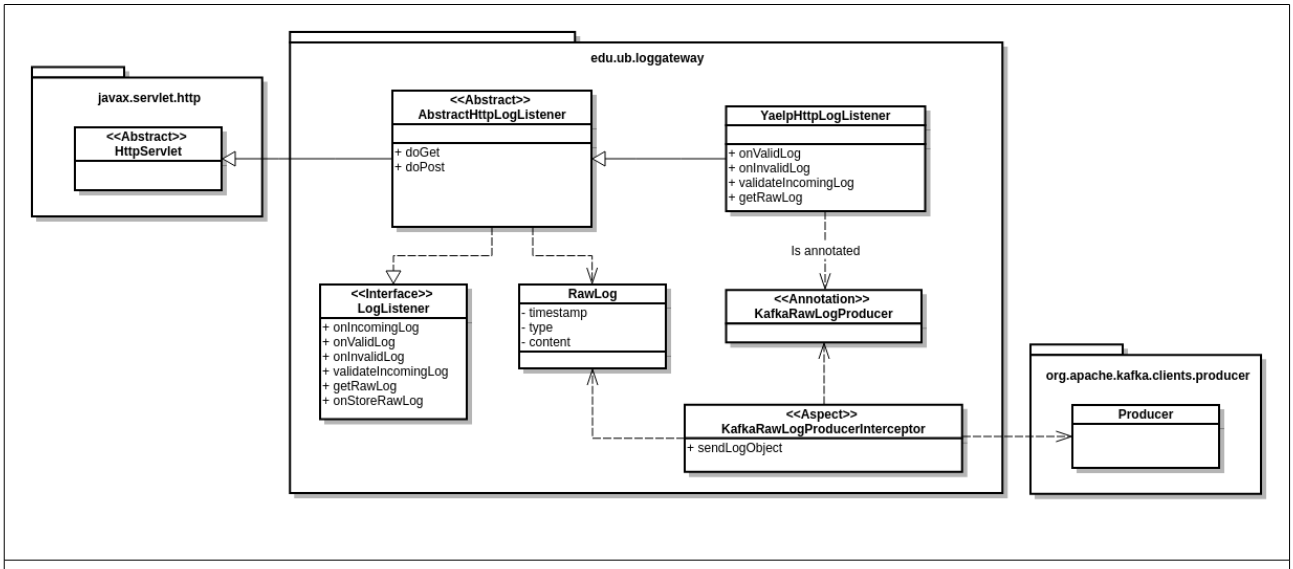


Figura 2.4.5.a. Esquema general de l'arquitectura presentada

### *2.4.6 AbstractHttpLogListenerServlet*

Aquesta classe és la implementació base que qualsevol integració amb un sistema que ens vulgui enviar evidències a través del protocol HTTP hi ha d'extendre. Integra el cicle de vida del tractament d'una evidència dins el tractament d'una transacció HTTP.

Queda subjecte a l'implementador de la classe que extengui de «**AbstractHttpLogListener**» de com tractar els diferents passos a fer per introduir una evidència en el «**data pipeline**».

### *2.4.7 CockroachHttpLogListenerServlet*

Aquesta implementació serà l'encarregada de tractar les evidències enviades en el protocol d'exemple que s'ha creat per aquest projecte, i guardar-les a qualsevol de les cues de Kafka disponibles, si en són vàlides.

Cal destacar que donat que s'està utilitzant Servlet 3.0 API, s'han utilitzat anotacions per a indicar que la classe és un Servlet mapejat a una URI concreta, i que es ha de carregar durant la inicialització del contenidor de servlets.

### *2.4.8 Aspectes, anotacions i interceptors*

Tot i que es podria proposar una implementació específica per a introduir les evidències al «data pipeline» via Kafka, per a permetre un millor desacoblament de la capa de

persistència de les evidències en el «**data pipeline**», s'ha utilitzat programació orientada en aspectes per a implementar l'acció de desar el «**raw log**» obtingut en diferents entorns. Això ens permet:

- Poder tenir un codi més fàcil de mantenir, i per tant, més còmode a l'hora de testejar.

- Compondre diferents integracions amb la mateixa operació sense fer canvis extensius a aquesta classe.

La programació orientada a aspectes és un paradigma que té com a finalitat afegir funcionalitat sobre un codi existent sense fer cap mena de modificació explícita sobre aquest. Aquesta funcionalitat que s'executa sobre un codi existent s'anomena consell o «**advice**». Per sí sol no és suficient per definir el paradigma. Es requereix un punt de tall, o «**pointcut**», que indica en quina part del codi s'executa el consell.

Amb aquests dos conceptes es pot definir un model de punts d'unió, o «**join point model**», amb el que es pot arribar a definir diferents punts de tall amb què aplicar consells. Aquests models acaben definint:

- **Quan s'ha d'executar un consell**. El model ha d'indicar quins punts d'unió es poden usar per entrellaçar el codi original amb el codi creat per un consell.

- Permetre **especificar els punts d'unió**. El model ha d'establir un llenguatge per a poder expressar els punts de tall a on s'entrellaçarà el codi.

- Permetre **especificar què executar en un punt d'unió**.

A Java, aquest tipus de programació està implementat per la llibreria **AspectJ**, que permet establir punts de tall al voltant dels mètodes de les classes per a executar codi, tant en temps de compilació com temps d'execució.

### *2.4.9 Anatomia d'un interceptor: KafkaRawLogProducerInterceptor i FileStorageInterceptor*

En el «**Log Gateway**», s'utilitzen interceptors per a definir consells que durant el temps de generació de l'artefacte del projecte, s'aplicarà el que s'anomena com a entrellaçat d'aspectes, o «**aspect weaving**», per a tractar aquelles classes que coincideixin amb els punts de tall definits en aquests interceptors, i aplicar-los el codi font implementat dins l'interceptor.

Els dos interceptors que s'han definit són els següents:

- «**KafkaRawLogProducerInterceptor**». S'utilitza per a capturar les instàncies de tipus «**RawLog**» per a enviar-les a una cua de Kafka.

- «**FileStorageInterceptor**». S'utilitza per a serialitzar els objectes «RawLog» al disc cada minut.

Amb la classe «**KafkaRawLogProducerInterceptor**», es pot veure com defineix tant el punt de tall com la implementació del consell que s'executarà sobre tots aquells mètodes que coincideixin amb el punt de tall definit:

```
@Aspect
public class KafkaRawLogProducerInterceptor {

  @Around("execution(* (@KafkaRawLogProducer edu.ub.bigdata.loggateway.LogListener+).onStoreRawLog(..))")
  public void sendToKafka(ProceedingJoinPoint pjp) throws Throwable {
    pjp.proceed();

    RawLog rawLog = (RawLog) pjp.getArgs()[0];

    System.out.println(String.format("rawLog [%s]", rawLog));
    RawLogProducer.getInstance().send(rawLog);
  }

}
```

Figura 2.4.9.a. Codi font exemple d'un interceptor d'un Aspecte

S'utilitza l'anotació **@Around** per a poder definir el punt de tall, i indicar que es vol implementar un consell al voltant del que es definirà. El que expressa és el següent:

«Executa el següent codi sobre tots aquells mètodes anomenats **onStoreRawLog**, siguin quins siguin els seus arguments, de totes aquelles instàncies que tinguin l'anotació **KafkaRawLogProducer**, siguin filles de **LogListener**, i siguin de qualsevol paquet de classes de Java»

Tot i marcar la classe com a aspecte amb l'anotació @Aspect, perquè realment es pugui generar un binari amb les classes afectades per aquest interceptor, es necessita modificar el codi binari generat durant la compilació de la classe amb l'entrellaçat d'aspectes en temps de compilació.

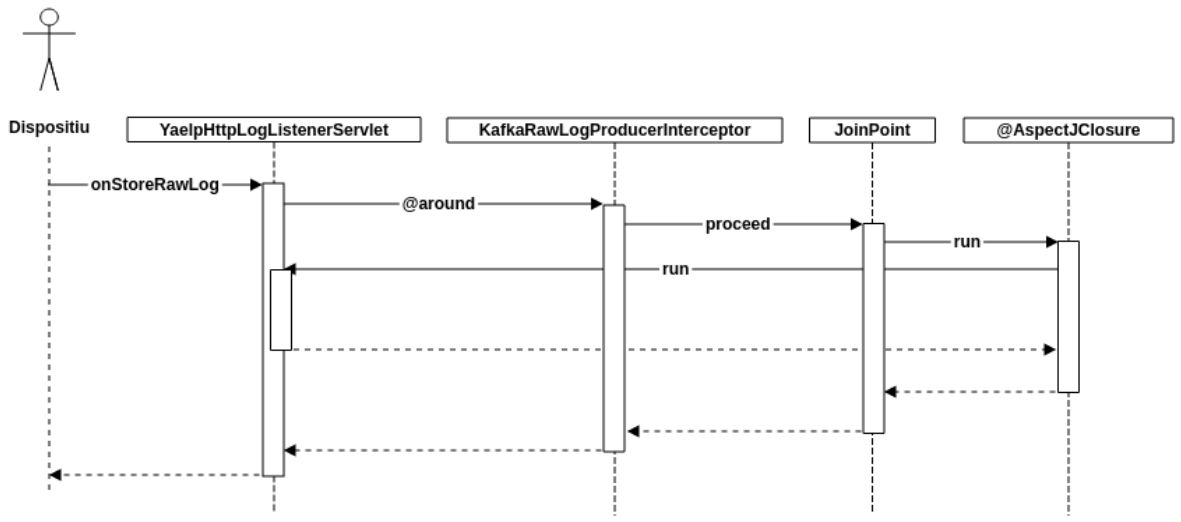


Figura 2.4.9.b. Diagrama d'execució d'un interceptor d'AspectJ

Quan AspectJ processa una classe per a dotar-li la funcionalitat de la classe interceptora, acaba creant una classe anònima que entrellaça el codi de l'interceptor amb el codi a on s'ha d'entrellaçar, quedant un diagrama d'execució com el que es veu a la figura **2.4.9.b**.

### *2.4.10 Kafka i l'acumulació de «Raw Log»*

Apache Kafka dóna una implementació de client de cua de Kafka que es pot usar immediatament per afegir missatges a un «topic» distribuït.

La implementació oferida és «thread-safe» i es recomana reutilitzar la mateixa instància el màxim possible. Per aquesta raó, la classe RawLogProducer que integra el client productor de Kafka implementació un «**singleton**» per assegurar que sempre s'utilitza la mateixa instància.

Perquè la quantitat de missatges acumulats a cada una de les cues estigui equilibrat, i no hi hagi problemes de càrrega no balancejada, durant la producció dels missatges s'utilitza una implementació diferent de particionador amb la classe «**LongValuePartitioner**».

Si s'utilitza el particionador per defecte de Kafka, acaba fent un hash de l'objecte passat per a calcular la partició a on acabaria el missatge, però això feia que es desequilibrés les particions del «topic», i gran part dels missatges acabaven a una sola partició.

### *2.4.11 Deserialització d'un «raw log»*

Els missatges que Kafka guarda a les seves cues són col·leccions de bytes, per tant, és obligació de la persona que s'integra amb Kafka definir com s'ha de transformar aquests missatges, tant per introduir-los com per extreure'ls.

Per això, a la implementació presentada s'ha utilitzat la llibreria **Kryo**, per a serialitzar i de-serialitzar els «raw logs». Aquesta llibreria és usada actualment per Storm per a la serialització i de-serialització de les tuples, atès que ofereix una velocitat i eficiència en aquestes operacions molt altes.

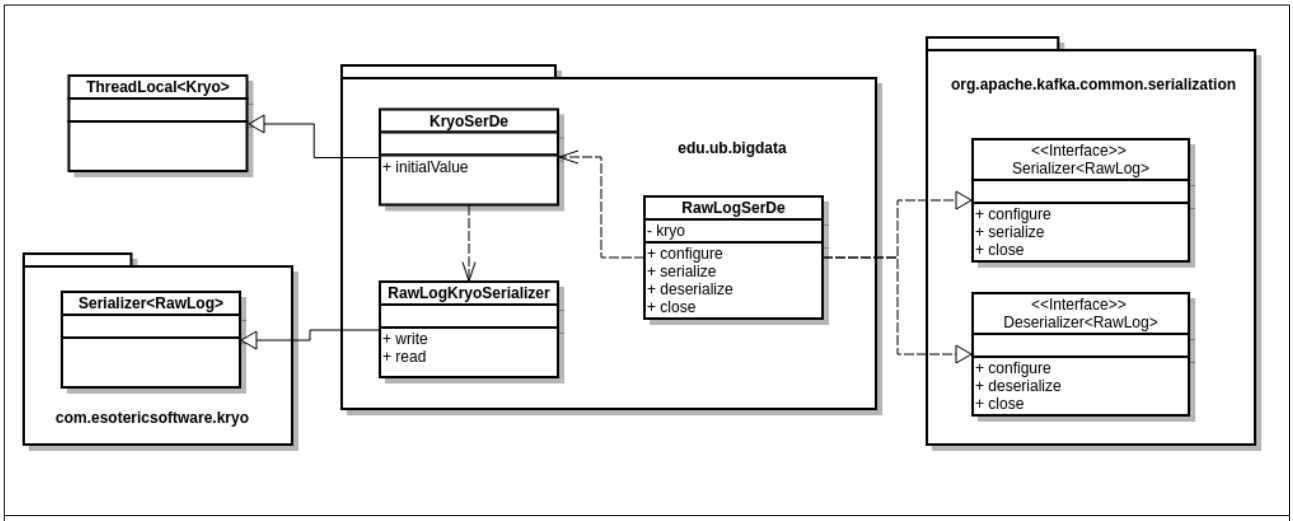


Figura 2.4.11.a. Esquema de les classes relacionades amb la serialització

La classe «**KryoSerde**» conté la implementació per a declarar tots els serialitzadors basats en Kryo, i s'utilitza com a ThreadLocal atès que només es vol utilitzar una sola instància per a cada fil d'execució d'una aplicació. De moment, l'únic serialitzador que s'enregistra és per la classe «**RawLog**»

Quan es vol enviar un «raw log» a una cua de Kafka, s'utilitza la classe «**RawLogSerde**», que és una integració dels mecanismes de serialització i deserialització de Kafka per a poder transformar els objectes «Raw Log» a vectors de bytes, o viceversa.

## 2.5 Log Pipeline

Tenint ja introduïdes les evidències captades pels dispositius en el «data pipeline», queda processar-les per a transformar-les de manera que les evidències siguin explotables per altres processos extractors de dades. Però, quin és el propòsit del «log pipeline»?

Un «log pipeline» és un classificador d'evidències que acaba decidint a quin lloc deuen quedar les dades, seguint els següents criteris sobre les operacions transformadores que pot aplicar:

- **Validació**. Tota evidència que es vol introduir ha de ser coherent, i correcte. No té sentit aplicar cap mena de càlcul sobre una dada estructuralment errònia o incorrecte des de la perspectiva de negoci.

- **Deduplicació**. Donat que estem treballant en un sistema altament distribuït, pot haver-hi problemes de comunicació, o errors humans que puguin provocar la duplicació d'una evidència.

- **Enriquiment**. Una vegada entra la dada en el sistema, es pot arribar a afegir més informació a aquesta que pugui servir a altres processaments de dades posteriors.

No només interessa definir les operacions transformadores, sinó que a més es vol definir punts adequats on es pugui monitorar el procés. És important definir quines mètriques es volen observar, per a saber exactament què està passant. Tots aquests punts queden reflectits a la figura **2.5.a** on es pot veure els diferents components processadors de la topologia de Storm que s'encarregarà de processar les evidències.

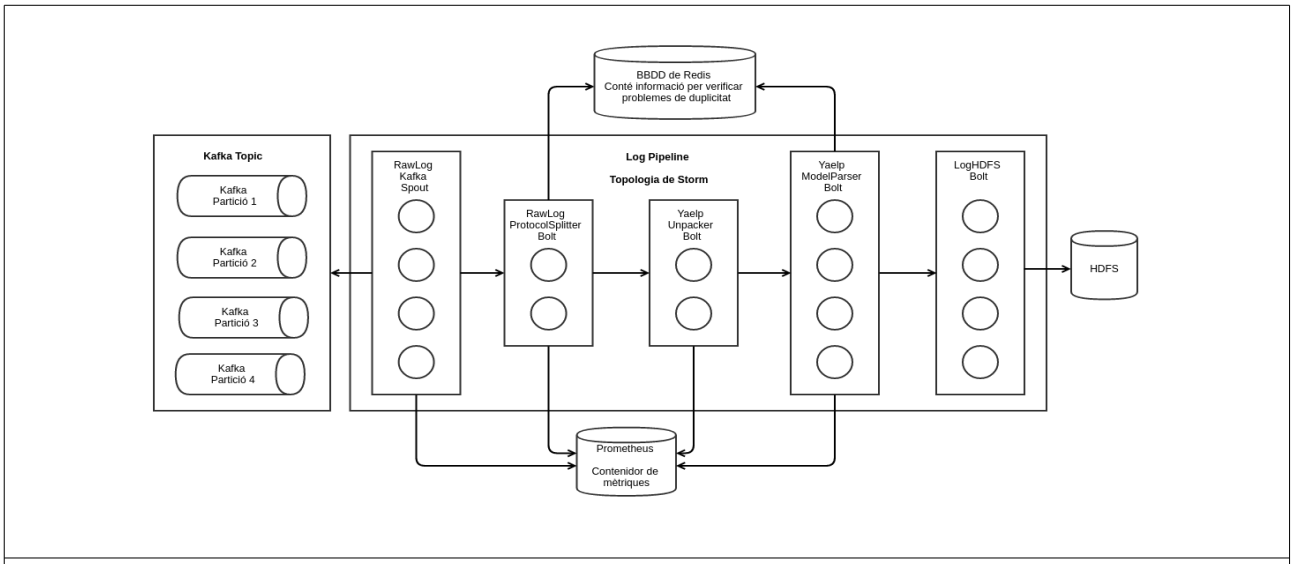


Figura 2.5.a. Diagrama dels bolts de la topologia

Tot i haver-hi processos de validació, és important tenir en compta que encara que una evidència sigui errònia, és important conservar-la. Segons en quin estat es trobi una evidència, és possible poder recuperar part o tota ella, i seria possible introduir-la en el «data lake». A la implementació presentada no indica cap manera per a recuperar aquestes evidències, però acaba desant-les a una altra part de l'HDFS perquè no afectin les altres evidències correctes.

### *2.5.1 Implementació*

Donat que s'utilitza Apache Storm per a tractar les evidències acumulades a Kafka, la implementació presentada és una topologia escalable per a processar cadascuna de les etapes transformadores d'una evidència.

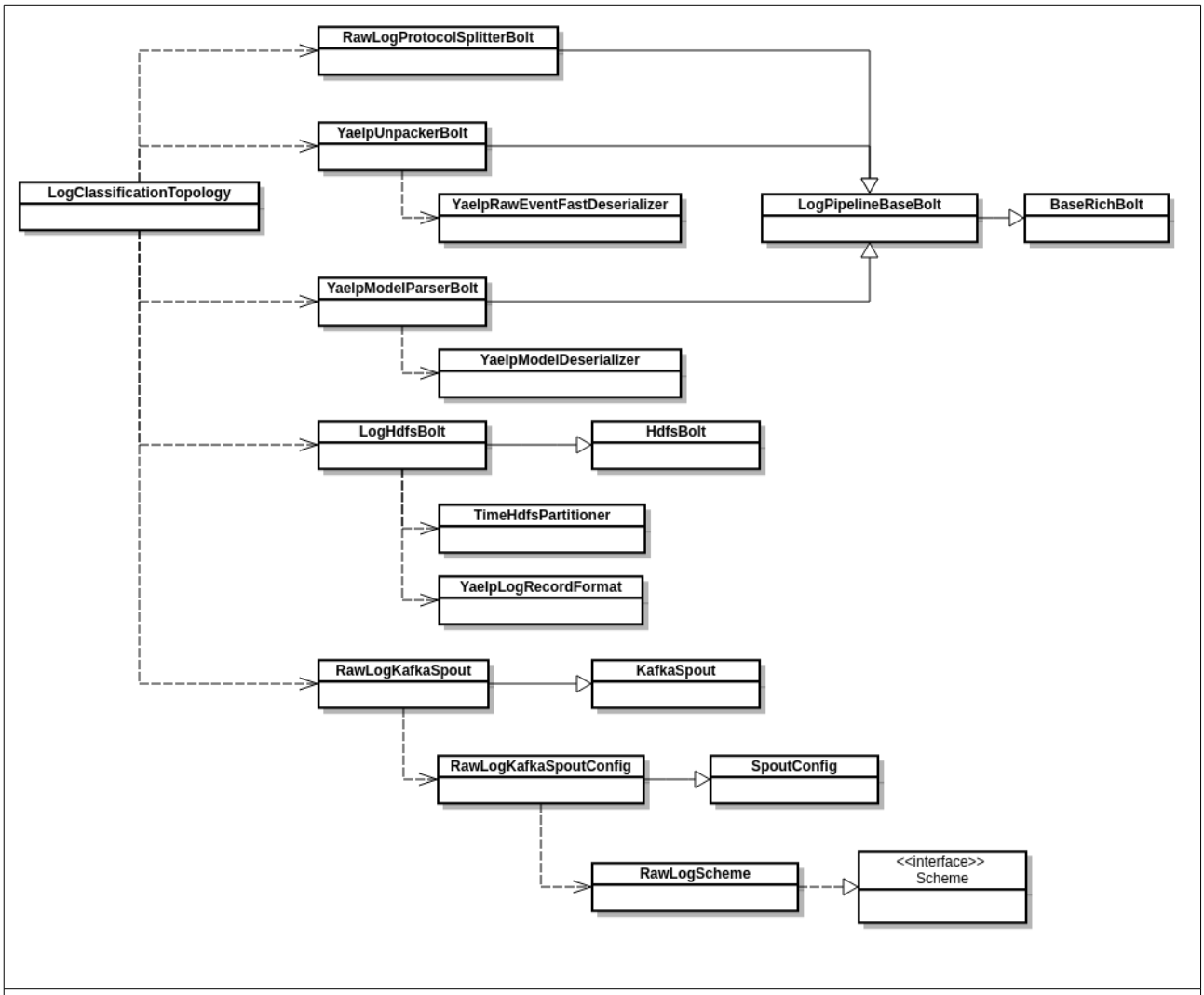


Figura 2.5.1.a. Diagrama de classes de la implementació de «log pipeline» presentada

La figura **2.5.1.a** ens mostra un diagrama amb les classes més rellevants del «log pipeline»,

amb la relació de cadascuna de les classes entre elles. Més endavant s'explicarà amb més detall les responsabilitats que té cadascuna, i quines són les seves funcionalitats.

Per a generar mètriques de la funcionalitat del sistema, i poder fer càlculs sobre el rendiment de la topologia, s'utilitza **Prometheus** per a guardar les mètriques.

A més, s'utilitza **Redis**, una base de dades en memòria, a on es desa «hashes» calculats dels «raw logs» i de les evidències identificades cada dia per a permetre cert grau de de-duplicitat d'evidències i impedir que es puguin desar evidències repetides a l'HDFS.

S'han utilitzat les següents llibreries per a integrar les diferents tecnologies entre si:

- «**storm-kafka**». Permet crear Spouts consumidors de les cues de Kafka.
- «**storm-hdfs**». Permet crear Bolts que acabaran persistin les evidències tractades a l'HDFS.
- «**prometheus-client**». Dóna la funcionalitat necessària per a generar mètriques que es podran desar a Prometheus.
- «**lettuce**». Permet integrar-se amb una base de dades Redis.

### *2.5.2 RawLogKafkaSpout*

Aquesta classe és el punt d'entrada d'una o més evidències a ser tractades, i té com a propòsit definir com transformar un «RawLog» a una tupla d'Storm.

Quan es configura el grau de paral·lelisme d'aquest «Spout», s'està configurant el nombre de consumidors que hi haurà en el mateix grup de consumidors al que atacarà les cues de Kafka.

Mitjançant la classe «**RawLogScheme**», es defineix com transformar els missatges de Kafka, que són «raw logs» serialitzats, en tuples consumibles per Storm.

### *2.5.3 LogPipelineBaseBolt*

Seguint una mica més en la figura **2.5.a**, tot «bolt» d'aquesta topologia extèn aquesta classe abstracta com a implementació base. En aquesta classe es defineix la integració amb el servidor de mètriques Prometheus, i dóna les eines necessàries perquè un «bolt» pugui

reportar el seu estat, així com la quantitat d'elements processats o elements erronis.

La idea és que tot «bolt», quan s'executa, independentment del resultat de l'execució del «bolt» sobre una tupla, s'intentarà enviar les mètriques guardades a Prometheus.

### *2.5.4 RawLogProtocolSplitterBolt*

Una vegada ja tenim un «RawLog» convertit a una tupla, es fa un cribratge inicial per a determinar, en funció del seu tipus, quines transformacions s'han d'aplicar sobre la tupla.

És en aquest punt a on es fa una primera verificació per si s'ha processat aquella evidència prèviament. A Redis, es desa una taula hash per a cada dia, a on cada clau és el «hash» calculat amb l'algorisme Murmur3 del «RawLog». La implementació d'aquest algorisme de «hashing» està provista per la llibreria «Guava» de Google. En cas que un «hash» ja estigui a la taula, el «raw log» serà descartat.

Tot i que aquesta validació no és suficient per a evitar «raw logs» repetits, i hi ha altres «bolts» a on es fa verificacions de tuples repetides.

### *2.5.5 YaelpUnpackerBolt*

Donat que un «raw log» en el protocol YAELP pot contenir múltiples evidències captades per un dispositiu, aquest «bolt» intentarà fer un primer processament del «raw log», extraient cadascuna de les evidències contingudes i emetent-les al «stream».

Per a cada tipus d'acció que ha generat aquella evidència es generarà mètriques per a saber quantes evidències s'han processat, i quan es triga a extreure les evidències d'un «raw log».

### *2.5.6 YaelpModelParserBolt*

Per a cada evidència que arriba d'un «YaelpUnpackerBolt» a una instància d'aquest «bolt», acabarà fent les següents accions:

- Validar l'evidència en funció de l'entorn i el disparador.
- Transformar l'evidència rebuda per a obtenir la representació final a ser desada a l'HDFS.

La validació de l'evidència es fa en l'àmbit estructural, és a dir, si totes les propietats

obligatòries de l'evidència hi són, i en l'àmbit lògic, verificant si conté una informació que té sentit. Per a fer aquesta validació, s'utilitza la llibreria «hibernate-validator» per a permetre definir anotacions sobre els models de dades finals.

Com també al RawLogProtocolSplitterBolt, aquest «bolt» és un altre punt a on verificar si s'han enviat evidències repetides. S'utilitza una altra taula «hash» de Redis per a verificar si les evidències han sigut desades.

A diferència de la verificació que s'executa al RawLogProtocolSplitterBolt, aquí és possible verificar sense cap mena de dubte si una evidència ha sigut processada prèviament, perquè una vegada ja es té una evidència validada, és segur que aquesta evidència és única en el temps, i per tant, la taula forçosament contindrà aquesta evidència.

### *2.5.7 LogHdfsBolt*

Finalment, s'utilitza aquest «bolt» per a desar qualsevol evidència processada a l'HDFS. Donat que tot model hi ha d'implementar la interfície HdfsLog per a definir com es guarda físicament l'evidència a l'HDFS.

La implementació d'aquest «bolt» controla la cadència amb la qual desa les evidències processades a l'HDFS, i la mida de cadascun dels fitxers HDFS generats.

Per a aconseguir que es poguessin desar les evidències a diferents carpetes l'HDFS, o «particions», s'ha hagut d'utilitzar una versió més recent de la llibreria «storm-hdfs». Per aquesta raó era necessari incloure-la dins del projecte.

## 2.6 Mètriques dels «log pipeline»

### *2.6.1 Objectius*

Per a estudiar el rendiment dels diferents «bolts», en primer lloc, es va mirar quines eren les mètriques més comunes que es necessitava calcular:

- El nombre d'èxits i el nombre d'errors a l'hora de processar una tupla en un bolt.
- En cas d'error, quins tipus d'errors són els més freqüents.
- La duració del processament d'una tupla, independentment de si hi ha hagut un error o no.
- Si un «bolt» està processant múltiples elements dins una mateixa tupla, quants en

processa.

Aquest tipus d'informació et dóna una primera idea sobre l'estat del «log pipeline» i es pot arribar a crear alarmes en casos excepcionals. En el cas que un dispositiu deixa d'enviar dades, o les dades enviades són incorrectes, aquest tipus d'anàlisi permet reaccionar més ràpidament i poder trobar la font del problema.

### *2.6.2 Infraestructura de les mètriques: Prometheus*

Per a mesurar diferents mètriques de rendiment dels «bolts» arran tota la topologia, es va plantejar utilitzar la integració de mètriques que dóna Storm per a automatitzar la generació de mètriques, i que després un altre servei les pogués explotar. Però ràpidament queda descartat, atès que la manera que Storm ofereix per a generar mètriques acaba sent simplista, o bé confús.

S'opta per utilitzar el sistema de monitoratge **Prometheus**. Aquest sistema permet acumular les mètriques de diferents serveis de dues maneres:

- Aquests serveis ofereixen una interfície que Prometheus pot consultar per obtenir-les, el que en la terminologia de Prometheus es coneix com a «**pull**»..

- Aquests serveis envien les seves mètriques cap a un servei intermediari que Prometheus el consultarà. En la terminologia de Prometheus, aquesta acció es coneix com a «**push**».

A més, aquest programari ofereix un llenguatge de consultes per a visualitzar les mètriques, segons el criteri que ens convingui, i també treu la necessitat d'usar altres components per a emmagatzemar les mètriques, i veure com analitzar-les.

### *2.6.3 Enviament de mètriques de Storm a Prometheus*

Per simplicitat de la resolució dels objectius proposats, s'utilitza un servei intermediari cap a on les diferents instàncies dels «bolts» de Storm enviaran les mètriques que requereixin ser monitorades. Aquest servei s'anomena «**Push Gateway**», i és un servei oferit per Prometheus.

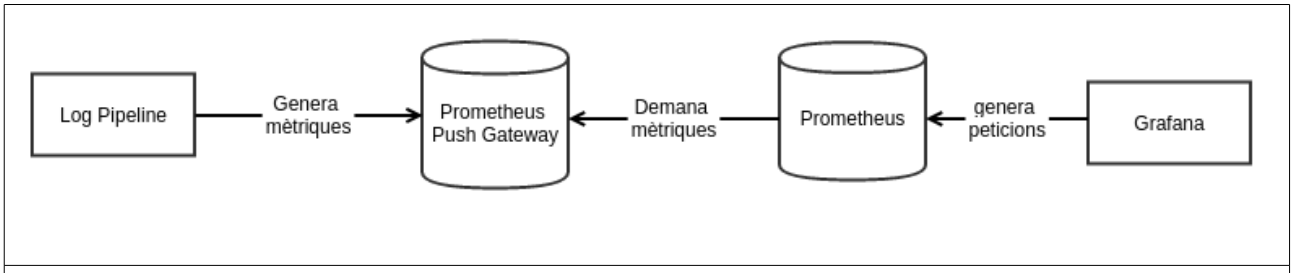


Figura 2.6.3.a. Integració de mètriques entre Storm i Prometheus

Tot «bolt» que estén de la classe **LogPipelineBaseBolt** conté un client de «push gateway», i un registre de mètriques, a on pot anar afegir les mètriques que requereixi.

Durant l'etapa de preparació d'un «bolt», l'implementador té la responsabilitat de definir les mètriques que voldrà generar, i en el processament de tuples, només cal que gestioni les tuples com la lògica de negoci indiqui.

Quan finalitza el processament de la tupla, la classe **LogPipelineBaseBolt** ja s'encarrega d'enviar les mètriques al «push gateway».

### *2.6.4 Visualització de l'estat del «log pipeline» amb Grafana*

Una vegada les mètriques que es van generant pels diferents «bolts» de la topologia es van desant a Prometheus, s'utilitza Grafana per a mostrar les mètriques, utilitzant el llenguatge de consultes propi de Prometheus.

Grafana ens permet dissenyar panells de control, o «dashboards», per a mostrar gràfiques en funció del temps, comptadors, o altres tipus d'objectes visualitzadors de dades, o «widgets», de múltiples fonts de dades que se li configurin.

En el disseny del panell de control presentat, s'ha creat diferents àrees contenidores de diferents gràfiques en funció de cadascuna de les etapes de transformació d'un «raw log» cap a una evidència processada.

### *2.6.5 Classificació de «raw logs»*

Aquesta àrea està dedicada a mostrar l'anàlisi de rendiment a l'hora de classificar els «raw logs» pel seu tipus i enviar-ho al «pipeline» de processament adequat.

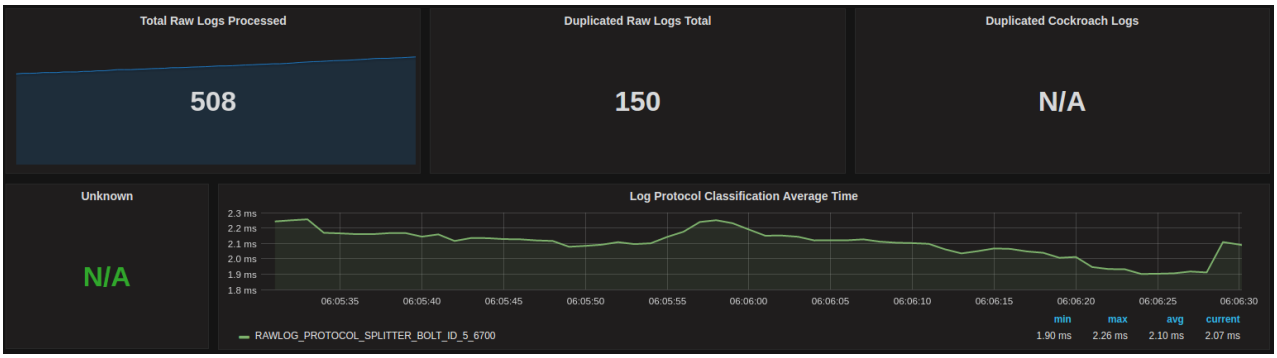


Figura 2.6.5.a. Àrea de classificació de «raw logs»

A l'àrea mostrada a la figura **2.6.5.a**, d'esquerra a dreta, i de dalt a baix:

- Un comptador del nombre de «raw logs» classificats correctament.
- Un comptador del nombre de «raw logs» duplicats.
- Un comptador amb el nombre d'evidències duplicades.
- Un comptador del nombre de «raw logs» no classificables.
- Una gràfica en funció del temps que triga a poder classificar un «raw log».

### *2.6.6 Desempaquetat de «raw logs»*

A l'hora de dissenyar aquesta àrea, és interessant veure quantes evidències es reben en funció de l'entorn d'on s'originen aquestes. A més, també interessa saber el nombre d'evidències que hi ha a cada moment en un «raw log» processat.

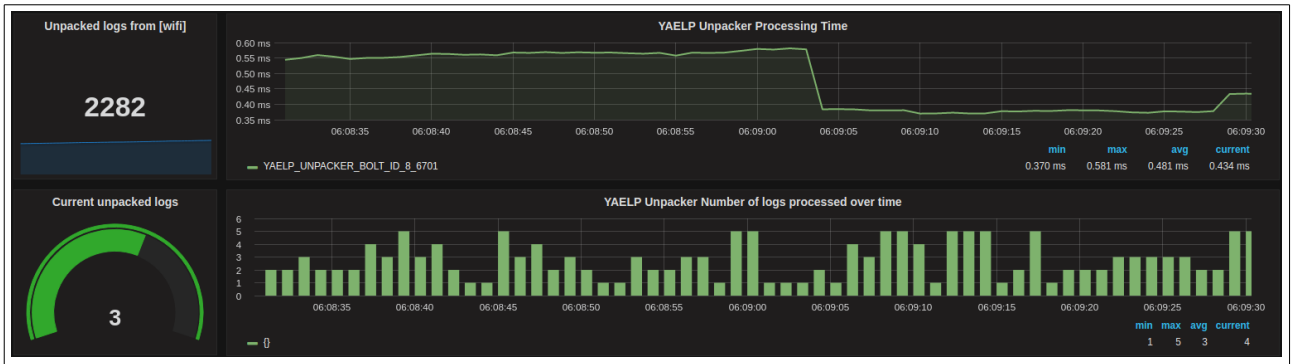


Figura 2.6.6.a. Àrea de desempaquetat de «raw logs»

A l'àrea mostrada a la figura **2.6.6.a**, d'esquerra a dreta, i de dalt a baix:

- Un comptador per saber el nombre d'evidències en protocol YAELP desempaquetades d'un «raw log» que són presències Wifi.

- Una gràfica en funció del temps que mostra el temps que triga a desempaquetar totes les evidències i a enviar-les al següent «bolt».
- Un comptador del nombre més recent d'evidències desempaquetades.
- Una gràfica en funció del temps que mostra el nombre d'evidències desempaquetades.

### *2.6.7 Interpretació de les evidències en protocol YAELP*

Semblant a l'àrea anterior, ens interessa saber tant la quantitat d'evidències processades i validades en funció de l'entorn, com també quines evidències enviades són errònies, ja sigui per problemes de validació lògica, o bé problemes d'estructura incompleta o carència d'informació.

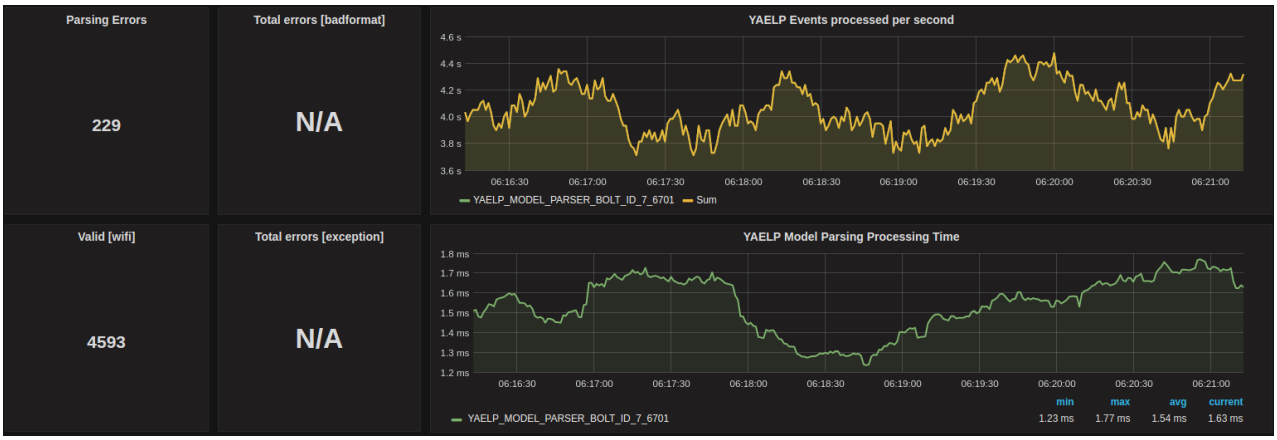


Figura 2.6.7.a. Àrea d'interpretació d'evidències

A l'àrea mostrada a la figura **2.6.7.a**, d'esquerra a dreta, i de dalt a baix:

- Un comptador amb el nombre global d'errors a l'hora d'interpretar evidències.
- Un comptador amb el nombre d'errors de format.
- Una gràfica en funció del temps que mostra la duració del processament de cadascuna de les evidències.
- Un comptador amb el nombre d'evidències de Wifi correctes.
- Un comptador amb el nombre d'errors per culpa de les excepcions dins l'execució del «bolt».
- Una gràfica en funció del temps que mostra quina quantitat d'evidències s'està processant al llarg del temps.

# 3 Experiments i jocs de proves

## 3.1 Requeriments de l'entorn d'execució

Per a provar la implementació presentada, s'ha creat un «cluster» de màquines virtuals, definit amb les següents consideracions:

- Des del punt de vista de la infraestructura del “cluster”:
- Quantes màquines hi han d'haver.
- Com tractar la xarxa virtual que es crearia per a connectar les màquines.
- Quin programari de l'ecosistema de Hadoop s'ha d'instal·lar a cadascuna de les màquines, i com s'ha de configurar.
- Quines màquines contenen serveis que poden ser utilitzats per altres màquines, i com automatitzar la seva configuració, sense tenir que explicitar la dependència entre aquestes màquines.
- Des del punt de vista del disseny del «cluster»:
- Quines màquines haurien de contenir l'HDFS.
- Quines màquines haurien de tenir els serveis de Kafka o Storm.
- Quines màquines farien de servidors web per a fer de «log gateways».

Tot això és possible automatitzar-ho gràcies a Vagrant i a la seva integració amb Puppet, que permet indicar de manera declarativa quin programari s'ha d'instal·lar i en quin estat ha de quedar els diferents fitxers de configuració per a cadascun dels serveis de Hadoop.

## 3.2 Hardware

Per a provar diferents configuracions de «cluster», s'ha utilitzat la següent configuració de hardware:

| Component | Model |
|---|---|
| Processador | Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz |
| Memòria RAM | 8x16GB (128GB) DDR4 @ 3.0GHz |
| Disc Dur | 1TB SATA3 SSD |
| Sistema Operatiu | Ubuntu 14.04 LTS |

Figura 3.2.a. Especificacions del hardware de proves

Tot i així, per a fer proves en entorns més limitats pel que fa a potència de processament i quantitat de memòria i discs durs disponibles, és requerit com a mínim un Intel Core i5 amb 16GB de RAM DDR3.

## 3.3 Programari requerit

### *3.3.1 Java*

És necessari indicar que el projecte està basat en Java 8, atès que la versió de la HDP, o «Hortonworks Data Platform», és la 2.4.0, i es treballa amb aquesta versió de Java.

### *3.3.2 Gradle*

La implementació presentada està basada en **Gradle**, i per tant, s'utilitza aquest programari per a gestionar els diversos projectes i les seves dependències. Per tant, és necessari tenir instal·lat prèviament aquest programari si es vol generar els artefactes dels diferents projectes.

Es pot descarregar des d'aquesta adreça:

http://gradle.org/gradle-download/

### *3.3.3 VirtualBox*

La primera eina que es necessita és **VirtualBox**, un proveïdor de màquines virtuals per a poder instanciar màquines a on es faran les diferents proves.

La versió que s'ha usat és la 5.0.16r105871, hi és disponible a:

http://download.virtualbox.org/virtualbox/5.0.16/virtualbox-5.0_5.0.16-105871~Ubuntu~trusty_amd64.deb

### *3.3.4 Vagrant*

**Vagrant** és un programari, que juntament amb VirtualBox, s'utilitza per a automatitzar la generació i configuració de màquines virtuals. Ofereix un DSL, o «Domain Specific Language», per a poder definir configuracions de màquines virtuals, que juntament amb imatges plantilla d'un sistema operatiu, Vagrant pot instanciar-les i configurar-les sense

cap mena d'interacció per part de l'usuari.

Hi és disponible a:

https://releases.hashicorp.com/vagrant/1.8.3/vagrant_1.8.3_x86_64.deb

### *3.3.5 Hortonworks Data Platform*

A l'ecosistema de Hadoop existeixen múltiples distribucions que ofereixen configuracions del «stack» de Hadoop, és a dir, són paquets amb versions específiques del programari de Hadoop, que garantitzen fins a cert grau d'estabilitat alhora de treballar amb elles.

En el cas d'aquest projecte, s'ha decidit utilitzar la distribució «**HDP**», o «**Hortonworks Data Platform**», però existeixen altres proveïdors com Cloudera o MapR. La versió que s'utilitzarà és la **2.4.0.0-169**, i a la figura **3.3.5.a** es pot veure les diferents versions del programari que s'utilitzarà.

| Programari | Versió |
|---|---|
| Hadoop | 2.7.1 |
| Kafka | 0.9.0.0 |
| Storm | 1.0.0 |
| Zookeeper | 3.4.6 |

Figura 3.3.5.a. Versió dels programari utilitzats al projecte

Hi ha una petita diferència a comentar amb la versió de Storm. Tot i que a la versió 2.4.0 de HDP s'utilitza la versió 0.10.0 de Storm, s'ha decidit utilitzar la versió més recent i estable de Storm.

A més, es pot consultar les versions de tot el programari publicat per a aquesta distribució de HDP a la següent adreça web:

https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.4.0/bk_HDP_RelNotes/content/ch_relnotes_v240.html

### *3.3.6 Structor*

Per aconseguir aquest grau de flexibilitat i automatització a l'hora de fer proves amb l'ecosistema de Hadoop, s'ha utilitzat el projecte Structor de Hortonworks.

Aquest programari, basat en Apache BigTop, ofereix una sèrie de receptes de Puppet que defineixen com instal·lar els diferents serveis de Hadoop sobre un conjunt de màquines virtuals instanciades mitjançant Vagrant.

Structor es basa en un fitxer de configuració, anomenat «current.profile», que defineix de quina manera han de quedar configurades les màquines virtuals que es volen instanciar.

```
{
  "domain": "local.vm",
  "realm": "LOCAL.VM",
  "hdp": "2.4.0",
  "security": false,
  "clients" : [ "hdfs", "yarn", "zk", "hbase", "hive", "tez" ],
  "mail": "oriol@alumnes.ub.edu",
  "shared_folders": [
  ],
  "storm": {
    "version": "1.0.0",
    "ui_port": 8800,
    "supervisor_slots_ports": [ 6700, 6701, 6702, 6703 ]
  },
  "vm_cpus": 1,
  "nodes": [
    {
      "hostname": "nn",
      "ip": "240.0.0.20",
      "vm_mem": 1024,
      "server_mem": 300,
      "client_mem": 200,
      "forwarded_ports": [
      ],
      "roles": [ "nn", "zk", "yarn", "hive-db", "hive-meta", "hive-server2", "client" ]
    },
    {
      "hostname": "data1",
      "ip": "240.0.0.22",
      "vm_mem": 1024,
      "server_mem": 300,
      "client_mem": 200,
      "forwarded_ports": [
        { "guest": 8042, "host": 8042 }
      ],
      "roles": [ "slave" ]
    },
    {
      "hostname": "st",
      "ip": "240.0.0.24",
      "vm_mem": 1024,
      "server_mem": 300,
      "client_mem": 200,
      "forwarded_ports": [
      ],
      "roles": [ "grafana" ]
    }
  ]
}
```

Figura 3.3.6.a. Exemple d'un perfil de «cluster» amb 3 nodes.
Un «namenode», un «datanode» i un servidor d'estadístiques amb Grafana.

Com es pot veure a la figura, cadascuna de les màquines té la seva pròpia IP, amb la quantitat de nuclis de CPU i memòria que poden consumir com a màxim, els rols que tenen, o els ports oberts a cadascuna de les màquines.

Els rols defineixen els diferents serveis que s'instal·laran en aquella màquina perquè tinguin totes les funcionalitats necessàries per a poder oferir les funcionalitats necessàries pròpies del rol.

Per exemple, quan s'especifica el rol «grafana», no només s'instal·larà la mateixa aplicació «Grafana», sinó que també s'instal·larà «Prometheus» i les altres dependències pròpies de «Prometheus» i es podrà integrar «Grafana» amb «Prometheus» per a poder visualitzar estadístiques.

## 3.4 Modificacions a Structor

A l'hora d'orquestrar tot el desplegament del «cluster» dissenyat, **Structor** utilitza Puppet per a definir què ha d'instal·lar, com ho ha d'instal·lar, i en quin ordre s'ha de fer.

Per això, s'han definit una sèrie de mòduls nous, i s'han agregat al manifest principal per a poder associar-los a nous rols de Structor.

Els nous mòduls hi són disponibles a la carpeta modules, del projecte structor, i el manifest principal és el fitxer «**manifests/default.pp**». En aquest últim fitxer es troba la configuració de dependències entre els diferents mòduls de Puppet, i indica l'ordre d'instal·lació de cadascun d'ells, així com l'associació entre els rols i els mòduls.

### *3.4.1 El rol «kafka-manager»*

Aquest rol instal·la en la màquina indicada el servei web **Kafka Manager** desenvolupat per Yahoo. Aquest servei permet monitorar els diferents «brokers» de Kafka disponibles en el «cluster», i poder fer petites gestions com reconfigurar les cues creades, o veure el nombre de missatges generats pels productors.

### *3.4.2 El rols «storm-nimbus» i «storm-supervisor»*

Aquests dos rols permeten instal·lar en una màquina o el servei Nimbus de Storm per a poder publicar topologies de Storm, o el servei supervisor que actua com a màquina executora de topologies sota el Nimbus.

És necessari indicar que no es pot instal·lar el tots dos serveis sota la mateixa màquina, per tant, si una màquina indica que té tots dos rols, agafarà el rol de «storm-nimbus».

El rol de «storm-nimbus» no només acaba instal·lant el mateix servei Nimbus, sinó també la interfície web per a poder monitorar sense necessitat d'usar el terminal per a interactuar amb el Nimbus.

### *3.4.3 El rol «grafana»*

Aquest rol instal·la diversos serveis per a permetre el monitoratge d'altres sistemes. Instal·la Grafana, Prometheus i el Push-Gateway de Prometheus en la mateixa màquina.

### *3.4.4 El rol «prometheus»*

Com el seu nom indica, permet instal·lar una instància de **Prometheus** en aquella màquina que se li afegeix aquest rol. A més, instal·la també una instància del servei **Pushgateway**, la qual és consultada per Prometheus per acumular les mètriques del Pushgateway.

### *3.4.5 El rol «redis»*

Aquest rol permet instal·lar la base de dades **Redis** en la màquina que se li assigna aquest rol.

## 3.5 Desplegament del projecte

### *3.5.1 Inicialització del cluster*

Una vegada hagi instal·lat el programari descrit prèviament, només cal crear el fitxer «**current.profile**» a l'arrel de la carpeta «**structor**» amb el contingut d'algun dels perfils que hi ha disponibles a la carpeta «**profiles**» del projecte Structor:

- «**4-node.profile**». És un perfil per a màquines de 16GB de RAM, i està fet per a fer una petita prova conceptual del projecte. És el que recomanaria per a veure cóm funciona tot, i fer-se una primera idea.
- «**18-node.profile**». Aquesta configuració serveix per a estudiar problemes de balanceig de càrrega, i poder estudiar entorns més distribuïts. Recomanat per a 128GB de RAM.

Una vegada editat el fitxer «**current.profile**» amb algun dels perfils que s'ofereixen, es pot inicialitzar el «cluster» executant la següent comanda, estant a la carpeta del projecte Structor:

```
#!/bin/bash
```

```
vagrant up
```

Figura 3.5.1.a. Comanda per inicialitzar el «cluster» virtualitzat

Pot trigar una estona a posar-se el «cluster», depenent de la configuració que s'hagi escollit, per la qual cosa seria bo tenir una tassa de cafè o te a mà.

### *3.5.2 Canvis a /etc/hosts*

Per a evitar haver d'estar editant les URLs del navegador cada vegada que s'accedeix a una interfície web, val la pena editar el fitxer «/**etc**/**hosts**» per a incloure els següents noms de domini:

```
127.0.0.1 nn.local.vm
127.0.0.1 hive.local.vm
127.0.0.1 hbase.local.vm
127.0.0.1 nimbus.local.vm
127.0.0.1 data1.local.vm
127.0.0.1 data2.local.vm
127.0.0.1 data3.local.vm
127.0.0.1 data4.local.vm
127.0.0.1 kafka1.local.vm
127.0.0.1 kafka2.local.vm
127.0.0.1 kafka3.local.vm
127.0.0.1 kafka4.local.vm
127.0.0.1 storm1.local.vm
127.0.0.1 storm2.local.vm
127.0.0.1 storm3.local.vm
127.0.0.1 storm4.local.vm
127.0.0.1 jt.local.vm
127.0.0.1 stats.local.vm
```

Figura 3.5.2.a. Àlies dels noms de domini de les màquines virtuals instanciades.

### *3.5.3 Llistat de serveis disponibles*

A la figura **3.5.3.a** es mostra un llistat de direccions Web que permet accedir als diferents serveis web que s'inicialitzen en el «cluster» de Hadoop, però depenent de com s'hagi configurat el fitxer «current.profile», és possible que tots els serveis no hi siguin disponibles:

| **Nom del servei** | **URL** |
|---|---|
| Hue | http://nn.local.vm:8000/about/ |
| Oozie | http://jt.local.vm:11000/oozie/ |
| Namenode Status Service | http://nn.local.vm:50070/ |
| Job History Service | http://nn.local.vm:19888/jobhistory/ |
| Storm's Nimbus | http://nimbus.local.vm:8800/ o http://nn.local.vm:8800/ |
| Kafka Manager | http://nn.local.vm:9001/ |
| Grafana | http://stats.local.vm:3000/ |
| Prometheus | http://stats.local.vm:9090/ |

| | |
|---|---|
| Prometheus Pushgateway | http://stats.local.vm:9091/ |
| Jetty Server ~ 1 | http://kafka1.local.vm:18081/ |
| Jetty Server ~ 2 | http://kafka2.local.vm:18082/ |
| Jetty Server ~ 3 | http://kafka3.local.vm:18083/ |
| Jetty Server ~ 4 | http://kafka4.local.vm:18084/ |
| Yarn ResourceManager | http://nn.local.vm:8088/cluster |

Figura 3.5.3.a. Llistat de serveis disponibles al «cluster»

### *3.5.4 Com connectar a una màquina virtual via SSH*

Hi ha diverses maneres per a connectar-se a una màquina virtual instanciada per Vagrant. La més senzilla és amb la comanda pròpia de Vagrant, com es pot veure en la figura **3.5.4.a**.

```
#!/bin/bash

vagrant ssh [NOM DE LA MÀQUINA]
```

Figura 3.5.4.a. Comanda per connectar a una màquina del «cluster» virtualitzat

En els dos perfils de «clusters» disponibles, s'utilitza el mateix criteri per anomenar les màquines, amb l'únic canvi en la enumeració quan hi ha múltiples màquines que tenen el mateix set de rols. La figura **3.5.4.b**. mostra un llistat dels noms usats als perfils amb el nom complet per identificar-los, així com una breu descripció de la seva funcionalitat.

| **Nom** | **Nom complert** | **Descripció** |
|---|---|---|
| `nn` | `Name Node` | Conté els serveis crítics per poder fer funcionar l'HDFS i MapReduce. Però pel perfil «4-node» pot contenir els serveis per a Hive, Hbase, el Nimbus de Storm. Conté un servei de Zookeeper. |
| `nimbus` | `Storm Nimbus` | Aquesta màquina serveix per a publicar les topologies de Storm. També tenen el servidor web que permet monitorar l'estat de les topologies. |
| `data` | `Data Node` | Normalment actua com a node contenidor de dades de l'HDFS i per a executar tasques MapReduce. Però en el perfil «4-node», pot contenir els «supervisors» de Storm. |
| `kafka` | `Kafka Broker` | Cada Kafka Broker està instal·lat en aquestes màquines. També s'instal·la en aquestes màquines el «log gateway». |
| `Storm` | `Storm Supervisor` | Són els nodes que instancien «worker processes» per a Storm. |
| `stats` | `Statistics` | És la màquina que conté serveis com Prometheus, Pushgateway i Grafana. També conté la base de dades Redis que usarà el «log pipeline». |

Figura 3.5.4.b. Llistat de noms de màquines amb descripcions.

### *3.5.5 Compilació i generació dels artefactes del projecte*

Una vegada el «cluster» estigui operatiu, es poden generar els artefactes que es publicaran mitjançant la comanda:

```
#!/bin/bash

make publish
```

Figura 3.5.5.a. Comanda per compilar el projecte

Aquesta comanda compilarà cada projecte i generarà els artefactes a l'arrel del projecte:

- «**log-gateway-1.0-SNAPSHOT.war**». Aquest artefacte s'ha de desplegar a totes aquelles màquines virtuals que tinguin un servidor web.
- «**log-pipeline-1.0-SNAPSHOT.jar**». Aquest artefacte s'ha de desplegar a la màquina virtual que tingui el rol de «storm-nimbus» al fitxer «current.profile».

### *3.5.6 Iniciar i configurar la base de dades Redis*

Per alguna raó que no acabo d'esbrinar perquè, quan s'inicia una màquina virtual amb **Redis**, tot i que el servei es crea correctament, no es pot arribar a posar en marxa. Per això, és necessari iniciar-ho manualment

```
#!/bin/bash

/etc/init.d/redis_6379 start
```

Figura 3.5.6.a. Comanda per a iniciar el servidor de Redis.

S'ha d'executar a la màquina a on està instal·lat Redis la comanda de la figura **3.5.6.a**. Això és necessari, perquè el «log pipeline» depèn d'aquest servei per a poder evitar evidències duplicades.

Per últim, Redis per defecte, escolta la IP 127.0.0.1, però realment no volem això, perquè si no, les altres màquines no podran treballar amb aquesta base de dades. Per això, s'ha d'editar el fitxer «/etc/redis/6379.conf» i canviar la propietat «bind» pel següent valor:

0.0.0.0

D'aquesta manera, ja escoltarà a qualsevol que es vulgui connectar a la màquina a on resideix Redis.

### *3.5.7 Creació de les cues de Kafka*

Tot i que el «cluster» està en marxa, ara mateix no hi ha cap cua distribuïda de Kafka configurada. Per configurar-la, s'ha de connectar a qualsevol dels nodes que tingui com a rol «kafka» i executar la comanda en la figura **3.5.7.a**:

```
#!/bin/bash

kafka-topics --create --topic raw-logs --partitions 4 --replication-factor 2 --config retention.ms=1000
```

Figura 3.5.7.a. Comanda per crear les cues de Kafka

Aquesta comanda crearà una cua amb quatre particions, és a dir, a quatre brokers es podrà anar desant missatges a la cua distribuïda, i aquestes quatre particions tindran dues còpies.

S'haurà d'ajustar aquest valor en funció del nombre de màquines amb el rol «kafka» hi hagi disponibles. Penseu que com es coordinen amb **Zookeeper**, és possible anar afegint màquines per rebalancejar el «cluster» de **Kafka**.

### *3.5.8 Configuració de «Kafka Manager»*

Aquest servei, quan s'instal·la, no detecta automàticament la instal·lació de Kafka instal·lada. Per això, s'ha de crear una nova configuració de «cluster» de Kafka a Kafka Manager.

En primer lloc, per a accedir a Kafka Manager, es pot anar a l'adreça:

http://nn.local.vm:9001/

Dins, es pot crear un nou «cluster» desplegant el menú «Cluster» i seleccionant la opció «Add cluster». En el formulari que apareixerà, es té que afegir els valors indicats a la figura **3.5.8.a**.

| **Paràmetre** | **Valor** | |
|---|---|---|
| Cluster Name | `log-pipeline` | |
| Cluster Zookeeper Hosts | `4-node` | `nn.local.vm:2181,stats.local.vm:2181` |
| | `18-node` | `nn.local.vm:2181,hive.local.vm:2181,nimbus.local.vm:2181` |
| Kafka Version | `0.9.0.0` | |
| Enable JMX Polling | `Activat` | |
| clusterManagerThreadPoolSize | `2` | |

| | |
|---|---|
| kafkaCommandThreadPoolSize | 2 |
| logkafkaCommandThreadPoolSize | 2 |
| brokerViewThreadPoolSize | 2 |
| offsetCacheThreadPoolSize | 2 |
| kafkaAdminClientThreadPoolSize | 2 |

Figura 3.5.8.a. Credencials per a accedir a Grafana

Una vegada s'ha creat la definició del cluster, Kafka Manager detectarà automàticament tots els «brokers» disponibles, i informarà de la quantitat de bytes enviats i rebuts a aquests. També detectarà els «topics» que s'hagin creat en aquest cluster i els mostrarà.

Per a monitorar l'estat del topic «raw-logs», es pot fer anant a la següent adreça:

http://nn.local.vm:9001/clusters/log-pipeline/topics/raw-logs

### *3.5.9 Publicació del «Log Gateway» a Jetty*

Ja generats tots dos projectes, per a desplegar el «log gateway», s'ha de connectar a cadascuna de les instàncies en el «cluster» amb el rol «log_gateway»[1], i executar la següent comanda:

```
#!/bin/bash

/opt/scripts/jetty/deploy_log_gateway.sh
```

Figura 3.5.9.a. Comanda per publicar el «log gateway» a un dels serveis web Jetty

El que acaba fent és copiar el fitxer «**log-gateway-1.0-SNAPSHOT.war**» de l'arrel del projecte (que en el cas de la màquina virtual, es troba a la ruta «/**vagrant**»), al contenidor de servlets **Jetty** desplegat, assegurant-se del fet que si hi havia una còpia anterior, l'esborra.

El servei **Jetty** configurat està preparat per a detectar els fitxers desats a la carpeta de destí, i publicar els contenidors servlets amb el nom proposat.

### *3.5.10 Publicació del «Log Pipeline» a Storm*

Per a publicar la topologia del «log pipeline» a Storm, s'ha de connectar al node que tingui

1 Si s'utilitza qualsevol de les configuracions de «cluster» presentades, els serveis web hi són a les mateixes màquines a on estàn els serveis de Kafka.

el rol «**storm-nimbus**» via SSH i executar la comanda en la figura **3.5.10.a**.

```
#!/bin/bash

JAR_PATH=/vagrant/log-pipeline-1.0-SNAPSHOT.jar
TOPOLOGY_CLASS=edu.ub.bigdata.logpipeline.storm.classification.topologies.LogClassificationTopology
TOPOLOGY_NAME=RawLogConsumerTopology
ZOOKEEPER_HOSTS=nn.local.vm:2181,stats.local.vm:2181
storm jar $JAR_PATH $TOPOLOGY_CLASS $TOPOLOGY_NAME $ZOOKEEPER_HOSTS
```

Figura 3.5.10.a. Comanda per publicar el «log pipeline» a Storm

Cal destacar de la figura **3.5.10.a** també que la variable «**ZOOKEEPER_HOSTS**» té un valor diferent en funció del perfil de «cluster» escollit. A la figura **3.5.10.b** es pot veure els valors que s'haurien d'indicar en funció del perfil de «cluster».

| **Perfil** | **Valor de la variable «ZOOKEEPER_HOSTS»** |
|---|---|
| 4-node | `nn.local.vm:2181,stats.local.vm:2181` |
| 18-node | `nn.local.vm:2181,hive.local.vm:2181,hbase.local.vm:2181,nimbus.local.vm:2181` |

Figura 3.5.10.b. Valors dels hosts de Zookeeper en funció del perfil de «cluster».

En pocs minuts, la topologia acaba sent publicada i ja està preparat el «cluster» per a processar les dades de dispositius.

S'ha de tenir en compte que en els perfils presentats, o s'ha de publicar la topologia en el node «**nn**» si és el perfil «4-node», o en el node «**nimbus**», si és el perfil «18-node».

### *3.5.11 Configuració de Grafana i creació del «dashboard»*

Queda configurar **Grafana**, per a poder importar el «dashboard» que ens mostrarà les mètriques de rendiment del «log pipeline» publicat. Fins que no hi hagi activitat a la topologia de Storm, encara que es configuri el «dashboard», no es mostrarà cap activitat.

Es pot accedir a Grafana via l'enllaç web:

http://stats.local.vm:3000/

Com que és la primera vegada que s'accedeix, ens demanarà crear un nou usuari amb contrasenya. Aquests credencials quedaran desats a la màquina virtual, pel que només cal inventar-nos un usuari amb els credencials que es pot veure a la figura **3.5.11.a**.

| **Paràmetre** | **Valor** |
|---|---|
| Usuari | `adminub@ub.edu` |

| | |
|---|---|
| Nom d'usuari | `Admin UB` |
| Contrasenya | `adminub` |

Figura 3.5.11.a. Credencials per a accedir a Grafana

Però una vegada hi estiguem a dins, és necessari crear una nova organització, on es crearà el «dashboard». Per això, s'ha d'anar a la icona de Grafana, i dins del desplegable del perfil d'usuari, seleccionar l'opció «**New organization**», tal com es pot veure a la figura **3.5.11.b**, i després indicar com a nom de la organització «**UB**».

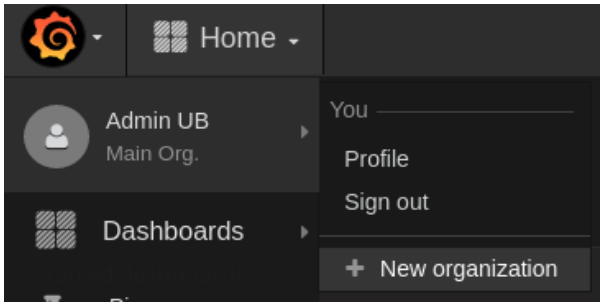


Figura 3.5.11.b. Menú per a crear una nova organització.

Una vegada creada l'organització, s'ha de crear una nova font de dades de la que begui Grafana per a mostrar les mètriques acumulades per Prometheus.

Per fer això, s'ha d'anar un altre cop a la icona de Grafana i seleccionar l'opció «**Data Sources**», i una vegada carregada aquesta nova secció, s'ha de seleccionar el botó «**Add data source**».

En la nova secció carregada, s'ha d'omplir les dades del camp amb les dades indicades a la figura **3.5.11.c**, i prémer el botó «**Add**».

| **Paràmetre** | **Valor** |
|---|---|
| Name | `Prometheus` |
| Default | `Activat` |
| Type | `Prometheus` |
| Url | `http://stats.local.vm:9090/` |
| Access | `proxy` |
| Basic Auth | `Desactivat` |
| With Credentials | `Desactivat` |

Figura 3.5.11.c. Valors a emplenar per a crear la font de dades per accedir a «Prometheus».

Per últim, s'ha d'importar el «dashboard» per a poder mostrar les mètriques.

### *3.5.12 Execució de la simulació*

Per últim, s'inclouen una sèrie de petits programes, que es troben a la carpeta **scripts** del projecte, fets en Python per a poder simular diferents dispositius que envien dades als «log gateways», i acaben enviant evidències en el protocol YAELP.

Cadascun d'aquests programes simula possibles problemes que es pot trobar a l'hora d'enviar evidències:

- «**device.py**». Aquest és el programa base d'un dispositiu que funciona bé. Teòricament, no hauria d'enviar evidències repetides, ni mal formades.

- «**device_badoui.py**». Aquest programa-simulador genera evidències amb l'OUI d'una MAC malformada. Aquestes evidències incorrectes quedaran descartades en el «log pipeline».

- «**device_dubs.py**». Aquest programa genera evidències duplicades, tant pel que fa a repeticions d'un conjunt d'evidències d'un mateix entorn enviades, com evidències repetides dins un conjunt.

Només cal executar, des de la màquina nadiua, qualsevol d'aquests programes amb Python per a començar a processar evidències i desar-les a l'HDFS (si són correctes). A la figura 3.5.12.a ensenya algunes maneres d'executar els scripts.

```
#!/bin/bash

python scripts/device.py > /tmp/device.01.out &
python scripts/device_badoui.py > /tmp/device_badoui.02.out &
python scripts/device_dubs.py > /tmp/device_dubs.03.out &
```

Figura 3.5.12.a. Comandes per a instanciar els dispositius que enviaran evidències.

És recomanable executar-los amb una comanda «**screen**», per a no tancar els processos si es tanca la sessió, o si s'accedeix a la màquina via **SSH**.

### *3.5.13 Monitorar l'enviament d'evidències al «log gateway»*

Una vegada els dispositius comencin a enviar evidències al «log gateway», per assegurar-nos que s'estan processant de forma correcta, es pot consultar de diverses maneres:

- Es pot mirar a la carpeta «/**tmp**/**log-gateway**» de les instàncies «**data1**» i «**data2**», si s'utilitza el perfil «4-node», o les instàncies «kafka1» a «kafka4» si s'utilitza el perfil «18-node». Dins aquesta carpeta, els «log gateways» desen cada minut les evidències que va processant.

- Al «**Kafka Manager**», es pot visualitzar l'estat de les cues que hi ha actualment, i veure si s'està processant missatges dins el cluster «**log-pipeline**» que s'ha definit en la secció **3.5.7**, i anant al topic «**raw-logs**».

### *3.5.14 Monitorar l'estat del «log pipeline»*

Per a monitorar la topologia de Storm, es pot avaluar l'estat d'aquesta de diverses maneres.

Una d'elles és via la interfície web, accessible a través de la següent adreça:

http://nimbus.local.vm:8800/

Dins, es pot accedir al resum de la topologia «RawLogConsumerTopology», i es pot veure diverses estadístiques sobre el nombre de tuples emitides pels «spouts», o el temps de processament mitjà dels «bolts».

Una altra manera, si es desitja veure en temps real l'activitat de les tasques de Storm, es pot fer accedint a cadascun dels nodes «worker» del «cluster» de Storm. Dins, es pot accedir a la següent ruta:

/usr/hdp/current/storm/storm/logs/workers-artifacts/

Dins aquesta carpeta, existeix una carpeta de cadascuna de les topologies publicades al «Nimbus». Dins, conté la carpeta d'un dels «workers» que estarà processant dades en aquell moment. El fitxer «**worker.log**» desa la informació en cru de l'estat actual d'aquell worker.

Per avaluar el rendiment de la topologia, es pot anar al servidor de Grafana, havent estat configurat prèviament seguint els passos de la secció 3.5.11, mitjançant el següent enllaç:

http://stats.local.vm:3000/

# 4 Conclusions i assoliment d'objectius

En principi, les mancances més importants trobades en l'anàlisi de la solució actual de processament de dades, descrit en el punt **2.2** d'aquesta memòria, són resoltes amb la introducció de programari més flexible.

El «**log gateway**» com l'alternativa en forma de micro-servei, sent més simple de mantenir i amb l'única finalitat d'introduir les dades al «log pipeline». La complexitat del codi disminueix qualitativament, al mateix temps que es recolza sobre Kafka per a persistir les evidències a ser processades.

«**Kafka**», com a substitut de Apache Flume, per acumular les evidències enviades pels dispositius. El fet de poder acumular les evidències, i permetre que grups de consumidors puguin processar les cues al seu ritme, i sense haver de gestionar cap mecanisme de bloqueig per accés concurrent simplifica tot el procés.

El «**log pipeline**» com a responsable de classificar i validar les evidències, a més de persistir-les finalment a l'HDFS. El fet d'utilitzat Storm per a processar cadascuna de les evidències, permet particionar de forma còmoda el processament, i només s'ha d'estudiar la proporcionalitat dels «bolts» per assegurar que no es desajusta la seva càrrega de feina. Tot i que Kafka està actuant com a un mecanisme de «**back pressure**», per a evitar col·lapsar la topologia de Storm, si hi hagués un pic de càrrega inesperat, segons cóm es distribueixi la càrrega que hi hagi en cadascun dels «bolts», és molt probable que sigui necessari ajustar l'índex de paral·lelisme dels nodes de la topologia.

La capacitat de monitorar tota la transformació d'una evidència a una dada continguda al «data lake» i explotable per altres processos de càlcul de KPI. El fet de tenir un servidor de mètriques com **Prometheus**, i explotar les seves mètriques amb **Grafana**, fa que avaluar el rendiment de tot el procés de transformació, i poder veure quan hi ha errors.

Mitjançant **Redis**, una base de dades en memòria que guarda un índex de totes aquelles evidències que s'han processat, ens permet mantenir un mecanisme de de-duplicitat d'evidències sota certs nivells. Tot i així, hi ha un manteniment moderat d'aquesta base de dades, atès que no es pot mantenir indefinidament les taules que guarden els «hashes» de les evidències, perquè exhaurien la memòria de màquina on resideix aquest servei. En fet, una potencial optimització sobre la quantitat de memòria consumida per Redis seria aplicar un «**Filtre de Bloom**», atès que s'aconseguiria disminuir la quantitat de memòria consumida, si es fes un us optimitzat de les estructures que ofereix Redis, com per exemple, els mapes de bits.