

PROYECTO FINAL DE INGENIERÍA

**SISTEMA DE MONITOREO DE SILOS MEDIANTE
SENSORES IOT Y PROCESAMIENTO DE BIG DATA**

Calace, Álvaro – LU1037968

Ingeniería en Informática

Zamudio, Octavio – LU1038254

Ingeniería en Informática

Tutor:

Calla, Cristian, UADE

[MES] [DÍA], 2018



UNIVERSIDAD ARGENTINA DE LA EMPRESA

FACULTAD DE INGENIERÍA Y CIENCIAS EXACTAS

Resumen

El presente Proyecto Final de Ingeniería propone el diseño e implementación de una herramienta de procesamiento capaz de transformar grandes cantidades de datos crudos en información relevante para el usuario, aplicado a la agricultura. Esta herramienta se complementa con una interfaz gráfica interactiva que hace uso de la misma, y permitirá a usuarios con conocimientos del área mencionada tomar mejores decisiones con mayor información disponible, con el fin de optimizar su actividad comercial.

Desde lo técnico, el proyecto se divide en tres partes distintas: por un lado, la selección de sensores climáticos y la implementación de una interfaz que interactúe con ellos; el montaje de una plataforma de procesamiento de Big Data en la nube; y finalmente, la implementación de una interfaz gráfica que consuma dicho sistema, y que le permita al usuario común operar ignorando la complejidad de los procesos internos. En ésta última será en donde más valor verá dicha persona, siendo su portal de ingreso a distintas funcionalidades que serán discutidas a continuación.

Dada la naturaleza de código abierto de las tecnologías y herramientas que seleccionaremos para apoyarnos en el desarrollo de este proyecto, consideramos valioso para la comunidad que el mismo sea de la misma índole, una herramienta extensible que permita contribuciones de terceros. Se pretende que las bases del sistema demostrado sirvan para su aplicación en distintos campos de trabajo.

Entre las conclusiones a las que arribamos tras el desarrollo del presente proyecto, destacamos la observación del gran potencial que tiene la aplicación de la tecnología, y más aún, del procesamiento de Big Data en el mundo real, no solamente por la implementación que debimos realizar, sino por toda la investigación que debimos invertir detrás de ello.

Abstract

The current Final Project of Engineering proposes the design and implementation of a processing tool capable of transforming large amounts of raw data into relevant information for the user, applied to agriculture. This tool is complemented with an interactive graphic interface that makes use of it, and will allow users with knowledge of the aforementioned area to make better decisions with more information available, in order to optimize their commercial activity.

From the technical aspect, the project is divided into three separate parts: on the one hand, the selection of climatic sensors and the implementation of an interface that interacts with them; the assembly of a Big Data processing platform on the cloud; and finally, the implementation of a graphical interface that consumes said system, and that allows the common user to operate ignoring the complexity of internal processes. In the latter will be where the greatest value is presented to that person, being their portal of entry to different functionalities that will be discussed below.

Given the open source nature of the technologies and tools that we will select to support us in the development of this project, we consider that for our platform to be valuable for the community, it should follow the same principles, by becoming an extensible tool that allows contributions from third parties. It is intended for the demonstrated system's basics to become useful in different work environments.

Among the conclusions that we have reached after development of the current project, we highlight the observation of the great potential in the usage of technology, and even more, of Big Data processing in the real world, not only because of the implementation we had to carry out, but also because of all the time that we had to invest in researching the subject.

Contenidos

Agradecimientos	2
Resumen	3
Abstract	4
Contenidos	5
Introducción	9
Antecedentes	11
Descripción	13
1) Conceptos iniciales	13
1.1) Resumen del proyecto	13
1.2) Objetivos del proyecto	14
1.3) Alcance del proyecto	15
1.3.1) Operaciones dentro del alcance del proyecto:	15
1.3.2) Operaciones fuera del alcance del proyecto	16
1.3.3) Entregables	16
Caso de Negocio	17
1) Introducción	17
2) Análisis de puntos de impacto	17
2.1) Aplicación en silos de almacenamiento	21
2.1.1) Variables a monitorear	21
2.1.2) Importancia de las variables seleccionadas	22
3) Aplicación propuesta	23
Marco Teórico	25
1) Arquitecturas Big Data	25
1.1) Introducción	25
1.1.1) Limitaciones de los sistemas tradicionales	25
1.1.2) Concepto de arquitectura	26
1.1.3) Propiedades deseables de un sistema de Big Data	27
1.1.4) Arquitecturas candidatas	28
1.1.5) Criterios de evaluación	29

1.2) Arquitectura Lambda	30
1.2.1) Capa de Batch	31
1.2.2) Capa de Servicio	31
1.2.3) Capa de Velocidad	31
1.2.4) Críticas	33
1.2.5) Calificación	33
1.3) Arquitectura Kappa	35
1.3.1) Logs y Streams	36
1.3.2) Críticas	38
1.3.3) Calificación	38
1.4) Decisión	39
2) Software	41
2.1) Mensajería	42
2.1.1) Apache Kafka	42
2.1.1.1) Tópicos y Logs	42
2.1.1.2) Productores y Consumidores	43
2.1.1.3) Garantías	44
2.1.1.4) Diseño	45
2.1.2) RabbitMQ	47
2.1.2.1) Colas de trabajo	47
2.1.2.3) Exchange	48
2.1.2.4) Brokers Distribuidos	49
2.1.2.5) Replicación	50
2.1.2.6) Otras Consideraciones	50
2.2) Procesamiento (Streaming y Batch)	51
2.2.1) Hadoop	51
2.2.2) Apache Storm	52
2.2.3) Apache Samza	54
2.2.4) Apache Spark	55
2.2.5) Kafka Streams	56
2.2.6) Comparativa de alternativas	57
2.2.6.1) Modelo de procesamiento	58
2.2.6.2) Garantía de entrega	59
2.2.6.3) Tolerancia a fallos	59
2.2.6.4) Manejo de estados	60
2.2.6.5) Rendimiento	62

2.6) Conclusión	63
3) Internet of Things	65
3.1) Arquitectura (Iot-A)	65
3.1.1) Modelo de referencia	66
3.1.2) Modelo de Dominio	67
3.1.3) Modelo de Información	70
3.1.3.1) Datos en un sistema IoT	71
3.1.4) Modelo Funcional	72
3.1.5) Modelo de Comunicación	75
3.1.6) Modelo de Seguridad	77
3.2) Conclusión	77
4) Arquitectura	80
4.1) Servidor	81
4.1.1) Servicios (API REST)	81
4.1.2) Mensajería (Cluster Kafka)	82
4.1.3) Streaming	83
4.1.4) Base de Datos (Vistas)	84
4.2) Dispositivos IoT	84
4.2.1) Sensor	85
4.2.2) Placa (Microcontrolador)	86
4.2.3) Implementación de un dispositivo IoT	87
4.2.4) Modelo de Dominio	91
4.3) Infraestructura	93
4.3.1) Componentes	93
4.3.2) Dimensionamiento de hardware	94
4.3.3) Ejecución de pruebas	95
5) Análisis Técnico del Prototipo	96
5.1) Configuración para desarrollo local	96
5.2) Instalación	98
5.3) Ejecución	100
5.4) Estructura	104
5.4.1) Proyectos	104
5.4.2) Librerías	105
5.4.2.1) Back-end	105
5.4.2.2) Front-end	106
Pruebas Realizadas	107

1) Estrategia de prueba	107
2) Ejecución de Pruebas	108
Discusión	110
Conclusiones	113
Bibliografía	115
Anexo	118
1) Modelo de capas arquitectónicas (IBM)	118
2) Mapeo de patrones atómicos a compuestos (IBM)	119
3) Arquitectura de la capa batch (LA)	120
4) IoT-A: Modelo de Dominio	121
5) IoT-A: Modelo de Información	122
6) Especificaciones técnicas de Arduino UNO REV 3	123
7) Programa de lectura análoga de una entrada en Arduino	124
8) Agregación por ventana de tiempo con Kafka Streams	124
9) Configuración de Kafka (sólo valores personalizados)	125
10) Configuración de Servidores Remotos	125

Introducción

A lo largo de las últimas dos décadas, la informática remodeló la mayoría de los aspectos de la vida de los seres humanos. Ya sea en el ámbito laboral como en el social, desde decisiones políticas de gran impacto mundial como en tareas domésticas, el manejo de la información se ve como protagonista en un gran abanico de situaciones que vivimos, y al pasar los años, esta influencia promete una tendencia de crecimiento sostenido e ininterrumpido.

Sin embargo, así como la tecnología de las computadoras progresa, también lo hacen las necesidades de los usuarios, quienes cada vez se ven más acostumbrados a encontrarse a segundos de la información que desean obtener. Esto conlleva a que las capacidades de procesamiento y distribución de la información que posee el software existente también deben aumentar, dado que existirá siempre un límite en el impacto de escalar el hardware sobre los atributos de calidad con los que los usuarios operan (velocidad de respuesta, disponibilidad, fiabilidad, etc.). Es por ello que nos encontramos en una era en la que no solamente alcanza con desarrollar equipos más potentes: debemos acompañar esto con mejores técnicas con las que utilizamos la tecnología con la que disponemos.

El desarrollo de mejor software es entonces, en realidad, una disciplina que pone en práctica no solamente la lógica para cumplir requerimientos, sino que también requiere un análisis metódico de los problemas para alcanzar una solución, y de las soluciones existentes e implementadas para solucionar problemas. Guiándonos entonces por los requerimientos no funcionales de los usuarios, restringimos y guiamos el software, con el fin de que este proceso no nos entregue un monolito que cumple con los requerimientos funcionales y ya, sino que lo logre de la mejor manera y que se encuentre constantemente evolucionando. De esta manera, en los últimos años surgieron “técnicas” como Big Data y IoT, que buscan proveer a los ingenieros con estructuras arquitectónicas y de pensamiento, con el fin de lograr un acercamiento mayor de los usuarios con la tecnología, aún cuando éstos no lo perciben tanto.

En cuanto a los campos de aplicación, tal y como hemos mencionado, no se ven limitados a la aplicación por profesionales informáticos, sino que también puede ser extendido a todo aquel que requiera información agregada, procesada y que le brinde valor, ya sea en el área de los negocios, como en sus actividades sociales y recreativas.

El proyecto que será descrito a lo largo de este informe se divide conceptualmente en dos partes: la primera es la investigación y justificación del uso de las herramientas que permiten llevar a cabo las tareas más demandantes en cuanto a la obtención y procesamiento de información en grandes cantidades. La segunda parte consta de la implementación de los conceptos teóricos desarrollados, en forma de una plataforma interactiva que permita visualizar la información “digerida”, con aplicación en un campo que, en la discusión, consideremos conveniente para la demostración de lo aprendido a lo largo del proyecto.

Antecedentes

Cada año que transcurre, las industrias dedican un mayor porcentaje del tiempo y presupuesto en comprender y aprovechar el valor de la información que circula dentro y fuera de las organizaciones. Ésto no se ve únicamente en compañías dedicadas a la ingeniería de información, sino que también podemos encontrarlo en distintos sectores, disparando así cada vez cambios sociales y culturales de mayor envergadura.

El planeamiento dentro de los gobiernos es un gran ejemplo, como en el caso de Boston que, ante indicadores de problemas en materia de seguridad, educación y tránsito, ha tomado la decisión de utilizar técnicas de relevamiento y procesamiento de información en tiempo real con el fin de direccionar la agenda política.

En el entretenimiento, mientras tanto, productores de Hollywood, estudios de televisión y financiadores del cine se encuentran ajustando analíticas que les sirven de guía en la creación de contenido. Vinny Bruzzese, estadista y “científico” de Hollywood, asegura que logra aumentar sus ingresos gracias a la gran disponibilidad de datos, extraídos de cientos de años de películas, mediante los cuales es capaz de identificar con precisión detalles minúsculos detalles con gran impacto en alzar una película en un éxito de proporciones sin precedentes. Según el estadista, él y su equipo de analistas examinan la estructura de la historia contada y el género de nuevos guiones, y los comparan con aquellos de películas que marcaron un antes y un después en la historia del cine, con el fin de señalar qué es lo que posibilita que un guión se asegure reconocimiento y premios.

Lo mismo para el deporte, en el cual el rol del concepto de “analytics” resulta prominente. La estadística ha sido un gran actor en el baseball desde sus comienzos, y es visto hoy como la llave que servirá para desbloquea el potencial de los equipos y sus jugadores. En los últimos años se ha dado un paso mayor, embebiendo dispositivos de monitoreo del rendimiento de los jugadores en tiempo real, integrando IoT y Big Data en un único producto, con el fin de proveer a los entrenadores y cazadores de talentos con herramientas que predicen la evolución de un jugador en el largo plazo.

Por otro lado, los estados se han visto utilizando estas técnicas para prevenir y recuperarse de los desastres naturales que amenazan con azotar a las distintas poblaciones del mundo. Un ejemplo claro es el de un terremoto que, en el año 2012, dejó devastada la ciudad Christchurch, Nueva Zelanda. Esta ciudad se encuentra utilizando procesamiento de datos en tiempo real para ayudar con los esfuerzos de recuperación y reconstrucción, mediante un proyecto que analiza información obtenida por sensores en tiempo real (sistemas de cámaras, tráfico, administración de construcción, calidad de aire y agua) para ponerla al servicio del planeamiento urbano y re envisionamiento de cómo debe operar la ciudad.

Queda evidenciado que son abundantes los ejemplos que se pueden encontrar del uso beneficioso de Big Data e IoT, aún cuando nos han quedado algunos por nombrar (salud, educación, igualdad social, energía). Es incluso sencillo, desde el punto de vista de una persona familiarizada con la tecnología, hallar que el manejo de información mediante Big Data posee un valor intrínseco: no depende del campo de aplicación, sino que es propio y se expresa por sí mismo. Es por ello que hemos decidido abordar este proyecto, para construir y distribuir conocimiento sobre cómo la tecnología afectará nuestras vidas de aquí en más. Nuestro propósito es, además, expandir sobre lo mencionado y realizar nuestro propio acercamiento a éste tipo de técnicas, mediante nuestra propia implementación de los lineamientos de Big Data e IoT, aplicados en un campo de aplicación que determinaremos más adelante en el desarrollo del proyecto.

Descripción

En la siguiente sección se justificará la realización de un proyecto para la implementación de un Sistema de Monitoreo de Silos. Para ello se definirá el alcance del proyecto, las necesidades de negocio que dan lugar al sistema, la estrategia adoptada para el desarrollo del sistema, y un plan a alto nivel para la organización de tareas.

1) Conceptos iniciales

En esta sección procederemos a presentar y describir la solución propuesta en base al objetivo inicial, considerando los antecedentes observados en la sección anterior. Dicho objetivo, como ya fue mencionado en la introducción, consiste en la entrega de un prototipo funcional que, aprovechando el uso de nuevas maneras de pensar la información de nuestros sistemas, consista en una plataforma usable por usuarios dentro de la agricultura, permitiendo así a la persona administrar de manera óptima el almacenamiento de sus productos en silos.

1.1) Resumen del proyecto

Durante los últimos años han ido cambiando las necesidades de datos de las empresas. Esto se debe principalmente al aumento significativo en la producción de datos a nivel global, que entre 2009 y 2015 se ha visto multiplicada por un factor de diez¹. Esto crea nuevas oportunidades para todas las industrias, incluyendo a la agronomía, hacia la cual está orientada el presente proyecto.

De las diversas áreas que componen las prácticas de la agronomía, se estará apuntando a solucionar la referida al monitoreo de las condiciones ambientales de silos de granos, mediante una red de sensores conectados por internet y un sistema apto para el procesamiento de volúmenes de datos masivos. Con esto se pretende ofrecer una herramienta de alta

¹ http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode

precisión, que, mediante reportes de estado en tiempo real, facilite la toma de decisiones a los productores de granos, teniendo en cuenta condiciones de almacenamiento de la producción.

A través de este proyecto se introducirá en el mercado, un servicio orientado al monitoreo remoto de silos de granos al que los clientes podrán acceder mediante sus computadoras, dispositivos móviles, u otras terminales habilitadas para la visualización de páginas web. El back-end del sistema se montará en la nube, por lo que el costo de introducir nuevos clientes no supondrá complejidad más allá de la capacidad de la arquitectura de escalar horizontalmente o verticalmente de acuerdo a la demanda, y la instalación de una red de sensores conectados al sistema central en los silos.

El servicio proveerá al operario reportes y gráficos en tiempo real, o en ventanas de tiempo determinadas, de las condiciones ambientales de sus instalaciones. También posibilitará la configuración de alarmas que notifiquen la detección de parámetros fuera del rango aceptable de operación a lo largo de períodos de tiempo determinados.

1.2) Objetivos del proyecto

Los objetivos del proyecto se categorizan de la siguiente manera:

- Comprensión del negocio: Se realizarán tareas de investigación para profundizar el conocimiento sobre las variables que se deben considerar para el mantenimiento de silos, y cómo pueden afectar a la calidad del grano.
- Especificación y análisis de requerimientos: Se deben identificar los requerimientos funcionales y no funcionales que deberá satisfacer el servicio.
- Diseño funcional: En esta etapa se generará la documentación relevante que describa el comportamiento del sistema.
- Diseño técnico: Se diseñará la arquitectura del sistema, especificando cada uno de los componentes del mismo, y justificando las decisiones tomadas.
- Desarrollo back-end: La primera parte del desarrollo se enfocará en el sistema central y su integración con la red de sensores.
- Desarrollo front-end: Se desarrollará una interfaz web para acceder a los servicios.

-
- Implementación: se realizarán las tareas de configuración necesarias para trasladar una parte del sistema a un entorno virtual hosteado por un proveedor.
 - Pruebas: se realizarán las pruebas que se consideren pertinentes para poder asegurar el buen funcionamiento del sistema, simulando y/o tomando datos reales para la justificación del comportamiento.

1.3) Alcance del proyecto

1.3.1) Operaciones dentro del alcance del proyecto:

Investigación

- Consulta con expertos y bibliografía para mejorar la comprensión del negocio
- Diseño funcional de alto nivel
- Estudio de alto nivel de las mejores arquitecturas aplicables al caso de negocio
- Elección de librerías de código abierto que se utilizarán en la implementación
- Estudio detallado de los sensores ambientales y cómo integrarlos con el sistema

Desarrollo

- Sistema de almacenamiento basado en colas
- Sistema de procesamiento (batch y streaming)
- Simulador de las entradas (input) de los sensores
- Aplicación web para visualizar los reportes

Implementación

- Consideraciones de la infraestructura necesaria
- Implementación real de uno de los sensores requeridos
- Montaje parcial del sistema en la nube
- Estudio de rendimiento del sistema en funcionamiento

1.3.2) Operaciones fuera del alcance del proyecto

- Análisis de cobertura de testing
- Implementación real de todos los sensores ambientales
- Documentación funcional de bajo nivel (casos de uso, diagramas de clase)
- Análisis de bajo nivel de las arquitecturas disponibles
- Análisis económico detallado de los costos de implementación

1.3.3) Entregables

- Documento de justificación del proyecto
- Investigación de arquitecturas Big Data
- Investigación de sensores IoT
- Documento de arquitectura del sistema
- Configuración de la red de sensores
- Código del sistema central
- Código de la interfaz web
- Código del microcontrolador del sensor de temperatura
- Análisis de performance del sistema integrado

Caso de Negocio

1) Introducción

Durante décadas, la producción de cereales se ha establecido como la base de toda una industria global, atrayendo emprendedores de todas las categorías, ya sea grandes corporaciones, como pequeños productores. En un sector donde los involucrados deben diariamente realizar grandes cantidades de decisiones de intrincada complejidad, y con innumerables factores que influyen sobre ellas, ha comenzado a prevalecer un sentido de necesidad por el correcto planeamiento de los cultivos, y de dar respuestas inmediatas con la mayor cantidad de información de la que se pueda disponer.

Una de las mayores dificultades con las que se ha cruzado esta actividad es el almacenamiento en buenas condiciones de sus productos. Prevenir el daño a los granos debido a condiciones climáticas supone un proceso que debe ser planificado con mucho cuidado y conocimiento. Es por ello que consideramos que la tradición agricultora se ubica como un objetivo apropiado para la utilización de Big Data y IoT, con el interés de construir una interfaz para lograr generar, a futuro y con suficiente certeza, los parámetros que optimicen la actividad.

2) Análisis de puntos de impacto

Para señalar la importancia que puede llegar a tener Big Data en la producción de cereales, y por lo tanto, cuál puede ser el impacto de no utilizarlo en nuestras vidas, analizamos el impacto de las incorrectas (o carentes) acciones llevadas a cabo como respuesta a situaciones anormales. Tomamos en cuenta lo sucedido con las plantaciones de arroz en Sri Lanka durante 2016 y 2017, en donde la economía del país se vió severamente afectada de manera negativa, alcanzando este efecto indeseado a la producción mundial de algunas variedades de cereales y arroz

Poniendo el problema en números, se calcula que para 2017 disminuirá aproximadamente un 45% la producción de arroz, estimado en 2.7 millones de toneladas, y provocando la nueva inseguridad alimenticia de casi 900.000 personas en el país. Así mismo, se necesitaría de millones de dólares en importación de cereal, harina y maíz, entre otros, solamente para satisfacer la demanda del país.

Por otro lado, se cree que el problema no termina ahí, sino que trascenderá sobre la producción de la temporada 2017/2018, empeorando la situación no sólo para los más necesitados, sino también en materia de exportación: un país que no logra satisfacer demanda local, pocas opciones tiene respecto en materia de exportación.

Las razones de esta situación se pueden ubicar, según el departamento meteorológico, en los cambios climáticos, particularmente en la variabilidad de los patrones de lluvia, de acuerdo a intensidad y frecuencia, lo cual afecta variables como la humedad y la temperatura, y aumenta las probabilidades de uno de los fenómenos naturales más temidos por los productores: las inundaciones.

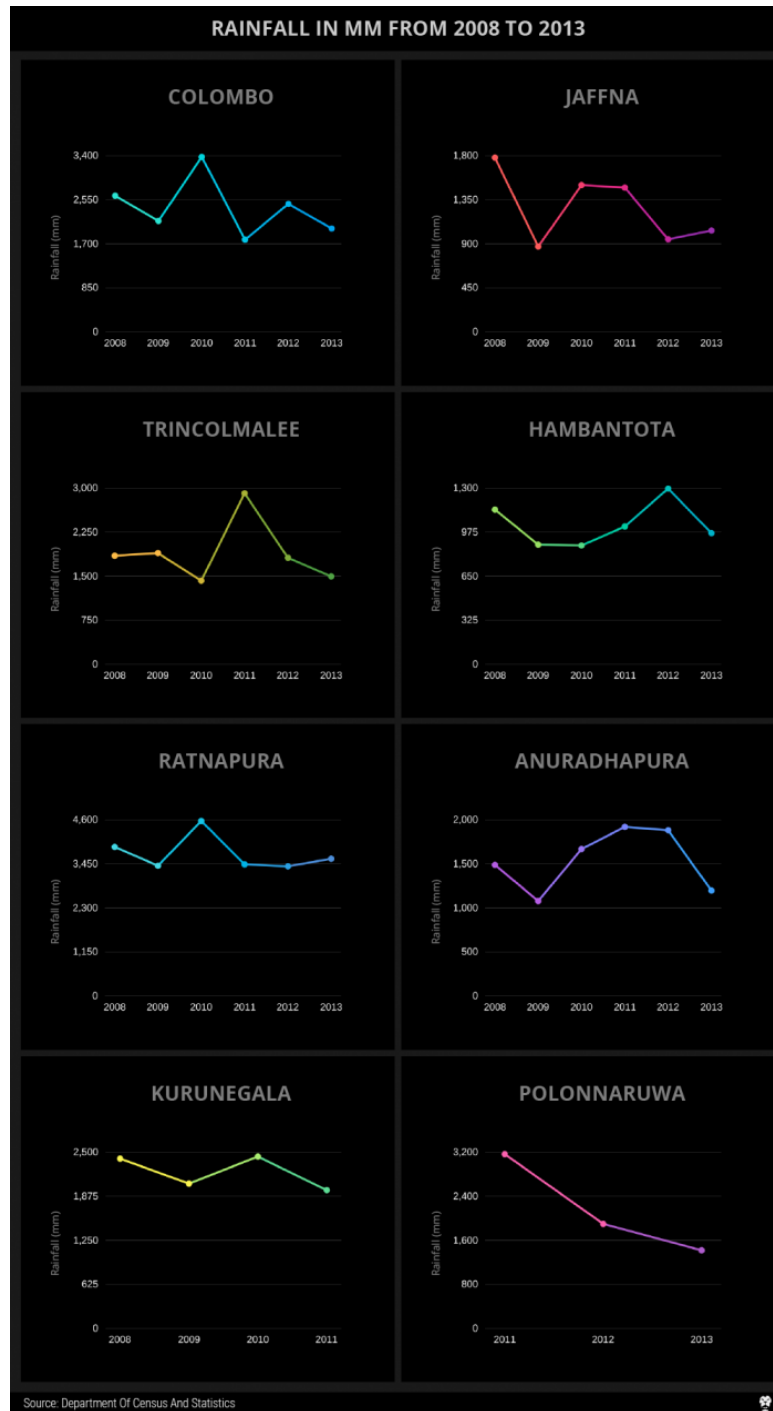


Figura 1. Disminución en intensidad de las lluvias (en mm)
(en distintos puntos del país, desde 2008 a 2013)

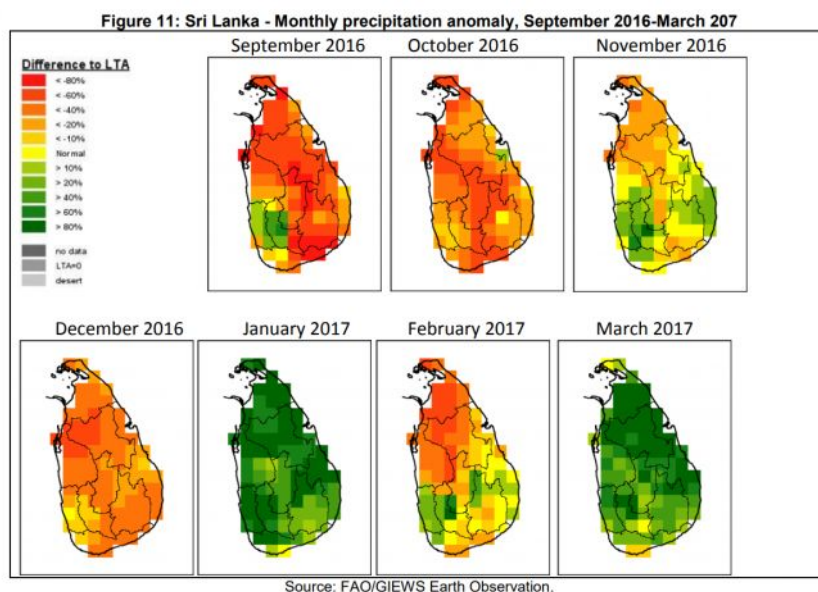


Figura 2. Variación en el promedio a largo plazo en intensidad de las lluvias (en mm)
(en distintos puntos del país, desde Septiembre 2016 a Marzo 2017)

Un caso similar ha sido el de Colombia, entre 2007 y 2012, viendo disminuida la producción de arroz de seis a cinco toneladas por hectárea. Frente a esta alarma, científicos del “Centro Internacional de Agricultura Tropical” solicitaron a la asociación de productores de arroz colombiana la información de monitoreo de las plantaciones, y pusieron a prueba avanzados algoritmos de explotación de información para desvelar patrones ocultos dentro de esos datos, con el fin de identificar la causa de la problemática, y trabajar en una solución.

Si bien los resultados que obtuvieron fueron muy específicos para cada locación en particular, también llegaron a la conclusión de que si existían ciertos factores particulares del clima que no debían ser pasados por alto, relacionados con la radiación solar recibida por el producto.

Si quisiéramos ubicarnos espacial y temporalmente en nuestra propia realidad, podemos tomar lo que varios expertos del agro argentino contemporáneo han expresado a favor de Big Data. Muchos agricultores de vanguardia consideran a la tecnología como un “puente”, el cual les permite conectar de manera más certera con su producción. Según los especialistas en el área, mediante la aplicación de las técnicas mencionadas logran llegar a

información lo suficientemente depurada para monitorear anomalías, plagas o necesidad de nutrientes, de manera tal que el productor puede tomar decisiones clave que no sólo producen un ahorro en tiempo y costos, sino que a su vez contribuyen en el cuidado del medioambiente. Esto significa que los beneficios no sólo significan ganancias económicas para quienes lo incorporan, sino que, en cierto sentido, nos beneficia a todos.

Desde aquí encontramos dos puntos a tratar: por un lado, la aplicación de Big Data específicamente en silos de almacenamiento, y por el otro, cómo construir una estructura que permita estos análisis.

2.1) Aplicación en silos de almacenamiento

Los silos de almacenamiento han sido actores presentes en el mundo, en alguna forma u otra, desde el Siglo VIII A.C.. Si bien las técnicas de construcción y mantenimiento de silos han ido evolucionando a lo largo del tiempo, y ya sea para almacenar arroz, madera, cereales, harina u otros, el concepto es el mismo: preservar elementos en condiciones controladas y óptimas, con el fin de entregar el mejor producto posible.

2.1.1) Variables a monitorear

Ya que los principios son, en general, los mismos, y teniendo nuestra incógnita de cómo puede la tecnología aportar en este área, podemos identificar las variables que nos interesa monitorear, aquellas que entregarán mayor valor a las aplicaciones que pudieran ser construidas sobre nuestro prototipo.

- Temperatura
- Humedad

2.1.2) Importancia de las variables seleccionadas

Con el fin de mantener el grano en condiciones saludables, es requerido lograr almacenarlo seco y fresco, es decir, mantener a raya los niveles de humedad y calor. De no lograrse contener estas variables dentro de umbrales aceptables, se generaría un ecosistema que aceleraría el crecimiento de insectos (plagas), hongos y, posiblemente, micotoxinas producidas por estos últimos, poniendo en peligro la salud de sus potenciales consumidores y de otras producciones con las que entrara en contacto.

Los granos y semillas son buenos aislantes, es decir, alojan bien la temperatura. Cuando la temperatura aumenta, se producen corrientes de convección: el calor se transfiere hacia arriba y aparecen “zonas de calor”. Así aumentan los niveles de humedad por condensación en la superficie, y se acumulan condiciones indeseables en ciertas ubicaciones fácilmente distinguibles.

Nos resulta interesante poder determinar en dónde se encuentran las “zonas de calor”, dado que los requerimientos del esquema de refrigeración, como la ubicación o presión estática ejercida por los ventiladores del silo, se vería afectado por ellas. No le resultará igual al usuario tener un incremento marcado de humedad y/o temperatura en una esquina del silo, que tenerlo en el centro, en cuanto a la respuesta que deberá dar frente a cada circunstancia, así como no es lo mismo que la producción sufra de altas temperaturas por un choque de calor (la respuesta sería ventilar), que por un exceso de uso del mecanismo de secado (la respuesta sería dejar de secar).

Es por todo lo mencionado con anterioridad, que encontramos un lugar para Big Data e IoT. Ayudarán en la tarea generar valor en forma de un mecanismo con la capacidad de monitorear grandes cantidades de información sobre estos dos parámetros básicos, con el fin de poder reaccionar o accionar anticipadamente a cambios extremos en las condiciones de almacenamiento, ya sea tanto por fallas mecánicas del sistema de refrigeración y secado, como por fenómenos ambientales tales como un choque térmico.

3) Aplicación propuesta

A través de este proyecto se introducirá en el mercado, un servicio orientado al monitoreo remoto de silos de granos al que los clientes podrán acceder mediante sus computadoras, dispositivos móviles, u otras terminales habilitadas para la visualización de páginas web.

La plataforma se encontrará dividida en tres módulos que interactúen entre sí, con el fin de brindar mayor valor de negocio que la sumatoria de sus partes individuales:

- Procesamiento en streaming y almacenamiento de datos obtenidos de los sensores.
- Visualización gráfica de los datos obtenidos históricos y en tiempo real, por silo y por zona espacial dentro del mismo.
- Configuración y activación de alarmas, mediante la definición de umbrales de aceptación a lo largo de un período de tiempo.

La aplicación deberá tener la capacidad de recibir todas las métricas medidas mediante varios sensores, en uno o más silos a la vez, y procesarla con el fin de transformar datos en información mediante las tecnologías seleccionadas anteriormente para esta tarea. El usuario deberá ingresar a la plataforma, y desde ella podrá visualizar en tiempo real las condiciones de las variables seleccionadas, diferenciada por silo y por zona en la que se encuentre desplegado cada sensor. Esto significa que la aplicación recibirá de éstos no solamente las variables climáticas, sino que también el identificador de cada sensor de acuerdo a la zona de cobertura que posee.

Además de poseer reportes gráficos de la temperatura y humedad media del silo, tanto en tiempo real como en ventanas de tiempo determinadas, posibilitará la visualización de esta misma información dentro de cada zona diferenciada dentro del silo. Con esto, pretendemos cubrir la necesidad de que el usuario pueda tomar decisiones de acuerdo a cada zona de análisis, de manera tal de que, a su vez, logre comprender los efectos que tienen sus acciones realizadas en un área del silo sobre otra distinta.

Por otro lado, se posibilitará la creación de alarmas que notifiquen la detección de parámetros fuera de los umbrales de aceptación definidos por el usuario, a lo largo de períodos de tiempo determinados.

Marco Teórico

1) Arquitecturas Big Data

1.1) Introducción

1.1.1) Limitaciones de los sistemas tradicionales

El crecimiento de la producción de datos pone en evidencia las limitaciones de los sistemas de base de datos tradicionales, generalmente basados en el modelo relacional. Estos sistemas y las técnicas de administración tradicionales limitan las capacidades de los negocios de escalar a Big Data.

Para afrontar los desafíos de Big Data, se desarrolló un nuevo conjunto de herramientas, que se suelen agrupar dentro del término NoSQL. Estos sistemas permiten escalar a sets de datos más grandes, pero cada tecnología por sí misma no es suficiente para lograr este objetivo, sino que se deben implementar dentro de un marco de trabajo que incluye múltiples herramientas.

Estas tecnologías tienen distintos orígenes. Algunas fueron creadas por Google, particularmente los sistemas de archivos distribuidos y el framework de computación MapReduce. Por otro lado, la comunidad de open source contribuye mediante proyectos como Hadoop, MongoDB, Cassandra, RabbitMQ, Kafka, Storm, entre otros.

Es necesario replantear completamente el diseño de los sistemas de datos para lograr adaptación al procesamiento de Big Data. Para justificar esto, primero es necesario analizar las situaciones en las que las técnicas tradicionales de escalabilidad de datos alcanzan su límite.

- *Implementación de colas intermediarias:* Consiste en evitar las operaciones individuales mediante un intermedio entre la aplicación y la base de datos, favoreciendo las operaciones en batches. A medida que aumenta la demanda se pueden añadir servidores para paralelizar las actualizaciones, pero eventualmente llega un punto en el que la base de datos se convierte en el factor limitante.
- *Fragmentación de la base de datos (particionamiento horizontal):* Consiste en la distribución de la carga de la base de datos en varios servidores. Si bien esto aumenta la capacidad de procesamiento de datos, demanda tareas de redistribución de los datos actuales en forma equitativa y la configuración de los distintos workers de la aplicación para utilizar el fragmento adecuado. A medida que aumenta el volumen de los datos se crean nuevas particiones, y se va volviendo más complicado responder a problemas como las fallas de disco y la corrupción de datos resultante de errores humanos.

El problema central reside entonces en la base de datos misma, que al no ser distribuida por naturaleza, obliga a los desarrolladores a implementar comportamientos complejos de paralelización, colas, particiones y réplicas. Los errores humanos son inevitables, por lo que la tolerancia a los mismos es una característica fundamental en un sistema escalable.

1.1.2) Concepto de arquitectura

Para los propósitos del presente análisis, una arquitectura es la disposición de los componentes tecnológicos en las distintas capas de un sistema respetando determinados patrones de diseño.

El primer concepto que aparece en la definición es el de componentes tecnológicos. Este término se refiere a todas las piezas de software que se integran en el sistema. En el contexto de Big Data se destacan los servicios de mensajería (colas), procesamiento batch y

streaming. Se incluye además el medio de visualización de los datos y las herramientas de aprendizaje automático (machine learning).

Cada componente opera en una capa determinada del sistema. Un grupo de arquitectos de IBM² define un modelo de cuatro capas arquitectónicas: de fuentes de datos, de mensajería y almacenamiento, de análisis y finalmente de consumo. Las arquitecturas que se analizarán no adoptan este modelo al pie de la letra, pero comparten varios de sus principios y por lo tanto la separación de responsabilidades que resulta tiene varios puntos en común.

El concepto que resta por definir es el de patrón de diseño. IBM diferencia los patrones atómicos de los compuestos. Los primeros responden a necesidades acotadas de negocio, como por ejemplo las notificaciones automáticas o el análisis de datos históricos. A estos los agrupa en patrones de almacenamiento, acceso, procesamiento, y consumo (nótese la similitud con el modelo de capas). Los patrones compuestos agrupan a los atómicos formando soluciones genéricas a situaciones de negocio comunes. El patrón compuesto que mejor se ajusta al presente caso es el de *Actionable Analysis - Manual Action*. Esto significa que el sistema recomienda acciones basado en el resultado de un análisis, pero es el usuario el que decide y ejecuta las acciones.

1.1.3) Propiedades deseables de un sistema de Big Data

La arquitectura de un sistema de Big Data debe ser tal que favorezca la escalabilidad y al mismo tiempo mantenga la complejidad dentro de niveles tolerables. Esto significa que no solo debe ser eficiente en el uso de los recursos sino que también debe ser entendible con facilidad. Para poder garantizar el cumplimiento de dichas necesidades, un sistema debe poseer las siguientes propiedades:

1. *Tolerancia a fallos*: Varias de las decisiones en el diseño están orientadas a definir el comportamiento ante las dificultades inherentes a los sistemas distribuidos, como las fallas aleatorias de hardware, las semánticas de consistencia de las bases de datos

² Mysore, D., Khupat, S., & Jain, S.. Understanding atomic and composite patterns for big data solutions, Big data architecture and patterns <<https://www.ibm.com/developerworks/library/bd-archpatterns4/index.html>>

distribuidas, la concurrencia y la duplicación de datos. Se suele referir a estas dificultades mediante el Teorema CAP, que será analizado más adelante.

2. *Lectura y escritura de baja latencia*: Ciertas operaciones exigen que la latencia sea mínima, típicamente menos de un segundo. El sistema debe asegurar que sea posible lograr esta latencia cuando es necesario, sin comprometer la robustez del sistema.
3. *Escalabilidad*: Habilidad para mantener el rendimiento del sistema frente a la demanda en aumento, agregando recursos al sistema. La arquitectura debe asegurar escalabilidad horizontal con baja complejidad.
4. *Extensibilidad*: El costo de desarrollar nuevas funcionalidades debe ser mínimo. Esto incluye los casos en los que es necesario realizar migraciones a gran escala de datos a un nuevo formato.
5. *Consultas Ad-Hoc*: La capacidad para realizar consultas arbitrarias sobre el set de datos es importante para descubrir oportunidades de negocio.
6. *Mantenimiento mínimo*: Se deben elegir componentes que tengan la menor complejidad de implementación posible, y que simplifiquen los procesos de agregado de nuevo hardware y solución de problemas en producción.

Existen otros factores de calidad como la depurabilidad y la generalización, pero se priorizan los listados. Se analizó previamente como la complejidad operacional en los sistemas de datos tradicionales crece con el volumen de los datos, y por qué es necesario un enfoque diferente para el procesamiento de Big Data. Establecer qué propiedades debe tener una buena arquitectura sirve como referencia al evaluar cuál es la más adecuada para un proyecto particular.

A continuación se estudiarán las arquitecturas de mayor renombre en la actualidad, con el objetivo de decidir cuál será adoptada en el desarrollo del sistema de monitoreo de silos.

1.1.4) Arquitecturas candidatas

La investigación inicial para este análisis consistió en identificar potenciales arquitecturas para el sistema. Condiciones excluyentes para la selección fueron el nivel de detalle de la documentación y la aprobación general de la comunidad. Los planteos sobre el diseño de sistemas de Big Data son varios, pero una gran parte no cuenta con información concreta de su implementación, limitándose a los conceptos puramente teóricos.

Las arquitecturas que satisfacen estas condiciones son Lambda y Kappa. Lambda, que tiene sus orígenes en un ensayo de Nathan Marz en 2011, es quizás la más popular actualmente, y está respaldada por un extenso libro del mismo autor con la coautoría de James Warren. Kappa nace en 2014 a partir de una crítica a Lambda, y define un enfoque distinto para atacar lo que el autor, Kreps Jay, considera limitaciones de la arquitectura. Ha crecido en popularidad desde entonces y existen varios recursos en línea para profundizar en sus conceptos.

1.1.5) Criterios de evaluación

Para llegar a una decisión fundamentada de manera objetiva, se establece un criterio de evaluación unificado para la comparación de arquitecturas. Este estará compuesto por valoraciones en una escala de 1 - 5, cada una con un peso entre 0 y 1 para la apreciación final. Los pesos asignados son subjetivos y acordes a la importancia que se le atribuye a cada uno en el desarrollo del sistema de monitoreo de silos.

Las valoraciones escogidas se refieren por un lado a algunas de las propiedades deseables del sistema enumeradas previamente, así como a otros factores que se consideran de relevancia, como la documentación disponible.

Tabla 1. Criterios de comparación de arquitecturas Big Data

Propiedad	Peso
Tolerancia a fallos	0.9

Latencia	0.75
Escalabilidad	0.95
Mantenimiento	0.8
Documentación	0.4
Extensibilidad	0.4
Consultas ad-hoc	0.5

1.2) Arquitectura Lambda

Nathan Marz acuña el término *Lambda Architecture (LA)* para una arquitectura genérica, escalable y tolerante a fallos, basado en su experiencia en sistemas distribuidos en Backtype y Twitter. Parte de la premisa de que no existe una solución única a los problemas de computación de funciones arbitrarias sobre datos arbitrarios, y que se debe combinar una variedad de herramientas y técnicas para construir un sistema de Big Data.

La idea principal de Lambda es construir un sistema como una serie de capas, cada una operando sobre funcionalidad provista por la anterior. Las capas que se distinguen son la de batch, la de servicio y la de velocidad. Esta decisión se fundamenta en la realidad de que es poco factible técnicamente, o económicamente, realizar consultas de manera consistente sobre sets de datos masivos. Por lo tanto, se debe proceder mediante *vistas de batch (batch views)*, de modo que las consultas no se ejecuten sobre todos los datos, sino sobre vistas indexadas que permiten acceso aleatorio. Dicho de otra manera, se descompone la ecuación $consulta = función(todos\ los\ datos)$ en dos operaciones: $vista\ de\ batch = función(todos\ los\ datos)$ y $consulta = función(vista\ de\ batch)$.

A primera vista, este enfoque tiene la desventaja de que habrá latencia entre la consulta y la generación de las vistas batch. Este tiempo puede ser considerable, especialmente si los datos acumulados aumentan. En los siguientes párrafos se describe el rol

de cada capa, detallando la solución que Lambda ofrece a esta problemática, así como a las demás dificultades propias de un sistema distribuido.

1.2.1) Capa de Batch

En esta parte del sistema se implementa la función *vista de batch* = *función(todos los datos)*. La capa de batch debe realizar dos tareas: mantener el set de datos maestro inmutable, y realizar funciones arbitrarias sobre el mismo. Para este tipo de procesamiento son ideales los sistemas de procesamiento batch como Hadoop.

El objetivo en esta capa es la implementación de un programa capaz de procesamiento paralelo distribuido en un cluster de servidores, y que escale con facilidad mediante la adición de nuevo hardware.

1.2.2) Capa de Servicio

El siguiente paso es cargar las vistas en alguna fuente donde puedan ser consultadas. La capa de servicio realiza esa tarea mediante una base de datos distribuida que carga la vista y permite accesos aleatorios, actualizando los datos a medida que son renovados por la capa de batch.

La base de datos en la capa de servicio solamente necesita soportar operaciones de actualización y lectura; no es necesaria la capacidad de escritura aleatoria. Esto es una ventaja significativa ya que las escrituras aleatorias son causa de la mayor complejidad en las bases de datos. Esta simplificación tiene como resultado un sistema robusto, predecible, fácil de configurar y de operar. Una de las recomendaciones del autor para cumplir esta función es ElephantDB.

1.2.3) Capa de Velocidad

La vista de batch que se presenta en la capa de servicio no incluye los datos que ingresaron mientras se realizaba el precálculo. La pieza faltante para un sistema en tiempo

real es una manera de compensar esta latencia, y es este el propósito de la capa de velocidad. Como el nombre lo sugiere, el objetivo es asegurar que los datos se incluyan en las funciones de consulta tan rápidamente como lo requiera el negocio.

Esta capa es similar a la de batch en cuanto a la producción de vistas a partir de los datos recibidos, pero difiere en los datos que maneja: la capa de batch opera siempre sobre todo el set de datos, mientras que la capa de velocidad solo lo hace sobre los datos recientes. Además, para minimizar la latencia, esta capa no procesa todos los datos nuevos en una sola corrida, sino que mantiene vistas en tiempo real (*realtime views*) mediante computaciones incrementales. Esto se representa en la función: *vista realtime = función(vista realtime, nuevos datos)*.

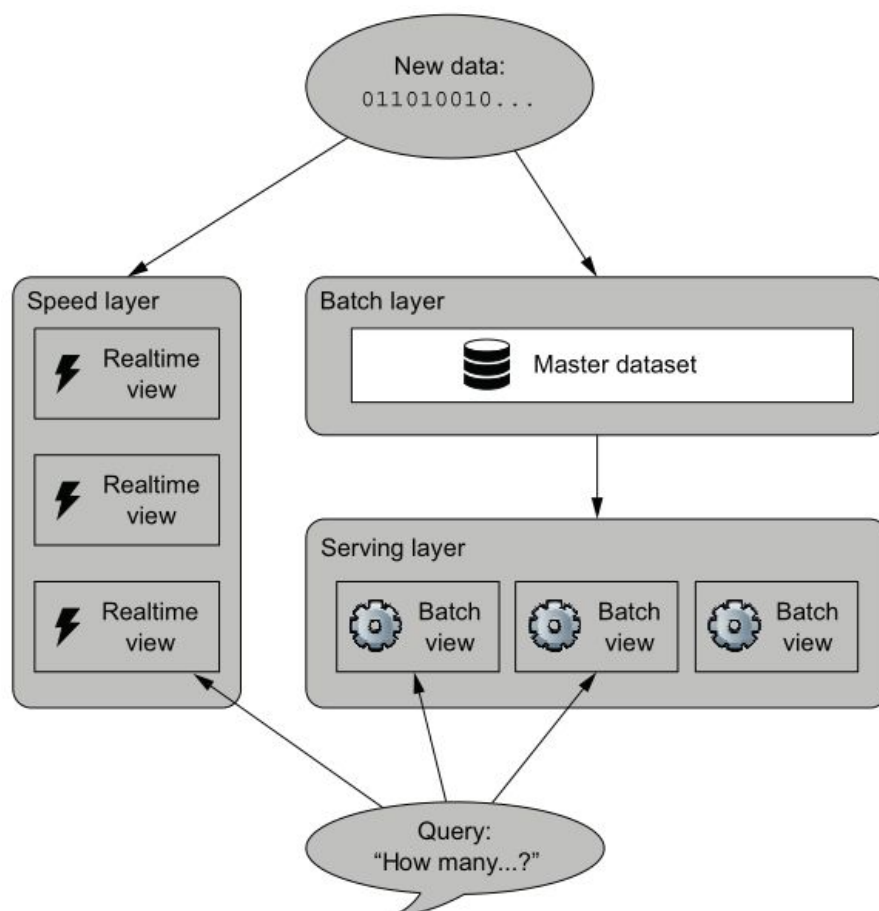


Figura 2: Diagrama de la Arquitectura Lambda

Una de las ventajas de este diseño es que una vez que los datos pasan por la capa de batch, las vistas en tiempo real asociadas pueden descartarse. Esto ofrece la ventaja de que los errores que ocurran en la capa de velocidad, que por naturaleza es más compleja y propensa a errores que las demás, serán sólo temporales. Esta propiedad es denominada por Marz como *aislamiento de la complejidad*.

Otra de las posibles disposiciones de las capas de batch y velocidad permite la ejecución de algoritmos exactos del lado de procesamiento en lotes y aproximados para el incremental. Esto es conveniente en casos donde el costo computacional del cálculo en tiempo real es muy elevado, y es aceptable entregarlos con un cierto margen de error para mejorar el rendimiento. A esta característica se la denomina *precisión eventual* - las vistas son casi exactas durante un tiempo, y completamente exactas cuando se generen en la capa de batch.

1.2.4) Críticas

La principal crítica que se le hace a Lambda es que se deben mantener dos bases de código distintas para la capa de batch y la de velocidad, y se debe asegurar que ambas produzcan exactamente los mismos resultados. La programación en sistemas distribuidos tiende a acoplarse a la tecnología de base, como Hadoop en batch y Storm en streaming. Esto genera dificultades para reutilizar el código, y resulta en una alta complejidad operacional.

Este diseño también limita la extensibilidad del sistema, ya que cualquier nueva funcionalidad va a estar acotada por la intersección entre las dos capas, es decir, para implementar una nueva abstracción en la capa de batch es necesario poder repetirlo para la de velocidad, y viceversa.

Se cuestiona además si realmente Lambda “vence al teorema CAP” como sostiene Marz. Una de las garantías que debe ofrecer un sistema distribuido según este teorema es la consistencia. Para reducir la latencia, en la capa de velocidad se sacrifica la exactitud de las vistas, y si bien asegura precisión eventual, al no ser inmediatamente consistente con los datos entrantes no cumple con la garantía de consistencia.

1.2.5) Calificación

- Tolerancia a fallos [4]: El componente principal en este diseño es Hadoop, que administra transparentemente el procedimiento contra fallos. Otra particularidad es que todas las capas son tolerantes a los errores humanos, ya que ante un fallo se pueden recomputar las vistas y remover los datos defectuosos.
- Latencia [4]: La capa de velocidad permite generar vistas en tiempo real, evitando las dificultades de procesar grandes volúmenes de datos. En algunos casos donde la computación incremental es compleja, esto significa sacrificar exactitud dependiendo que sea más costoso para el negocio.
- Escalabilidad [3]: Las capas de bache y de servicio están conformadas por sistemas completamente distribuidos, por lo que para escalar bastaría con agregar recursos físicos y un mínimo de configuración. La desventaja es que es una tarea que se debe realizar por duplicado para las distintas capas.
- Mantenimiento [2]: Si bien se utilizan herramientas que de por sí abstraen las complejidades del mantenimiento (Hadoop, bases de datos de acceso aleatorio sólo de lectura, frameworks de streaming), existe un problema fundamental en la separación de las bases de código en dos capas distintas. Replicar los cambios de una capa en la otra es un esfuerzo adicional, y más aún lo es corregir cualquier inconsistencia que pueda surgir.
- Documentación [5]: La principal referencia es el libro del mismo autor de la arquitectura, que explica con detalle y ejemplos (que incluyen demostraciones de código) la implementación de cada una de las capas. La popularidad de Lambda se ve reflejada además en distintos recursos en línea que describen experiencias con la misma y patrones para evitar problemas comunes.
- Extensibilidad [2]: Para extender la funcionalidad del sistema mediante nuevas vistas o fuentes de datos requiere la introducción de cambios en las capas de batch y de

velocidad, y si sucede que una de las dos no es factible o es muy costosa la modificación, se verán limitadas las capacidades de extensión de toda la aplicación.

- Consultas ad-hoc [4]: Al contar con un set de datos maestro completamente inmutable, realizar consultas arbitrarias es solo cuestión de implementar las vistas necesarias.

1.3) Arquitectura Kappa

Kappa tiene sus inicios en un artículo publicado por Jay Kreps, por entonces arquitecto de la infraestructura de datos en línea de LinkedIn, en O'Reilly, un repositorio de libros, conferencias y publicaciones relacionadas a IT. Una particularidad de este escrito es que la primera mitad consiste en un análisis, con crítica incluida, de Lambda, que por entonces era la arquitectura por excelencia.

La alternativa propuesta por Kreps es una que busca evita utilizar sistemas separados para el procesamiento batch y en tiempo real, para evitar los inconvenientes que se notaron previamente en Lambda. Para ello sugiere ir en contra de la presunción de que el procesamiento de streams no es capaz de operar en datos históricos, basándose en la noción de computación paralela que poseen las tecnologías de streaming modernas.

La estrategia para conseguir que el autor plantea para trasladar las actividades de reprocesamiento a los jobs de streaming se resume en los siguientes puntos:

1. Utilizar un sistema que permita retener logs completos durante el tiempo que sea necesario, como Apache Kafka.
2. El reprocesamiento se realiza por una instancia de streaming distinta a la que opera con los datos nuevos, y cuyas salidas se dirigen a tablas diferentes.
3. Cuando la segunda instancia alcanza a la primera, la aplicación es notificada para comenzar a leer datos de la nueva tabla.
4. Se detiene la instancia anterior y se elimina la tabla correspondiente.

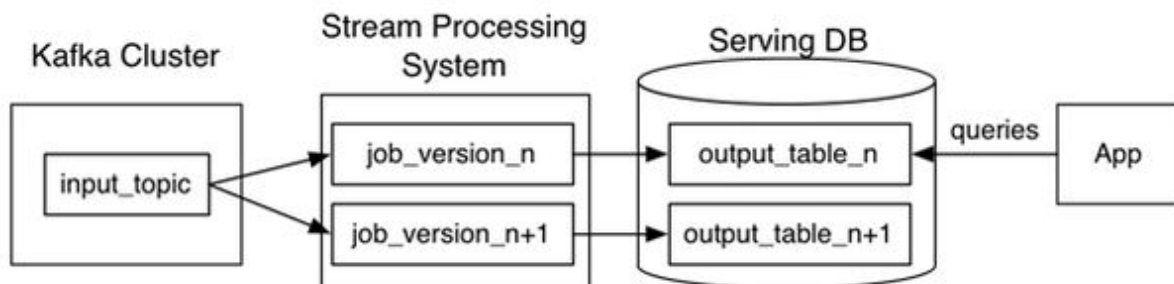


Figura 4: Arquitectura Kappa

Mediante este enfoque el reprocesamiento no se realiza periódicamente sino solo cuando ocurren cambios en el código y es necesario recomputar los resultados. Incluso en estos casos la transición es sencilla ya que el reprocesamiento es realizado de manera similar, salvo las diferencias que se hayan introducido en el código mejorado.

1.3.1) Logs y Streams

Los dos componentes fundamentales en una arquitectura Kappa son el log y los streams. El concepto de log aparece en el centro de varias aplicaciones de software, ya sean de base de datos relacionales y NoSQL, almacenes clave-valor o control de versiones. En sí, un log no es más que una secuencia de registros ordenados por tiempo, y que solo permite operaciones de anexación (*append-only*).

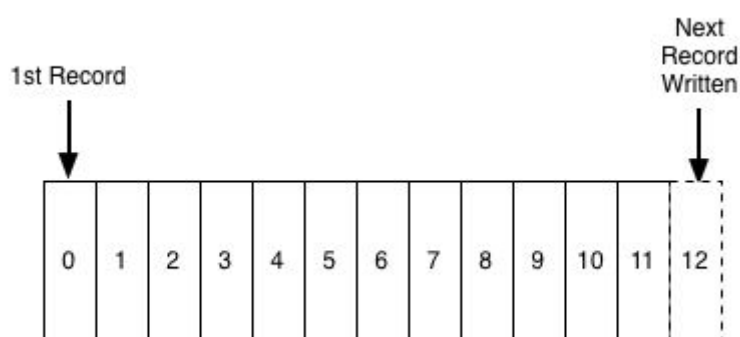


Figura 5: Estructura de un log

Basar un modelo de computación distribuida en el log tiene como ventaja la abstracción de las operaciones de integración de datos y de procesamiento en tiempo real, simplificando el diseño del sistema distribuido.

La estrategia de integración de datos en un sistema de big data debe administrar de manera eficiente el flujo entrante de eventos, ya sean clicks o acciones en una página web, reportes de actividad de servidores o mediciones de sensores IoT. Esto significa que estos datos se pongan a disposición de los consumidores de manera centralizada, con una única *fuentes de verdad*, y con interfaces de acceso definidas. El establecimiento de un sistema de almacenamiento que asegure el acceso a datos de manera segura y sencilla es la primera condición para la posterior realización de procesamiento de mayor complejidad como análisis y predicción de patrones. Un sistema basado en logs simplifica este proceso de integración ya que cada consumidor puede leer el log de manera independiente, y a partir de la posición que necesite, así como escribir nuevos registros para ser consumidos por otras aplicaciones.

Las operaciones de transformación de datos que se realizan a partir de los logs tienen forma de *stream jobs*. Streaming consiste básicamente en el procesamiento continuo de datos, diferenciándose del más tradicional procesamiento batch que opera en ventanas de tiempo determinadas. El auge de las aplicaciones en tiempo real ha aumentado la importancia de las herramientas de procesamiento en streams, dando lugar a frameworks como Samza y Storm.

Uno de los beneficios de un sistema orientado a streaming es que permiten extender las propiedades del sistema de logs mediante jobs que consumen dichos logs y generan otros por su cuenta, conservando las mismas políticas de acceso de los logs originales. Por lo tanto, los demás clientes se verán abstraídos de la complejidad de las computaciones realizadas por el job intermedio y podrán acceder al resultado de la misma forma que se accede a cualquier otro log. Esto da lugar a la generación de grafos de flujos de datos donde intervienen múltiples logs y stream jobs. Otro punto a favor de este enfoque es que si uno de los jobs llegase a fallar, los consumidores podría seguir operando hasta agotar el log, es decir, cuentan con un buffer hasta que se restaure el eslabón conflictivo.

Una faceta que requiere consideración especial en el contexto de streaming es el procesamiento dependiente de estado (*stateful*). Para realizar agregaciones en ventanas de

tiempo en tiempo real se debe almacenar algún tipo de estado, como podría ser la suma de mediciones de cada sensor en una red en la última hora. La solución más sencilla sería almacenarlo en memoria, pero ante una eventual falla este estado se perdería y, según las exigencias del negocio, podría no ser aceptable perder esta información. Una alternativa más robusta es almacenar este estado local en tablas o índices que permitan restaurarlo ante fallas. Esto permite además mantener un log de la evolución de este estado, que puede ser a su vez consultado por otros clientes de ser necesario. Luego, para controlar el tamaño del almacén de estados, se puede realizar compactación de logs, como hace por ejemplo Kafka.

1.3.2) Críticas

La simpleza de Kappa acarrea el costo de mantener múltiples tablas durante las ejecuciones paralelas de los stream jobs, demandando mayor capacidad de hardware de almacenamiento. Esto no es un punto menor si el costo de almacenamiento es elevado, como lo puede ser si se utilizan discos de estado sólido que poseen mayor velocidad de lectura.

Otra limitación que presenta esta arquitectura es que existe un lapso de tiempo durante el que los consumidores pueden llegar a leer datos desactualizados, específicamente mientras se espera que la segunda ejecución del proceso de streaming alcance los datos más recientes.

1.3.3) Calificación

- Tolerancia a fallos [4]: El log como centro de la arquitectura permite que los procesos que transforman los datos actúen de manera asincrónica e independiente entre sí, de manera que si uno falla el funcionamiento del sistema no se detiene por completo. La capacidad de replicar los logs en distintos servidores de manera transparente asegura un alto nivel de disponibilidad en todo momento.
- Latencia [3]: El foco en el procesamiento en tiempo real mediante streams busca minimizar el lapso de tiempo entre el arribo de un evento y la materialización del mismo en las vistas. Las vistas que necesitan recomputarse periódicamente se generan

en forma paralela, por lo que es posible que en ciertos contextos pase cierto tiempo hasta que los datos más recientes se pongan a disposición de los consumidores.

- Escalabilidad [4]: La capacidad del sistema de poder escalar proporcionalmente con el volumen de datos que procesa estará atada a la complejidad de incrementar el espacio de almacenamiento disponible para la retención de logs y la potencia computacional de los stream jobs. Afortunadamente, las herramientas como Kafka (logs) y Samza/Storm (streams) fueron diseñadas con la escalabilidad horizontal en consideración, simplificando el proceso de añadir nuevos nodos al sistema.
- Mantenimiento [5]: La forma en la que Kappa aborda los problemas de desarrollar un sistema distribuido se destaca por su simpleza en representar todas las transformaciones de datos en forma de streams que operan sobre un único tipo de almacén de datos en forma de log. Esto consigue que todos los procesos se representen de la misma manera, independientemente de que estos operen en tiempo real o sobre ventanas de tiempo, y no se deben mantener repositorios de código distintos para ambos casos.
- Documentación [2]: La arquitectura en sí no cuenta con documentación detallada. La principal referencia se encuentra en las publicaciones de Kreps, donde se plantean los conceptos generales de Kappa y se dan algunos indicios de cómo implementarla de manera concreta.
- Extensibilidad [4]: La independencia entre consumidores y productores facilitan la adición de nuevas funcionalidades simplemente mediante la implementación de los procesos necesarios que consultarán los logs existentes y potencialmente escribir logs nuevos, conformando grafos de flujos de datos como se mencionó previamente.
- Consultas ad-hoc [3]: Para realizar consultas arbitrarias se deben implementar los stream jobs que generen las tablas necesarias. Esta actividad supone cierto grado de esfuerzo, pero no hay restricciones en la capacidad de realizar consultas ya que se cuenta con el registro de todos el historial de eventos en el log.

1.4) Decisión

Los resultados de aplicar el criterio de evaluación definido son los siguientes:

Tabla 2. Comparación de arquitecturas Big Data

Propiedad	Peso	Lambda	Kappa
Tolerancia a fallos	0.9	4	4
Latencia	0.75	4	3
Escalabilidad	0.95	3	4
Mantenimiento	0.8	2	5
Documentación	0.4	5	2
Extensibilidad	0.4	2	4
Consultas ad-hoc	0.5	4	3
Final		15,85	17,55

El valor final indica que, para los fines del proyecto bajo análisis, será más adecuado el diseño del sistema a partir de los principios de la arquitectura Kappa. Se puede observar que las principales ventajas de Kappa en términos cuantitativos están en los atributos de escalabilidad, extensibilidad y mantenibilidad - un reflejo de la simplicidad del diseño sugerido por esta arquitectura en comparación con Lambda.

Ambos modelos definen lineamientos en el desarrollo para controlar la complejidad en sistemas distribuidos, pero el diseño basado en logs y streams propuesto por Kafka resulta más apto para sistemas donde el foco se encuentra, casi exclusivamente, en el procesamiento de eventos en tiempo real. Esta propiedad justifica en gran parte la decisión tomada, teniendo en cuenta que es esta la naturaleza del sistema de monitoreo de silos bajo análisis.

2) Software

El siguiente paso en la planificación del sistema es la elección del software de base necesario para la implementación de los principios arquitectónicos estudiados. Para ello se analizarán algunas de las distintas herramientas que ofrece el ecosistema de Big Data, que ha crecido en popularidad estos últimos años y cuenta actualmente con una gran variedad de herramientas profesionales de código libre para todas las áreas del procesamiento (mensajería, streaming y batch).

Se optará por aquellas herramientas que se considere son más adecuadas para la arquitectura Kappa y que permitan construir una base sólida sobre la cual introducir el código personalizado que implementa las funcionalidades del sistema. Serán condiciones excluyentes para esta selección que las implementaciones estén correctamente documentadas, que estén siendo mantenidas con soporte de manera activa, que cuenten con cierto grado de respaldo profesional habiéndose aplicadas exitosamente en proyectos anteriores, y que permitan la integración de los distintos bloques del sistema. Otros factores a considerar incluyen la compatibilidad con el sistema operativo Linux y las facilidades que ofrece el software para lograr escalabilidad en la aplicación.

2.1) Mensajería

Para la capa de mensajería se evaluarán dos alternativas: Apache Kafka (de *Apache Foundation*) y RabbitMQ (de *Pivotal*).

2.1.1) Apache Kafka

En la documentación se introduce a Kafka como una “plataforma de lectura continua distribuida”. Las capacidades claves de Kafka incluyen:

1. Publicación y suscripción a secuencias de registros.
2. Almacenamiento de secuencias de registros en forma tolerante a fallos.
3. Procesar los registros a medida que van ingresando.

Estas propiedades hacen que Kafka sea útil en la construcción de canales de datos entre distintos sistemas de manera confiable, así como para el diseño de aplicaciones que transforman estos datos en tiempo real.

Kafka se ejecuta como un conjunto (*cluster*) repartido en uno o más servidores. Cada conjunto almacena secuencias de registros en categorías llamados tópicos, y cada registro consiste de una clave, un valor y una marca de tiempo. La interacción con el servidor se realiza mediante APIs de publicador y consumidor, con implementaciones disponibles para distintos lenguajes de programación. La comunicación cliente-servidor se realiza a través de un protocolo especializado de alto rendimiento sobre TCP.

2.1.1.1) Tópicos y Logs

La principal abstracción para el almacenamiento de registros ofrecida por Kafka es el tópico. Un tópico es una categoría identificada por un nombre en el que se van publicando registros, y permite que la suscripción de múltiples clientes. Cada tópico se puede particionar mediante estrategias arbitrarias (aleatorias o semánticas) y distribuirse de manera transparente en distintos servidores del cluster. Cada partición contiene un log inmutable de registros, cada

uno identificado por un offset. Se pueden definir políticas de retención para controlar eliminar registros tras cierto tiempo y limitar el tamaño de la partición, aunque una de las características de Kafka es que el rendimiento no se ve afectado por el tamaño del log.

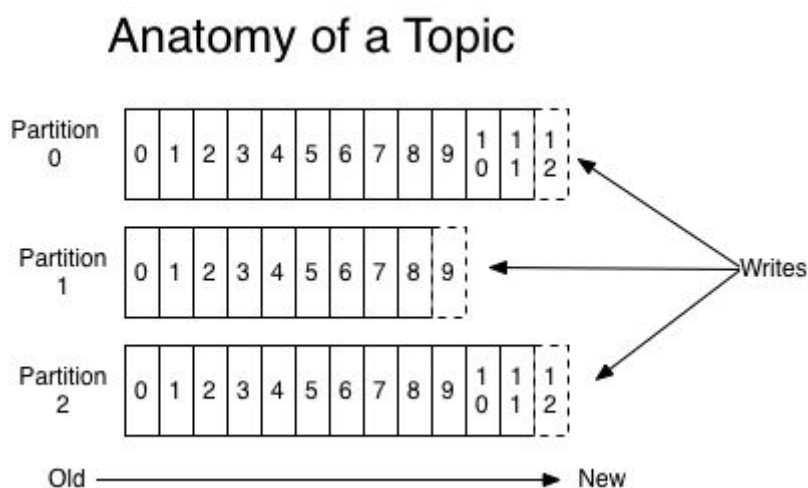


Figura 6. Tópico de Kafka

Cada servidor en un cluster de Kafka es responsable de administrar los datos y peticiones para un grupo de particiones. A su vez, cada partición se puede replicar en una cantidad configurable de servidores para lograr tolerancia a fallos. En el esquema de replicación cada partición define un servidor líder y cero o más seguidores, donde el líder gestiona todas las lecturas y escrituras a la partición mientras que los seguidores lo replican pasivamente. Si llegase a fallar un líder, uno de los seguidores lo reemplaza automáticamente. Otro efecto de esta distribución es que se consigue un buen balanceo de la carga.

2.1.1.2) Productores y Consumidores

Los productores pueden publicar datos en tópicos a elección, y son además responsables de elegir a qué partición escribir, aplicando alguna estrategia semántica (particionar por id de usuario, por ejemplo) o en forma aleatoria.

Los consumidores se agrupan en grupos con un nombre determinado, y es mediante estos grupos que se suscriben a tópicos. Cada registro que se publica en un tópico se entrega a una de las instancias de consumidores en el grupo para balancear la carga entre las instancias disponibles. La estrategia que implementa Kafka para la distribución de mensajes consiste en repartir las particiones a cada instancia de consumidor en forma equitativa. Estas asociaciones se mantienen dinámicamente, por lo que si se agregan o quitan instancias se redistribuyen automáticamente.

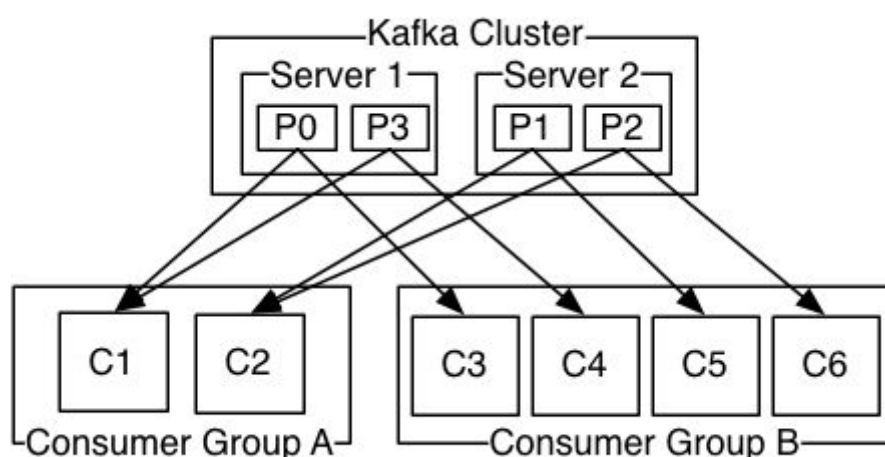


Figura 7. Grupos de Consumidores

2.1.1.3) Garantías

Kafka ofrece ciertas garantías respecto a la entrega de mensajes. En primer lugar, se asegura que todos los mensajes publicados por un productor se anexan al log en el orden que fueron enviados. Esto se refleja del lado del consumidor, que leerá los registros en el orden exacto en el que fueron almacenados. Finalmente, se asegura que un cluster de Kafka configurado con un factor de replicación de N puede tolerar hasta $N-1$ fallas de servidores sin perder datos.

2.1.1.4) Diseño

Una sección de la documentación es dedicada a explicar el razonamiento detrás de las decisiones de diseño de Kafka. A continuación se listan algunos conceptos que se consideran relevantes para la evaluación en curso.

- **Persistencia:** Kafka delega la mayor parte de las operaciones de almacenamiento al sistema operativo, en lugar de mantener caches complejos en memoria. Esto se debe a que las escrituras secuenciales en el disco son veloces por naturaleza al no realizar accesos aleatorios, y son operaciones altamente optimizables por el sistema operativo debido a su predictibilidad.
- **Eficiencia:** Se atacan dos causas comunes de ineficiencia en los sistemas de mensajería. La primera se refiere al manejo de las operaciones I/O pequeñas, que deben manejarse en baches para minimizar la cantidad de round-trips realizados. La otra dificultad se encuentra en el copiado innecesario de bytes en la transmisión de datos en red. La ineficiencia en este caso radica en que una transmisión tradicional requiere sucesivos copiados entre el espacio de kernel y el espacio de usuario hasta que finalmente se copia al buffer de red. Kafka aprovecha la llamada a sistema *sendfile* (disponible en los sistemas operativos modernos) que permite realizar esta operación puramente dentro del espacio de kernel, logrando transmisiones sin copiados innecesarios.
- **Productor:** Los productores envían datos al líder de la partición de destino sin ruteo entre medio. Para detectar cuál es el líder, todos los nodos de Kafka responden peticiones de metadatos sobre qué servidores están activos y cuáles son los líderes de cada partición. Otro detalle del productor es que los envíos se realizan asincrónicamente en baches de tamaño configurable.
- **Consumidor:** Las conexiones al servidor son iniciadas por el cliente, es decir, se utiliza una estrategia de *pull* del lado del consumidor. Las ventajas de este enfoque incluyen que el cliente nunca se ve saturado de mensajes y que las lecturas son más eficientes al

posibilitar el envío de todos los mensajes hasta alcanzar la posición actual del log. El control de posición de lectura en el log es mantenido tanto por el cliente como por el servidor para minimizar la pérdida de mensajes y reducir la posibilidad de recibir un mensaje más de una vez ante un eventual fallo.

- Semánticas de envío de mensaje: Kafka soporta, por defecto, semánticas de envío *at-least-once*, donde los mensajes se enviarán al menos una vez, pudiendo haber mensajes duplicados como resultado de fallos de sistema. Sin embargo, este control se puede refinar si los clientes realizan *commit* de sus estados entre las operaciones, permitiendo restaurar al estado deseado luego de un fallo y evitar la retransmisión de mensajes. Si bien esto agrega overhead a la comunicación, permite realizar implementaciones que garantizan semánticas de *at-most-once* y *exactly-once*.
- Replicación: Kafka fue diseñado para utilizar replicación por defecto. La unidad de replicación es la partición de tópico; cada tópico tiene un líder y cero o más seguidores. Las escrituras se realizan primero sobre el líder y luego son replicadas por los seguidores. Para manejar las fallas automáticamente, el administrador de recursos - Zookeeper en el caso de Kafka - mantiene un estado sobre el conjunto de seguidores *in-sync* con el líder, es decir, que están activos y no se encuentran “atrasados” en la replicación del log (el punto en el que se considera que una réplica está “atrasada” es configurable).
- Comparación de logs: La compactación de logs asegura que Kafka siempre retiene, al menos, el último valor conocido para cada clave de mensaje en una partición. Esta práctica es útil cuando se realizan computaciones dependientes de estado, como conteos y agregaciones, y se busca asegurar alta disponibilidad persistiendo la secuencia de cambios del estado interno. La estrategia de mantener el último estado conocido permite liberar espacio de almacenamiento sin descartar la totalidad de la información, como si se hace mediante las políticas de retención estándares.

2.1.2) RabbitMQ

RabbitMQ es un intermediario (*broker*) de mensajes. Además de cumplir las funciones básicas de comunicación asincrónica entre productores y consumidores mediante colas, facilita la distribución del trabajo entre distintos workers con alta disponibilidad. La estrategia de mensajería de RabbitMQ es administrada por unidades denominadas *Exchange* que se ubican entre el productor y las queues. Las interfaces para la comunicación entre procesos se especifican en el *Advanced Queue Messaging Protocol* (AQMP).

2.1.2.1) Colas de trabajo

Una cola de trabajo almacena mensajes y los distribuye a consumidores aplicando alguna estrategia de planificación, como round-robin. En RabbitMQ, el servidor es el que inicia la comunicación con un consumidor cuando hay mensajes disponibles, por lo que se utiliza un enfoque de *push* en vez de *pull*.

Una de las prácticas empleadas en la comunicación servidor-consumidor para lograr consistencia en la entrega de mensajes es el *acknowledgement* de mensajes. Un mensaje despachado se elimina de la queue solamente cuando el productor notifica al servidor que éste ha sido procesado correctamente. Si el cliente falla, o si se pierde la conexión antes de que el *ACK* haya sido enviado, el broker enviará el mensaje nuevamente a cualquier otro consumidor que esté disponible.

El balanceo de carga es logrado especificando la cantidad de mensajes que un consumidor puede recibir en forma simultánea. Para evitar sobrecargar a los clientes, el servidor no enviará otro mensaje hasta que no haya recibido el *acknowledgement* del mensaje anterior. Esto permite lograr una distribución más pareja entre varios consumidores cuando el tiempo de procesamiento necesario para cada mensaje no es constante.

Otro parámetro configurable en una queue es su durabilidad. Por defecto, las colas en RabbitMQ no son persistentes, por lo que si se reinicia una instancia se descartan los datos temporales, incluyendo mensajes posiblemente no enviados. Si la criticidad de los mensajes

determina que este comportamiento no es adecuado, se puede optar por persistir el estado de la queue a disco para poder restaurar los datos de ser necesario.

2.1.2.3) Exchange

La implementación de distintos patrones de mensajería se logra mediante la adición de nodos de *exchange* a la comunicación. El exchange es el encargado de distribuir los mensajes recibidos en las queues según una estrategia definida, entre ellas: *fanout*, *direct*, *topic* y *headers*.

El exchange *fanout* es el más simple; consiste en publicar un mensaje recibido a todas las colas conocidas. Las colas se enlazan a un exchange en el momento de su declaración, permitiendo la implementación del patrón publicador-suscriptor. En estos casos no es necesario que las colas tengan nombres concretos, ya que el publicador solamente necesita conocer el identificador del exchange al cual están asociadas.

Un exchange *direct* se comporta de manera similar al *fanout*, con la salvedad de que posibilita mayor granularidad en el ruteo de mensajes a través de claves. Una cola puede suscribirse a un exchange y solo recibir aquellos mensajes que sean enviados con una clave determinada. Es posible que más de una misma cola utilice la misma clave, y en estos casos el comportamiento es idéntico al ruteo *fanout*.

Direct exchange routing

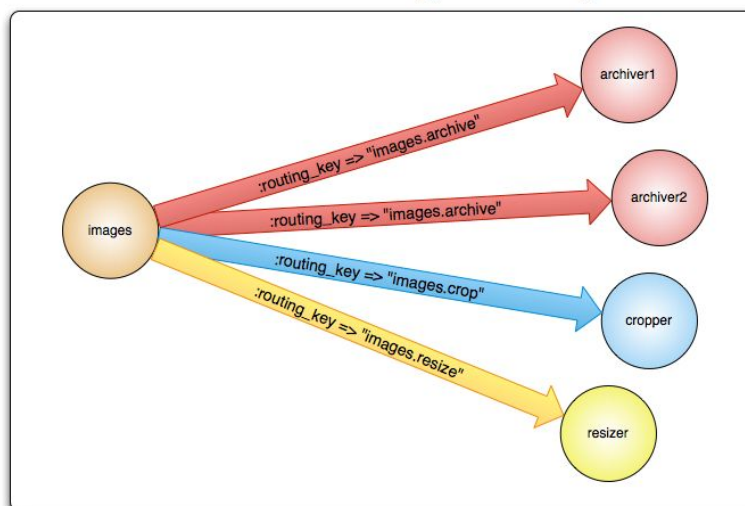


Figura 8. Exchange del tipo *direct* en RabbitMQ

Los exchange del tipo *topic* permiten implementar estrategias de ruteo de mayor complejidad. Son similares a los *direct* ya que también se basan en el uso de claves de ruteo, pero en este caso las claves son compuestas y pueden refinar aún más el subconjunto de mensajes que desea consumir la cola. Esto se logra mediante el uso de los caracteres que son sustituidos durante el ruteo: un asterisco (*) es sustituido por exactamente una palabra, y el numeral (#) puede ser sustituido por cero o más palabras. Cada palabra en la clave se separa por un punto (.).

El ruteo mediante *headers* consiste en dirigir los mensajes según valores en la cabecera del mensaje enviado y no por una clave aparte. Se lo puede considerar similar a *direct*, con la salvedad de que se pueden incluir varios headers en un mensaje, mientras que la clave es solo una, y se puede optar por condiciones OR/ANY, donde basta con que coincida un solo valor de los headers, o AND/ALL.

2.1.2.4) Brokers Distribuidos

AMQP es distribuido por naturaleza, los casos de uso más comunes involucran a varios clientes dispersos en varias máquinas. En algunas situaciones, sin embargo, es necesario que el intermediario mismo esté distribuido para aumentar el rendimiento y mejorar la disponibilidad. Existen tres formas de lograr esto: agrupamiento (*clustering*), federación y “la pala” (*the shovel*).

Mediante *clustering*, se conectan varios brokers físicos para formar uno solo lógico, y automáticamente se replican los exchanges, usuarios y permisos en todos los nodos. Las colas pueden permanecer en un solo nodo o ser replicadas en todo el cluster. Los mecanismos de federación y *the shovel* se utilizan para redirigir los mensajes publicados a un exchange o cola en un nodo a un exchange o cola en otro. La diferencia entre ambos es que en federación la comunicación entre nodos es directa, mientras que *the shovel* introduce un enlace que simplemente consume los mensajes de un nodo y los publica en el otro.

2.1.2.5) Replicación

La replicación de colas en un cluster es configurable de manera no obtrusiva mediante políticas que especifican que colas deben replicarse y en cuantos nodos. Cada cola replicada cuenta con un maestro que es el que recibe todas las operaciones que se realizan y luego réplica en los *mirrors*.

Si llegase a fallar el nodo maestro, se promueve a maestro el mirror con mayor antigüedad, el más probablemente sincronizado al previo maestro. Existe la posibilidad de que no exista mirror sincronizado al maestro y por lo tanto se produzca pérdida de mensajes. También es posible que ante un fallo se produzca una situación en la que un conjunto de mensajes hayan sido enviados pero no se haya recibido aún el ACK correspondiente, por lo que el nuevo maestro intentará re-enviarlos. En este caso donde las semánticas de entrega son de *at least once*, queda del lado del cliente detectar el mensaje duplicado.

2.1.2.6) Otras Consideraciones

RabbitMQ permite definir distintas variables de configuración que permiten mejorar la adaptación a un entorno particular, mediante el establecimiento de límites de memoria y uso de disco de acuerdo a los recursos disponibles, por ejemplo. Este mismo medio permite también configurar los canales de comunicación con los clientes, incluyendo los mecanismos de autenticación, autorización y encriptación mediante TLS o SSL.

La comunidad alrededor de RabbitMQ desarrolla activamente plugins que extienden las funcionalidades del sistema, por ejemplo, introduciendo interfaces web de administrador. Otras implementaciones permiten la utilización de WebSockets para permitir integración directa con clientes que soportan dicho protocolo, notablemente los navegadores modernos.

Otro aspecto a resaltar es la existencia de implementaciones de clientes en varios lenguajes de programación, incluyendo Python, Java, Ruby, PHP, C#, JavaScript, Go, Elixir, Objective-C y Swift.

2.2) Procesamiento (Streaming y Batch)

En cuanto a la capa de procesamiento de los datos, se evaluarán Hadoop, Apache Storm, Apache Samza, Apache Spark y Kafka Streams.

2.2.1) Hadoop

Hadoop nace como un framework y ecosistema de código abierto, utilizado para el almacenamiento y procesamiento distribuido de grandes volúmenes de sets de datos. La base del concepto de Hadoop se encuentra en asumir que las fallas de hardware pueden y suelen ocurrir, por lo que pretende generar un ambiente de rápida respuesta y reacción ante estos incidentes.

La problemática se ataca mediante la técnica conocida como “clustering”, la cual consiste en facilitar distintos nodos cuyo fin es balancear la carga ante la indisponibilidad de cualquiera de ellos.

Se puede decir que éste producto está constituido por dos partes:

1. *Almacenamiento*: provee un sistema de archivos llamado HDFS (Hadoop Distributed File System), dividiendo los archivos en bloques y distribuyendolos entre los diversos nodos del cluster. Los nodos del cluster pueden hablar entre ellos, manejando el balance de la carga y los niveles de replicación.

Esto puede utilizarse en conjunto con otras tecnologías de procesamiento, e incluso no es requerida para procesar información en Hadoop, ya que este permite utilizar otros sistemas de almacenamiento, como S3 de Amazon en la nube. A su vez, se conoce que HDFS sufre problemas de escalabilidad y “cuello de botella” ante volúmenes exagerados de metadata. El acceso a los archivos es posible sobre diversas tecnologías y lenguajes de uso común (C++, Java, Python, PHP, HTTP, etc.).
2. *Procesamiento*: provee un sistema llamado MapReduce, que se encarga de procesar los bloques de manera paralela, localmente en cada nodo del cluster, y aprovechando la ventaja de tener data “local” en cada uno. El “job tracker” distribuye el trabajo entre

los distintos “task tracker” de cada nodo, a conciencia de la localización de la información disponible.

Este procesamiento de tipo batch es incompatible con la arquitectura Kappa.

A pesar de no poseer un procesamiento aplicable a los principios y preceptos de Kappa, resulta ser un ecosistema muy completo y que ofrece una solución bastante amigable de implementar. Se trata de una solución estable y conocida, y su sistema de almacenamiento puede ser integrado con procesamiento de “streaming” (el utilizado en arquitecturas Kappa). Por esto, consideramos que era imperativo mencionarlo brevemente.

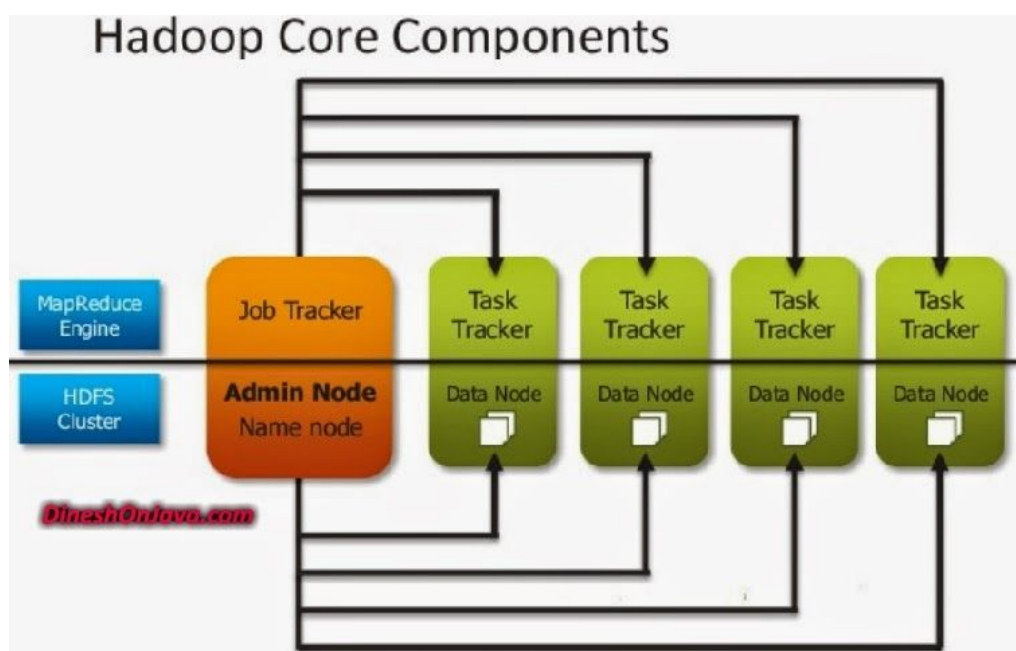


Figura 9. Componentes centrales de Hadoop

2.2.2) Apache Storm

Storm surge como un framework distribuido de procesamiento en “streaming” de código abierto, adaptable a casi cualquier lenguaje de programación, y utilizado mayormente para analíticas en tiempo real y “machine learning” en línea. Es mayormente utilizado por Yahoo y Twitter.

Esta herramienta se dispone en una estructura de grafo dirigido no cíclico, lo que permite una topología similar a la de MapReduce (en Hadoop), pero sumando la capacidad de procesar en tiempo real con baja latencia, de escalar incrementalmente el volumen de cómputo, y de entregar resultados al cliente de manera sincrónica y distribuida. Dispone de un módulo que funciona bajo el enfoque de “fallar rápido, auto reiniciar”, y se distribuye en “spouts” (streams de entradas en tuplas) que logran adaptarse a casi cualquier fuente de información, y “bolts” (módulos de proceso, transformación y salida de datos). Este tipo de procesamiento puede ser utilizado junto con el sistema de archivos propio de Hadoop, si esto fuera deseado.

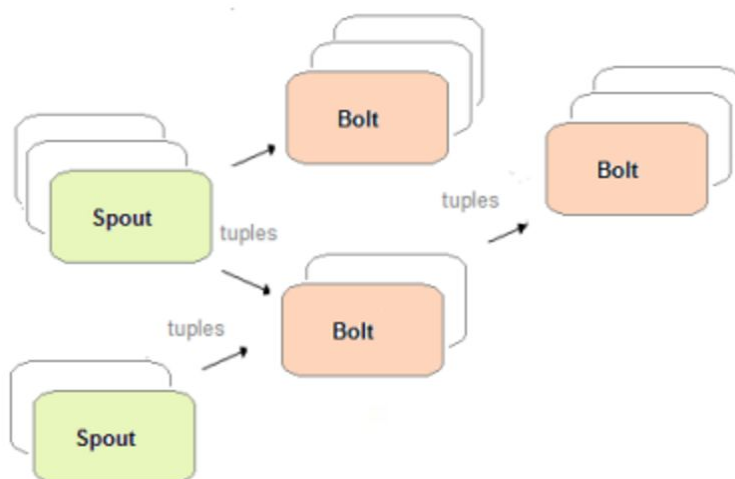


Figura 10. Grafo de “spouts” y “bolts” de Storm

No se encuentra tan especializado para trabajar con Kafka como Samza por haber sido desarrollado con anterioridad y no aplicar particionamiento similar al que suele ser utilizado en conjunto con Kafka, y suele requerir un almacenamiento externo que no viene incluido en la solución cuando se requiere mantener un gran volumen de estados para procesar tuplas. Por otro lado, se lo puede hacer funcionar bajo un modelo sql (SparkSQL).

2.2.3) Apache Samza

Al igual que Storm, Samza es un framework de procesamiento distribuido, frecuentemente utilizando en conjunto con Kafka para la mensajería, y Hadoop YARN (un negociador de recursos) para proveer tolerancia ante fallas, administración de recursos y seguridad. Se encuentra en el ambiente de producción de LinkedIn, llegando a procesar más de un millón de mensajes por hora en los momentos de mayor tráfico sostenido. Se podría decir que, en cierto sentido, Samza se centra mucho en la explotación de Kafka, pero permite mayor flexibilidad de ecosistema al usuario, ya que no posee topología definida como Storm.

Samza resuelve la problemática no mediante el procesamiento a tiempo real de tuplas (Storm), sino con el procesamiento de mensajes con la cadencia de uno a la vez, y en el orden en el que vienen. La corriente de entrada se divide en particiones compuestas por secuencias con identificadores unívocos (provee a sus tareas con sus propios pares llave-valor en disco), y procesándolos en “eventos” que asemejan el comportamiento basado en tiempo real de Storm.

El caso de uso común de Samza es aquel en el cual se poseen enormes cantidades de trabajo (por ejemplo, muchos gigabytes por partición), en el cual se irán localizando y procesando los datos en la misma máquina de una manera tal en la que se es eficiente cuando la información no cabe en memoria. También ofrece una API que el usuario puede decidir no utilizar, permitiendo que cada uno reemplace la ejecución, mensajería y almacenamiento por uno especializado para el caso de negocio particular.

2.2.4) Apache Spark

Spark busca analizar la información en una plataforma distribuida, principalmente orientado a acelerar las operaciones no queriendo trabajar en tiempo real, sino manejando la información en lo que se llama “micro-batch”. Incluye “machine-learning” (aprendizaje de máquina) iterativo y procesamiento gráfico. A diferencia de Hadoop, no opera en pasos, sino en procesar todo en una única barrida de n iteraciones más pequeñas, acercándose así mucho

más a simular procesamiento en tiempo real. Por otro lado, a diferencia de Storm, ejecuta cómputo paralelo a nivel de dato, y no a nivel de tarea.

Para asegurar tolerancia a fallos, se vale de unos objetos en los que almacena los datos, llamados RDD (“Resilient distributed datasets”), proveyendo recuperación total a los objetos de información almacenados en discos rígidos y/o memoria. Sin embargo, a diferencia de Hadoop, Spark no posee un módulo nativo y primario que se dedique exclusivamente al almacenamiento distribuido, por lo que debe ser integrado con tecnologías de almacenamiento, y su procesamiento “en memoria” puede resultar en un cuello de botella cuando se busca una fuerte relación de “costo-eficiencia”.

2.2.5) Kafka Streams

Describimos Kafka Streams como una librería construida sobre la popular plataforma de ingesta de datos, Kafka, siendo su código fuente disponible como parte de aquella, surgiendo de un emprendimiento fundado por ex-empleados de LinkedIn, la cual nos provee una API (interfaz de programación) que nos permite escribir aplicaciones de procesamiento de streams en tiempo real.

Este software de procesamiento busca en particular poder tratar con eficiencia el procesamiento mediante la transformación de tópicos de Kafka, de input a output, en especial en un esquema de microservicios no sincrónicos que entregan información de manera eficiente y compacta. Esto significa que la librería está diseñada para ser integrada con la lógica de negocio de una aplicación, en lugar de ser parte de un gran proceso batch de analíticas.

Existen diversas ventajas que nos promete Kafka Streams, encontrándose entre las más notorias la capacidad de re-procesamiento distribuido para calcular nuevos outputs cuando el código cambia, su propio DSL (lenguaje específico de dominio) y soporte para encriptación y autenticación, además de su naturaleza de código abierto.

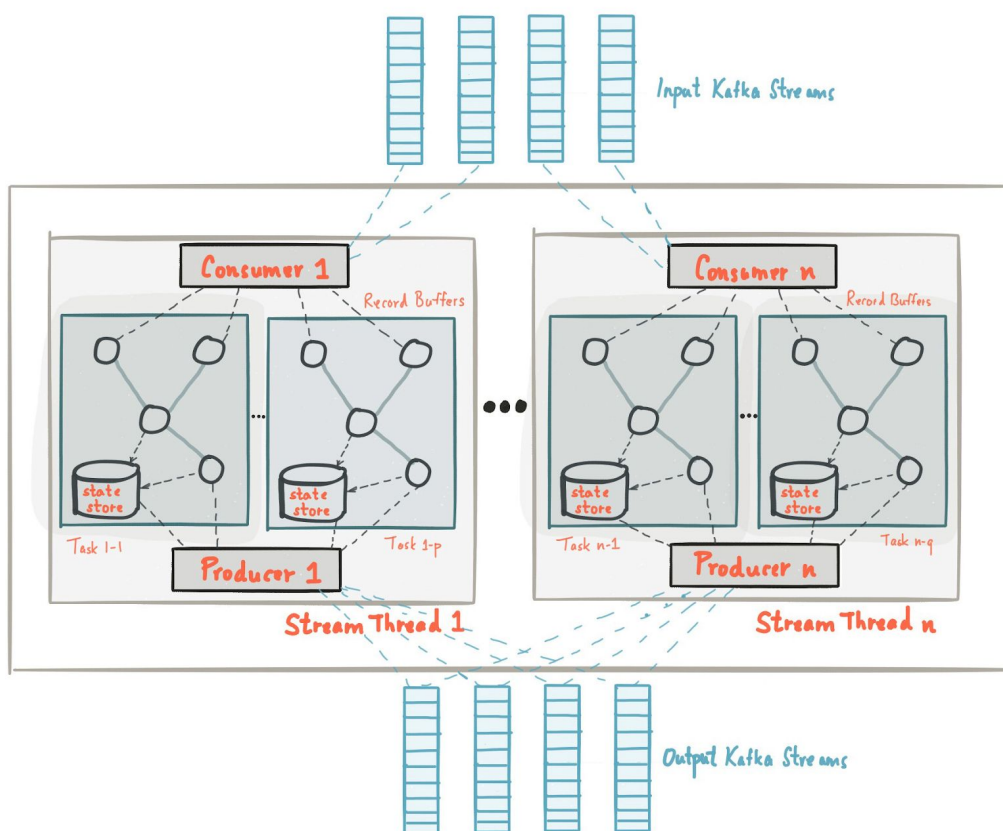


Figura 11. Topología de Kafka Streams

La compatibilidad nativa con Apache Kafka, dado por un modelo completamente integrado con las abstracciones provistas por la plataforma, y la simplicidad de las configuraciones que se deben realizar, lo convierte en una opción muy fuerte cuando se decide utilizar dicha plataforma como núcleo de una implementación.

2.2.6) Comparativa de alternativas

Consideramos que de los mencionados anteriormente, todos aquellos con capacidad de procesamiento de streams se encuentran altamente calificados para procesar de manera eficiente y continúa la información que reciben. No existen reglas “duras” sobre cuál de ellos es superior a los demás, pero si podemos determinar algunos lineamientos que nos servirán.

Tabla 3. Comparación de sistemas de procesamiento

	Storm	Spark	Samza	Kafka Streams
Modelo de streaming	Stream nativo	Micro-batching	Stream nativo	Stream nativo
Garantía de entrega	Al menos una vez	Una única vez	Al menos una vez	Al menos una vez
Tolerancia a fallos	Acuse de recibo (ACK)	RDD Checkpointing	Registro en logs	Reinicio de tareas fallidas con backup de data en Kafka
Manejo de estados	Stateless	Stateful	Stateful	Stateful
Latencia	Milisegundos (asincrónico)	Segundos (dependiendo del tamaño de muestra)	Milisegundos	Milisegundos
Lenguajes soportados	Virtualmente cualquiera	Scala, Java, Python	Scala, Java	Virtualmente cualquiera

2.2.6.1) Modelo de procesamiento

Echando una mirada a la tabla de arriba, encontramos dos enfoques en la manera en la que procesamos la información entrante: nativo y de micro-batching. Esta categoría se vuelve de primer plano, cuando consideramos que es la que definirá en un mayor porcentaje el costo y limitaciones en las operaciones que podremos realizar.

En el primer caso, el nativo, todos los registros obtenidos son procesados a medida que arriban al sistema, uno por uno. Esto reduce los costos de las operaciones y el esfuerzo de implementación, ya que no trata con abstracciones de la información, y a su vez también la latencia, por la misma razón. Los sistemas de procesamiento nativo de streams suelen poseer

un menor ‘throughput’ (ó ancho de banda) y significan un esfuerzo considerable si se quisiera resguardar tras balanceo de carga o tolerancia de fallos.

Por otro lado, el del micro-batching, dividir los registros en subconjuntos aumenta el esfuerzo de operaciones de tipo ‘join’ o ‘split’, ya que se debe operar sobre el batch completo. La tolerancia a fallos y el balanceo de carga son más sencillos de incluir: se soluciona enviando un micro-batch a cada nodo.

2.2.6.2) Garantía de entrega

Cuando hablamos de ‘garantía de entrega’, nos referimos a la capacidad del framework de asegurarnos si los mensajes llegarán en tiempo y forma, y bajo qué esfuerzo. Aquí encontramos dos categorías:

- *Al menos una vez*: existirán múltiples intentos de envío de cada mensaje que ingresa al sistema, de manera tal de que al menos uno tenga éxito. Se admite que los mensajes se dupliquen, pero no que se pierdan.
- *Una única vez*: sólo se realizará un intento de envío por cada mensaje que ingresa al sistema. No se admiten duplicados, ni perdidos.

Más allá de la simple comparación, estas dos estrategias nos llevan a varios aspectos que son más críticos de la fiabilidad, como la respuesta ante fallos, u otros relacionados con la performance, como la latencia, ‘throughput’ (rendimiento y ancho de banda) y la escalabilidad.

2.2.6.3) Tolerancia a fallos

Este aspecto resulta inherentemente más complejo en streaming que en sistemas de batch. Cuando tratamos con un modelo del tipo de este último, sencillamente podemos reiniciar la parte del cómputo que falló, solucionar los problemas de manera aislada, y eso es todo. Sin embargo, en escenarios de tipo streaming, corremos con la desventaja de que aún

optando por esta reacción, los registros seguirán entrando al sistema, mientras que podríamos estar corrompiendo la consistencia de información.

Los casos presentados utilizan tres mecanismos de tolerancia a fallos:

- Acuse de recibo (ACK), utilizado por Storm, consiste en cada transformador enviando un ‘acknowledgment’ al anterior por cada registro procesado, de manera que se utilizan pocos bytes para almacenamiento de los acuses de recibo por cada registro, y se mantiene una copia de respaldo hasta que se reciban todos los ‘acknowledgments’ del siguiente operador. Si no se reciben todos, se vuelven a enviar. Cuando se reciban todos, se elimina la copia de respaldo, la cual había generado un problema de duplicación de información y ocupación del espacio de memoria.
- Para micro-batching de Spark, cada nodo intenta procesar un subconjunto de la información. Si alguno de estos falla, sencillamente se recomputa, aprovechando que los micro-batch son persistentes e inmutables. Así va alcanzando “checkpoints”.
- En el caso de Samza, se utiliza un mecanismo de monitoreo de los offset de sus tareas, y los mueve cuando son completadas. Se pueden generar puntos de recuperación que pueden ser restaurados en caso de falla.
- Kafka Streams hace uso de las capacidades provistas nativamente por Kafka, el cual preserva una copia cada partición de datos procesada. En caso de fallo en alguna tarea en una instancia en particular, reinicia dicha tarea y la vuelve a ejecutar en alguna de las otras instancias disponibles de la aplicación. Además se mantienen réplicas locales de los logs de cambios de estado de los tópicos, los cuales se pueden compactar para facilitar la purga de tópicos que podrían crecer indefinidamente.

2.2.6.4) Manejo de estados

La mayoría de las operaciones en aplicaciones no triviales de streaming tienen algún tipo de estado. Por ello los datos no solamente tienen una entrada, un procesamiento y una salida, sino que incluyen el estado persistido de la data que procesan, van a procesar o ya procesaron, cumpliendo con la definición de procedimiento “stateful”. Como en todos los puntos anteriores, tenemos distintas estrategias:

- Dado que Storm provee garantía de entrega de “al menos una vez” con múltiples envíos de cada mensaje, este punto se vuelve una dificultad, dado que resulta muy complicado recrear un estado a partir de múltiples instancias del mismo mensaje. No posee la capacidad para realizar manejo de estados nativamente, por lo que se optaría por utilizar herramientas externas, como Trident, que define abstracciones como agregaciones, agrupamiento y, entre otras más cosas, lo necesario para realizar procesamiento incremental con manejo de estados, con consistencia de garantía de entrega de “única vez”.
- Spark, siendo un sistema de procesamiento en micro-batches, trata la problemática utilizando otra stream de micro-batches para poder determinar en todo momento el estado de un micro-batch coincidente entre la información a procesar. Durante el procesamiento de cada micro-batch toma el estado actual y una función representando la operación a realizar, y entrega el estado actualizado junto con el micro-batch ya procesado.

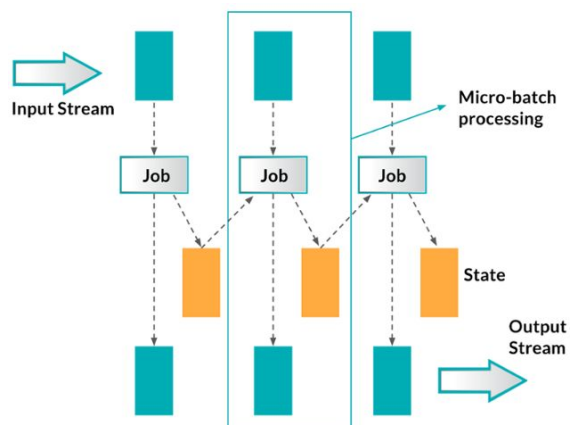


Figura 12. Manejo de estados en Spark

- La solución del lado de Samza es directamente empujar la data a Kafka, funcionando así en el contexto de manejo de estados. Posee operadores con estado y el mismo es transmitido a Kafka, de manera tal de que si fuera necesario recrear un estado, esto se pudiera hacer de manera sencilla desde tópicos de Kafka. Además, mantiene su copia local clave-valor para evitar incesantes llamadas a Kafka y agilizar los tiempos.

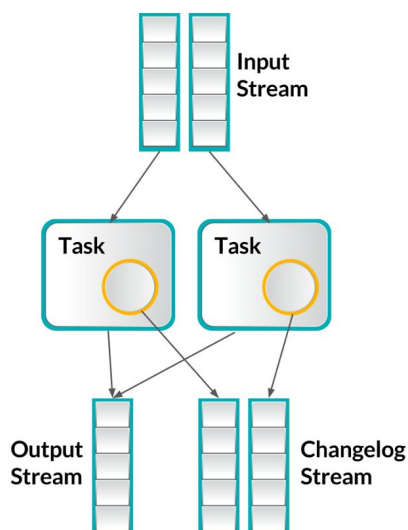


Figura 13. Manejo de estados en Samza

2.2.6.5) Rendimiento

El rendimiento de estas alternativas se mide con distintos conceptos, tales como latencia y capacidad de procesamiento. Depende de múltiples variables, y en el caso de

micro-batching, la latencia se mide en segundos, mientras que para streaming nativo se la mide en el espectro de los mili-segundos. Esto significa que Spark pertenece al primer grupo, mientras que Storm, Samza y Kafka Streams al segundo.

Resulta importante también tener en cuenta el costo de los otros aspectos para cada caso, como garantía de entrega, tolerancia de fallos y manejo de estados.

2.6) Conclusión

Todas las plataformas que fueron analizadas tienen un rol específico y no existe una solución única y estandarizada a la cuestión de procesamiento de Big Data. Es necesario analizar el contexto de aplicación concreto y a partir de ello conformar el software stack que dará soporte al sistema. Entran en juego entonces variables que ya han sido evaluadas durante la investigación previa: rendimiento, facilidad de uso, capacidad de integración, documentación disponible.

La arquitectura base de la aplicación ya fue definida, por lo que algunas alternativas quedan relegadas a un estado de menor prioridad. Hadoop cae en esta categoría, ya que sus aplicaciones para procesamiento en lotes entra en conflicto con los principios de Kappa, que está orientada a streams para el procesamiento de datos.

La primera fase del estudio de software se refirió a las plataformas de retención de logs. De las dos herramientas evaluadas, se optará por Apache Kafka. La principal motivación en este caso es la pre-existencia de interfaces para integrarse con las demás capas del sistema. Los frameworks para procesamiento de datos analizados tienen en común la capacidad de integrarse casi transparentemente con Kafka, ya sea mediante librerías de código libre de terceros o como parte del diseño mismo del sistema (como es el caso de Samza o Kafka Streams). Se concluye en cambio que RabbitMQ no posee las mismas facilidades, ya que si bien es extensible y se pueden implementar las interfaces necesarias, esto involucra un costo de desarrollo adicional, cuyo beneficio no resulta suficiente como para justificarlo en este proyecto.

Por razones similares, la elección para la capa de streaming será Kafka Streams. Se considera que las demás alternativas satisfacen en menor medida y bajo un mayor costo condiciones como la facilidad de uso e integración, la capacidad para procesar datos con baja latencia, o la compatibilidad con la arquitectura subyacente del sistema, como ya se mencionó es el caso de Hadoop. A su vez, se considera positivo la eliminación de dependencias

Otra de las posibilidades era Spark, cuyo diseño no está orientado estrictamente al procesamiento en tiempo real, sino que introduce una latencia, que si bien es considerablemente menor a la de sistemas de procesamiento batch, es un factor que se tuvo en cuenta para la decisión. Spark exige además la persistencia en un sistema de archivos distribuido externo, lo que deriva en un sistema con mayor complejidad. Por otro lado, a diferencia de Kafka Streams, requiere configurar y administrar clusters cuya única función será el procesamiento de streams, agregando complejidad y una dependencia externa de frameworks que no necesitamos en nuestro caso de uso. Si bien no formará parte del núcleo del sistema, no se descarta su eventual utilidad en otras facetas, particularmente la de *machine-learning* para la identificación de patrones de comportamiento en los datos.

Las otras dos soluciones, Samza y Storm, poseen similitudes entre sí e incluso con Kafka Streams. Todas están orientadas específicamente al procesamiento vía streaming de baja latencia y ofrecen garantías adaptables en el envío de mensajes, permitiendo equilibrar performance y consistencia de datos según sea necesario. Sin embargo, existen ciertos factores que hacen que Samza y Storm sean consideradas menos convenientes que Kafka Streams.

Storm no incluye por defecto persistencia de estados durante el procesamiento, una propiedad fundamental para asegurar la tolerancia a fallos. La solución de Samza a este problema es persistir sus estados intermedios directamente dentro de Kafka, algo natural teniendo en cuenta que fue diseñado para ser compatible *out of the box* con la plataforma. Sin embargo, los jobs de Samza no son ejecutados en el mismo contexto que Kafka, teniendo en cuenta que no comparten el mismo gestor de recursos (YARN para Samza y Zookeeper para Kafka). Esto no es un impedimento demasiado crítico, pero de todos modos significa introducir otro componente en el sistema.

Las debilidades que se enumeraron de las distintas herramientas no son limitaciones absolutas, si fuera este el caso no serían reconocidas globalmente como lo son. Pero finalmente la decisión se reduce a cual se ajusta de la mejor manera al sistema en cuestión, y no hay razón para no utilizar aquella que ofrezca las características deseadas con la menor complejidad posible. Kafka Streams es prácticamente una extensión de Kafka por lo que la integración es trivial, y si bien es menos abarcativa que herramientas como Spark en algunos aspectos, ofrece todas las funcionalidades necesarias para el caso de uso que precisamos cubrir.

3) Internet of Things

El concepto de Internet of Things es utilizado cotidianamente para referirse al concepto de elementos conectados para lograr algún objetivo particular. Uno de las aplicaciones más conocidas está relacionada a las casas inteligentes donde, por ejemplo, las ventanas se cierran cuando el aire acondicionado está encendido. También se ha visto utilidad en distintas industrias, como la química donde se puede aplicar para monitorear la emisión accidental de componentes nocivos al aire.

La agricultura también ha encontrado utilidad en este instrumento, brindando facilidades para monitorear la humedad de la tierra y optimizar el microclima en invernaderos. Para el proyecto en curso, se buscará una configuración que permita aplicar esta estrategia de monitoreo a silos de granos, cuyas variables ambientales son un factor importante en la calidad final del producto.

3.1) Arquitectura (Iot-A)

El objetivo de esta sección es llevar el concepto abstracto de IoT a un modelo concreto mediante investigación de los modelos de referencia de mayor aceptación en la comunidad. Al igual que para la mayoría de las áreas de estudio, no existe una fuente única y estándar

para el diseño de soluciones IoT; sin embargo, se podrá notar que hay un conjunto de ideas comunes que describen en líneas generales las distintas capas de una arquitectura de este tipo.

El concepto de IoT en sí fue concebido en 1999 por el investigador británico Kevin Ashton (MIT) para referirse a una red conectada al mundo físico mediante sensores ubicuos. Los avances tecnológicos desde entonces han fomentado la adopción del concepto en la industria, dando lugar a soluciones personalizadas que responden a problemáticas específicas donde la interoperabilidad es de baja prioridad.

Esta situación llevó a distintos participantes a proponer modelos que permitan cierto grado de estandarización en las implementaciones. Uno de los modelos resultantes fue IoT-A, cuya introducción se refiere precisamente a esta realidad como “insostenible” en el largo plazo. El documento de IoT-A será la principal referencia en este estudio ya que ofrece un grado de detalle superior a las otras fuentes públicas que se han encontrado. Además, IoT-A no es llevado adelante por una única organización sino por una sociedad de varias, disminuyendo el riesgo de que el contenido esté influenciado por los intereses particulares de la entidad autora.

3.1.1) Modelo de referencia

El modelo de referencia en IoT-A se divide en 3 submodelos principales, particularmente el de Dominio, de Información y el Funcional. Se definen además dos modelos para detallar la implementación de funcionalidades clave de Comunicación y Seguridad.

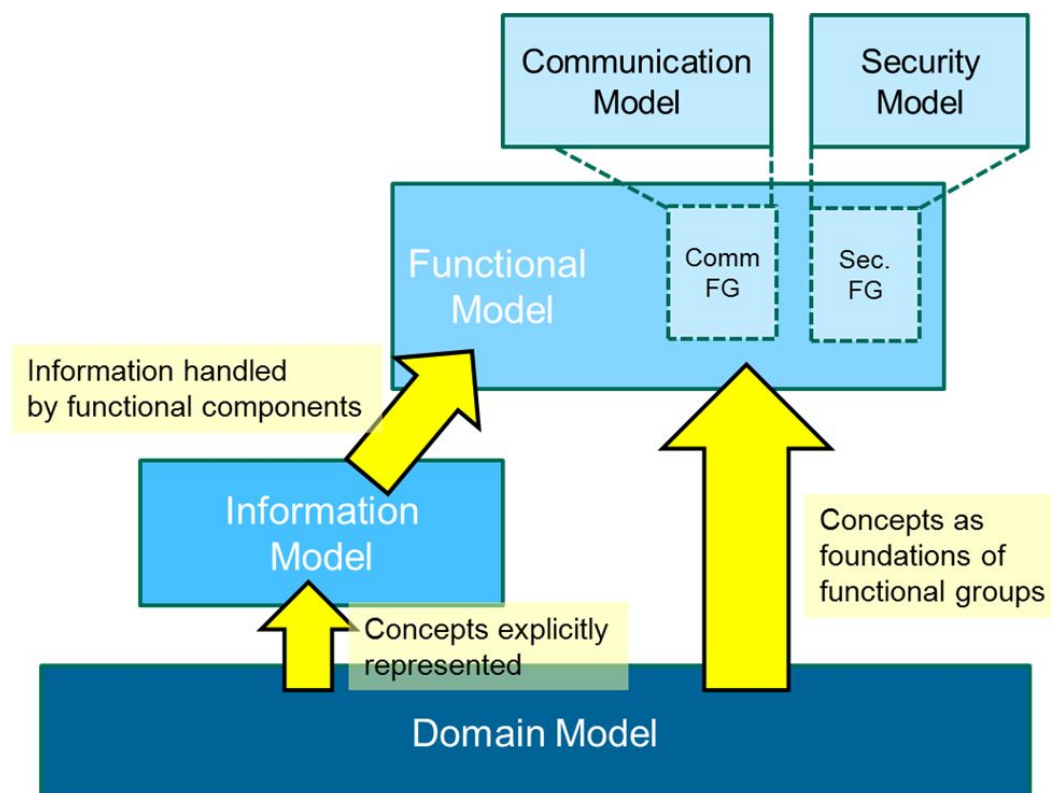


Figura 14. Modelo de referencia (IoT)

El modelo de Dominio introduce los conceptos principales de la IoT y las relaciones entre los mismos. El nivel de abstracción que se utiliza permite separar las definiciones de tecnologías particulares, y definir los conceptos de una manera lo suficientemente genérica como para mantenerse relevante en el paso del tiempo. El modelo de Información modela los conceptos que se presentan en el de Dominio, especificando atributos y relaciones de toda la información que se maneja en un sistema de IoT.

El modelo Funcional expone las funcionalidades necesarias para interactuar con las instancias de los conceptos definidos en los modelos anteriores. Una de las funcionalidades clave en los sistemas distribuidos es la comunicación entre los distintos componentes, y un sistema de IoT generalmente debe integrar diferentes protocolos de comunicación entre componentes (ya sean propios o ajenos al mismo), por lo que se define un modelo de Comunicación. Finalmente, se dedica un grupo de funcionalidades a los mecanismos de Seguridad, de fundamental importancia al igual que en todos los sistemas interconectados.

A continuación se describirán en mayor detalle los objetivos y conceptos de cada uno de estos modelos, con la finalidad de, una vez concluido el análisis, ser capaces de definir una arquitectura que respete la separación de responsabilidades recomendada.

3.1.2) Modelo de Dominio

El modelo de Dominio define los conceptos abstractos, sus responsabilidades y relaciones. El nivel de detalle empleado debe enfocarse en aquello que no es propenso a variar en el tiempo; algunos elementos como los *dispositivos* probablemente no cambien, pero el tipo de dispositivo dependerá de la tecnología disponible en el momento. Este es el caso con instrumentos como los lectores de códigos de barra, QR o reconocimiento por imágenes.

En un escenario típico de IoT, un Usuario busca interactuar con una Entidad Física del mundo real. Al hablar de Usuario, uno se puede referir a un Usuario Humano o a algún Artefacto Digital (por ejemplo, una aplicación u otro agente de software). El contacto con la Entidad Física en IoT no se realiza directamente sino mediante un servicio que intermedia la comunicación.

Las Entidades Físicas pueden ser casi cualquier objeto o entorno, desde humanos y animales a autos y edificios. Estas poseen a su vez una contraparte en el mundo virtual, a las que se denominan Entidades Virtuales. Estas pueden asumir distintas formas según la naturaleza del objeto físico; un edificio por ejemplo podría verse digitalizado en un modelo tridimensional, así como un humano puede estar asociado a una cuenta en una red social.

Las Entidades Virtuales poseen dos propiedades fundamentales. En primer lugar, se las consideran Artefactos Digitales, en el sentido que están asociadas a una única Entidad Física, aunque estas no necesariamente están asociadas a una única entidad virtual (se lo puede ver como una relación 1 a 1...n en UML). Cada Entidad Virtual es única e identificable unívocamente. Los Artefactos Digitales pueden ser *activos*, como es el caso de agentes de software que acceden servicios, o *pasivos*, como lo es una representación digital dentro de una base de datos. La segunda propiedad importante es que una Entidad Virtual es una

representación sincronizada de su contraparte física, e idealmente un cambio en alguna de las dos partes sería reflejada automáticamente en la otra.

Para explicitar la unión entre las Entidades Físicas y Virtuales se utiliza el término Entidad Aumentada. Cuando se conforma esta unidad un objeto está habilitado para integrarse en la red digital, por lo que es esto a lo que se puede considerar la “cosa” (*thing*) en la IoT. En la práctica, esta relación se logra asociando un dispositivo que provea la interfaz tecnológica para interactuar con u obtener información acerca de la Entidad Física. Se dice entonces que el Dispositivo actúa de intermediario entre las Entidad Físicas que no tienen proyección en el mundo digital y las Virtuales, que no tienen proyección en el mundo físico.

Los Dispositivos proveen capacidades monitoreo, sensoreo, actuación, computación, almacenamiento y/o procesamiento. De acuerdo a estas capacidades, se pueden agrupar en:

- Sensores: proveen información acerca de la Entidad Física que monitorean. Esta información suele referirse al estado físico de la entidad (variables como temperatura, humedad).
- Etiquetas: Identifican la Entidad Física a la cual están asociadas. El proceso de lectura en sí es realizado por Sensores.
- Actuadores: Modifican el estado físico de la entidad, por ejemplo mediante movimiento o encendido/apagado de funcionalidades.

Las Entidades Virtuales vinculadas a Entidades Físicas de gran tamaño (como puede ser un silo agropecuario) pueden depender de una gran cantidad de Dispositivos. A su vez, un Dispositivo puede en sí estar formado por uno o más Dispositivos que interactúan para lograr una funcionalidad particular.

Otro concepto que forma parte del modelo de Dominio es el de Recurso. Un Recurso es un componente de software de bajo nivel que permite la interacción con las Entidades Físicas mediante interfaces nativas. Generalmente, este se encuentra embebido en el Dispositivo vinculado a la Entidad Física, en forma de código ejecutable para acceder a la información del objeto o actuar sobre el mismo. Puede llegar a encontrarse, sin embargo, en algún sistema de back-end lejos del Dispositivo en sí, como una base de datos en la nube. Los

Recursos se relacionan con una Entidad Virtual para posibilitar la interacción con la Entidad Física que representa.

Como la implementación de un Recurso depende en gran medida del hardware del Dispositivo, es necesaria la existencia de un Servicio con una interfaz estandarizada y correctamente definida, accesible a través de la red. La capa de Servicios permite la invocación de funcionalidades de bajo nivel, que operan directamente sobre los Recursos, y de alto nivel, como puede ser la ejecución de un determinado proceso de negocio. El nivel de abstracción del servicio permite categorizarlo a nivel de Recurso o de Entidad Virtual. En el primer caso, el Servicio expone la funcionalidad del Recurso, administrando además aspectos no funcionales como la seguridad, disponibilidad y rendimiento. Los Servicios a nivel de Entidad Virtual directamente proveen acceso y capacidades de modificación de los atributos de la Entidad Virtual. Finalmente, existe una categoría de Servicios Integrados, que son aquellos compuestos por una combinación de ambos tipos.

La documentación de IoT-A utiliza una sintaxis basada en UML para ilustrar las relaciones entre los conceptos definidos, con ciertas modificaciones como el significado del símbolo “hereda de”, que debe leerse como “es un” en este contexto. En el Anexo D puede encontrarse el diagrama correspondiente a los conceptos del modelo de Dominio.

3.1.3) Modelo de Información

El modelo de Información define la estructura de los datos que maneja el sistema a nivel conceptual. El diagrama a continuación muestra la estructura que se utiliza para describir los datos procesados en un sistema IoT.

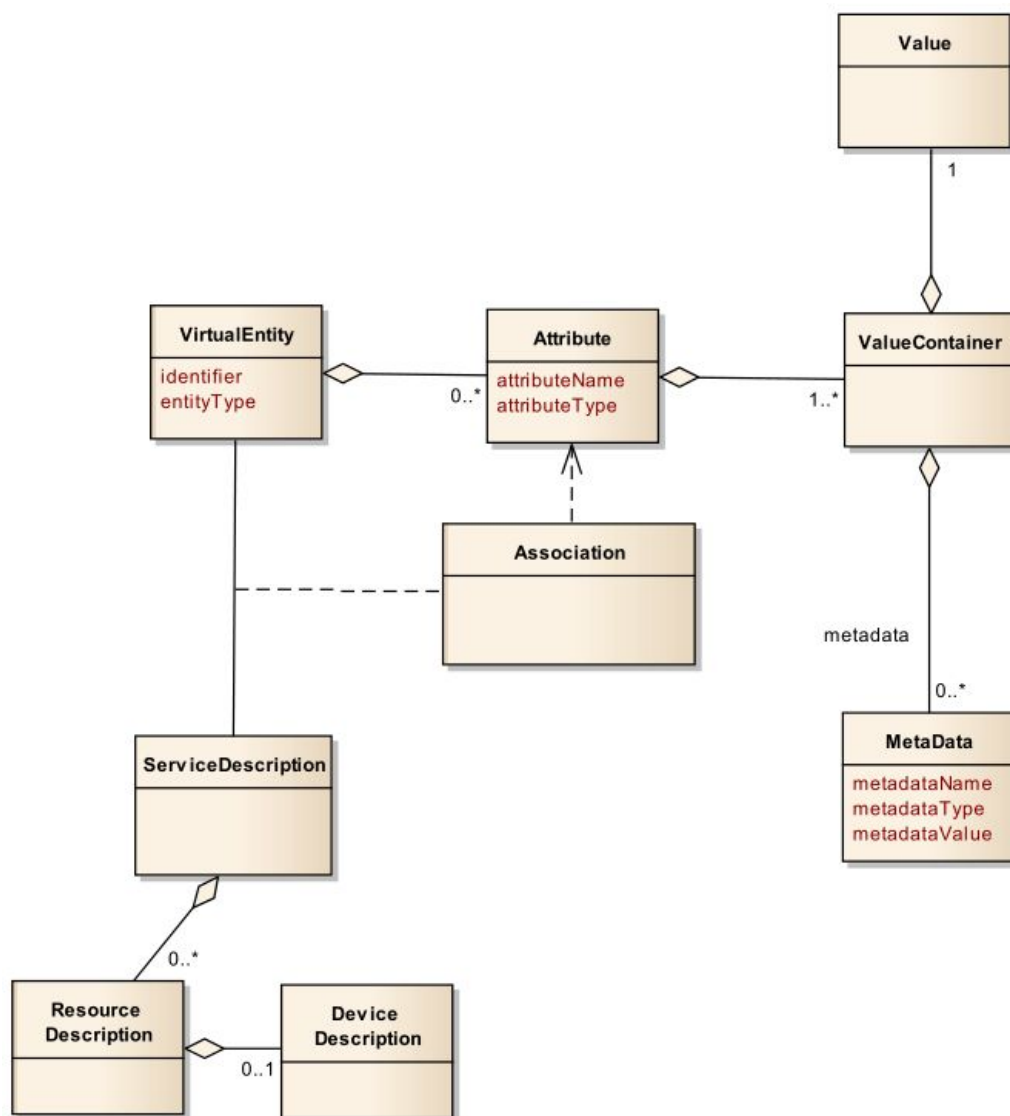


Figura 14. Modelo de Información (IoTA)

Los elementos principales en este modelo son la Entidad Virtual y la Descripción del Servicio. Como se mencionó en el análisis del modelo de Dominio, la Entidad Virtual se refiere a una Entidad Física determinada, mientras que el Servicio actúa como unión entre el mundo digital y el físico.

Las Entidad Virtuales poseen un identificador único y un tipo determinado, de acuerdo a la clasificación que se utilice. La descripción se realiza mediante la asignación de Atributos, cada uno con uno o múltiples Contenedores de Valor, que además de poseer el valor en sí pueden ser enriquecidos con metadatos. La flexibilidad del modelo para la definición de

atributos y valores permite representar relaciones complejas donde la multiplicidad no necesariamente es de uno a uno.

El otro aspecto fundamental del modelo es la Descripción del Servicio, que describe los aspectos relevantes del Servicio que permite comunicación con la Entidad Virtual, incluyendo la interfaz que este ofrece. Esta descripción puede contener referencias a los Recursos y Dispositivos que encapsula. Las Asociaciones permiten vincular un Servicio a una Entidad Virtual en forma de Atributo de esta última.

3.1.3.1) Datos en un sistema IoT

Un sistema de IoT frecuentemente tiene que operar con datos de distinta naturaleza. Se pueden distinguir distintos tipos de dato, particularmente:

- Datos en tiempo real: son aquellos que reflejan el estado actual del sistema. Los sensores generalmente producen datos de este tipo.
- Datos derivados: son el resultado de ejecutar procesos de distinto tipo sobre otros datos, como agregaciones. Dentro de esta categoría se encuentran los datos calculados por un proceso de batch o streaming típicos de Big Data.
- Datos inferidos: a partir de ciertas aplicaciones de hechos e inferencias se puede generar datos que representan algún conocimiento. Este tipo de datos puede ser generado, por ejemplo, por un sistema de Inteligencia Artificial.
- Datos reconciliados: aquellos datos que no se manipulan en su forma “cruda”, sino que pasan por algún proceso de purificación o enriquecimiento para aumentar su calidad. Esta técnica es común en el área de Análisis de Datos.

3.1.4) Modelo Funcional

El modelo Funcional define los grupos funcionales que conforman el modelo de referencia de IoT-A, y las relaciones que existen entre ellos. Dividir las responsabilidades en

distintos grupos funcionales sirve para facilitar el entendimiento de cada área del sistema, reducir la complejidad y volverlo más fácil de gestionar.

La definición de los grupos funcionales produce una serie de capas ordenadas según su grado de abstracción. Algunas denominaciones de estos grupos están directamente asociados a sus contrapartes en el modelo de Dominio, particularmente los grupos de Aplicación, Entidad Virtual, Servicio y Dispositivos. Además de estos, se definen otras áreas que deben considerarse en un sistema de IoT: Comunicación, Organización de Servicios y Gestión de Procesos de Negocio IoT. Finalmente, se presentan dos grupos más que operan en forma transversal en el modelo, ofreciendo funcionalidades a las demás capas longitudinales. Tales son los grupos funcionales de Administración y Seguridad.

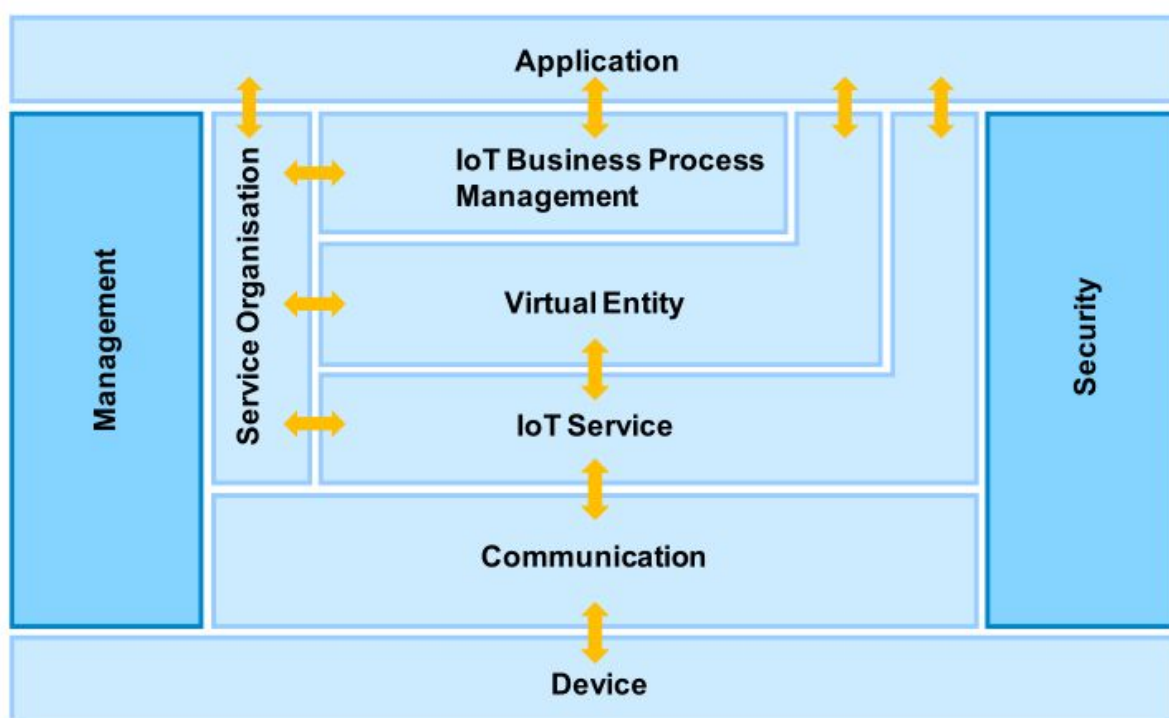


Figura 15. Modelo Funcional (IoT-A)

Como se observa en el diagrama, no solo es importante definir el rol de cada grupo de funcionalidad, sino también las relaciones entre cada uno de ellos. Cabe aclarar que dos de las capas del modelo, particularmente las de Aplicación y Dispositivo no son descritas con mayor detalle en la documentación de IoT-A ya que se consideran demasiado genéricas como para aportar valor.

El primer grupo funcional que se describirá es el de Gestión de Procesos de Negocio IoT (IoT BPM por sus siglas en inglés). El objetivo de este grupo es definir los conceptos e interfaces que se necesitan para introducir sistemas de IoT en los negocios tradicionales en forma estándar y respetando buenas prácticas, evitando el encerramiento en soluciones propietarias con capacidad limitada de integración. Se pretende que las aplicaciones que interactúan con la capa de IoT BPM no necesiten conocer los detalles específicos de las capas inferiores, logrando menores costos de integración. Para la implementación concreta de los procesos, este grupo invoca las funcionalidades de menor nivel expuestas de las capas de Organización de Servicios y Entidad Virtual.

El grupo de funcionalidad de Organización de Servicios actúa de central de comunicación para otros grupos. Teniendo en cuenta que la unidad principal de comunicación en el modelo es el Servicio, esta capa se encarga de organizar los servicios de diferentes niveles de abstracción, permitiendo por ejemplo que aplicaciones externas se comuniquen con grupos de bajo nivel como la de Entidad Virtual o incluso la de Servicios IoT. Uno de los patrones utilizados es la composición de servicios, donde se combinan varios servicios básicos (como aquellos que se encuentran en los dispositivos mismos) para responder consultas con un mayor nivel de abstracción.

El acceso directo a los Recursos es mediado por los Servicios IoT. Aquí se exponen los valores de cada dispositivo individualmente, por lo que se podría, por ejemplo, solicitar las mediciones de un sensor determinado. El significado de este valor no forma parte del intercambio en sí, sino que debe ser conocido de antemano por el solicitador. Un nivel por encima se encuentra el grupo de Entidad Virtual, que pone a disposición los datos asociados a las Entidades Virtuales. Esto se logra abstrayendo el acceso a los dispositivos individuales, responsabilidad del grupo de Servicios IoT, por lo que debe modelarse la comunicación entre estos dos grupos.

El direccionamiento y ruteo de los dispositivos en la red es coordinado por el grupo de Comunicación. Esto no solo involucra el traslado de datos sino también cuestiones de control como la calidad de servicio y el manejo de errores, así como la optimización del uso de energía mediante técnicas de bacheo de mensajes.

El grupo transversal de Administración es responsable de gestionar aquellas tareas que involucren dos o más grupo funcionales, como el apagado del sistema IoT completo durante tareas de mantenimiento. Esto no incluye los controles de seguridad del sistema, que por ser un área crítica es asignada al otro grupo transversal, el de Seguridad. Las responsabilidades de este grupo están conformadas por el aseguramiento de la seguridad y la privacidad del sistema, la registración de clientes, la anonimidad y desvinculabilidad de los datos, y el establecimiento de comunicaciones seguras entre pares mediante autorización (garantizando integridad y confidencialidad de los datos).

3.1.5) Modelo de Comunicación

La definición de los principales paradigmas de comunicación en un sistema IoT, tanto interna como externa, es definida por el modelo de Comunicación. Este modelo se basa en una representación en capas similar al modelo OSI para redes, donde las capas inferiores operan más cerca del hardware, y las superiores a nivel de aplicación.

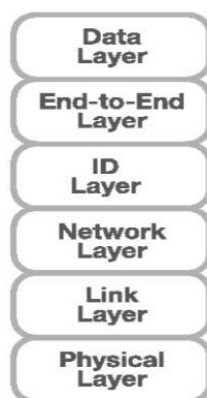


Figura 16. Modelo de Comunicación (IoTA)

Algunas capas comparten los roles de sus contrapartes en el modelo OSI. Tal es el caso de la capa física, cuyo comportamiento es altamente dependiente de las implementaciones específicas de cada solución particular. La integración en un marco común

ocurre gradualmente en las capas superiores. En la de Enlace, por su parte, se apunta a lograr interoperabilidad dentro de esta heterogeneidad de tecnología definiendo ciertas interfaces comunes para el intercambio de paquetes. Similar es el caso de la capa de Red, que al igual que en el modelo OSI busca establecer formas de comunicación para integrar soluciones de diverso origen.

A partir de la capa de ID el modelo comienza a especificarse en el contexto IoT, y por lo tanto se comienza a observar diferencias respecto al modelo OSI. En esta capa se establece el concepto de identificador de Entidad Virtual (VE-ID), que provee un marco de resolución de direcciones para IoT. La capa que le sigue, la de Punto a Punto, es responsable de varias tareas relacionadas a la configuración de redes: ruteo mediante proxies y puertas de enlace, traducción de peticiones, comunicación entre redes. La capa de Datos que se ubica en la cima del modelo corresponde al punto de entrada a los datos, cuya descripción es realizada mediante el modelo de Información.

Además de distribuir responsabilidades mediante la separación en capas, el modelo de Comunicación hace hincapié en una dificultad inherente a las redes de IoT, particularmente las limitaciones técnicas de los dispositivos. Para entender el problema, en primer lugar se realiza una distinción entre las redes restringidas (*constrained*) y las no restringidas. La Internet tradicional es un ejemplo de una red no restringida, ya que está conformada por enlaces de alta velocidad (mayor a 1 Mb/s) y poca latencia. Una red restringida, en cambio, es caracterizada por tener baja velocidad de transferencia (menor a 1 Mb/s) y latencias más elevadas. Estas redes son limitadas por las tasas de transferencia de la tecnología en la capa física de los dispositivos, así como por la necesidad de minimizar el consumo de energía.

Combinando estos tipos de redes, se pueden plantear diferentes escenarios para un sistema IoT. En el caso más sencillo existe solo una red restringida de comunicación exclusivamente interna, pero a medida que se introducen más partes surgen problemáticas que requieren alguna estrategia de resolución. En una estructura IoT típica, la Internet será un intermediario en alguna fase de la comunicación. Este sería el caso, por ejemplo, de un sistema de monitoreo donde los datos son provistos por dispositivos dentro de una red restringida, enviados a través de Internet a un servidor remoto (previamente ruteados por

alguna puerta de enlace, como un router), para luego derivar en alguna señal que provoque una acción en los actuadores de los mismos u otra red de dispositivos.



Figura 17. Configuración de dos redes restringidas mediadas por Internet

La complejidad en esta situación radica en que intervienen distintos protocolos, por lo que en algún punto de la comunicación, generalmente en las puertas de enlace, debe realizarse una traducción a un lenguaje entendible por las diferentes partes. Idealmente, para las aplicaciones que interactúan con las entidades mediante interfaces de alto nivel estos detalles serán transparentes.

3.1.6) Modelo de Seguridad

El proyecto IoT-A reafirma la importancia de los conceptos de confianza, seguridad y privacidad en el sistema. Esta noción aplica a todo sistema distribuido y no solo al contexto IoT, por lo que no se considera necesario profundizar en mayor detalle sobre las implicaciones de cada faceta de este modelo. A grandes rasgos, una implementación de IoT debe respetar los siguientes principios relacionados a la seguridad:

- **Confianza:** La identidad de la comunicación entre dos partes del modelo de Dominio debe ser verificable, y el intercambio de datos, íntegro.
- **Autenticación y Autorización:** Los niveles de acceso al sistema deben controlarse mediante un sistema de roles para prevenir accesos indeseados.
- **Anonimidad y desvinculabilidad:** Un atacante que intercepte un conjunto de datos no debe poder trazar su origen a un usuario particular.

Si bien estos conceptos no son específicos al IoT, debe tenerse en cuenta que la naturaleza heterogénea y las limitaciones técnicas de los dispositivos pueden dificultar su implementación. Además, debe asegurarse que los protocolos de seguridad permitan interoperabilidad con sistemas externos; caso contrario se iría en contra de uno de los pilares de IoT-A.

3.2) Conclusión

Una de las cuestiones que busca resolver IoT-A es la tendencia de las soluciones IoT a encerrarse en su propio ecosistema, resultando en *Intranets* aisladas en lugar de una Internet de Cosas relativamente unificada. Hubiera sido muy fácil caer en esta falencia, ya que el conocimiento sobre el área es aún inmaduro y cada solución propietaria impone sus propios fundamentos.

Durante la concepción de la idea original de un sistema de monitoreo de silos, simplemente se visualizó el aspecto de IoT en forma de un conjunto de sensores cuya única responsabilidad sería recolectar datos ambientales y enviarlos al nodo central de procesamiento. Si bien esto en teoría podría ser operable, la estructura resultante sería altamente incompatible con otros servicios o sistemas externos que puedan introducirse en el largo plazo. Se entiende entonces la importancia de basar la arquitectura en un modelo de referencia con principios sólidos y fundamentados en la experiencia de proyectos anteriores.

Cada modelo de IoT-A abarca una perspectiva del sistema. El de Dominio le asigna un nombre y responsabilidades a todos los elementos que forman parte del sistema. Uno de los beneficios de realizar estas distinciones es que se le quita ambigüedad a lo que representa IoT, sin ir más lejos permite entender de manera concreta que constituye una *cosa* en este contexto (que como se comentó previamente, se refiere una Entidad Aumentada). De esta manera se comienza a descomponer en partes fácilmente entendibles aquello que conforma un sistema IoT.

Los modelos que siguen enuncian mejores prácticas para el diseño de la arquitectura, profundizando cada área específica por separado. En el modelo de Información se explica cómo describir los atributos y relaciones de cada elemento del sistema, y en el Funcional se definen ciertas capas de abstracción para administrar la complejidad. Los modelos de Comunicación y Seguridad por su parte atacan ciertas cuestiones de bajo nivel propias de cualquier sistema distribuido.

Ahora bien, para entender la utilidad de este estudio para el proyecto en curso, se debe tener en cuenta que el aspecto de IoT del prototipo es de poca complejidad, y por lo tanto sería excesivo intentar cubrir todos los aspectos del modelo de referencia propuesto por IoT-A. Se debe entender el análisis anterior como uno de fines principalmente teóricos, que permita tener una base sólida sobre la cual construir un sistema con capacidad de escalar y extenderse en el futuro. Para los fines concretos del presente trabajo, no se considera de valor implementar, por ejemplo, los mecanismos de seguridad recomendados por IoT-A. Sí se definirá el modelo de Dominio, ya que se considera el más importante para mejorar el entendimiento del sistema. Asimismo, el desarrollo del prototipo contemplará en lo posible la separación en grupos funcionales según el nivel de abstracción respecto al dispositivo en sí.

En lo que a implementación física se refiere, se trabajará con un único dispositivo real. Este será una placa electrónica Arduino con un sensor de temperatura que permita emular en cierta medida una situación real, mientras que las demás variables serán producidas por simuladores exclusivamente basados en software.

4) Arquitectura

El siguiente diagrama ofrece una vista de alto nivel de la arquitectura diseñada a partir de la investigación de las distintas capas que conforman el sistema.

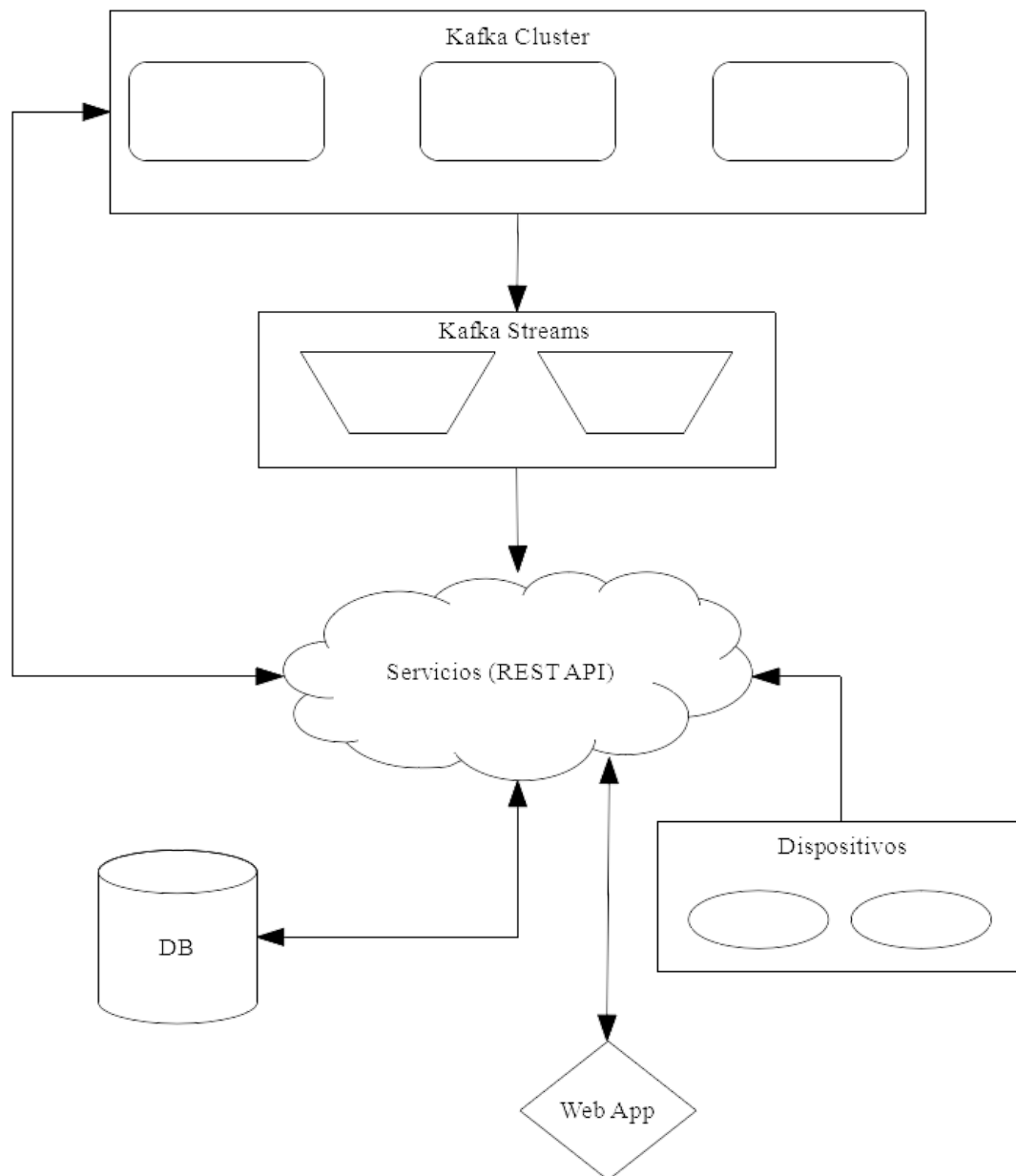


Figura 18. Arquitectura del sistema

4.1) Servidor

La implementación de la arquitectura Kappa del lado del back-end se separa en cuatro capas principales. A continuación se describirán las responsabilidades de cada una, así como los componentes que las conforman.

4.1.1) Servicios (API REST)

En esta capa se encuentran los puntos de entrada y salida del sistema, respetando los principios de abstracción sugeridos por IoTA. Para esto se implementó un conjunto de servicios REST cuyos endpoints permiten la inserción de datos en el sistema así como el consumo de los mismos.

Para enviar datos como mediciones de sensores, los dispositivos envían peticiones HTTP a la API de sensores, enviando los datos en forma de JSON con una estructura determinada. Esta operación es poco costosa y fácil de implementar en los clientes con pocos recursos.

El consumo de la información del sistema está orientado principalmente a la visualización en un cliente web, pero es lo suficientemente modular como para ser utilizable por otras implementaciones de cliente. La API ofrece acceso a las agregaciones de datos realizadas por los jobs de streaming del sistema mediante peticiones HTTP tradicionales. Además, cuenta con un endpoint de alertas capacitado para la comunicación mediante WebSockets, para informar a los usuarios de situaciones de riesgo con la menor latencia posible.

La implementación de los servicios fue realizada utilizando J2EE como base y Jersey para los servicios REST. La conectividad mediante WebSockets es facilitada por librerías que ofrecen por defecto los JDK más recientes. Para acceder a los datos se utilizan dos estrategias distintas, según el tipo de funcionalidad deseada. Las agregaciones pre-calculadas se

encuentran en una base de datos relacional a la cual se accede mediante JDBC y *prepared statements* de SQL. Para la lectura de datos en tiempo real, y para las alertas, se instancia un consumidor de Kafka que se conecta directamente a los tópicos relevantes para hacer llegar los datos al cliente en forma transparente.

Los archivos estáticos de la interfaz web (HTML, JS, CSS) son también servidos en esta capa. El desarrollo de esta parte del sistema está basado en el framework AngularJS de Google.

Idealmente, esta capa estaría conformada por varios nodos detrás de un balanceador de cargas. Esto inevitablemente exige tomar otras decisiones relacionadas a la arquitectura del sistema, como podría ser la estrategia de balanceo de cargas. Sin embargo, esta cuestión está más ligada a otras áreas de estudio y no se analizará en mayor detalle en este proyecto.

4.1.2) Mensajería (Cluster Kafka)

Los datos que ingresan al sistema se almacenan de manera inmutable en distintos tópicos de Kafka. La inserción de datos es realizada por productores de Kafka ubicados en la capa de servicios, que intermedian la comunicación entre los sensores y los tópicos.

Los aspectos técnicos de Kafka fueron analizados con detalle en una de las etapas de investigación previas. A grandes rasgos, se pueden identificar cuatro componentes principales:

- Gestor de Recursos: La sincronización del comportamiento distribuido de Kafka es realizada por una instancia de Zookeeper. Zookeeper es configurable para adaptarse al entorno de ejecución (por ejemplo, los puertos a utilizarse en las conexiones).
- Servidores: Los servidores de Kafka se ejecutan en distintos nodos, y con la configuración necesaria pueden operar en forma conjunta, haciendo posible las operaciones de particionado y replicado de los datos.
- Sistema de Archivos: En el núcleo de Kafka se encuentra el *log*, que como se explicó anteriormente es el almacén central de los datos.

- Clientes (Productor/Consumidor): Estos clientes se ejecutan en otros contextos mediante librerías que facilitan la lectura y escritura a los tópicos.

La configuración utilizada tanto para Zookeeper como para Kafka es, en su mayor parte, la recomendada por los autores del proyecto. Algunas modificaciones fueron introducidas para adecuar la distribución de brokers definida para este proyecto. Para poner a prueba las capacidades de particionado y replicado de Kafka, se optó por elevar el factor de replicación y el máximo de particiones por defecto a 2.

El rendimiento del sistema de mensajería será evaluado mediante un test de estrés utilizando JMeter en una etapa posterior. Según un artículo referido por la documentación oficial de Kafka³, el diseño *append-only* de los tópicos permite lograr un alto nivel de producción incluso en equipos modestos en cuanto a hardware.

4.1.3) Streaming

La capa de streaming está compuesta por aplicaciones Java de ejecución constante, que utilizan la librería de Kafka Streams para implementar la funcionalidad requerida. Kafka Streams abstrae los aspectos de bajo nivel de la interacción con el tópico, y permite al desarrollador concentrarse en el comportamiento concreto.

La mayoría de las operaciones pueden implementarse bajo el paradigma funcional, definiendo una *pipeline* de operaciones sobre los datos que van ingresando en el tópico. Estas operaciones incluyen transformaciones, agrupamientos y agregaciones para obtener el resultado deseado.

Uno de los objetivos del sistema es procesar los datos para presentarlos en forma relevante al usuario. Para ello, una de las funcionalidades consiste en agregar las mediciones de los sensores en ventanas de tiempo de tamaño variable. Este tipo de agregación es

³ Kreps, J.. Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)

<<https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>>

soportado por defecto por Kafka Streams, por lo que se puede implementar con pocas líneas de código (referirse al anexo G para más detalles).

Otra funcionalidad clave es la emisión de alertas cuando se detecta que una medición no cae dentro de un rango determinado. Esto se implementa mediante un stream que monitorea los datos que ingresan a los tópicos y notifica a la capa de servicios cuando se detecta un caso de riesgo.

4.1.4) Base de Datos (Vistas)

Para la persistencia de las *vistas*, que en Kappa representan el producto de la capa de streaming, se utiliza una base de datos relacional. Por facilidad de acceso se optó por una de código abierto, particularmente MySQL Community Edition. Las escrituras y lecturas se realizan desde los demás componentes mediante JDBC.

Siguiendo los lineamientos de Kappa, las vistas generadas son de carácter transitorio, ya que pueden regenerarse a partir de los datos almacenados en Kafka (la única fuente de verdad), de ser necesario. Este sería el caso, por ejemplo, de alguna corrección en los jobs de streaming.

4.2) Dispositivos IoT

La proyección del sistema en el plano físico es realizada mediante los dispositivos. Estos pueden ser considerados los puntos de entrada de un sistema IoT, y como se analizó previamente, pueden ser de distintas naturalezas: sensores, etiquetas, actuadores. Para el sistema de monitoreo de silos, se necesitarán principalmente sensores de distintos tipos que permitan digitalizar el valor de las distintas variables ambientales relevantes. En esta sección se analizará brevemente cual es la oferta en el mercado de esta clase de instrumentos electrónicos. Se concluirá con una explicación de la implementación real de un sensor de temperatura que se realizó a modo de prototipo, así como su integración al sistema central.

Se debe tener presente que el aspecto electrónico de los sensores cae, en su mayor parte, fuera del área de estudio del presente proyecto, por lo que se evaluarán respecto a la utilidad que ofrecen desde un punto de vista informático.

4.2.1) Sensor

Lo único que se necesita es obtener los dispositivos (hardware) necesario, por lo que se descartan ciertas opciones que ofrecen soluciones integrales que incluyen los componentes de software. Mediante búsqueda en línea se identificaron algunos grandes proveedores, incluyendo IBM, Molex y TE Connectivity. No es de interés comparar las ofertas en términos económicos, sino que se pretende adquirir un grado básico de conocimiento respecto las implicaciones de instalar estos sensores.

Se tomará como punto de referencia (en forma arbitraria) a TE Connectivity. En el catálogo de productos hay un área dedicada para sensores de distintos tipos. Entre los tipos de sensores, se encuentran aquellos que miden las variables que se identificaron como relevantes para el caso de negocio, incluyendo: temperatura, humedad, vibración, presión, etc. Algunas de estas categorías están divididas a su vez por la implementación particular del sensor; esto es evidenciable principalmente en los sensores de temperatura, de los que hay más de 200 tipos.

Yendo más a detalle en el caso de los sensores de temperatura, se observa que los distintos tipos de sensor ofrecen distintos niveles de precisión, interfaces de conectividad, rangos de operación y consumo de energía. Eligiendo un sensor determinado, se pueden observar los siguientes datos:

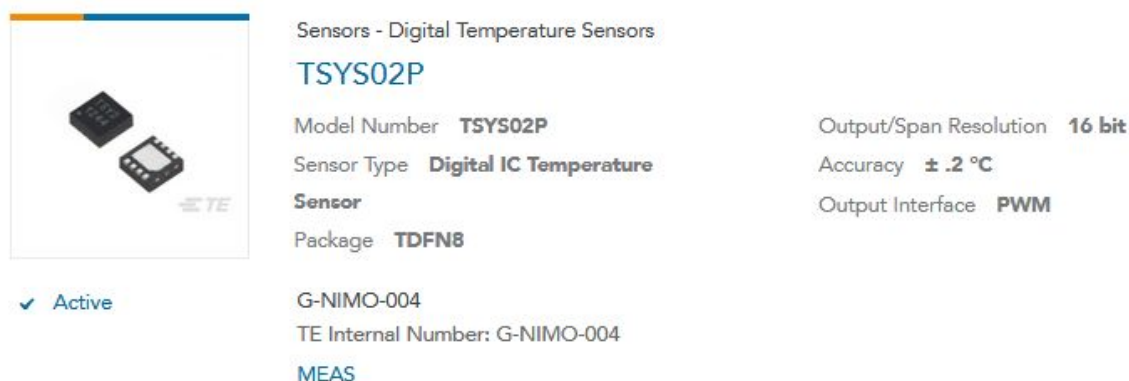


Figura 19. Especificaciones principales de un sensor digital de temperatura (TE Connectivity)

4.2.2) Placa (Microcontrolador)

Una de las preguntas que surgen habiendo visto esto, es “de qué manera se integra este dispositivo con el resto del sistema”. Por sí solo, el sensor de poca utilidad, ya que necesita una fuente de energía para operar y un destino al cual enviar su *output* en forma de señales eléctricas, así como un componente que interprete estas señales y permita realizar operaciones significativas sobre ellas. Esto significa que falta un componente más para constituir lo que se definió previamente como dispositivo IoT.

Estos requerimientos pueden ser cubiertos por una placa con microcontrolador. Estas unidades poseen capacidad de procesamiento (CPU), memoria de trabajo (ROM y RAM) mediante el microcontrolador, y generalmente cuentan con los circuitos necesarios para conectarse con otros elementos compatibles. En las instalaciones reales, la conexión entre la placa y el sensor se realiza mediante soldaje, aunque para prototipar es normal utilizar un *breadbord* que permite realizar la conexión en forma transitoria mediante *jumper wires*, que no necesitan ser soldados. Según la placa, se puede además conectar más de un sensor al mismo tiempo, potencialmente reduciendo costos de adquisición de hardware, consumo de energía, implementación y mantenimiento.

Los microcontroladores son programables. La arquitectura del procesador (e.g. ARM) dicta qué tipo de software puede ejecutar, y este generalmente es almacenado en alguna

memoria permanente como puede ser una ROM de tipo *flash*. El controlador ofrece interfaces para que el código acceda al valor del voltaje de las señales eléctricas emitidas por el sensor, conociendo de antemano la ecuación para traducir dicho voltaje en una magnitud de significado real. Esto se ejemplifica a continuación en la explicación del sensor implementado para el prototipo.

El costo de la placa estará asociado a variables como la velocidad del procesador, memoria disponible, cantidad y tipo de periféricos (análogos o digitales), entre otras.

4.2.3) Implementación de un dispositivo IoT

Uno de los objetivos definidos en el alcance del proyecto era la implementación real de al menos un sensor y su integración con el sistema. El prototipado de estos dispositivos ha sido facilitado por plataformas como Arduino, que integran distintas partes del proceso en una manera accesible para aquellos no especializados en el área.

Arduino es una compañía y proyecto de software y hardware libre; sus productos son distribuidos bajo la Licencia Pública General GNU (GPL). Se pueden fabricar placas basándose en dichos productos, pero el público general accede a ellas mediante la compra de pre-ensamblados. En este caso, se adquirió una placa Arduino UNO Rev3, uno de los modelos más básicos, a través un distribuidor independiente.

El software que se ejecuta en el microcontrolador se escribe en C++ tradicional, con la salvedad de que se incluye dentro de las librerías básicas aquella que administra el I/O con los componentes conectados, como los sensores. El programa debe definir además dos funciones estándares de *setup* y *loop*. Como sus nombres indican, la primera debe inicializar el programa, esto generalmente implica por lo menos abrir una conexión con la terminal central, y en *loop* se especifica el procedimiento que se ejecutará mientras la placa permanezca encendida. Arduino recomienda el uso de un IDE estándar que facilita el compilado de los archivos (*sketches*) y su escritura a la memoria del controlador.

Para realizar las conexiones necesarias, fue necesario obtener un *breadbord* (placa de pruebas con orificios conectados electrónicamente) y un conjunto de cables de terminaciones

macho-macho. Finalmente, se eligió un sensor de temperatura de fácil acceso tanto de instalación como económico, particularmente del modelo LM35. El mismo cuenta con una precisión de 0.5°C y opera en el rango de -55 a 150°C . El sensor posee 3 pines, uno de alimentación (5V), uno de tierra y uno de salida.

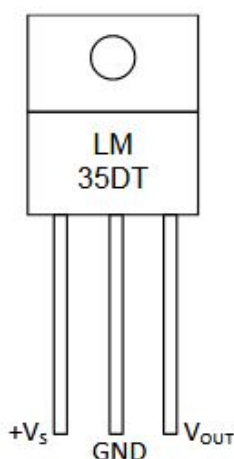


Figura 20. Pines de un sensor de temperatura modelo LM35

La instalación final puede observarse en las siguiente fotografías, siendo:

1. Placa Arduino UNO Rev3
2. PIN entrada análoga
3. PIN de tierra (GND)
4. Breadboard
5. Sensor de temperatura LM35
6. Microcontrolador (ATmega 328P)
7. PIN de alimentación de 5V

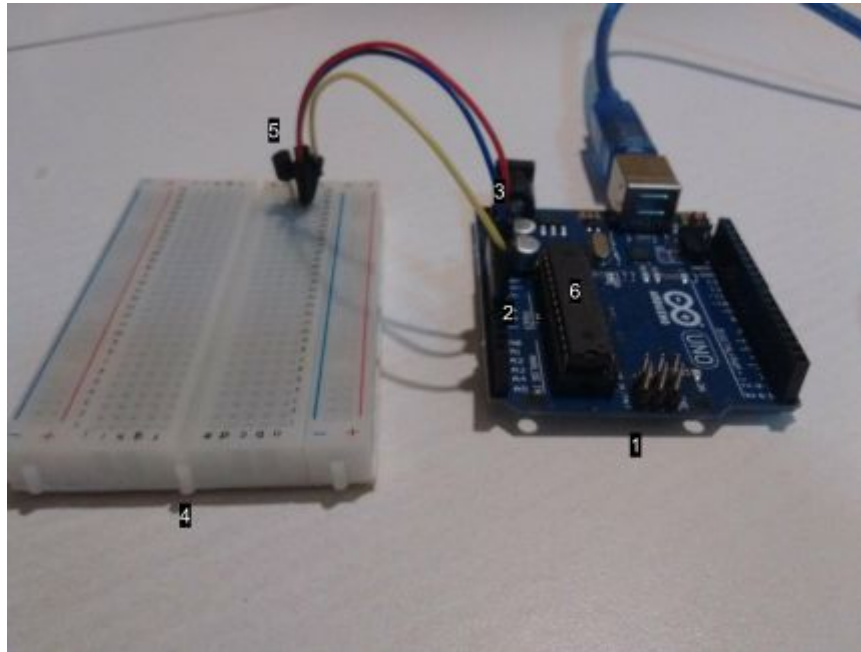


Figura 21. Vista lateral del dispositivo

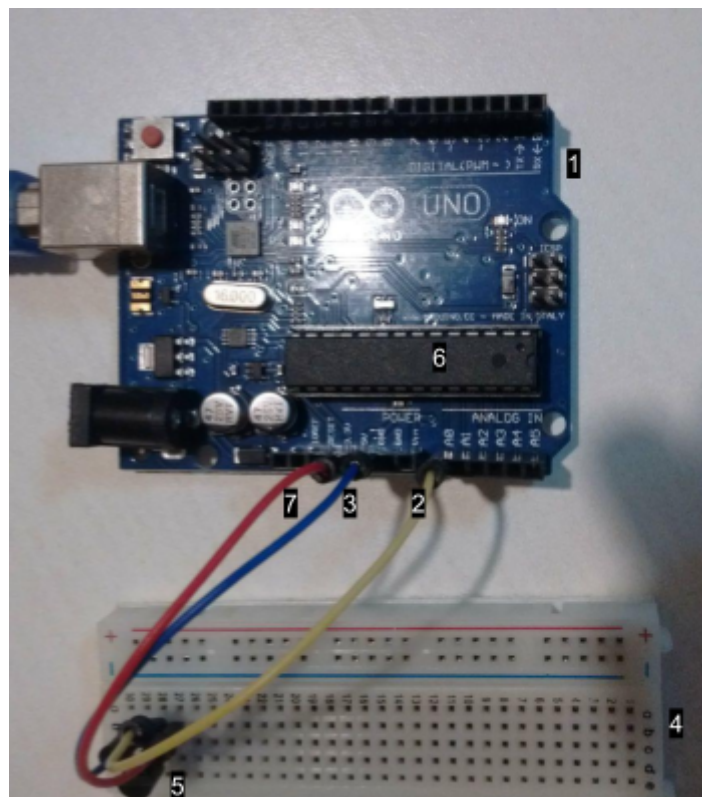


Figura 22. Vista frontal del dispositivo

Una vez realizadas las conexiones, se debe escribir un programa que verifique que se realizó correctamente. A tal fin, se puede utilizar un *sketch* sencillo que lea el valor de la entrada con y lo envíe al terminal principal mediante comunicación serial. Se denomina terminal principal a la estación de trabajo donde se conectó la placa (en este caso, una PC de escritorio). El código puede encontrarse en el anexo F). El funcionamiento del programa se puede corroborar mediante el monitor serial incluido en Arduino IDE, que muestra los bytes enviados en pantalla.

Ahora bien, el valor que se lee del PIN de entrada en forma analógica no es semántico por su propia cuenta. La función *analogRead(pin)* expuesta por la librería básica de Arduino mapea voltajes de 0 a 5 voltios a un entero entre 0 y 1023 (1024 unidades). De esta relación, se deduce que, siendo U la unidad de lectura analógica:

$$1024U = 5000mV$$

$$1U = \frac{5000mV}{1024}$$

$$1U = 4,88mV \quad (1)$$

Por lo que cada U vale equivale a aproximadamente 4,88mV. Se sabe además, de la documentación de LM35, que la transferencia de voltios a grados centígrados se realiza mediante la ecuación:

$$V_{OUT} = 10 \frac{mV}{^{\circ}C} \times T \quad (2)$$

De (1), se sabe que:

$$4,88 \frac{mV}{U} \times U_{OUT} = V_{OUT} \quad (3)$$

Sustituyendo (3) en (2):

$$4,88 \frac{mV}{U} \times U_{OUT} = 10 \frac{mV}{^{\circ}C} \times T$$

$$\frac{4,88 \frac{mV}{U} \times U_{OUT}}{10 \frac{mV}{^{\circ}C}} = T$$

$$0,488 \frac{^{\circ}C}{U} \times U_{OUT} = T$$

Por lo tanto, basta con multiplicar el resultado de la lectura analógica por una constante de 0,488 para convertirlo en grados centígrados.

Para mantenerse en línea los principios de abstracción propuestos por IoTA, la integración con el sistema principal se debe realizar en una manera poco intrusiva. Se optó por un enfoque basado en eventos, con una interface común (*Driver*) para todos los sensores. La implementación dependerá del hardware específico - en el caso de Arduino, se realizan lecturas bloqueantes sobre el puerto serial al cual está conectado el dispositivo, disparando un evento con el valor de la medición cada vez que se encuentran datos.

Una de las ventajas de esta estrategia es que se pueden utilizar implementaciones que simulen esta clase de eventos de manera transparente para los demás componentes. Esto será necesario para ingresar en el sistema datos de otras variables como la humedad y presión, dado que no se utilizarán sensores reales.

4.2.4) Modelo de Dominio

Para definir el modelo de dominio según la recomendación de IoTA, se deben asociar los elementos de la red a los conceptos clave (Entidad Física, Virtual, Servicio, Recurso, etcétera). A su vez, se deben establecer los atributos básicos de los conceptos que permitan identificarlos, así como las relaciones que tienen entre sí. Se pretende mediante la definición de este modelo eliminar ambigüedades en cuanto al entendimiento del sistema desde el punto de vista de IoT.

El diagrama resultante se muestra a continuación. Algunos puntos que merecen aclaración:

- El acceso a los datos medidos por los sensores se realiza comunicándose con el controlador de las placas, por lo que se lo categoriza como Recurso.
- Se puede observar una jerarquía en los servicios según niveles de abstracción, pero en menor o mayor medida todos cumplen la función de abstraer el acceso al Recurso.
- Como explica IoTA, un Usuario en el modelo puede ser un agente de software, como lo es en esta arquitectura el conjunto de procesadores de Streams.

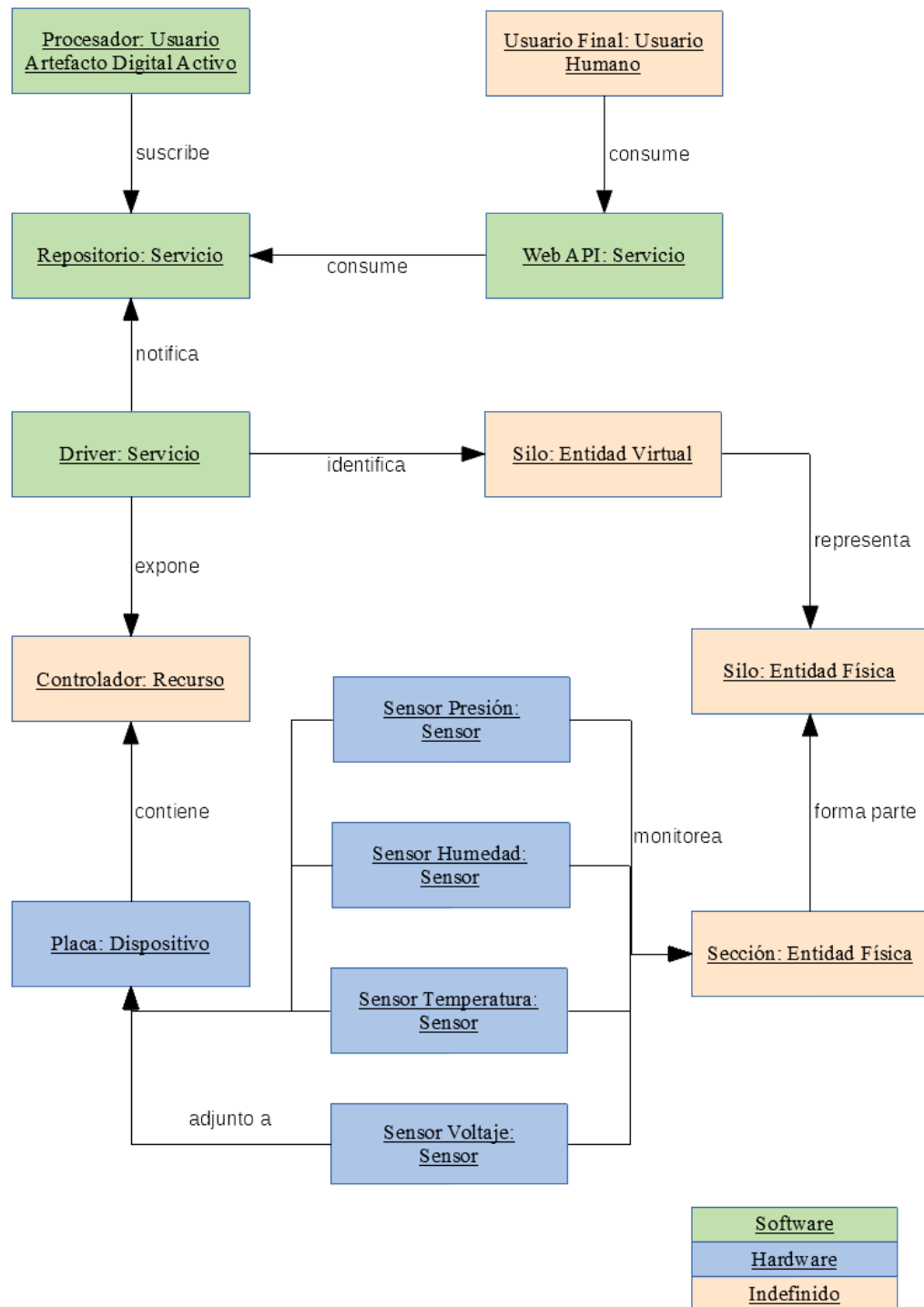


Figura 23. Modelo de dominio del sistema

4.3) Infraestructura

El objetivo de esta sección es distinguir claramente los elementos que constituyen el sistema, y que deben ser tenidos en cuenta a la hora de ejecutar la aplicación en un entorno de remoto (cloud). Una vez identificados, se sugerirá una instalación que permita realizar pruebas sobre la aplicación en funcionamiento - actividad que será el tema central de uno de los apartados finales del presente documento.

4.3.1) Componentes

La implementación del sistema requiere una infraestructura conformada por varios componentes, que son listados a continuación:

- Servidor de gestión de recursos: Ejecuta el servicio Zookeeper, coordinador central entre los distintos nodos de Kafka, por lo que debe ser accesible a estos.
- Nodos de Kafka: Ejecutan un broker de Kafka y almacenan los eventos en disco. La velocidad de escritura de Kafka es alta por la estrategia *append-only*, por lo que es más importante concentrarse en propiedades como la memoria y la capacidad de almacenamiento.
- Nodos de Streams: Ejecutan una o más aplicaciones de Java tradicionales que utilizan la librería de Kafka Streams para procesar los datos que ingresan en el log. Requieren conexión con los nodos de Kafka para operar.
- Servidor Web: Para los fines prácticos del proyecto solamente se utiliza un servidor web, sin implementar una estrategia de balanceo de cargas. Este servidor debe contar con una IP pública mediante la cual los usuarios humanos y digitales acceden a los servicios del sistema.
- Base de datos MySQL: Contiene las vistas generadas por los jobs de streaming.
- Servidor local de sensores: Se prefiere que los sensores no realicen la conexión con el servidor central en forma directa, sino que lo hagan mediante un servidor instalado en las proximidades de las redes de sensores vía LAN.

- Sensores: Los dispositivos ubicados directamente en las instalaciones deben poder comunicar las mediciones al servidor local. Idealmente estarían capacitados para conexión inalámbrica.

Todos los componentes que cumplan función de servidor deben tener instalado un sistema operativo GNU/Linux y un Runtime de Java (OpenJDK preferentemente).

4.3.2) Dimensionamiento de hardware

Para ejecutar los servidores del sistema se escogerá un servicio de hosting que cumpla los requisitos definidos en el ítem anterior. Cabe aclarar que el servidor de sensores se ejecutará en forma local.

El proveedor de hosting elegido fue OVH⁴ ya que cuenta con un paquete de VPS con características aceptables a un costo mensual relativamente bajo de 4U\$. El servidor accesible mediante Internet y cuenta con 1 núcleo (virtual) de 2.4GHz, 2GB de RAM y 10GB de SSD. Además, viene pre-instalada una distribución de Linux a elección con lo mínimo necesario para iniciar una sesión de terminal, permitiendo personalizar el sistema libremente. Las tareas de administración, como la liberación de la aplicación, pueden realizarse mediante SSH.

Una de las cuestiones que mayor importancia tomó durante el análisis del proyecto es la capacidad del sistema de operar en un entorno distribuido. Para evaluar este concepto se contrataron dos unidades del paquete mencionado en el párrafo anterior. La distribución de componentes entre las dos máquinas se realizó de la siguiente manera:

- Máquina I: nodo Kafka I, Base de Datos MySQL, Streams de Agregación
- Máquina II: nodo Kafka II, Zookeeper, Servidor Web Tomcat, Streams de Alertas

⁴ <https://www.ovh.com/us/cloud/>

Se puede observar que ambos servidores ejecutan un broker de Kafka, por lo que será posible corroborar el particionamiento y replicación de los datos. Para concluir, el componente que se ejecutará en forma local será el servidor de sensores. Como se mencionó previamente la mayoría de los sensores serán simulados mediante software, pero uno de ellos (de temperatura) será integrado mediante una placa Arduino UNO.

4.3.3) Ejecución de pruebas

Una vez verificado el funcionamiento de la aplicación en esta instalación se realizarán pruebas de rendimiento que permitan obtener algunas métricas representativas. La descripción de los casos de prueba y los resultados se especificarán en la sección de **Pruebas Realizadas**.

La ejecución de las pruebas se realizará mediante JMeter⁵, realizando peticiones a la Web API expuesta a la Internet creando mediciones de prueba, generando carga sobre los nodos de Kafka y los jobs de Streams. Se intentará obtener indicadores de la cantidad de registros procesados por unidad de tiempo, con el fin de estimar las limitaciones de la instalación actual y poder producir aproximaciones sobre lo que sería un caso de uso real.

⁵ La aplicación Apache JMeter™ es software de código abierto, 100% en Java, diseñada para realizar pruebas de carga sobre el comportamiento funcional y medir rendimiento (traducido de <https://jmeter.apache.org/>).

5) Análisis Técnico del Prototipo

En el siguiente apartado se realizarán comentarios sobre la estructura del código del prototipo, así como los requerimientos de sistema para ejecutar y desarrollar la aplicación. El objetivo principal de esta sección es proveer el material necesario para que alguna parte interesada pueda trabajar sobre el prototipo.

5.1) Configuración para desarrollo local

El desarrollo del sistema fue realizado en una máquina virtual de Linux para posibilitar la portabilidad entre distintas plataformas. El hipervisor utilizado fue VirtualBox, por su disponibilidad en distintos sistemas operativos, además de ser de código abierto. La distribución de Linux elegida fue Ubuntu 16.10 por su facilidad de instalación y estabilidad. El software de base requerido para ejecutar/modificar la aplicación incluye:

- MySQL Community Server & Client: Se puede instalar mediante el gestor de paquetes de Ubuntu (apt). Está incluido en el repositorio por defecto de la distribución. La creación del usuario de la base de datos de la aplicación así como las tablas es facilitado por un shell script que se encuentra en el repositorio de git del sistema.
- Apache Tomcat: Cumple el rol de webserver para exponer los servicios del sistema y servir los archivos estáticos del front-end.
- Git: Necesario para clonar el repositorio de código, así como para subir modificaciones al mismo. Disponible en el repositorio de paquetes de Ubuntu.
- Maven: Es el sistema de build automatizado por excelencia para el desarrollo en Java, también se puede encontrar en el repositorio de Ubuntu. Se puede utilizar mediante la línea de comandos o a través de un IDE que soporte Maven.
- Eclipse IDE (Java EE): Por familiaridad y disponibilidad, el desarrollo se realizó utilizando Eclipse como IDE.

-
- Apache Kafka: Los binarios de Kafka se pueden obtener en su página principal. El desarrollo se realizó utilizando las librerías compatibles con la versión 0.10.2.0 de Kafka, que al comienzo del proyecto era la versión más reciente.
 - NodeJS: Para el build de la aplicación web, así como la gestión de librerías utilizadas en el frontend se utilizó el gestor de paquetes de NodeJS, *npm* (Node Package Manager).
 - Gulp: Es un conjunto de herramientas que permite definir tareas para automatizar el build de aplicaciones, generalmente de frontend. El entregable incluye el conjunto de tareas necesarias para generar la versión distribuible del front-end del sistema.

5.2) Instalación

Antes que nada se deben instalar los componentes necesarios. Utilizando *apt* y *wget* en Ubuntu, esto se puede realizar en unos pocos comandos.

```
sudo apt install git
```

```
sudo apt install mysql-server
```

```
sudo apt install nodejs && sudo apt install nodejs-legacy && npm install -g gulp
```

```
wget http://apache.dattatec.com/tomcat/tomcat-9/v9.0.4/bin/apache-tomcat-9.0.4.tar.gz &&  
tar -xf apache-tomcat-9.0.4.tar.gz
```

```
wget https://archive.apache.org/dist/kafka/0.10.2.0/kafka_2.11-0.10.2.0.tgz && tar -xf  
kafka_2.11-0.10.2.0.tgz
```

```
pfi@pfi:~/Desktop$ wget https://archive.apache.org/dist/kafka/0.10.2.0/kafka_2.11-0.10.2.0.tgz && tar -xf kafka_2.11-0.10.2.0.tgz  
--2018-02-01 10:13:44-- https://archive.apache.org/dist/kafka/0.10.2.0/kafka_2.11-0.10.2.0.tgz  
Resolving archive.apache.org (archive.apache.org)... 163.172.17.199  
Connecting to archive.apache.org (archive.apache.org)|163.172.17.199|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 37630750 (36M) [application/x-gzip]  
Saving to: 'kafka_2.11-0.10.2.0.tgz'  
  
kafka_2.11-0.10.2.0 100%[=====>] 35,89M 627KB/s in 50s  
  
2018-02-01 10:14:35 (732 KB/s) - 'kafka_2.11-0.10.2.0.tgz' saved [37630750/37630750]
```

Luego, se procede a clonar el repositorio de la solución. Para ello se utiliza el comando *git clone*:

```
git clone https://github.com/aci2n/isms.git
```

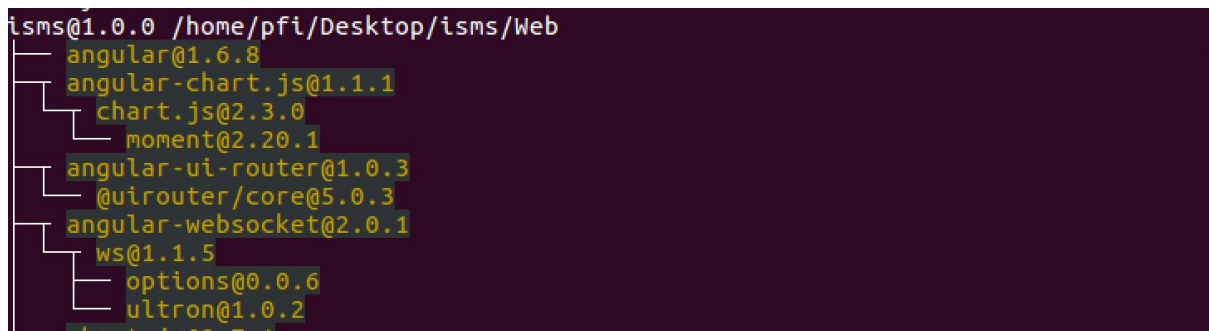
```
pfi@pfi:~/Desktop$ git clone https://github.com/aci2n/isms  
Cloning into 'isms'...  
remote: Counting objects: 1831, done.  
remote: Total 1831 (delta 0), reused 0 (delta 0), pack-reused 1831  
Receiving objects: 100% (1831/1831), 506.92 KiB | 271.00 KiB/s, done.  
Resolving deltas: 100% (894/894), done.  
Checking connectivity... done.
```

En el repositorio clonado hay una carpeta nombrada *Database*. Dentro de esa carpeta hay dos scripts de Bash que preparan la base de datos para el uso en la aplicación.

```
cd Database && ./pre-setup.sh && ./setup.sh
```

Volviendo al directorio principal, hay otra carpeta nombrada *Web*. Para instalar las dependencias del front-end, se utiliza *npm*, y luego *gulp* para compilar los archivos del proyecto a un formato distribuible.

```
cd Web && npm install && gulp
```



```
isms@1.0.0 /home/pfi/Desktop/isms/Web
├─┬ angular@1.6.8
│   └─┬ angular-chart.js@1.1.1
│       └─┬ chart.js@2.3.0
│           └─┬ moment@2.20.1
│               └─┬ angular-ui-router@1.0.3
│                   └─┬ @uirouter/core@5.0.3
│                       └─┬ angular-websocket@2.0.1
│                           └─┬ ws@1.1.5
│                               └─┬ options@0.0.6
│                                   └─┬ ultron@1.0.2
│                                       └─┬ chart.js@2.3.0
```

El último paso antes de poder iniciar el sistema consiste en copiar los archivos de configuración de Kafka personalizados al directorio de configuración de Kafka. Para ello, basta con el siguiente comando (*\$KAFKA_DIR* siendo el directorio de instalación):

```
cd Streams/resources && cp isms-server-1.properties isms-server-2.properties
$KAFKA_DIR/config
```

5.3) Ejecución

La ejecución del sistema puede dividirse en tres partes - la iniciación de los brokers de Kafka, de los Streams y el back-end del sistema en sí.

Para iniciar los brokers de Kafka, primero se debe ejecutar Zookeeper, que intermedia la comunicación entre los distintos brokers. Junto con los binarios de Kafka, se incluyen algunos scripts de administración que facilitan esta tarea. Para iniciar Zookeeper se debe ejecutar el siguiente comando, dentro del directorio de instalación de Kafka.

./bin/zookeeper-server-start.sh config/zookeeper.properties

```
pfi@pfi:~/Desktop/kafka_2.11-0.10.2.0$ ./bin/zookeeper-server-start.sh config/zoo
ookeeper.properties
[2018-02-01 10:43:06,468] INFO Reading configuration from: config/zookeeper.prop
erties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2018-02-01 10:43:06,471] INFO autopurge.snapRetainCount set to 3 (org.apache.zo
ookeeper.server.DataDirCleanupManager)
[2018-02-01 10:43:06,471] INFO autopurge.purgeInterval set to 0 (org.apache.zook
eeper.server.DataDirCleanupManager)
[2018-02-01 10:43:06,471] INFO Purge task is not scheduled. (org.apache.zookeep
er.server.DataDirCleanupManager)
[2018-02-01 10:43:06,471] WARN Either no config or no quorum defined in config,
running in standalone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)
[2018-02-01 10:43:06,488] INFO Reading configuration from: config/zookeeper.prop
erties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2018-02-01 10:43:06,488] INFO Starting server (org.apache.zookeeper.server.ZooK
eeperServerMain)
```

A continuación se pueden arrancar los brokers de Kafka. Para el desarrollo generalmente alcanza con iniciar uno solo. Esto se logra ejecutando otro script de administración, y utilizando el archivo de configuración que se copió previamente.

./bin/kafka-server-start.sh config/isms-server-1.properties

```
pfi@pfi:~/Desktop/kafka_2.11-0.10.2.0$ ./bin/kafka-server-start.sh config/isms-s
erver-1.properties
[2018-02-01 10:51:59,411] INFO KafkaConfig values:
    advertised.host.name = null
    advertised.listeners = null
    advertised.port = null
    authorizer.class.name =
    auto.create.topics.enable = true
    auto.leader.rebalance.enable = true
    background.threads = 10
    broker.id = 1
```

Habiendo iniciado los servicios necesarios, ya es posible iniciar el servidor web. Esto puede realizarse utilizando cualquier contenedor de JavaEE; en este caso se utiliza Apache Tomcat. Eclipse puede integrarse con este servidor y permite inicializarlo desde la UI - este es el método recomendado, por su facilidad, durante el desarrollo.

```
Tomcat v9.0 Server at localhost [Apache Tomcat] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Feb 13, 2018, 11:26:29 PM)
Feb 13, 2018 11:26:31 PM org.apache.tomcat.util.digester.SetPropertiesRule begin
WARNING: [SetPropertiesRule]{Server/Service/Engine/Host/Context} Setting property 'source' to 'org.ecl
Feb 13, 2018 11:26:31 PM org.apache.catalina.startup.VersionLoggerListener log
INFO: Server version: Apache Tomcat/9.0.0.M19
Feb 13, 2018 11:26:31 PM org.apache.catalina.startup.VersionLoggerListener log
INFO: Server built: Mar 27 2017 12:09:59 UTC
Feb 13, 2018 11:26:31 PM org.apache.catalina.startup.VersionLoggerListener log
INFO: Server number: 9.0.0.0
Feb 13, 2018 11:26:31 PM org.apache.catalina.startup.VersionLoggerListener log
INFO: OS Name: Linux
Feb 13, 2018 11:26:31 PM org.apache.catalina.startup.VersionLoggerListener log
```

Para ver la aplicación en funcionamiento se deben iniciar los simuladores de datos que básicamente son clientes que generan peticiones HTTP al servidor central para almacenar mediciones de sensores ficticias. Para esto se creó una clase auxiliar denominada TestSimulatorSensor que simula 10 sensores de distintos tipos y locaciones simultáneamente.

```
TestSimulatorSensor [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Feb 13, 2018, 11:28:04 PM)
simulating: 0.7070944015612668
simulating: -2.600600323077556
simulating: 1.8114187256654668
simulating: -3.104470865301098
simulating: 0.16370009021074047
simulating: 0.015097049337680373
simulating: 2.722915121738634
simulating: 2.410743109247547
simulating: 0.3793000123333214
simulating: -0.021407493335058
```

A esta altura ya puede visualizarse la entrada de datos en tiempo real al sistema mediante la función de monitoreo. Para ello, se debe acceder a la URL <http://localhost:8080/app/dashboard/monitor/>.⁶ Allí podrán observarse los distintos tipos de sensores y las ventanas de tiempo de agregación disponibles.

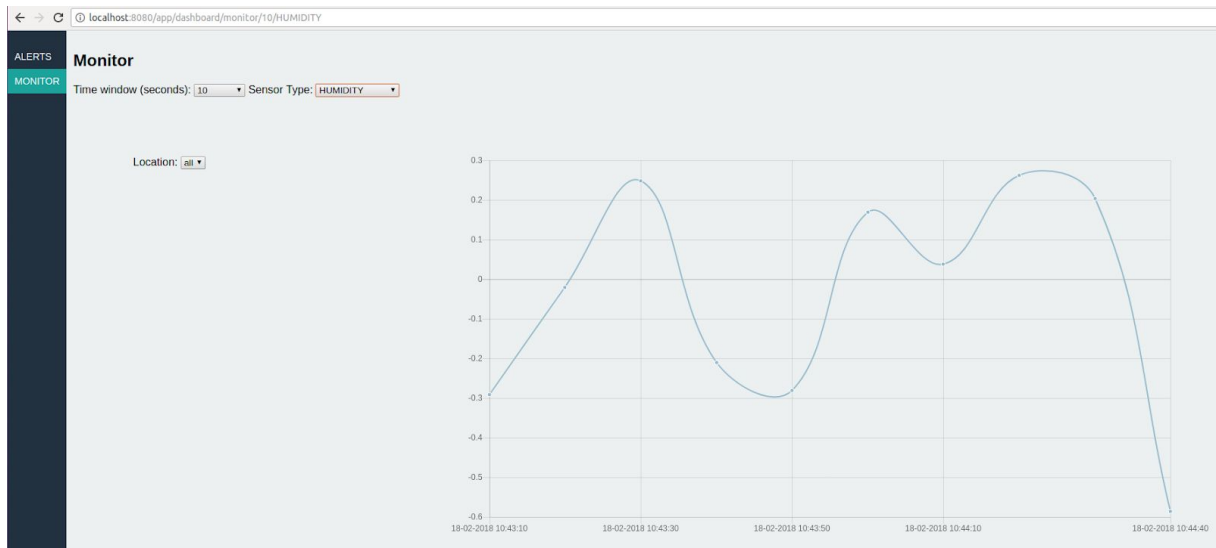
⁶ El hostname y puerto de la URL es configurable, la mencionada aquí es la utilizada por defecto.



En este punto solo está disponible la ventana de tiempo *realtime*. Para comenzar a generar agregaciones en distintas ventanas de tiempo, deben iniciarse los jobs de streaming. Esto se realiza mediante una aplicación de Java tradicional. En Eclipse, basta con ejecutar el archivo `AggregatorTest.java`.

```
AggregatorTest (2) [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Feb 18, 2018, 10:44:14 PM)
Launching stream with window size: 10
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

De esta manera, los streams comienzan a poblar la base de datos con las vistas de agregación de datos por ventana de tiempo. En la aplicación esto se puede visualizar seleccionando la opción “10” en el dropdown.



5.4) Estructura

5.4.1) Proyectos

El código del proyecto se divide en distintas unidades con responsabilidades claramente definidas, que en cierta medida se corresponden con las distintas áreas del sistema. Cada unidad está conformada por un proyecto Java distinto, e interactúan entre sí para lograr el funcionamiento del sistema. A continuación se listan estas unidades y se describe brevemente el rol de cada una.

- *Common*: Este módulo contiene aquellas clases que son utilizadas frecuentemente en todo el sistema. Esto incluye clases de utilidad y modelos de los datos.
- *Database*: Contiene las clases que abstraen el acceso a la base de datos para actualizar y consultar las vistas precalculadas por los streams.
- *Drivers*: Define las clases que administran la comunicación de bajo nivel con los sensores. Para este prototipo, esto implica la conexión mediante puerto serial con el controlador Arduino.
- *KafkaSerialization*: Define las implementaciones de las interfaces utilizadas por los clientes de Kafka para serializar y deserializar los datos. Como se utilizan tanto en la capa de Streams como en la de Web, se extrajeron a un módulo independiente.
- *Sensor*: Contiene los ejecutables (clases con métodos *main*) que inician el proceso de captura y envío de datos al sistema central.
- *Streams*: Define las abstracciones utilizadas en el proyecto para definir topologías de los streams, así como una interfaz de línea de comandos para iniciar streams de manera sencilla.
- *Web*: Define las APIs utilizadas por la aplicación web para realizar consultas y aquellas utilizadas por los sensores para enviar mediciones. Además contiene todos los archivos estáticos de HTML, (S)CSS y JS.

5.4.2) Librerías

A continuación se listan las librerías externas de las cuales depende el sistema. Algunas de ellas, como las relacionadas a Kafka, son centrales al funcionamiento de la aplicación, mientras que otras simplemente se utilizan para acelerar el desarrollo evitando la reimplementación de soluciones comunes.

5.4.2.1) Back-end

Las dependencias del Back-end son gestionadas mediante Maven, y especificadas concretamente en los archivos *pom.xml* de cada proyecto. A continuación se listan todas las librerías de terceros, independientemente del módulo del sistema al cual están relacionadas.

- Unirest: Conjunto de librerías livianas de HTTP mantenidas por Mashape.⁷ Utilizada del lado de los clientes que envían datos al servidor, como los sensores.
- jSerialComm: Librería de Java que provee acceso estandarizado, independiente de plataforma⁸. Utilizada para la comunicación con el microcontrolador de Arduino.
- Kafka Clients: Kafka Clients ofrece una forma estandarizada de leer y escribir datos a clusters de Kafka⁹. Se prefiere utilizar la librería de Streams ya que esta se ubica un nivel más arriba en cuanto a abstracción, pero para ciertas operaciones de más bajo nivel se utilizan los clientes directamente. Estas operaciones incluyen por ejemplo la escritura a Kafka de mediciones de sensores que el servidor web recibe mediante la API REST.
- Kafka Streams: Kafka Streams es una librería de cliente para construir aplicaciones donde las entradas y salidas son almacenadas en clusters de Kafka¹⁰. Utilizada para la implementación de los jobs de streaming.

⁷ <http://unirest.io/java.html>

⁸ <https://fazecast.github.io/jSerialComm/>

⁹ <https://docs.confluent.io/current/clients/index.html>

¹⁰ <https://kafka.apache.org/documentation/streams/>

- Tomcat JDBC Pool: Facilita la implementación de una pool de conexiones a la base de datos en un servidor Tomcat¹¹. Optimiza el uso de recursos del servidor web reutilizando conexiones existentes.
- MySQL Connector/J: Driver oficial de JDBC para MySQL¹².
- Jersey: Implementación de JAX-RS que facilita el desarrollo de servicios REST¹³. Ofrece abstracciones de estilo declarativo (anotaciones) para especificar los endpoints del servicio.

5.4.2.2) Front-end

Las librerías utilizadas para el desarrollo de la interfaz web se administran mediante *npm*, y se encuentran declaradas en el archivo *package.json* como es convención de este gestor de paquetes.

- AngularJS: AngularJS is un framework de JavaScript de código abierto mantenido principalmente por Google que simplifica algunas de las dificultades de desarrollar *single-page applications*¹⁴.
- Angular-ChartJS: Define directivas para AngularJS basadas en ChartJS, para la generación de gráficos de distintos tipos¹⁵.
- Angular-UIRouter: Framework de ruteo de *single-page applications* para AngularJS¹⁶.

¹¹ <https://tomcat.apache.org/tomcat-7.0-doc/jdbc-pool.html>

¹² MySQL Connector/J is the official JDBC driver for MySQL.

¹³ <https://jersey.github.io/>

¹⁴ <https://angularjs.org/>

¹⁵ <https://jtblin.github.io/angular-chart.js/>

¹⁶ <https://github.com/angular-ui/ui-router>

Pruebas Realizadas

Para realizar pruebas de carga sobre el backend se diseñó un plan de JMeter que realiza peticiones constantemente al servidor y registra los tiempos de respuesta. La cantidad de peticiones que se realizan en forma paralela es configurable, lo que permite estudiar cómo responde el sistema ante distintas cargas a lo largo de un tiempo determinado.

Esta sección consistirá de dos partes; en primer lugar se describe la estrategia de prueba, especificando detalles como qué partes del sistema se evaluarán, qué métricas se recolectarán y sobre qué ambientes se ejecutarán las pruebas. En la segunda parte se plasmarán los resultados junto con una breve descripción. En el apartado de **Discusión** se realizará un análisis más detallado.

1) Estrategia de Prueba

El aspecto del sistema a evaluar es la capacidad de introducción de datos en el tópico de Kafka. Este proceso es intermediado por el webserver, es decir, los clientes no tienen comunicación directa con el cluster de Kafka. Normalmente, el servidor no espera la respuesta del lado de Kafka antes de responder la petición, por lo que en este sentido es asincrónico (no tiene sentido demorar al cliente más de lo necesario). Sin embargo, para esta prueba resulta necesario realizar este proceso de manera sincrónica, para incluir en las mediciones el tiempo de computación que implica la escritura de registros en el tópico.

El plan de prueba en sí consiste en realizar peticiones constantemente con un grado determinado de paralelismo (usuarios concurrentes). Teniendo en cuenta el tiempo de respuesta de cada petición, y aprovechando ciertas características de JMeter que facilitan el agregado de resultados, se pueden determinar métricas como:

- Tiempos de respuesta
- Respuestas por unidad de tiempo (throughput)
- Porcentaje de errores

La terminal desde la que se ejecutarán las pruebas será una de las máquinas de desarrollo personales. El sistema en sí estará ejecutándose en un servidor en la nube, utilizando la infraestructura definida en el apartado de **Infraestructura** del **Marco Teórico**. De todos modos, se realizarán pruebas preliminares en una red local para obtener resultados donde el impacto de la latencia de red sea menor, y que sirvan para contrastar con los resultados recolectados del sistema remoto.

2) Ejecución de Pruebas

A continuación se plasmarán los resultados recolectados mediante JMeter. Aprovechando las capacidades de JMeter para agregar resultados, los tiempos de respuesta se registrarán en forma de promedio, mediana y los percentiles de 90, 95 y 99. Los planes de prueba utilizados se encuentran en el repositorio de la aplicación dentro del proyecto *Test*.

Además de variar la cantidad de usuarios concurrentes en cada escenario, también se realizarán pruebas utilizando las versiones sincrónicas y asincrónicas de la API. Calculando las diferencias entre ambos casos se puede aproximar el tiempo efectivo de procesamiento. La única diferencia entre ambos modos es que en el sincrónico, el servidor web espera la respuesta de Kafka antes de responder la petición del cliente. De esta manera, se puede percibir con mayor precisión el tiempo que demora Kafka en escribir el registro en el tópico.

Adicionalmente, se escribió un programa sencillo que escribe continuamente en un tópico de Kafka directamente, sin servidor web de por medio. El resultado de la ejecución de dicho programa también será tenido en cuenta durante la etapa de Discusión, principalmente para evaluar el impacto de la capa web en la latencia. Esta prueba se ejecutará contra el servidor remoto, por lo que también se verá afectada por la latencia de la red.

En la tabla, cada caso está identificado con un identificador de la siguiente manera:

<red><sincronicidad><concurrency>

<red> = L (local) o R (remoto)

<sincronicidad> = A (asincrónico) o S (sincrónico)

<concurrency> = número de usuarios paralelos

Ejemplo: **LS100** = Red local, API sincrónica, 100 usuarios paralelos

Caso	Muestras	Promedio (ms)	Mediana (ms)	Percentil 90 (ms)	Percentil 95 (ms)	Percentil 99 (ms)	Error (%)	Throughput (Respuestas/segundo)
LS100	2581981	23	18	33	42	96	0.004%	4307.31465
LS200	2921500	59	43	90	129	195	0.012%	5060.44072
LS500	3496245	85	77	130	160	223	0.014%	5822.83404
LA100	2475152	24	17	37	51	164	0.000%	4125.19146
LA200	3119463	56	32	83	167	537	0.000%	4530.67300
LA500	3815507	78	60	121	174	319	0.000%	6356.26505
RS1	2874	208	183	343	359	419	0.000%	4.78904
RS100	273133	219	184	345	363	422	0.00%	454.98811
RS200	279273	432	295	567	627	1738	0.05%	461.06337
RS500	272298	1101	487	2341	3284	5996	0.27%	453.90414
RA100	282117	216	183	340	361	403	0.002%	442.71313
RA200	312367	384	299	575	638	1731	0.005%	431.57723
RA500	293482	1052	418	1941	2850	5421	0.162%	465.36317

Tabla 4. Resultados de pruebas

La prueba de escritura directa a Kafka - de un único escritor concurrente - arrojó, por su parte, un throughput de 5,45 registros por segundo (3274 escrituras en 600 segundos). Esto se traduce a una latencia de aproximadamente **183 ms**.

Discusión

En la arquitectura del sistema bajo estudio, se pueden distinguir dos procesos donde el rendimiento es clave. El primer proceso se refiere a la escritura de datos en el repositorio central, que en este caso se implementa mediante un cluster de Kafka. El segundo proceso abarca el consumo de estos datos para generar las vistas relevantes al negocio, que como se explicó anteriormente se implementa mediante streaming.

En las pruebas realizadas, el objetivo fue entender las limitaciones del sistema respecto al primer proceso. La metodología utilizada para esto fue una prueba de carga tradicional donde se evalúa el impacto de la carga sobre los tiempos de respuesta del sistema.

El segundo proceso también posee ciertas métricas que describen su rendimiento, como podría ser el tiempo de llegada de un registro al tópic de Kafka hasta su inclusión en las vistas pertinentes. Esta medición, sin embargo, no se puede realizar mediante un plan de pruebas de JMeter únicamente. Este proceso tiene menos exposición hacia afuera (mediante la API) que la carga de datos, por lo que para evaluarse se deben introducir modificaciones en el código que permitan realizar las mediciones necesarias. Si bien aportaría valor conocer más a detalle esta parte del sistema, no se llevará a cabo esta parte del análisis en esta etapa.

Los resultados obtenidos de la prueba de carga verifican algunas afirmaciones que se habían establecido en la parte de investigación de Kafka. En primer lugar, se observa que debido a la estrategia *append-only* empleada por Kafka la cantidad de registros no tiene ningún impacto en los tiempos de escritura. Esto es evidente más que nada en las pruebas realizadas localmente, donde se introdujeron, entre todas las pruebas, más de 20 millones de registros sin provocar aumentos de latencia notables.

La separación de los casos de prueba entre aquellos donde se utiliza la API sincrónica del sistema y aquellos que usan la asincrónica permite verificar que el tiempo de escritura efectivo en Kafka es mínimo, incluso cuando se cuenta con múltiples escritores concurrentes. Esto se debe en parte a la manera en la que las librerías de Kafka optimizan estos escenarios mediante el *batching* de operaciones de I/O pequeñas. Si se comparan los resultados entre los casos sincrónicos y asincrónicos, tanto en las pruebas locales y remotas, se puede observar que no hay mayor latencia en los casos sincrónicos, como se esperaría naturalmente. Por ejemplo, en los casos con 500 usuarios concurrentes utilizando el servidor remoto, la latencia promedio en el caso sincrónico resultó incluso mayor que en el asincrónico (1101 a 1052 ms). Esto resulta aún más notable cuando se tiene en cuenta que el servidor remoto está configurado para replicar los registros en otro tópico. A priori es esperable que esto afecte los tiempos de respuesta de la API sincrónica, pero no parece ser este el caso guiándose por los resultados.

Una de las decisiones que podría criticarse a la arquitectura del sistema es la de abstraer el acceso al tópico de Kafka a los clientes detrás de un servidor web. Si bien la ganancia en cuanto a simplicidad del sistema es evidente¹⁷, resulta necesario determinar el impacto de esta capa en los tiempos de escritura. El programa de prueba ejecutado aparte de los planes de JMeter se comunica directamente con el cluster de Kafka, y comparando los tiempos de respuesta con los resultados obtenidos mediante JMeter se puede apreciar la latencia introducida por el servidor web. Comparando la media de 183 ms de latencia calculada por el programa con los 208 ms calculados en el caso **RS1**, se puede estimar que el servidor efectivamente introduce cierta latencia en la escritura¹⁸. Nuevamente, la decisión en este aspecto se reduce a determinar hasta qué punto la sencillez de tener un solo punto de entrada a la API del sistema justifica el ligero aumento en el tiempo de procesamiento.

Una última observación que es importante realizar es que resulta evidente que el principal limitante a la hora de trabajar con varios clientes en forma concurrente es la red.

¹⁷ Es común encontrar en cualquier sistema librerías que faciliten la comunicación mediante HTTP(S), mientras que la compatibilidad con las librerías de Kafka no se puede asegurar tan inmediatamente.

¹⁸ Sin embargo, si se utilizase la mediana del caso de JMeter como referencia se podría decir que no se introduce latencia alguna, ya que ambas pruebas indicarían 183 ms de tiempo de respuesta.

Observando como el throughput en la red local aumenta proporcionalmente con la cantidad de usuarios mientras que en el servidor remoto el throughput tiene un límite de alrededor de 450 respuestas por segundo independientemente de la cantidad de usuarios es suficiente para confirmar esto. Las limitaciones de la red también explican el porcentaje de errores en las pruebas, particularmente con el servidor remoto, donde el aumento de peticiones concurrentes resulta en que algunas de ellas terminan en timeout y por lo tanto son consideradas fallos por JMeter.

Las pruebas realizadas permiten en cierta medida evaluar el costo en cuanto a rendimiento de las distintas abstracciones y propiedades del sistema. El rendimiento que se considera aceptable depende del área de aplicación y de los recursos disponibles, pero para el alcance del presente prototipo, se considera que un throughput de 450 escrituras por segundo es un indicador aceptable, teniendo en cuenta que se realizó con hardware modesto y donde la red es el mayor limitante.

De todos modos, el sistema está diseñado con la escalabilidad en mente desde el comienzo, y lograr mejor rendimiento es tan sencillo como añadir más nodos al cluster de Kafka. Esto es una tarea casi trivial, más aún en la actualidad donde provisionar máquinas es un proceso tercerizable con costos adaptables a las exigencias del sistema.

Conclusiones

Si bien concluimos el desarrollo del proyecto, nos resulta imperativo mencionar que el problema que hemos tratado a lo largo de los últimos 16 meses no ha sido de carácter sencillo, y ha requerido amplios conocimientos del marco teórico que hemos presentado. Siendo que gran parte de este contexto no lo poseíamos, el proyecto presentó un arduo esfuerzo de investigación, ya sea mediante la consulta de bibliografía y profesionales de la industria, como de prueba y corrección de soluciones generadas por nosotros mismos. El resultado de esto puede verse en lo extensa que resulta la sección de marco teórico.

A su vez, consideramos de suma importancia que este tipo de aplicaciones estén siendo utilizadas en nuestros ambientes laborales, dado que sin ese acercamiento la referencia teórica y práctica hubiese sido mucho más complicada de obtener. La problemática requiere de gran manejo de las herramientas disponibles con el fin de asegurar que los requerimientos no funcionales que limitaron y guiaron la arquitectura se cumplieran.

Por otro lado, la realización del proyecto nos ayudó a comprender que la magnitud de problemas que Big Data e IoT pueden solucionar se encuentra muy lejos de su límite, existiendo amplio potencial en el uso de este tipo de técnicas para mejorar la calidad de vida de la población mundial. Por eso mismo, no podemos negar que el desarrollo que realizamos se encuentra acotado, dado que sería imposible explotar mucho más allá las capacidades de una tecnología en el plazo de tiempo pertinente que implica un proyecto final de Ingeniería. Si bien nos hemos limitado a un único campo de aplicación (silos de almacenamiento de granos), lo hemos tratado brevemente y sin mucha extensión con el fin de generar una implementación extensible y documentada que puede ser aplicada a casi cualquier contexto, requiriendo poco esfuerzo en la modificación del código.

Nos parece destacable que, a pesar del tiempo y esfuerzo que hemos consumido en la realización del proyecto, jamás perdimos la motivación para llegar a la finalización del mismo. Fue determinante el deseo de seguir aprendiendo y aplicando técnicas que, hasta el momento, resultaban relativamente desconocidas para nosotros, con el fin de lograr un mejor

entendimiento del mundo que nos rodea y de cómo podemos transformarlo con nuestras decisiones sobre el manejo de información.

Hemos determinado que los resultados obtenidos mediante la implementación de la solución planteada en este proyecto serán de carácter público, distribuidos mediante una licencia GPLv3.0 en la cual cualquiera puede contribuir y utilizar el código, siempre y cuando mantenga su naturaleza de software libre. De esta manera, seguiremos involucrados en el mantenimiento del mismo, y los resultados obtenidos definitivamente nos servirán para progresar en nuestras carreras profesionales.

Bibliografía

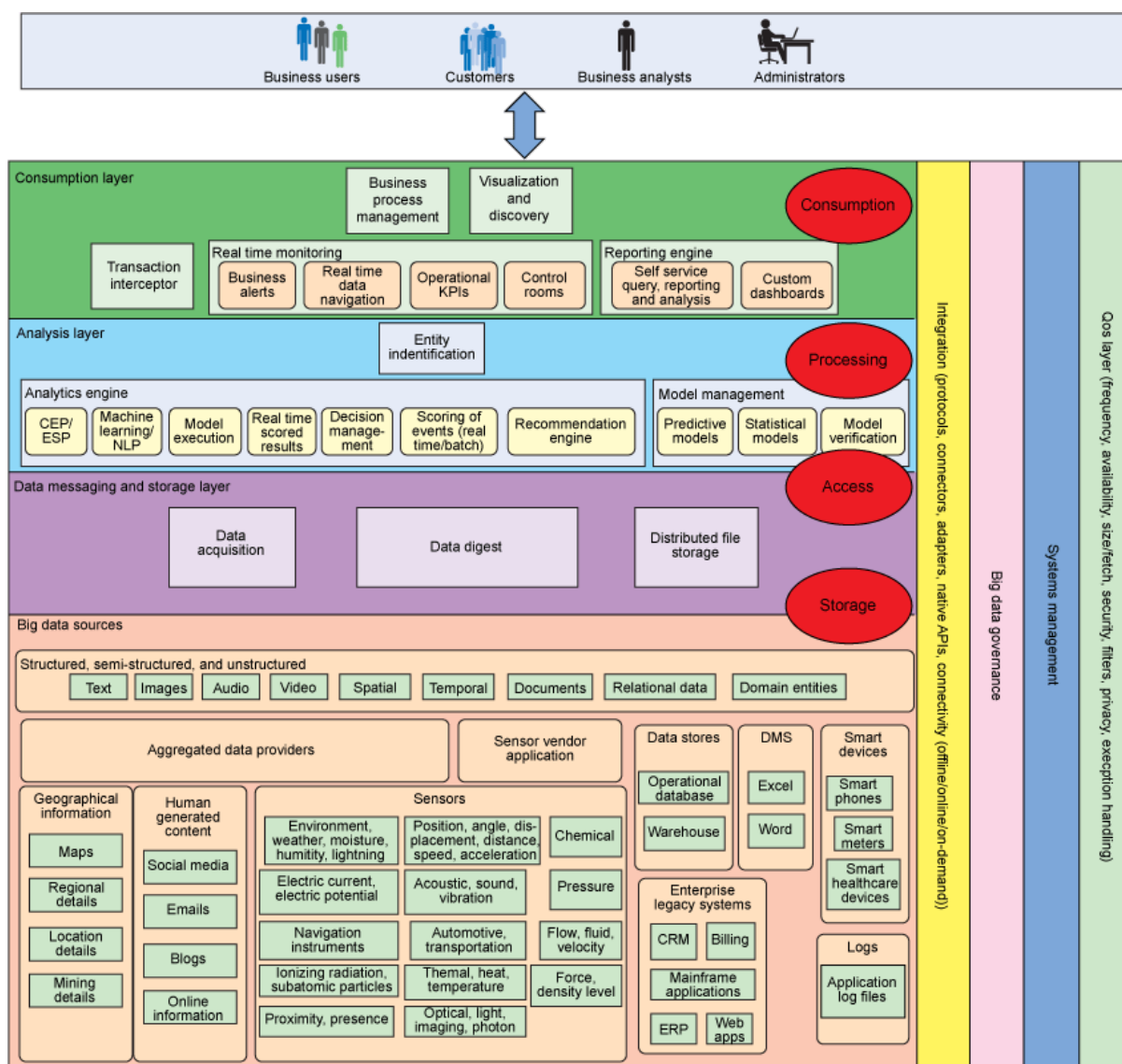
- Marz, N., & Warren, J.. *Big Data Scalable Real-Time Systems*. New York: Manning Publications Co., 2015.
- Greiner, R.. CAP Theorem: Revisited [en línea]. 2015 [fecha de consulta: 2 de enero de 2017]. Disponible en: <<http://robertgreiner.com/2014/08/cap-theorem-revisited>>
- Kreps, J.. *Questioning the Lambda Architecture* [en línea]. 2014 [fecha de consulta: 2 de enero de 2017]. Disponible en:
<<https://www.oreilly.com/ideas/questioning-the-lambda-architecture>>
- Marz, N.. How to beat the CAP theorem [en línea]. 2011 [fecha de consulta: 2 de enero de 2017]. Disponible en:
<<http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html>>
- Mysore, D., Khupat, S., & Jain, S.. Understanding atomic and composite patterns for big data solutions, Big data architecture and patterns [en línea]. 2013 [fecha de consulta: 8 de enero de 2017]. Disponible en:
<<https://www.ibm.com/developerworks/library/bd-archpatterns4/index.html>>
- Kreps, J.. The Log: What every software engineer should know about real-time data's unifying abstraction [en línea]. 2013 [fecha de consulta: 10 de enero de 2017]. Disponible en:
<<https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>>
- Castro Fernandez, R., Pietzuch, P., Kreps, J.. Liquid: Unifying Nearline and Offline Big Data Integration. Sin especificar [fecha de consulta: 15 de marzo de 2017]. Disponible en: <http://cidrdb.org/cidr2015/Papers/CIDR15_Paper25u.pdf>
- Hannelore, M., Karzel, D., Tuan-si, T.. *A Reference Architecture for the Internet of Things*. 2016 [fecha de consulta: 15 de marzo de 2017]. Disponible en:
<<https://www.infoq.com/articles/internet-of-things-reference-architecture>>

-
- Apache Software Foundation. Apache Kafka, A distributed streaming platform [en línea]. 2017 [fecha de consulta: 1 de abril de 2017]. Disponible en:
<<https://kafka.apache.org/documentation/>>
 - Nettsträter, A.. Internet of Things Architecture, Updated reference model for IoT v1.5 [en línea]. 2017 [fecha de consulta: 1 de abril de 2017]. Disponible en:
<http://www.meet-iot.eu/deliverables-IOTA/D1_3.pdf>
 - Pivotal Software, Inc.. RabbitMQ Documentation [en línea]. Sin especificar [fecha de consulta: 3 de abril de 2017]. Disponible en:
<<http://www.rabbitmq.com/documentation.html>>
 - Kreps, J.. Introducing Kafka Streams: Stream Processing Made Simple [en línea]. 2016 [fecha de consulta: 8 de abril de 2017]. Disponible en:
<<https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/>>
 - Confluent, Inc.. Kafka Streams Quick Start v4.0.0 [en línea]. 2017 [fecha de consulta: 10 de mayo de 2017]. <<http://docs.confluent.io/current/streams/index.html>>
 - Roar Media Pte Ltd.. How Big Data can help rice production [en línea]. 2017 [fecha de consulta: 5 de julio de 2017]. Disponible en:
<<https://roar.media/english/tech/insights/how-big-data-can-help-rice-production/>>
 - El Cronista. Tecnología y agro: el Big Data, la clave para ganar tiempo y dinero [en línea]. 2017 [fecha de consulta: 19 de septiembre de 2017]. Disponible en:
<<https://www.cronista.com/negocios/El-Big-Data-al-servicio-de-mayores-rindes-en-el-agro-20170907-0063.html>>
 - Arte Gráfico Editorial Argentino, Clarín digital. Agricultura: el “Big Data”, clave para ajustar los pronósticos y ahorrar mucha plata [en línea]. 2017 [fecha de consulta: 19 de septiembre de 2017]. Disponible en:
<https://www.clarin.com/rural/agricultura/big-data-clave-ajustar-pronosticos-ahorrar-mucha-plata_0_BkN18HcvW.html>

-
- Canadian Grain Commission. Monitoring grain temperature and aerating grain [en línea]. 2013 [fecha de consulta: 13 de octubre de 2017]. Disponible en: <https://www.grainscanada.gc.ca/storage-entrepot/mta-stv-eng.htm>
 - Kreps, J.. Benchmarking Apache Kafka: 2 million writes per second (on three cheap machines) [en línea]. 2014 [fecha de consulta: 14 de octubre de 2017]. Disponible en : <https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>
 - Brown, B.. 9 ways Big Data and Analytics are changing the world [en línea]. 2014 [fecha de consulta: 10 de noviembre de 2017]. Disponible en: <https://www.targit.com/en/blog/2014/11/9-ways-big-data-analytics-changing-the-world>

Anexo

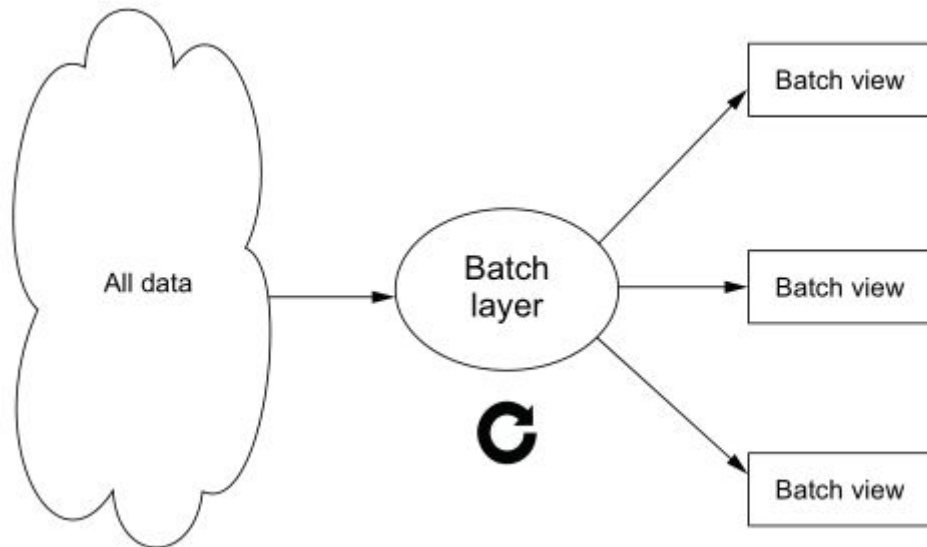
1) Modelo de capas arquitectónicas (IBM)



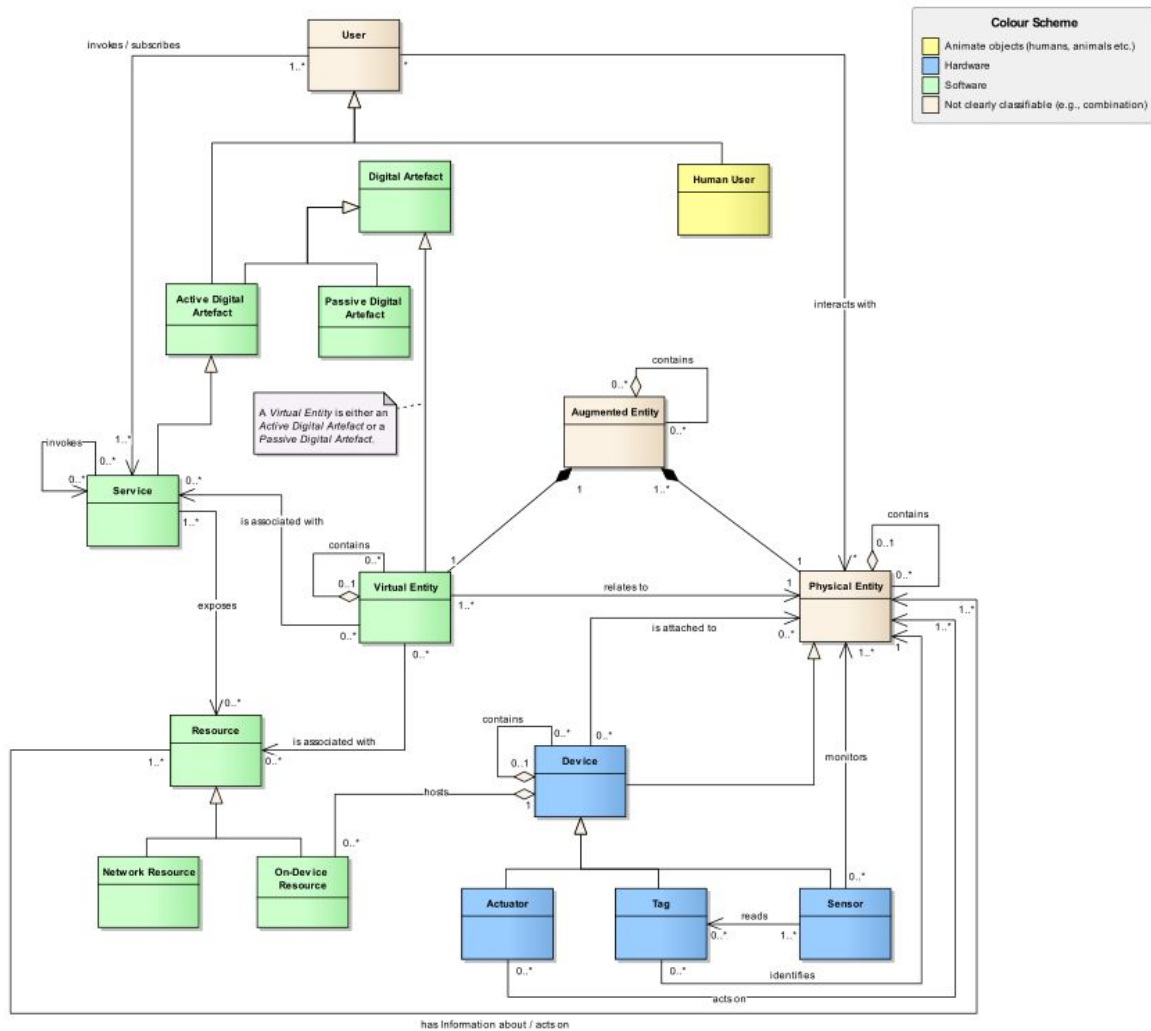
2) Mapeo de patrones atómicos a compuestos (IBM)

Atomic Patterns →		Consumption patterns				Processing patterns				Access Patterns			Storage Patterns				
Composite Patterns ↓		Visualization	Ad-hoc discover	Augment traditional data stores	Notifications	Initiate Automated Response	Historical data analysis	Advanced analytics	Pre-process raw data	Ad hoc analysis	Web and Social media	Device	Transactional, Operational and warehouse data	Distributed-Unstructured	Distributed-Structured	Traditional data stores	Cloud
Store & explore																	
Data source	Existing																
	New										✓	✓	✓				
Usage	Storage only													✓	✓	✓	✓
	Processing & consumption	✓	✓	✓			✓		✓	✓							
Purposeful and predictive analysis																	
Augmentation target	Human	✓	✓		✓												
	System			✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	
Processing	Standard						✓		✓	✓	✓	✓	✓	✓	✓		✓
	Predictive							✓	✓	✓	✓	✓	✓	✓	✓		✓
Timing	Decisioning							✓	✓	✓	✓	✓	✓	✓	✓		✓
	Batch			✓		✓	✓	✓	✓		✓	✓	✓	✓	✓		✓
	Real time	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓
Actionable analysis																	
Action	Full automated			✓	✓	✓	✓	✓			✓	✓	✓	✓	✓		✓
	Partial automation	✓		✓	✓		✓	✓			✓	✓	✓	✓	✓		✓
	Full manual	✓		✓	✓		✓	✓			✓	✓	✓	✓	✓		✓

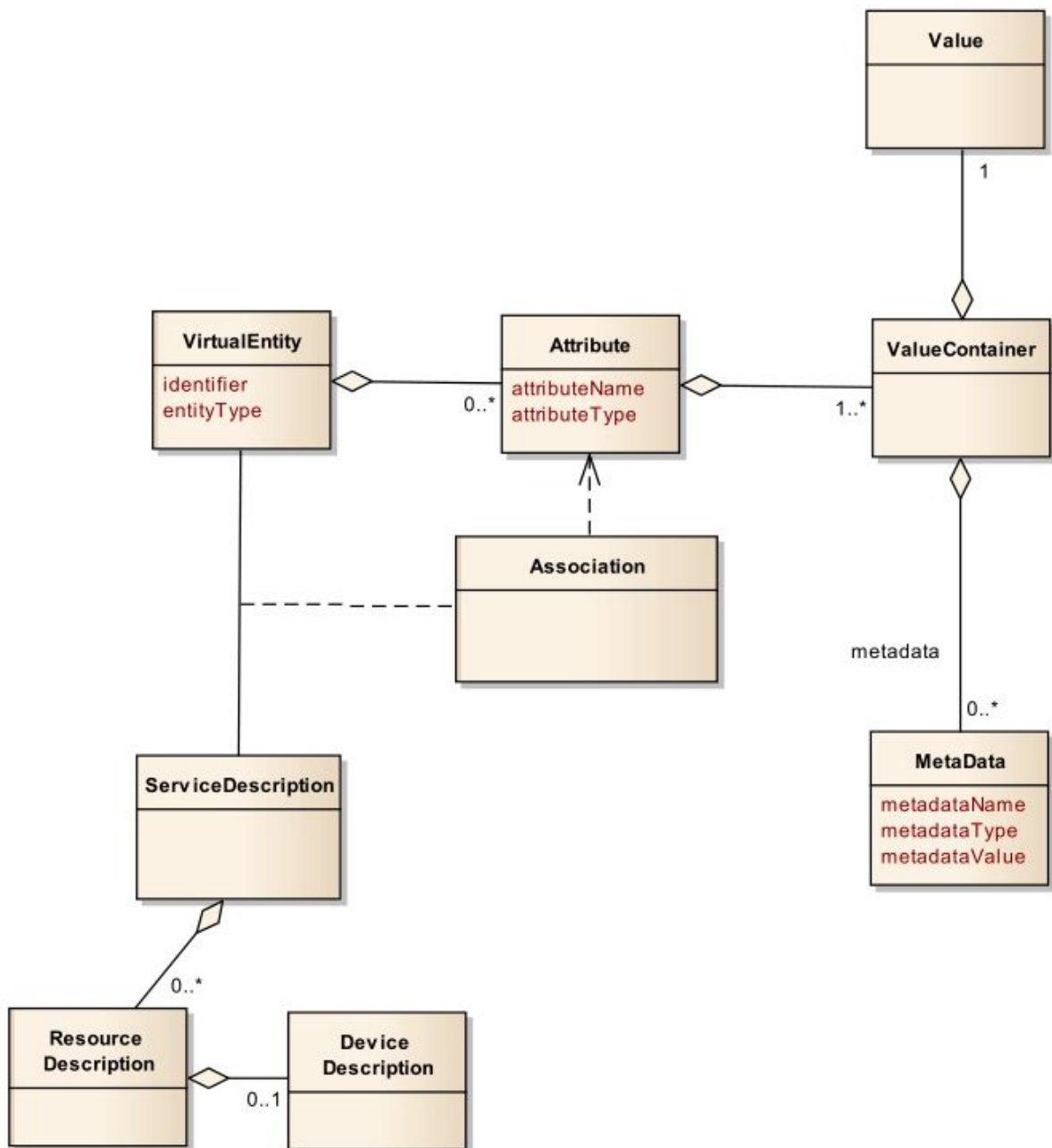
3) Arquitectura de la capa batch (LA)



4) IoT-A: Modelo de Dominio



5) IoT-A: Modelo de Información



6) Especificaciones técnicas de Arduino UNO REV 3

Microcontroller	ATmega328P
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz
LED_BUILTIN	13
Length	68.6 mm
Width	53.4 mm
Weight	25 g

7) Programa de lectura análoga de una entrada en Arduino

```
void setup() {  
  Serial.begin(9600);  
}  
  
void loop() {  
  Serial.write(analogRead(0));  
  delay(1000);  
}
```

8) Agregación por ventana de tiempo con Kafka Streams

```
final SensorAggregationKeySerde aggregationKeySerde = new SensorAggregationKeySerde();  
final SensorMetricSerde metricSerde = new SensorMetricSerde();  
final SensorRecordSerde recordSerde = new SensorRecordSerde();  
  
KStreamBuilder builder = new KStreamBuilder();  
KStream<String, SensorRecord> input = builder.stream(Constants.SENSOR_RECORDS_TOPIC);  
  
KGroupedStream<SensorAggregationKey, SensorRecord> grouped = input.groupBy(  
  (key, value) -> new SensorAggregationKey(value.getOwnerId(), value.getType()),  
  aggregationKeySerde,  
  recordSerde);  
  
KTable<Windowed<SensorAggregationKey>, SensorMetric> aggregation = grouped.aggregate(  
  SensorMetric::new,  
  (key, value, metric) -> metric.update(value),  
  TimeWindows.of(windowSize),  
  metricSerde,  
  Constants.SENSOR_AGGREGATIONS_STORE);
```

9) Configuración de Kafka (sólo valores personalizados)

broker.id=1

log.dirs=/tmp/kafka-logs-1

num.partitions=2

listeners=PLAINTEXT://localhost:9092

default.replication.factor=2

zookeeper.connect=localhost:2181