
Gestores de recursos

Manual

PID_00284065

Iraitz Montalbán Pontesta

Tiempo mínimo de dedicación recomendado: 3 horas



Iraitz Montalbán Pontesta

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

Primera edición: septiembre 2021
© de esta edición, Fundació Universitat Oberta de Catalunya (FUOC)
Av. Tibidabo, 39-43, 08035 Barcelona
Autoria: Iraitz Montalbán Pontesta
Producción: FUOC
Todos los derechos reservados

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea este eléctrico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita del titular de los derechos.

Índice

Introducción.....	5
1. Componentes principales de un gestor de recursos.....	7
1.1. Unidades de ejecución	7
1.1.1. Contenedores Docker	8
1.2. Colas de recursos	9
1.3. Planificador	10
1.3.1. <i>Single queue scheduling</i>	11
1.3.2. <i>Capacity scheduling</i>	11
1.3.3. <i>FAIR scheduling</i>	12
1.4. Infraestructura	13
1.5. <i>Framework</i> de procesado	15
2. Apache YARN.....	17
2.1. Fases de un proceso en YARN	18
2.2. Uso de YARN	19
2.3. Docker en YARN	22
3. Apache Mesos.....	24
3.1. Roles	27
3.2. Aislamiento	27
3.3. Uso de Mesos	28
4. Otras plataformas.....	30
4.1. Apache Myriad	30
4.2. Kubernetes	31
Resumen.....	33
Bibliografía.....	35

Introducción

Los equipos y sistemas informáticos que empleamos habitualmente se componen de unidades destinadas al almacenamiento y procesado de información. Los programas que en estos equipos se ejecutan requieren y comparten el uso de estos recursos finitos. Cuando hablamos de recursos nos referimos a estos y los gestores son los encargados de que los programas hagan un uso racional y equitativo de los mismos.

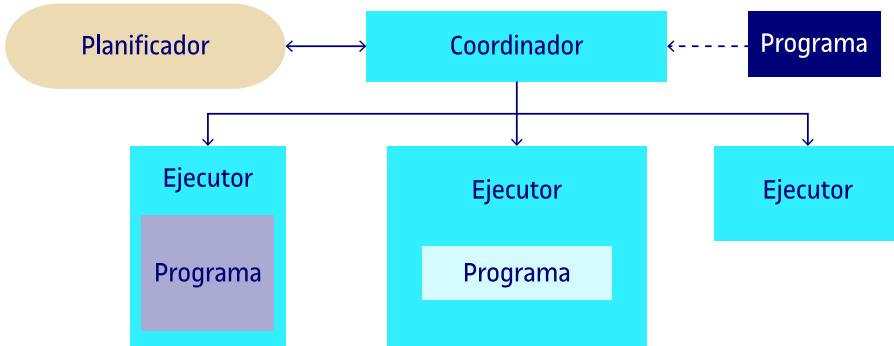
Los sistemas operativos de vuestros ordenadores, por ejemplo, se encargan de saber cuántos recursos físicos (memoria RAM, CPU disponible o almacenamiento de disco duro) restan. Del mismo modo, también deben desempeñar el papel de saber cómo estos han de ser repartidos a discreción para las tareas requeridas por los programas que se ejecutan en el equipo, dado que no todos podrán ejecutarse a la vez (al menos no habitualmente).

Este escenario se complica cuando hablamos de centros de datos y de clústeres de máquinas, ya que encontraremos tanto servicios como programas de usuario final, cada uno con sus necesidades computacionales y prioridades, y además se pueden ejecutar de forma distribuida, en distinto momento y/o ubicación física. Este hecho eleva enormemente la complejidad de la coordinación y gestión de procesos.

Dado que los recursos son finitos y múltiples los programas que requieren usarlos, acciones como las de priorizar las tareas, retener las que no dispongan de suficientes recursos en un momento dado, o incluso decomisionar programas con altas tasas de fallo son cuestiones clave para un uso eficiente de la infraestructura subyacente.

Esta delicada tarea de mediar entre las necesidades de los procesos y los recursos disponibles en las máquinas es la encomendada a los **programas gestores de recursos**. Podemos ver un esquema básico de un gestor de recursos en la figura 1, en el que los programas son gestionados por el gestor que planifica y lanza a ejecutar los programas a su debido tiempo, garantizando, como hemos dicho, un uso óptimo de los recursos disponibles.

Figura 1. Arquitectura tipo de un gestor de recursos



En resumen, los gestores de recursos son el vínculo que une el software a las máquinas y hacen que toda esta maquinaria trabaje de forma coordinada, como veremos más adelante.

1. Componentes principales de un gestor de recursos

A continuación detallaremos según lo anteriormente descrito qué piezas base, y comunes en todos los gestores, veremos implementadas de forma habitual, aunque seguramente de forma algo distinta en cada caso.

1.1. Unidades de ejecución

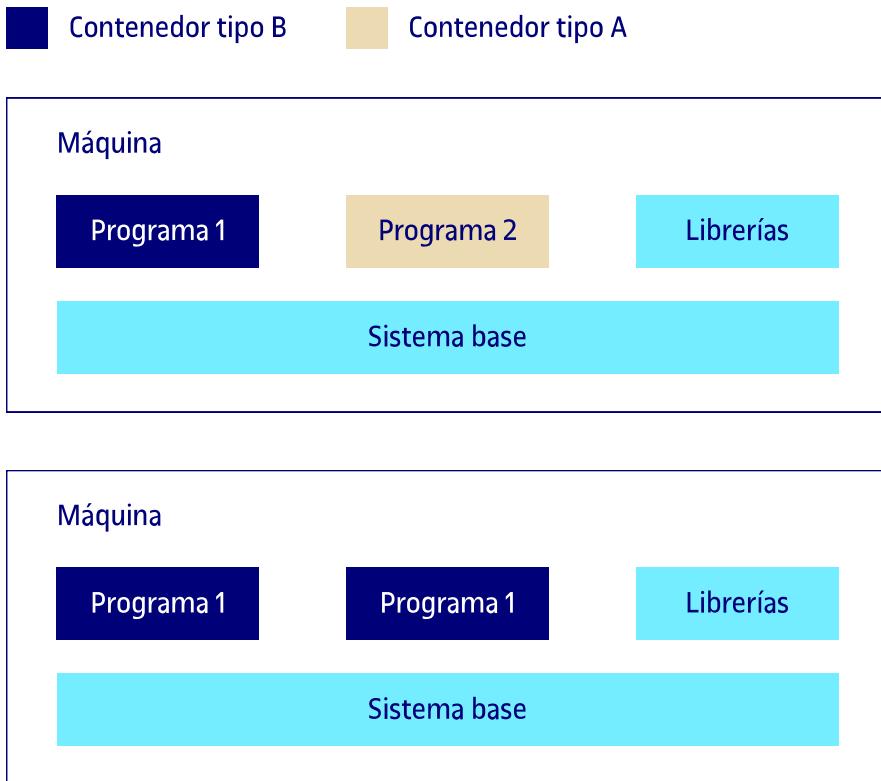
Llamaremos *unidades de ejecución base* o *contenedores* (de forma habitual en este contexto) a las abstracciones lógicas que permiten tratar aquello que estamos gestionando sin conocer su naturaleza. Podemos considerarlas como las unidades mínimas que serán desplegadas para poder contener las tareas que componen un programa o servicio.

Es como si introdujéramos las partes de nuestro programa en cajas, de forma que podemos llevarlas a cualquier punto sin conocer qué parte estamos transportando. Este punto es muy útil cuando trabajamos en sistemas horizontalmente escalables, como los sistemas para el procesado de datos *big data*.

Estas unidades son la base que emplean los gestores de recursos y son el objeto al que poder asociar los recursos disponibles en el sistema destino (CPU, RAM y disco duro en la mayoría de los casos). En esencia el gestor dispondrá de un número de unidades, y las unidades de ejecución harán uso de un subconjunto de ellas. Estas unidades pueden ser desde hilos de proceso Java con una memoria fija asignada (como veremos en el caso de YARN) hasta sistemas virtualizadores más complejos.

Veremos que existen distintos niveles de abstracción o independencia del sistema base dependiendo de la implementación concreta de la que estemos hablando. En muchos casos, las tareas correspondientes a un programa distribuido se ejecutan en un sistema donde las librerías o dependencias para su ejecución son compartidas entre varios (como se muestra en la figura 2). El cometido principal es el de poder reservar y disponer de recursos de forma lo más uniforme posible.

Figura 2. Ejemplo de programas distribuidos en contenedores en varias máquinas



Ejemplo

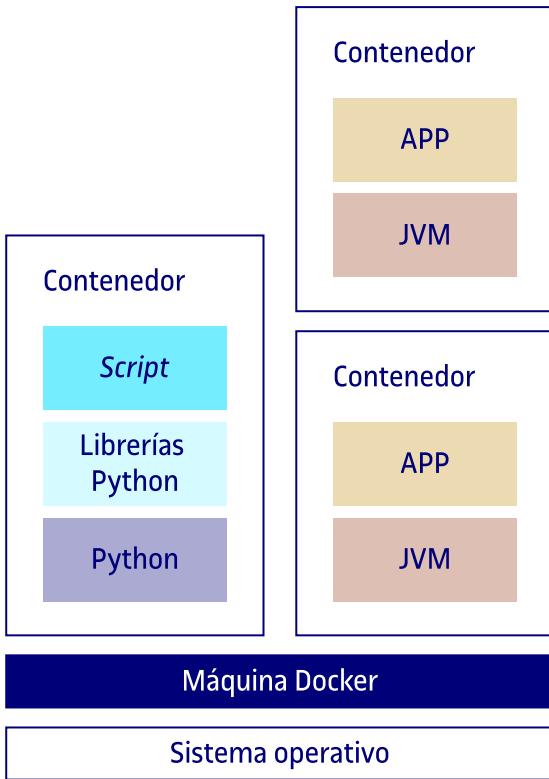
Podríamos imaginar un proceso que dependa de librerías Java específicas para leer ficheros (ficheros Parquet, por ejemplo) y que precise de 4 GB de memoria RAM. Para ellos es necesario que todas las unidades de ejecución cumplan estos requisitos, ya que si no, fallarían en su labor.

Para contextualizar el concepto de unidades de ejecución, resulta pertinente introducir el **proyecto Docker**. El proyecto Docker, al que haremos referencia en varias ocasiones a lo largo del módulo, supone actualmente un estándar en los gestores de recursos y es el principal motivo por el cual empleamos la palabra *contenedor* en estos (y otros) contextos.

1.1.1. Contenedores Docker

Docker nace, como muchas de las tecnologías que manejamos, de empresas de nueva generación con necesidades de mover aplicaciones y servicios a escalas nunca antes conocidas. El hecho de poder desplegar programas con las dependencias que estas puedan requerir puede derivar en tareas de mantenimiento elevadas cuando este mantenimiento lo realizamos de forma manual.

Figura 3. Ilustración de diversas aplicaciones en contenedores dentro de una máquina



Como se muestra en la figura 3, Docker nos permite encapsular nuestras aplicaciones en un contexto donde puedan convivir con otras aplicaciones de distinta naturaleza sin interferir unas con otras y conteniendo todo lo necesario para funcionar allá donde sean emplazadas. Esto permite aislar las dependencias que indicamos en el ejemplo del proceso Java como una unidad que contenga todos los requisitos y dependencias al margen del sistema operativo base, pudiendo así además correr el mismo proceso en distintos sistemas operativos sin modificación alguna.

Pese a ser un sistema más cercano a los mecanismos de virtualización como puedan ser las máquinas virtuales, su ligereza permite envolver cualquier aplicación con un mínimo sobrecoste maximizando la reproducibilidad y portabilidad de estas. De ahí su éxito en la actualidad.

1.2. Colas de recursos

Como ya hemos dicho, el mayor problema que debemos abordar es el uso compartido de unos recursos que se asumen finitos, con lo que directamente surge la pregunta: ¿cómo podemos repartirlos de forma ordenada, eficiente y justa?

Nota

Aquellos que queráis introducirnos en este mundo podréis descargaros la versión desktop (<https://www.docker.com/products/docker-desktop>) para multitud de sistemas e iniciarnos con los ejemplos que encontraréis en su página web (<https://www.docker.com/play-with-docker>).

Las colas son las abstracciones que permiten disponer de distintos niveles de priorización o acceso a recursos según la tipología o perfil del proceso demandante de estos.

Estas colas y distintos niveles deberán ser diseñados para habilitar la correcta convivencia de aplicaciones o servicios con distintos niveles de criticidad o naturaleza de ejecución.

Ejemplo

Programas críticos que deben ser ejecutados de inmediato. Estos deben disponer de recursos reservados solo destinados a su tipología particular de aplicaciones.

Figura 4. Ilustración de un sistema con dos colas de procesos



En la separación en dos colas indicada en la figura anterior, los recursos críticos solo serán empleados por aplicaciones críticas y no se comparten con otras aplicaciones de menor prioridad. En esta política se asume que es mucho peor que una aplicación crítica no disponga de recursos suficientes cuando los necesite que una aplicación de carácter general deba esperar a que se liberen los recursos asignados a su tipología.

1.3. Planificador

La unidad de planificación es la encargada de secuenciar y priorizar las tareas a ejecutar en la cola (o colas) de recursos disponibles.

Es una cuestión de organización poder extender el eje temporal de forma lo más eficiente posible a fin de que todo proceso encuentre cabida. En sistemas operativos clásicos existen gran cantidad de planificadores: FIFO, Earliest Deadline First, Round Robin, etc., aunque, debido a sus características particulares, no todos son aplicables al campo de los sistemas distribuidos y *big data*. En entornos distribuidos donde existen *frameworks* destinados a la separación de programas en tareas (tareas *map* y *reduce* en el caso de MapReduce, por ejemplo) requerimos que el gestor de recursos pueda planificar y controlar la ejecución y finalización de cada una de esas tareas en las que se desgranará nuestra aplicación o servicio.

Ved también

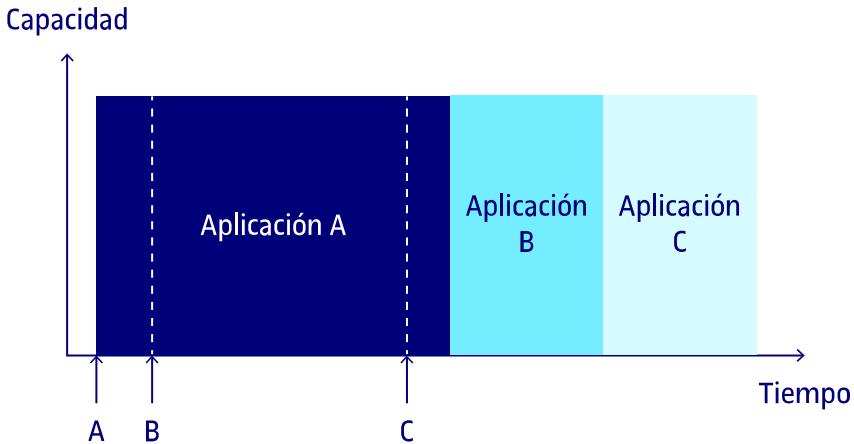
Encontraréis más información en la sección 2.4 de **Tanenbaum, A. S. (2003). Sistemas operativos modernos.** México D.F.: Pearson Educación, y en **White, T. (2015). Hadoop: The Definitive Guide.** O'Reilly Media Inc.

Deberemos establecer unas políticas de uso que son las que cumplirá nuestro planificador. Plantear la mejor estrategia dependiendo del caso concreto no es baladí y, de hecho, existen variedad de opciones como veremos a continuación.

1.3.1. *Single queue scheduling*

Es quizás la política de planificación más sencilla, dado que solo existe una cola de recursos y la antigüedad es la que establece qué aplicación será la siguiente en acceder a la cola de recursos pertinente.

Figura 5. Ilustración del ciclo de vida de tres aplicaciones



Las flechas indican el momento de entrada de la aplicación en el sistema. Como vemos, la aplicación B y C deberán esperar para ser ejecutadas.

Vemos en la figura 5 como según los recursos son liberados por la aplicación A, la asignación se hace a la aplicación con mayor antigüedad (la B en nuestro ejemplo de la figura 5), al margen de las prisas o necesidades de las aplicaciones a la espera.

En esta línea de políticas existen diversas variaciones relativas a la ordenación, dado que podemos igualmente optar por políticas que prioricen por antigüedad (FIFO, *First-In-First-Out*) o al más reciente (LIFO, *Last-In-First-Out*). A pesar de que son sencillas y pueden ser muy convenientes, en sistemas con alta concurrencia de tareas y disparidad de tiempos de ejecución (aplicaciones de larga y de corta duración) es preferible optar por otras políticas algo más complejas pero que ofrezcan unas mínimas garantías de ejecución o respuesta.

Para solventar estos problemas solemos dividir los recursos en varias colas y asignar perfiles o agrupaciones de programas que pueden hacer uso de una u otra cola dependiendo de la necesidad.

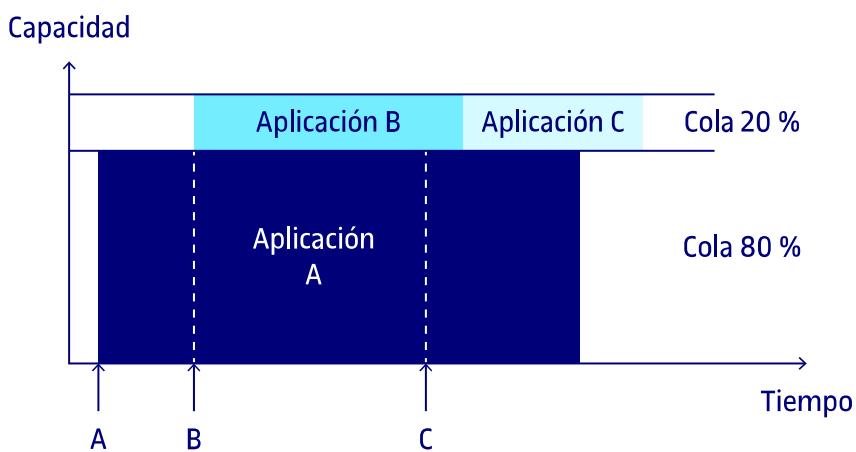
1.3.2. *Capacity scheduling*

Esta estrategia de planificación es muy útil cuando existe competición por los recursos compartidos por distintas unidades o equipos dentro de una misma organización. Tal y como se definía con anterioridad, la existencia de más de

una cola de capacidad permite separar distintos porcentajes de recursos ante distinta tipología de aplicaciones, de modo que la ejecución de multitud de ellas pueda convivir. La contraprestación con respecto a la política anterior es la de que a pesar de disponer de recursos libres, puede haber aplicaciones que empleen solo parte del clúster y por tanto su ejecución demore más en el tiempo a expensas de que otra aplicación pueda requerir esa reserva latente.

Podemos observar este hecho en la figura 6. En el hueco anterior a la ejecución de la aplicación B, existen recursos disponibles que en lugar de ser empleados por la aplicación A quedan a la espera de que una aplicación destinada a esa cola en concreto los demande.

Figura 6. Ilustración del ciclo de vida de tres aplicaciones en dos colas



Dentro de la propia capacidad asociada a la cola, otra subpolítica predominante puede considerarse, por ejemplo, según el tiempo de llegada.

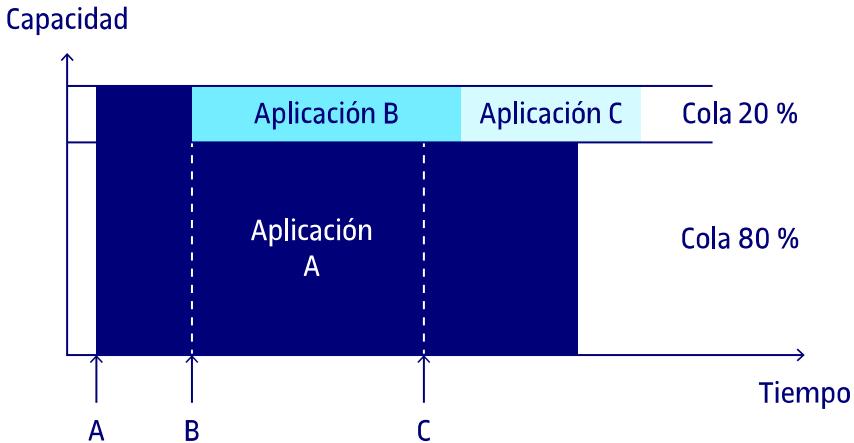
En organizaciones con suficiente complejidad suele ser habitual encontrarse con estructuras jerárquicas de recursos más enrevesadas aún. En estos casos, puede resultar complejo conseguir un nivel de equilibrio suficiente como para que el planificador pueda realizar una distribución justa entre los distintos procesos que deban ejecutarse.

1.3.3. FAIR scheduling

Esta última opción que veremos es popular por realizar un reparto justo de los recursos en promedio. Las aplicaciones reciben al menos un mínimo de recursos con los que iniciar sus procesos.

Permite exceder la capacidad por defecto asignada a una tipología de proceso según su cola de recursos asignada, siempre y cuando los recursos extra no sean reclamados por otras aplicaciones. En el ejemplo de la figura 7, la aplicación A toma recursos asignados a la cola B hasta que la aplicación B es iniciada y demanda estos recursos.

Figura 7. Ilustración del ciclo de vida de tres aplicaciones en dos colas



En contextos en los que las aplicaciones se ejecutan en tiempos previamente determinados y la concurrencia de aplicaciones es alta, esta estrategia suele ofrecer un buen balance entre el aprovechamiento de los sistemas y el rendimiento de las aplicaciones. Por ejemplo, los horarios de oficina en los que los empleados pueden generar muchos procesos puntuales en competencia de recursos.

Este planificador parte de la premisa de poder revocar un recurso ya asignado a una aplicación en ejecución, con lo que puede generar errores en la ejecución habitual de un programa. Por ello es requisito indispensable que sea empleado en contextos en los que la tolerancia a fallos en la ejecución de los programas sea una característica asumible. Es decir, que el *framework* sobre el que las aplicaciones son montadas pueda gestionar los fallos en tareas mediante reintentos.

1.4. Infraestructura

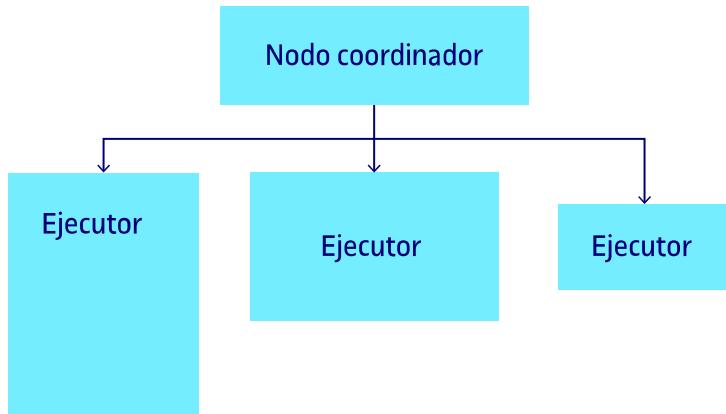
Entrando en materia de infraestructura, una de las arquitecturas más comunes en sistemas distribuidos es la denominada maestro-trabajador (*master-slave*).

Se trata de disponer de máquinas con distintos roles donde se establece una jerarquía: el rol de coordinador del servicio y el de ejecutores del servicio.

Esta arquitectura es común en multitud de servicios tales como el HDFS de Apache Hadoop o Apache HBase, entre otros. Si tomamos como ejemplo el HDFS, el *namenode* es el nodo responsable de la gestión de los ficheros, mientras que los *datanodes* se encargan de almacenar y servir los bloques de datos que componen un fichero en Hadoop.

Los gestores de recursos distribuidos presentan una arquitectura similar y suelen dividirse en **roles** de forma que existan **procesos o nodos maestros** encargados de las tareas administrativas o de coordinación; y **nodos trabajadores** encargados de hacer el trabajo efectivo, es decir, reservar el espacio y los recursos para poder ejecutar las tareas indicadas por el nodo maestro. Gracias a la coordinación de ambos roles, estos sistemas permiten ofrecer escalabilidad horizontal gracias a la adición de nuevas máquinas trabajadoras (*slave*) mientras el rol del nodo coordinador o maestro se encarga de orquestar y coordinar la actividad de todos los nodos ejecutores de forma que el sistema funcione como un todo.

Figura 8. Ilustración jerárquica entre los distintos roles de los nodos



Un programa como tal se encarga de realizar una serie de acciones con un fin, pero cuando se trata de aplicaciones que vayan a ejecutarse en entornos distribuidos hay otras consideraciones que tener en cuenta.

Los programas también deben ser capaces de coordinar sus tareas pendientes y negociar con el nodo coordinador dónde pueden ser ejecutadas estas. En esencia, se trata de negociar los siguientes pasos y recursos necesarios para realizarlos.

Los nodos ejecutores son los encargados de realizar en última instancia la tarea encomendada y por ello es relevante que existan condiciones mínimas de coherencia entre los mismos, dado que una disparidad muy elevada entre la tipología de recursos o condiciones base relativas a librerías o versionado del software puede generar inconsistencias que nos obligarán a realizar tareas de mantenimiento de forma regular.

Para facilitar la tarea de coordinación entre las acciones definidas en un programa y la gestión de recursos existen los conocidos **frameworks de procesado**.

1.5. Framework de procesado

A pesar de no ser una pieza incluida en los gestores, está estrechamente ligado a estas plataformas, dado que supone la contrapartida en la negociación que deben realizar los programas.

Es decir, si el gestor de recursos es un mecanismo al que pedir los recursos, el *framework* es el que viste el programa para poder demandar dichos recursos.

Son los mecanismos de coordinación a nivel de programa o aplicación. Cada programa dentro de estos *frameworks* dispondrá de un proceso coordinador de tareas y planificador propio (maestro-esclavo a su vez) para gestionar la ejecución de todo el grafo del proceso asociado a un programa dado. Más adelante lo veremos, pero reflexionemos sobre el *framework* MapReduce, si solo existen funciones *Map* y funciones *Reduce*, ¿quién coordina cuál debe ejecutarse cada vez?

Ejemplo

Supongamos que nos disponemos a lanzar un programa que cuente las palabras de un fichero. El programa en sí se encargará de leer las palabras y asociar un sumatorio según cada ocurrencia. Cuando nos dispongamos a lanzar este programa en un entorno distribuido como es un entorno Hadoop, habrá varias máquinas donde existan partes de nuestro fichero objetivo a leer. Una vez leída la información, esta debe ser agregada en una segunda etapa.

Precisamente el *framework* MapReduce se encarga de negociar con el gestor de recursos dónde ubicar las tareas, controlar cuántas tareas restan, y de ser necesario en caso de fallo de las tareas renegociar otra ubicación donde poder reintentar dicha tarea.

Pensemos que dependiendo del gestor de recursos que empleemos, la forma o el tiempo en el que estos son provistos puede variar e incluso hacerse insuficientes para la tarea que resta por parte de la aplicación. En estos casos la aplicación sería rechazada y quedaría a la espera de una nueva oferta por parte del gestor. Del mismo modo que un gestor eficiente de recursos permite explotar al máximo la capacidad computacional disponible, la aplicación por su parte prima la ejecución basándose en los términos necesarios (suficiente memoria o unidades de proceso, por ejemplo). Por lo tanto, dado que existen por ambas partes unos requisitos mínimos, podemos inferir que ni todo *framework* tiene cabida en cualquier gestor de recursos, ni cualquier gestor de recursos puede albergar cualquier *framework*. Deben de existir unos mecanismos comunes mediante los cuales puedan llevar a cabo esas negociaciones.

El *framework* es, por decirlo de algún modo, el marco de trabajo que establece las bases por las que ambos sistemas podrán entenderse. De este modo encontramos *frameworks* específicos como MapReduce que requieren de gestores concretos al otro extremo para poder ser ejecutados, siendo estos ecosis-

⁽¹⁾Ved «Qué es Apache Spark?». Microsoft Docs. Disponible en <https://docs.microsoft.com/es-es/dotnet/spark/what-is-spark>

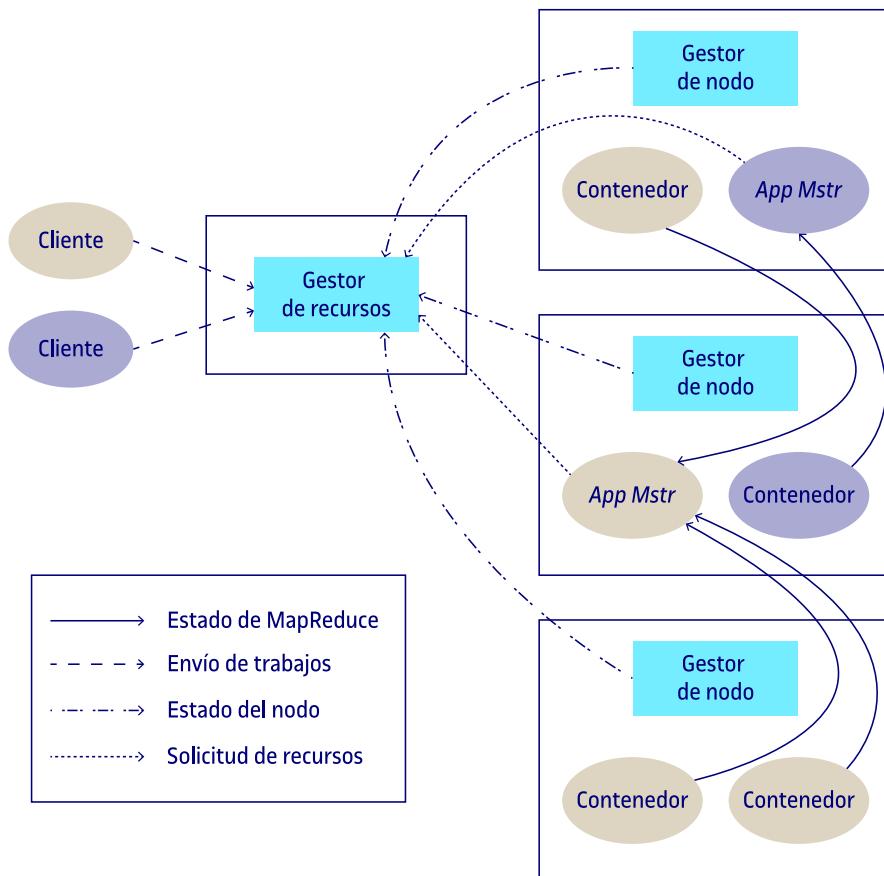
temas habituales en los entornos Hadoop (aunque no los únicos). Dentro de los *frameworks* más comunes en la actualidad encontraremos Apache Spark,¹ que además de ser un *framework* de computación distribuida donde prima la ejecución en memoria, es compatible con multitud de gestores de recursos.

Una vez vistas las piezas clave que componen un gestor de recursos y el contexto de los programas que en estos sistemas encuentran cabida, a continuación pasaremos a ver dos ejemplos concretos que son de uso habitual en las infraestructuras corporativas y mencionaremos también la evolución que han sufrido estas plataformas tecnológicas en los últimos años.

2. Apache YARN

Apache YARN (*Yet Another Resource Negotiator*) nace en el seno del proyecto Apache Hadoop. Fue incluido en la versión 2.x como una capa de abstracción entre el sistema de almacenamiento propio (*Hadoop Distributed File System*) y su capa de procesamiento distribuido basada en MapReduce.

Figura 9. Arquitectura de Apache Yarn



Como podemos ver en la imagen, YARN dispone de un nodo gestor en cada máquina dentro del clúster Hadoop. Los llamados **node manager** son los trabajadores encargados de atender los mandatos del **nodo central o gestor de recursos** (*resource manager*). Este último es el nodo coordinador, el que negocia con las aplicaciones cuándo y dónde pueden ejecutar sus tareas pendientes.

Los *frameworks* destinados a ejecución distribuida, como son Apache MapReduce o Apache Spark, despliegan un proceso principal (llamados *application master* o *driver*) destinado a negociar con el gestor de recursos la ubicación de sus tareas según la aplicación vaya avanzando en su ejecución. Esto resulta especialmente relevante cuando nuestros sistemas conviven con un sistema

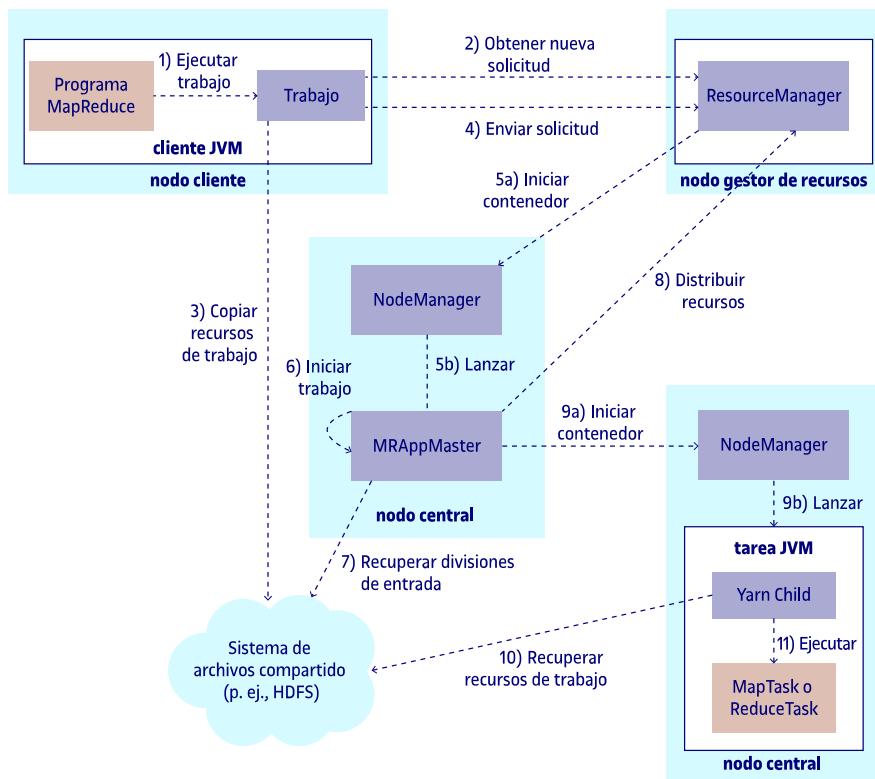
de datos distribuido como es el caso del sistema de ficheros distribuido de Hadoop. Cuando las tareas requieren un conjunto de datos concreto, pueden beneficiarse de ser ubicadas allá donde los datos residan, minimizando el movimiento de estos.

YARN solo es capaz de gestionar un clúster Hadoop, un conjunto de máquinas en concreto. Veremos más adelante otros gestores de carácter general que pueden ser empleados para controlar distintos grupos de *framework*.

2.1. Fases de un proceso en YARN

En un ciclo de vida tipo, como en el ejemplo mostrado a continuación, las aplicaciones son lanzadas demandando un contenedor en primera instancia. Este primer hilo de ejecución será el encargado de lanzar el proceso maestro del *framework* que se emplee en la aplicación y, a partir de este, iniciar las negociaciones con el gestor para disponer de más recursos.

Figura 10. Ciclo de vida de una aplicación M/R en YARN



En la figura arriba indicada podemos ver como una aplicación solicitante (paso 1) pasará a ser una aplicación NEW en primera instancia. Se le pedirá un identificador de aplicación al gestor de recursos (paso 2) y se solicitará información sobre la aplicación (tipología, *framework*, etc.). Esta información será registrada como parte del trabajo a realizar en el paso 3 y pasando su estado a SUBMITTED en el paso 4.

El gestor de recursos consultará al planificador sobre los recursos disponibles y la cola de recursos sobre la que la aplicación deberá ser ejecutada. Si existen recursos suficientes, la aplicación pasará al estado **ACCEPTED** y finalmente, una vez lanzado el contenedor inicial (paso 5), pasará a estar **RUNNING**, el proceso maestro del *framework* (MRAppMaster en la figura 10) ya está creado e iniciado.

A partir de este punto, el *framework* en cuestión de la aplicación (mediante el *application master*) se encarga de conocer dónde están ubicados los bloques de información, acorde a lo indicado por el HDFS (*Hadoop Distributed Files System*) en el paso 7, de modo que pueda pedir un número de contenedores idéntico y en las mismas ubicaciones donde se encuentre la información en el paso 8. Así, los *mappers* o tareas de lectura del *framework* MapReduce son ubicados en los nodos trabajadores.

Finalmente, cuando la aplicación haya finalizado, informará al gestor de recursos de YARN para decomisar los contenedores restantes y pasar a **FINISHED** disponiendo en el histórico de aplicaciones toda la información relevante para su posterior consulta.

Evidentemente, en caso de fallo o petición de finalización anticipada también serán registrados estos escenarios, **FAILED** y **KILLED**, respectivamente.

2.2. Uso de YARN

El gestor de recursos habitualmente es empleado de forma transparente para los usuarios que interactúan mediante otras aplicaciones.

Ejemplo

En el contexto de Hadoop, cuando lanzamos una consulta Hive, esta es convertida, de forma transparente al usuario, a un programa MapReduce y ejecutada contra el gestor de recursos. Hive se encarga de realizar las interacciones necesarias y respondernos con el resultado de nuestra consulta.

Sin embargo, es interesante conocer cómo poder interactuar a bajo nivel con el gestor de recursos en caso de requerir lanzar nuestros propios programas. Estos programas suelen ser habitualmente aplicaciones Java o Scala que emplean los *frameworks* MapReduce o Spark y son lanzados al gestor de recursos gracias a la consola de comandos o terminal.

En un sistema Hadoop al uso, la invocación del comando *yarn* es la que permite interactuar con el gestor de recursos. Podremos desplegar nuestras aplicaciones gracias al subcomando *jar* e indicando posteriormente el artefacto compilado (JAR⁽²⁾) que queremos ejecutar. Del mismo modo podemos usar el comando *application* para realizar acciones sobre las aplicaciones en curso o espera, de modo que podamos listar, mover entre distintas colas de recursos o incluso terminar su ejecución de forma anticipada.

⁽²⁾Ved https://es.wikipedia.org/wiki/Java_Archive

Figura 11. Ayuda del comando *yarn application* y sus opciones

```
~$ yarn application -help
20/03/07 15:24:48 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
usage: application
  -appStates <States>          Works with -list to filter applications
                                based on input comma-separated list of
                                application states. The valid application
                                state can be one of the following:
                                ALL,NEW,NEW_SAVING,SUBMITTED,ACCEPTED,RUN
                                NING,FINISHED,FAILED,KILLED
  -appTypes <Types>           Works with -list to filter applications
                                based on input comma-separated list of
                                application types.
  -help                      Displays help for all commands.
  -kill <Application ID>    Kills the application.
  -list                      List applications. Supports optional use
                                of -appTypes to filter applications based
                                on application type, and -appStates to
                                filter applications based on application
                                state.
  -movetoqueue <Application ID> Moves the application to a different
                                queue.
  -queue <Queue Name>        Works with the movetoqueue command to
                                specify which queue to move an
                                application to.
  -status <Application ID>   Prints the status of the application.
```

Sin embargo, estas interacciones de tan bajo nivel suelen ir destinadas al mantenimiento o resolución de incidencias como el de matar una aplicación que se pueda haber quedado consumiendo recursos de forma continua por un error en el programa. Como comentábamos al inicio, el *framework* y el gestor son capaces de interactuar y gestionar los ciclos de vida de forma autónoma, pero de cara a los usuarios suele ser más habitual el uso de servicios de más alto nivel que no requieran actuación sobre estos (Hive, Oozie o Sqoop son herramientas habituales en este sentido). Como hemos comentado, cuando lanzamos una consulta en Hive, por detrás se lanzan una serie de tareas MapReduce de forma que al usuario le es ajena toda esta interacción que realizan tanto el *framework* (MapReduce) como el gestor (YARN) a la hora de resolver su consulta.

Es el caso del *framework* Spark, por ejemplo, permite el despliegue de aplicaciones sobre YARN de forma nativa mediante la invocación de un comando propio llamado *spark-submit*. Simplemente, indicando que el maestro de nuestra aplicación deberá negociar con el gestor de recursos *yarn*,³ este se encargará de iniciar el ciclo de negociaciones para poder disponer de los recursos necesarios.

⁽³⁾Ved «Running Spark on YARN». *Spark 3.1.1 Documentation (apache.org)*. Disponible en <https://spark.apache.org/docs/latest/running-on-yarn.html>

Un ejercicio sencillo si se dispone de un entorno Hadoop es realizar una invocación de la consola de Spark, llamada *spark-shell* (que no deja de ser una aplicación con unas necesidades concretas) indicando una serie de recursos necesarios para realizar nuestras tareas:

```
./bin/spark-shell \
  --master yarn \
  --deploy-mode client \
  --driver-memory 4g \
  --num-executors 5 \
  --executor-memory 2g \
  --executor-cores 1 \
  --queue default
```

El comando anterior iniciará un intérprete de comandos Scala, preparado para lanzar operaciones en Spark. Existe también un comando muy similar, con parámetros equivalentes, para iniciar un intérprete Python: pyspark.

Bajo estos comandos se esconden las negociaciones que realizará nuestro *driver* de Spark (así se denomina la parte coordinadora de tareas en dicho *framework*) que deberá pedir un contenedor con 4 GB de memoria RAM para sí mismo y 5 contenedores ejecutores donde se realizarán los trabajos invocados en la consola con 2 GB de memoria RAM asignados a la cola de prioridad *default*.

De cara a facilitar la interacción humana, YARN incluye también una interfaz web en el puerto 8088 que nos permite realizar muchas de las acciones habituales sin tener que emplear la consola (una opción que muy pocos gestores presentan). En la propia interfaz web podemos comprobar que se lista nuestra aplicación⁴.

(4)Como hemos comentado, mediante el comando *yarn application-list* podemos también ver el estado de la aplicación que hemos arrancado.

Figura 12. Imagen de la aplicación web que expone el uso de recursos disponibles y aplicaciones desplegadas

The screenshot shows the 'All Applications' section of the Hadoop YARN interface. It lists two applications:

- Spark shell**: User: parambris, Application Type: SPARK, Queue: default, Start Time: Tue, 20 May 2014 06:34:52 GMT, Finish Time: N/A, State: RUNNING, Final Status: UNDEFINED, Progress: 0%, Tracking URL: [link]
- org.apache.spark.examples.SparkPi**: User: parambris, Application Type: SPARK, Queue: default, Start Time: Tue, 20 May 2014 06:12:50 GMT, Finish Time: Tue, 20 May 2014 06:12:15 GMT, State: FINISHED, Final Status: SUCCEEDED, Progress: 100%, Tracking URL: [link], History

At the top left, there's a navigation bar with links like 'About', 'Nodes', 'Node Labels', 'Applications', 'Scheduler', and 'Tools'. The 'Applications' link is highlighted.

Aquí también podremos conocer algo más en detalle con qué otras aplicaciones comparte recursos, en el caso de aplicaciones *batch* el nivel de progreso de sus tareas e incluso podremos llegar a pedir su finalización anticipada (*kill application*) si clicamos sobre el nombre de nuestra aplicación y observamos el botón en la parte superior izquierda bajo la letra **p** del logo de Hadoop.

Figura 13. Vista en detalle de una de nuestras aplicaciones

The screenshot shows the 'Application application_1496483081907_0001' page. At the top, it displays basic application details:

- User: root
- Name: grep-search
- Application Type: MAPREDUCE
- YarnApplicationState: FINISHED
- FinalStatus Reported by AM: SUCCEEDED
- Start Date: Jun 01, 2017 05:45:11 -0400 2017
- Elapsed: 1m 44.4sec
- Tracking URL: [link]

Below this is the 'Application Metrics' section, which includes:

- Total Resource Preempted: <memory:0, vCores:0>
- Total Number of Non-AM Containers Preempted: 0
- Total Number of AM Containers Preempted: 0
- Resource Preempted from Previous Attempt: <memory:0, vCores:0>
- Number of Non-AM Containers Preempted from Current Attempt: 0
- Aggregate Resource Allocation: 475353 MB-seconds, 373 vcore-seconds

At the bottom, it shows the log entries for attempt ID appattempt_1496483081907_0001_000001, with the first entry being 'Attempt ID: appattempt_1496483081907_0001_000001, Started: Sat Jun 3 15:15:11 +0550 2017, Node: http://edfc63fc9fde:8042'.

Este ejemplo es válido ante cualquier *framework* de procesado compatible con Apache Hadoop, y los habituales son MapReduce, Tez o Spark. Algunos como Spark permiten esa opción interactiva, pero dada la naturaleza de los procesos

en Hadoop es algo más habitual encontrar procesos desatendidos (*detached*). Es decir, procesos *batch* que no requieren de interacción humana para finalizar su tarea.

Como indicamos antes, la idea es que ambos sistemas, gestor y aplicación, se entiendan de manera que se hagan un uso eficiente de recursos para obtener el resultado deseado al finalizar la aplicación. A modo de ejemplo, así sería como podríamos lanzar un proceso PySpark para ejecutar el cálculo del número pi.⁵

(5) Podéis acceder al código del script en la parte baja de «Examples». Apache. Disponible en <https://spark.apache.org/examples.html>

```
./bin/spark-submit \
  --master yarn \
  --deploy-mode client \
  --driver-memory 4g \
  --num-executors 5 \
  --executor-memory 2g \
  --executor-cores 1 \
  pi.py 10
```

El número final se refiere al número de particiones que serán empleadas. Podemos jugar con estos números relativos al número de nodos ejecutores que requerimos para ejecutar nuestra aplicación o la capacidad de memoria requerida, cosa que será muy necesaria cuando los recursos computacionales de nuestra aplicación sean elevados.

2.3. Docker en YARN

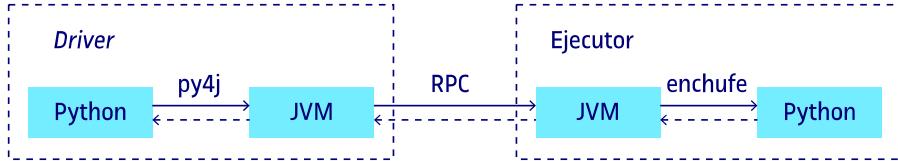
A pesar de que tradicionalmente los contenedores en YARN no han sido más que contextos lógicos de ejecución en máquinas virtuales Java (o JVM), cada vez era más demandada una capacidad de abstracción mayor con respecto al sistema base de los ejecutores. En particular en tareas provenientes del mundo de la ciencia de datos donde lenguajes como R o Python eran más demandados.

Cuando lanzamos un proceso Java en una máquina, no deja de ser un hilo de ejecución que puede verse afectado por el resto de hilos en la máquina, lo que no presenta un gran aislamiento entre las tareas de las distintas aplicaciones que se encuentren ejecutando en el clúster.

El poder disponer de unidades ejecutoras con un mayor nivel de aislamiento entre ellas permite además corregir problemas derivados de la resolución de dependencias en sistemas que emplean aplicaciones con lenguajes de programación distintos y no interpretables por la propia JVM. Este es el caso de **PySpark** o **SparkR**, las opciones que permiten especificar tareas en Python y R sobre el *framework* distribuido Spark. Cuando estas tareas se ejecutan necesitamos que el intérprete local las entienda (se muestra un ejemplo en la figura 14) y disponga de todas las librerías instaladas. Y si alguna librería faltara, el proceso fallará y se intentará en otra máquina donde con suerte pueda estar la

librería en cuestión. Esto nos obliga a gestionar todas las librerías en todas las máquinas, lo que puede suponer un arduo trabajo en un entorno con muchas máquinas.

Figura 14. Mapa de comunicación para una aplicación PySpark



Para solucionar este problema, desde la versión 3.0 de Hadoop se incorporó la posibilidad de ejecución de contenedores Docker⁶ desde YARN.

⁽⁶⁾Ved Docker sobre YARN.
Disponible en <https://www.youtube.com/watch?v=mEr-XQJzPOM>

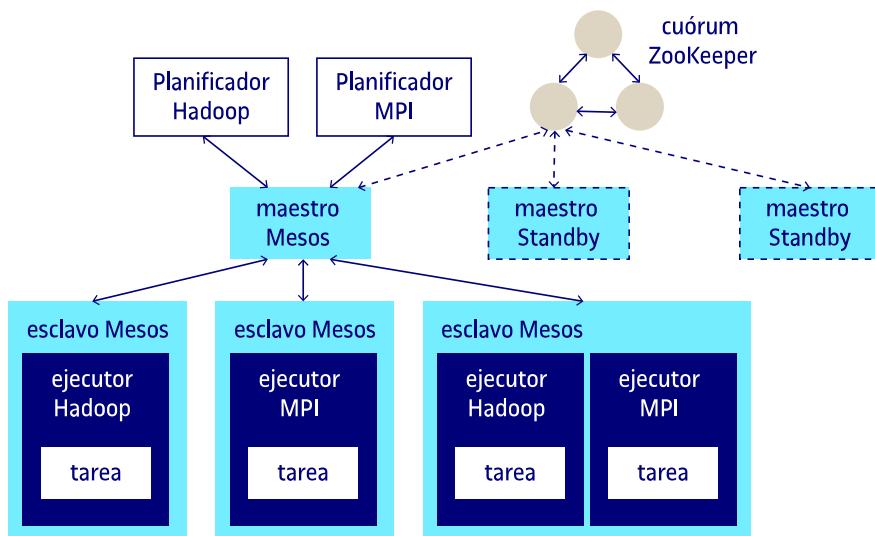
Gracias a tecnologías como la disponible en los contenedores Docker es posible emplazar aplicaciones en una máquina donde todas sus dependencias relativas a intérpretes o librerías vayan contenidas en la misma aplicación y sean contextos aislados de ejecución. Este nivel de aislamiento mejora la empleabilidad de los recursos aumentando los niveles de independencia entre aplicaciones, razón por la cual este tipo de soluciones son empleadas en la actualidad en multitud de gestores.

3. Apache Mesos

Mesos es un proyecto que vio sus inicios en la Universidad de California, Berkeley, con la intención de ofrecer un sistema de gestión de más bajo nivel en plataformas de recursos distribuidos multipropósito. De hecho, uno de sus creadores fue Matei Zaharia, creador también del *framework* Apache Spark tan usado hoy día. A diferencia de YARN, Mesos no dispone de un planificador de tareas como tal, el objetivo es trasladar esta responsabilidad a los propios *frameworks* de computación. De hecho, así como YARN pretende una correcta gestión de clústeres Hadoop, el propósito de Mesos va más allá y da cabida a procesos y sistemas de diversa naturaleza.

Mesos permite un acceso con un nivel de control superior sobre los recursos y facilita además la convivencia entre *frameworks* provenientes del mundo Hadoop con *frameworks* MPI (*Message Passing Interface*⁷), habituales en el mundo de la computación de altas prestaciones⁸ (HPC). Podéis ver un esquema de la arquitectura de Mesos en la figura 15. El enfoque primordial es el de disponer de un mayor aprovechamiento de los recursos disponibles en el centro de datos, es decir, en el grueso total de las máquinas disponibles por una organización. De hecho, esta necesidad fue identificada también por usuarios de los entornos Hadoop existentes en las compañías donde se iniciaron sus usos a escala corporativa. Vieron la necesidad de combinar pesadas ejecuciones *batch* con acciones interactivas o procesos *streaming*.

Figura 15. Visión de la arquitectura Mesos



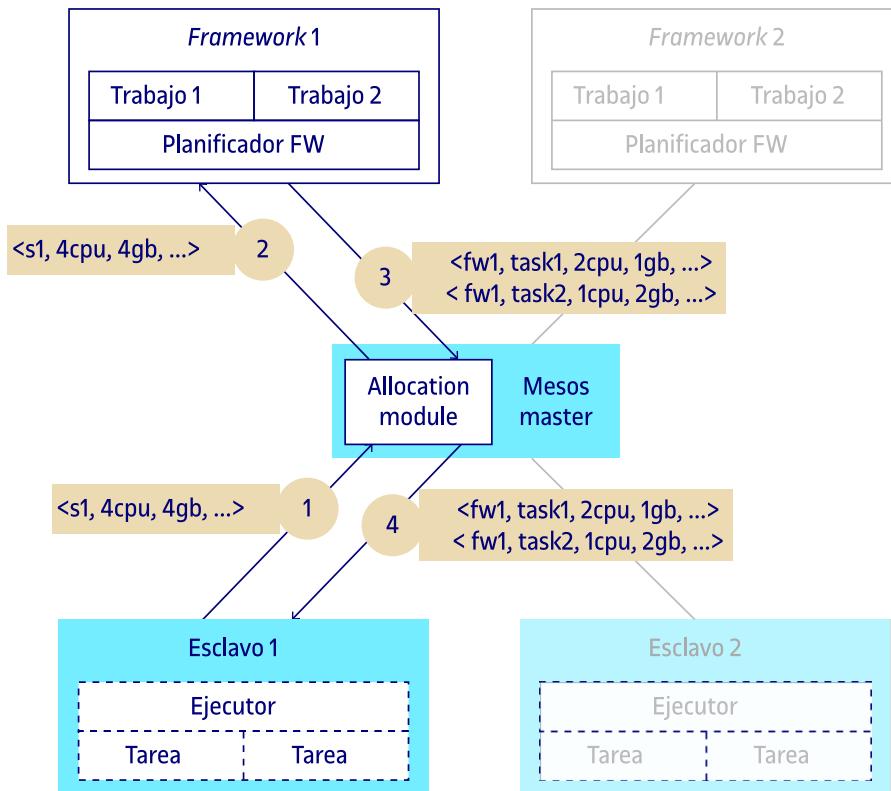
(7)Ved https://en.wikipedia.org/wiki/Message_Passing_Interface

(8)Ved «Computación de alto rendimiento». Wikipedia, la encyclopédie libre. Disponible en https://es.wikipedia.org/wiki/Computaci%C3%B3n_de_alto_rendimiento

Mesos implementa una gestión de dos niveles donde a cada *framework* de ejecución se le aplican unas políticas de oferta de recursos similar a las de las colas de gestión. Es decir, cuando una aplicación asociada a un *framework* demanda una serie de recursos, el planificador (*scheduler*) propio para dicho *framework* selecciona qué recursos y dónde ofrecerlos, de forma que esta aplicación únicamente pueda elegir entre los recursos ofrecidos para su entorno.

Podemos ver esta dinámica representada en la figura inferior, donde un nodo esclavo detalla sus recursos disponibles al nodo maestro en Mesos (paso 1), de forma que estos recursos puedan serle ofrecidos al *framework* 1 (paso 2), que indica cuáles son los recursos que quiere utilizar para las tareas necesarias por este a continuación (paso 3), y estas le son transferidas al nodo (paso 4), reservando esos huecos inicialmente libres.

Figura 16. Oferta de recursos acorde al *framework* demandante



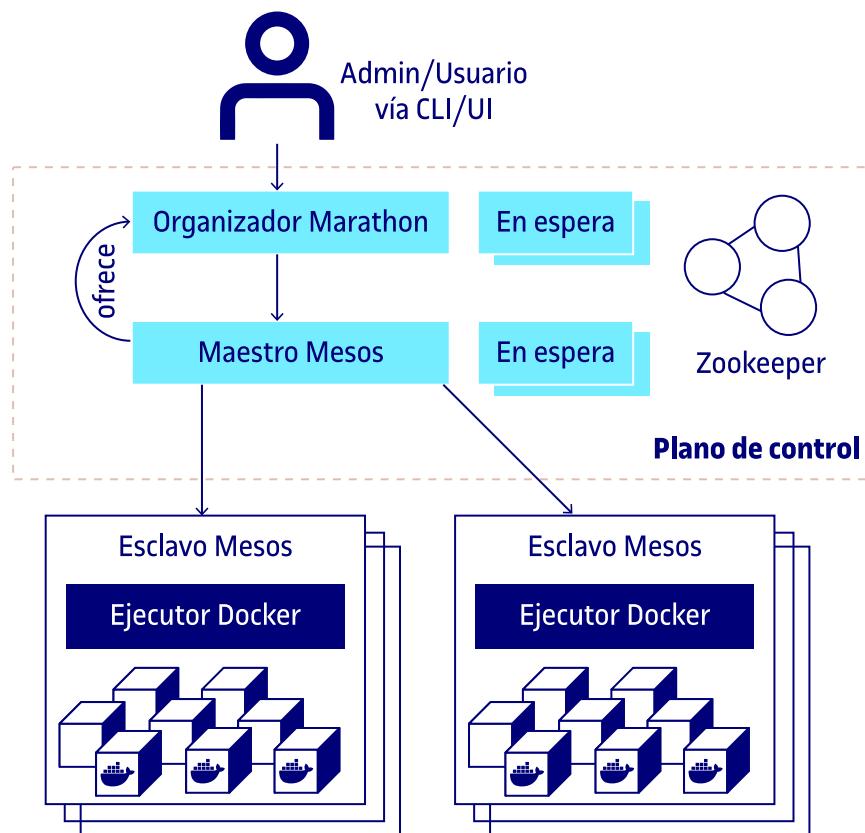
Este mecanismo permite una convivencia aislada de procesos de naturaleza muy diversa aplicando políticas transversales sobre la plataforma distribuida.

Una de las ventajas diferenciadoras de Mesos frente a otros gestores es que permite que los *frameworks* de aplicación acepten o declinen la oferta de recursos del gestor central. Gracias a incluir ambas opciones, se pueden llegar a definir *frameworks* específicos para Mesos con estrategias de negociación más complejas que las de aquellos gestores que no lo permiten.

Sin duda, Mesos arroja un orden de complejidad superior a lo visto anteriormente con YARN, hecho que permite a su vez un mejor aprovechamiento de los recursos en contextos de procesos heterogéneos.

Precisamente, uno de los *frameworks* más usados y versátiles de los disponibles en el ecosistema Mesos es **Marathon** (<https://mesosphere.github.io/marathon/>). No solo por implementar la convención que le permite negociar con Mesos sobre la disponibilidad de recursos para aplicaciones distribuidas, sino porque además incorpora la orquestación de contenedores para aplicaciones de uso cotidiano como pudieran ser servidores web o servidores de aplicación en general. Podemos ver la función ejercida por Marathon sobre Mesos en la siguiente figura. Marathon implementa aquellas particularidades no existentes en Mesos, complementándolo para poder ser usado con contenedores Docker (ejemplo de la figura 17).

Figura 17. Ilustración de la arquitectura de Marathon sobre Mesos



Dado el bajo nivel que presenta Mesos y exceptuando casos puntuales como pudieran ser aquellos destinados a la computación de altas prestaciones, en el contexto empresarial suele ser más habitual encontrarlo junto a alguna solución que permita una gestión superior mediante *meta-frameworks* como Marathon o Chronos (<https://mesos.github.io/chronos/>), por ejemplo.

3.1. Roles

Precisamente cuando hablamos de cuotas y gestión de recursos, en estos escenarios es habitual hablar de **roles**. Los roles suelen ser la piedra angular para habilitar el acceso a cuotas de recursos: garantías de acceso y acceso a recursos concretos sobre la base de las autorizaciones asociadas a estos.

Cualquier equipo, *framework* o usuario puede suscribirse mediante el rol que le corresponda y eso le indicará a Mesos la cuota de recursos a la que puede optar empleando un reparto equitativo y acorde a lo diseñado. Algunos tipos de recursos pueden ser algo más escasos que otros y esto puede generar una dominancia para *frameworks* que sean particularmente consumidores de dicho recurso, como es el caso de Spark y la memoria RAM. Gracias a estos roles se permite un control mucho más pormenorizado y se evita que un tipo de aplicaciones concretas fagociten recursos concretos.

3.2. Aislamiento

Es importante recalcar que el aislamiento entre aplicaciones cuando estas son de una naturaleza dispar es algo muy relevante. A sabiendas de los problemas que esto podría ocasionar cuando uno pretende establecer un gestor de naturaleza heterogénea, Mesos procuró solventar de raíz las potenciales problemáticas implementando mecanismos de aislamiento superior como son (y siguen siendo) los contenedores de software existentes en plataformas Linux. Es decir, desde un inicio introdujo el concepto de *contenedor*, tal y como lo conocemos en la actualidad y adherido a estos paradigmas ha evolucionado hasta la actualidad.

En 2011 Mesos entró a formar parte de los proyectos graduados de primer nivel en la fundación Apache bajo la versión 1.0 y ofreciendo entre otras capas de abstracción la ejecución vía contenedores Docker. Multitud de grandes organizaciones lo emplean precisamente dada su versatilidad en cuanto a aplicaciones de distinta tipología.

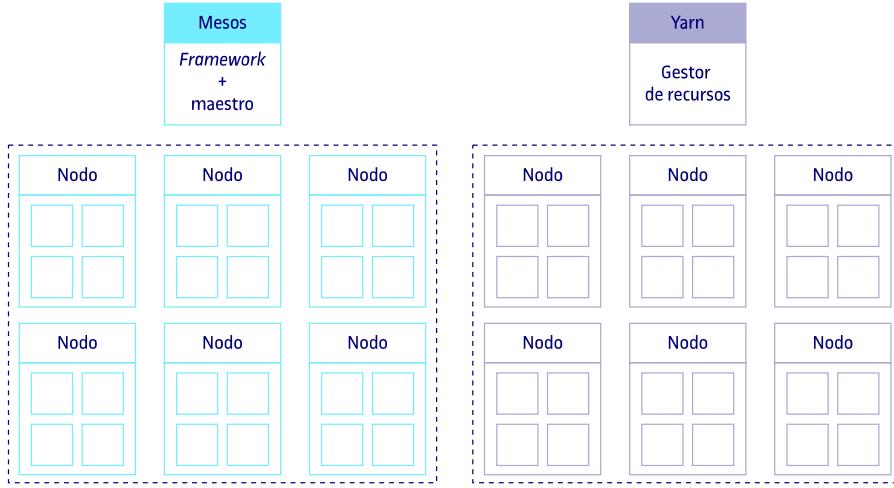
Ejemplo

Una de las grandes empresas que emplean Mesos es Netflix. Podéis verlo en el artículo «Distributed Resource Scheduling with Apache Mesos», en *Netflix Tech-Blog*. Disponible en <https://netflixtechblog.com/distributed-resource-scheduling-with-apache-mesos-32bd9eb4ca38>

Cabe destacar que ambos gestores, pese a estar destinados a derribar muros y silos de datos, en la mayoría de las organizaciones han sido empleados de forma independiente. YARN como clúster destinado a procesos pesados intensivos en el uso de datos (el caso de los clústeres Hadoop) con un foco muy orientado a la analítica y el soporte a la toma de decisiones. Mesos, por otro lado, es empleado en clústeres destinados a procesos más cercanos a la eficien-

cia operacional o computación de alto rendimiento. De este modo es habitual encontrar varios sistemas paralelos y diferenciados, sin interacción aparente, como se muestra en la figura 18.

Figura 18. Ilustración de clústeres independientes Mesos y Yarn



Ambos sistemas han servido de base para proyectos interesantes y relevantes que en la actualidad, pese a que no gozan de la madurez de estos proyectos, resulta relevante mencionar. Otro sistema que también ha servido de base es Apache Myriad, que veremos más adelante.

3.3. Uso de Mesos

Al igual que sucede con YARN, Mesos no es una herramienta amigable a la hora de ser empleada. Deberemos implementar nuestra aplicación como parte de algún *framework* que sepa cómo interactuar y negociar sus recursos con Mesos.

Gracias al *framework* Spark esto no resulta excesivamente complejo, ya que es capaz de interactuar con Mesos de forma nativa⁹, de modo que podríamos lanzar la consola anteriormente lanzada sobre YARN contra Mesos únicamente modificando la ruta del nodo maestro al que hacer referencia (en este caso deberemos indicar la dirección completa de la máquina maestra junto con el puerto donde se expone la interfaz de acceso).

⁽⁹⁾Ved <https://spark.apache.org/docs/latest/running-on-mesos.html>

```
./bin/spark-shell --master mesos://host:5050
```

En un estado inicial la aplicación Spark se suscribe (**SUSCRIBE**¹⁰) al maestro en Mesos de forma que pueda recibir ofertas (**OFFERS**¹¹) sobre los recursos disponibles a emplear. El *framework* puede decidir si para esta aplicación en cuestión hay alguna oferta que pueda aceptar y se lo transmite al maestro (**ACCEPT** o **DECLINE**). Una vez aceptada la oferta el *framework* puede emplear los recursos aceptados para llevar a cabo las tareas necesarias.

⁽¹⁰⁾Ved <https://mesos.apache.org/documentation/latest/scheduler-http-api/#subscribe>

⁽¹¹⁾Ved <https://mesos.apache.org/documentation/latest/scheduler-http-api/#offers>

Esta negociación es la que llevan a cabo el *framework* y el gestor para poder coordinarse y por ello es importante que ambos conozcan el protocolo a seguir. No todos los *frameworks* nos ofrecen la flexibilidad de Spark y en el caso de *frameworks* nativos en Hadoop deberemos adaptarlos o recurrir a plataformas intermediarias que nos habiliten la interacción entre el *framework* deseado y el gestor de recursos.

4. Otras plataformas

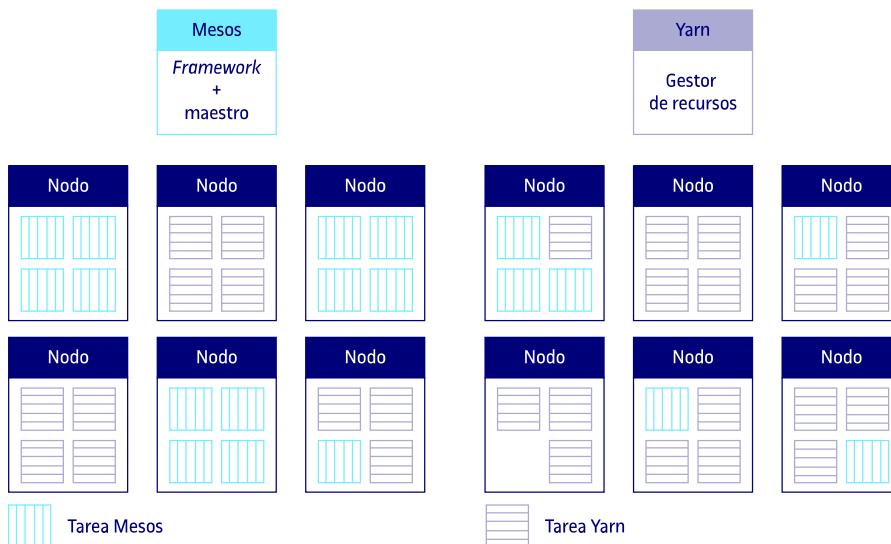
4.1. Apache Myriad

Una de las iniciativas de cara a derribar los silos anteriormente mencionados entre Mesos y YARN fue el proyecto Apache Myriad. Proyecto que a mediados del 2021 se encuentra todavía en fase de incubación, lo cual lo cataloga como un producto inmaduro para el público general.

Myriad aboga por ofrecer mecanismos que permitan integrar a YARN como un *framework* ejecutable más dentro de Mesos.¹² Mediante esta aproximación simplista uno puede disponer de un clúster Hadoop dentro de Mesos como si de una matrioska de capas de abstracción hardware se tratara.

⁽¹²⁾Ved «A tale of two clusters». Disponible en <https://www.oreilly.com/content/a-tale-of-two-clusters-mesos-and-yarn/#>

Figura 19. Convivencia de Mesos y YARN bajo la misma plataforma



Mientras que Mesos permite la integración de sistemas de computación de bajo nivel y el aprovechamiento al máximo de los recursos disponibles como gestor base, Myriad habilita que parte de estos recursos sean empleados como si de una plataforma Hadoop se tratara, aunando lo mejor de ambos mundos y rompiendo, de forma efectiva, los muros existentes entre ambos sistemas.

No todo son ventajas, dado que realizar esta interacción de forma lógica hace que parte de los recursos disponibles estén dedicados precisamente a dar ese nivel de abstracción. Pero en la actualidad esto compensa con creces, dada la proliferación de aplicaciones de diversa índole que podemos encontrar en prácticamente todas las organizaciones.

Nota

Aquellos que queráis indagar algo más en este proyecto, podéis recurrir como siempre a las documentaciones disponibles en la web del producto, en <https://cwiki.apache.org/confluence/display/myriad>.

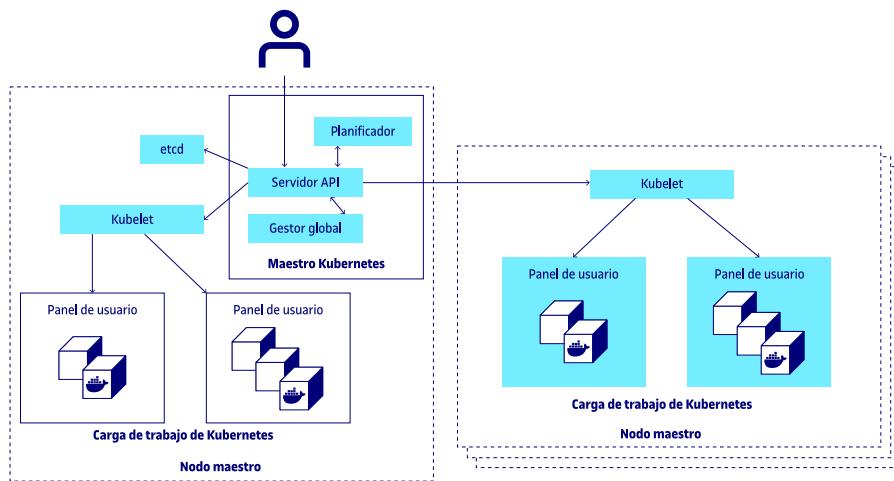
4.2. Kubernetes

A principios del 2000 Google necesitaba gestionar clústeres de máquinas con multitud de tareas, más allá de las imaginables por cualquier empresa por aquella época. Una vez más, crearon su propio proyecto para ello, llamado Borg¹³. Partiendo de esta base, el sistema evolucionó renombrado como Omega¹⁴ y finalmente en 2014 se dio a conocer al público general el proyecto Kubernetes. Un proyecto capaz de dar servicio a multitud de tareas compartiendo una misma infraestructura y coordinando las tareas que en ellas ocurren de forma eficiente.

⁽¹³⁾Ved <https://www.youtube.com/watch?v=0W49z8hVn0k>

⁽¹⁴⁾Ved <https://cutt.ly/imWI8pS>

Figura 20. Ejemplo de arquitectura tipo en Kubernetes



Podemos ver en la figura 20 que la piedra angular que sustenta Kubernetes es precisamente Docker (indicados como Pods en la imagen) dada su capacidad de contener una pléthora de aplicaciones de forma eficiente y con mínima huella sobre los sistemas huésped. Kubernetes es la serie de servicios desplegados en distintas máquinas a modo de contenedores, encargados del despliegue y la coordinación de las aplicaciones.

Dentro de los servicios maestros desplegados como contenedores también podemos observar los servicios de planificador (*scheduler*), conectividad tanto dentro como fuera de la red de la infraestructura (API Server) y el gestor global (*controller manager*) cubriendo las funciones principales que hemos visto en el resto de gestores de cara a mantener los servicios vivos y gestionar la labor de los nodos esclavos por medio del servicio *kubelet*.

Habitualmente es empleado para el despliegue de aplicaciones, pero cada vez más supone el reemplazo de las plataformas Hadoop corporativas para unificar bajo el paraguas de Kubernetes toda la gestión del centro de datos y la diversidad de aplicaciones que encuentren cabida en él.

Podemos pensar en el HDFS como una aplicación redundante de datos que emplea primordialmente disco y nos encontraremos con el proyecto Ceph¹⁵ dentro del marco de Kubernetes. Ceph es la evolución del almacenaje distri-

⁽¹⁵⁾Ved https://es.wikipedia.org/wiki/Ceph_File_System

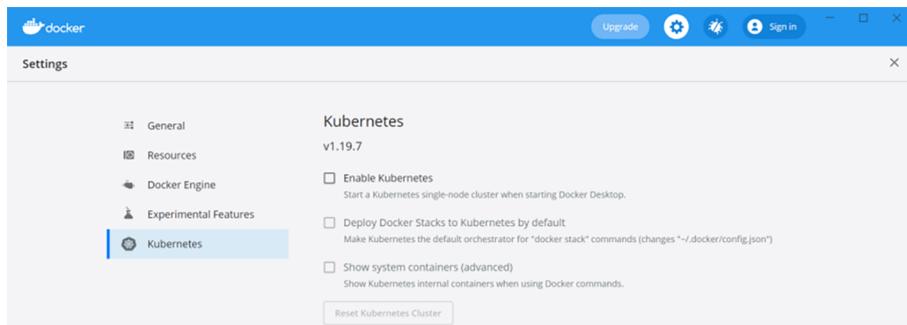
buido que corrige varios de los problemas que presentaba el HDFS al crecer el número de máquinas de forma importante. Esto nos permite pensar en Kubernetes como el gestor de recursos sobre el que poder desplegar nuestras aplicaciones y, de hecho, desde hace varias versiones Spark permite el despliegue de aplicaciones indicándolo como gestor.¹⁶ Sin duda es un recurso muy práctico principalmente cuando requerimos de capacidades dispares, tal y como veíamos anteriormente en el caso de Mesos y su extensión vía Myriad.

⁽¹⁶⁾Ved «Running Spark on Kubernetes». *Spark 3.1.1 Documentation (apache.org)*. Disponible en <https://spark.apache.org/docs/latest/running-on-kubernetes.html>

Sí que merece prestar atención al hecho de que, en efecto, no es un gestor de recursos como tal, es decir, dispone de capacidad de desplegar aplicaciones en distintas máquinas por medio de la abstracción que ofrecen los contenedores, pero en el caso de Kubernetes prima la persistencia. Esto quiere decir que bajo un acuerdo especificado en ficheros de configuración, acordamos con el gestor de Kubernetes cuántos contenedores requiere nuestra aplicación para funcionar (y el perfil de redes asociado), de modo que el gestor del clúster se encarga de que esto sea así hasta que se indique lo contrario sin negociación que valga. La premisa es que las aplicaciones desplegadas no terminen nunca y el coordinador se encargue de que así sea en la medida de lo posible, desplegando nuevas instancias cuando las anteriores fallen o escalando el número de contenedores si la carga en los sistemas se eleva.

No entraremos en excesivo detalle con respecto a la instalación de Kubernetes, dado que no es en absoluto trivial y, aquellos que lo vayáis a emplear, seguramente optaréis por el despliegue del recurso en su modalidad nube, lo cual resulta muy conveniente. Pero para que podáis probar esta tecnología aquellos que dispongáis de una instalación Docker Desktop, podéis habilitar la pestaña de Kubernetes en el apartado de configuración.

Figura 21. Habilitar Kubernetes en versiones de escritorio



Una vez desplegado, únicamente deberéis indicar cómo desplegar vuestra aplicación. Esto se realiza mediante ficheros de configuración *yaml*, que además permiten la definición de los recursos como código y pueden ser versionados, manteniendo así un mejor control. Una cualidad muy deseable para las organizaciones hoy día englobada dentro del paradigma DevOps tan omnipresente en todas las disciplinas.

Resumen

Los gestores de recursos, como hemos visto, nos permiten hacer un uso eficiente de la infraestructura disponible para la diversidad de fines que pueda perseguir una organización: operaciones intensivas en datos, lanzar aplicaciones basadas en flujos constantes (*streaming*) o simplemente exponer un servicio consumible.

Inicialmente, dentro de los ecosistemas Hadoop, nos encontramos con YARN. Su labor es la de coordinar las tareas requeridas por las distintas unidades de negocio que lancen procesos Spark sobre los datos del HDFS. En organizaciones maduras donde la productivización de sus conocimientos se aplique al vuelo en *streams* de datos, posiblemente estas aplicaciones están disponibles mediante contenedores Docker puestas en servicio gracias a los mecanismos de sistemas como Kubernetes o Marathon en Apache Mesos.

La esencia de un buen gestor de recursos y todo el desarrollo relativo a este ámbito es lo que ha dado cabida a la nube tal y como la conocemos. En el día a día de grandes proveedores como Amazon, Azure o Google existe esta problemática de tener que dar servicio a multitud de aplicaciones que requieren recursos y deben hacerlo con suficientes garantías empleando únicamente los recursos y las máquinas disponibles en sus centros de datos masivos. Una tarea nada sencilla que requiere de las herramientas adecuadas, dado que, según añadimos nuevos elementos a nuestro ecosistema tecnológico, surgen necesidades que sin duda harán avanzar estas plataformas según las necesidades que surjan.

En estas líneas hemos podido observar cómo esta necesidad ha ido creciendo y variando basándose en el uso de organizaciones concretas imperantes en cada momento. Posiblemente, en el futuro, según los actores y los usos de la información varíen, los gestores de recursos evolucionarán del mismo modo para dar cabida a nuevas formas de computación (GPU, TPU...) o cargas de trabajo todavía inimaginables que, en la actualidad, no son más que los inicios de grandes cambios potenciales en el mundo de la computación.⁹

⁽⁹⁾Ved «Computación cuántica». *Wikipedia, la enciclopedia libre*. Disponible en https://es.wikipedia.org/wiki/Computaci%C3%B3n_cu%C3%A1ntica

Bibliografía

- Aggarwal, M.** (2018). *Learn Apache Mesos*. Nueva York: Packt Publishing.
- Burns, B.; Beda, J.; Hightower, K.** (2019). *Kubernetes: Up and Running*. O'Reilly Media Inc.
- Ghodsi, A.; Zaharia, M.; Hindman, B. y otros** (2011). «Dominant resource fairness: fair allocation of multiple resource types. En: *Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI'11)*. Berkeley: USENIX Association.
- Hindman, B.; Konwinski, A.; Zaharia, M. y otros** (2011). «Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. En: *Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI'11)*. Berkeley: USENIX Association.
- Silberschatz, A.; Baer, P.; Gagne, G.** (2006). *Fundamentos de sistemas operativos* (7a. edición). Madrid: Mc Graw-Hill.
- Tanenbaum, A. S.** (2003). *Sistemas operativos modernos*. México D.F.: Pearson Educación.
- White, T.** (2015). *Hadoop: The Definitive Guide*. O'Reilly Media Inc.

