Sistema híbrido de almacenamiento de datos en el PIC

Alba Vendrell Moya

Resumen– Recientemente el *Port d'Informació Científica* (PIC) ha contemplado la posibilidad de desarrollar una alternativa al sistema actual del centro, tradicional y completamente físico. Se pretende lograr un sistema híbrido de datos que integre el sistema dCache con el servicio de almacenamiento en la nube AWS S3, en el cual se puedan almacenar diferentes tipos de copias de forma transparente al usuario. El proyecto resulta beneficioso y de interés debido a la gran incertidumbre existente respecto el futuro del mercado de la tecnología en cinta, así como las ventajas de crecimiento dinámico e instantáneo que ofrecen los servicios en la nube frente al complejo proceso administrativo cuando se pretende crecer de forma física. El sistema híbrido logrado no se trata de una cuestión únicamente práctica, sino que también implica la gestión de riesgos tanto de migración como de recuperación frente a desastres, además de ser un asunto económico y medioambiental.

Palabras clave– Amazon Web Services, computación en la nube, dCache, funciones Lambda, PIC, redundancia de datos, sistema de almacenamiento, sistema híbrido de datos.

Abstract– Recently the *Port d'Informació Científica* (PIC) has been contemplating the possibility of developing an alternative to the current system of the centre, which is traditional and completely physical. For this reason, the aim is to achieve a hybrid data system that integrates the dCache system with the AWS S3 cloud storage service, in which different types of copies can be stored transparently to the user. The project is beneficial and of interest due to the great uncertainty regarding the future of the tape technology market, as well as the advantages of dynamic and instantaneous growth offered by cloud services as opposed to the complex administrative process when growth is intended to be physical. The achieved hybrid system is not only a practical matter, but also involves risk management for both migration and disaster recovery, in addition to being an economic and environmental issue.

Keywords– Amazon Web Services, cloud computing, data redundancy, dCache, Hybrid system, lambda functions, PIC, Storage System.

1 Introducción

El *Port d'Informació Científica* (PIC) es un centro de procesamiento de datos de alto rendimiento con un centro de datos propio. Actualmente se encuentra contemplando la posibilidad de hacer uso de servicios de almacenamiento externos debido a que hay muchos proyectos en los que trabajan que requieren tener dobles copias, y obtener esta redundancia de datos mediante un servicio de almacenamiento en la nube resulta de gran utilidad. Existe una gran incertidumbre respecto el futuro coste de cinta o incluso su existencia, debido a la desaparición de la competencia en los últimos años del principal y único fabricante Fujifilm [1]. Por este motivo, el proyecto implica una gestión de riesgo frente una posible migración. La posibilidad de crecer dinámica e instantáneamente es una de las mayores ventajas que presenta la conexión de herramientas en la nube con la infraestructura del centro. Además, es preciso ser consciente del elevado consumo energético que presentan los centros de procesamiento de datos, lo que implica que se trata de una cuestión no solo práctica, sino también económica y medioambiental.

Dentro de un entorno que cada vez se mueve más hacia *cloud*, con grandes compañías dando servicio (Amazon Web Services (AWS) [2], Microsoft Azure [3] o Google

• E-mail de contacto: alba.vendrellm@e-campus.uab.cat

• Mención realizada: Enginyeria de Computadors

• Trabajo tutorizado por: Eduardo Cesar Galobardes (Computer Architecture Operating Systems Department) y Jordi Casals Hernández (Port d'Informació Científica)

• Curso 2019/20

Cloud Platform (GCP) [4]) y siendo esta una filosofía totalmente opuesta al concepto de centro de datos privado, tiene sentido para el PIC investigar la integración de sus sistemas con este tipo de herramientas.

En los últimos años el centro ha hecho diferentes pruebas con servicios *cloud*, participando en distintos proyectos. Por un lado en Helix Nebula Science Cloud [5], donde se exploraban las posibilidades de los sistemas híbridos de computación para trabajos con alta densidad de transferencia de datos, y por otro lado en una colaboración con Amazon Web Services (AWS), plataforma de la compañía Amazon que ofrece servicios en la nube pública. Esta última se ha basado en la realización de pruebas de reducción de costes y crecimiento temporal de la granja de computación basado en HTCondor [6], así como otras de funcionamiento y coste sobre el archivado de datos a largo plazo, trabajando con herramientas como Amazon S3 [7].

Este proyecto se enmarca dentro de la investigación de las herramientas y servicios que ofrece AWS, con la finalidad de encontrar aquellos que en su conjunto resulten beneficiosas para la creación de un sistema híbrido mediante la integración con los sistemas de almacenamiento local que actualmente dispone el centro.

Siguiendo la aproximación de extensión del centro de datos con recursos *cloud*, se ha analizado el escenario de hacer crecer las opciones de almacenamiento local con espacio *cloud*, centrándose en las herramientas de Amazon Web Services S3, S3 Glacier y S3 Deep Archive. El servicio AWS S3 puede tener diferentes políticas de retención, acceso e inmediatez de los datos.

Se ha encontrado la manera de automatizar el movimiento de datos entre las diferentes opciones de S3, pero más importante, la automatización del movimiento de datos de diferentes sistemas de almacenamiento que hay en el PIC hacia el tipo de almacenamiento de AWS S3 que más convenga en función de las necesidades de acceso, de la popularidad de acceso y de la importancia del conjunto de datos, obteniendo así el sistema híbrido planteado. Concretamente en el proyecto se ha creado una integración del servicio S3 de AWS con el sistema distribuido dCache [8], mediante el uso de herramientas como el SDK boto3 [9] o las funciones Lambda de AWS S3 [10].

El artículo está estructurado de la siguiente forma. En la segunda sección se identifican los objetivos, tanto principales como específicos y complementarios. A continuación se encuentra el estado del arte y la metodología, donde se contextualiza el proyecto y el procedimiento realizado. A partir de la quinta sección empieza el desarrollo principal del proyecto, mediante una introducción a los sistemas de almacenamiento, la integración entre ambos, y una serie de tareas paralelas también realizadas respecto la gestión de eventos. Finalmente, se exponen los resultados obtenidos y las conclusiones del trabajo, cerrando con una sección de tareas futuras, agradecimientos y referencias bibliográficas.

2 OBJETIVOS

A continuación, se exponen los diferentes objetivos, tanto principales como específicos y complementarios, identificados en el proyecto.

2.1. Objetivos principales

El proyecto consta de cuatro objetivos principales que en su conjunto forman la base del trabajo a realizar:

O.P. 1. Obtener un sistema híbrido de almacenamiento entre *cloud* y centro de datos.

O.P. 2. Dotar al sistema de la capacidad de realizar segundas copias, copias simples o copias de seguridad a largo plazo con una interfaz transparente al usuario.

O.P. 3. Definir los niveles de servicio basados en los diferentes tipos de requisitos [11]. Esto implica determinar la cantidad de copias que se van a hacer y el resto de servicios que se van a ofrecer en función de la petición de cada experimento.

O.P. 4. Automatizar el movimiento de datos entre las diferentes opciones de almacenamiento, combinando local con AWS S3, en función de una serie de parámetros (Cantidad de acceso, antigüedad de los ficheros, criticidad o disponibilidad requerida).

2.2. Objetivos específicos y complementarios

Con la finalidad de lograr el sistema deseado se han determinado una serie de objetivos específicos y necesarios:

O.E. 1. Administrar sistemas a nivel básico tanto de máquinas físicas o virtuales en el centro de datos, como en los diferentes servicios de AWS.

O.E. 2. Analizar y comprender el funcionamiento básico de las diferentes herramientas de almacenamiento que se plantean en el trabajo, como dCache, Enstore [12] y Ceph [13], y de los diferentes servicios de almacenamiento de AWS.

O.E. 3. Estudiar las posibilidades que ofrecen la librería de Python boto3 y la interfaz de línea de comandos de AWS.

O.E. 4. Gestionar eventos mediante el sistema de notificaciones de dCache y las funciones AWS Lambda.

Así mismo, para añadir funcionalidades que faciliten la gestión del sistema y plantear un cierre de proyecto ambicioso, se han establecido dos objetivos complementarios.

O.C. 1. Emplear herramientas de monitorización y graficación como Nagios [14], Grafana [15] o Amazon CloudWatch [16] para analizar el estado del sistema y los resultados obtenidos.

O.C. 2. Poner la herramienta desarrollada en producción dentro del centro de datos.

3 ESTADO DEL ARTE

A causa de la constante evolución e integración de la nube pública con la infraestructura empresarial, el mercado se encuentra en una transformación tecnológica en busca de la mayor innovación, eficiencia y productividad gracias a las herramientas *cloud*. De entre todas las existentes se pueden remarcar tres grandes plataformas de computación en la nube, las cuales han experimentado un gran crecimiento en el uso de sus servicios: Microsoft Azure [3], Google Cloud Platform [4] y Amazon Web Services [2].

Por otro lado también se encuentran los sistemas de ficheros distribuidos (DFS) administrados localmente, como es el caso de Ceph [13] o dCache [8].

En este proyecto se ha decidido desarrollar un sistema híbrido entre los sistemas especificados debido a conocimientos previos del personal del centro y disponibilidad de créditos gratuitos para hacer pruebas en la plataforma AWS, así como la importancia que tiene dCache en el centro, ya que es el sistema que gestiona el sistema principal de disco.

Actualmente existe una interfaz de integración de dCache con AWS S3 programada por los propios desarrolladores del sistema físico en *Go* [17]. Esta proporciona la funcionalidad básica, pero presenta limitaciones frente a las necesidades de nuestro sistema ya que no permite trabajar con diferentes clases de objetos de AWS S3. Realizar una interfaz propia en lenguaje Python implica una mayor flexibilidad de integración y facilidad de desarrollo por diversos motivos.

Por un lado, la facilidad de integración con cualquier sistema operativo que supone Python es mucho mayor que la del sistema realizado con *Go*, ya que este segundo como mínimo requiere ser compilado cada vez que se realiza un cambio. Al mismo tiempo implementar el sistema desde cero proporciona unos buenos cimientos que implican una mayor comprensión del sistema en cuestión. Esto último resulta beneficioso a la hora de integrar las tareas externas a la creación del sistema híbrido, también necesarias para el desarrollo del proyecto, ya que compartir lenguaje de programación y herramientas en los diferentes módulos del proyecto ofrece un mayor control y conocimiento de estas.

Por otro lado, de esta forma no hay necesidad de recompilar cada vez que se realicen modificaciones, lo que permite un trabajo más cómodo y eficaz. Además se disponen diversas herramientas que facilitan su implementación como Jupyter Notebook [18] o la librería boto3, la cual se caracteriza por tener un gran potencial y disponer de documentación de calidad.

En este punto ya se conocen las herramientas principales sobre las que se basa el proyecto, así como la situación y limitaciones de la interfaz de integración identificada.

4 METODOLOGÍA

Debido a la naturaleza del proyecto se ha establecido una metodología de desarrollo en espiral. El paradigma de modelo de espiral permite reaccionar frente a los riesgos que detectamos en cada iteración. Este presenta un ciclo de vida flexible y dinámico.

La característica principal que define el proyecto es el alto grado de incertidumbre que presenta. Esta propiedad suele verse relacionada con una mayor dificultad a la hora de realizar una buena planificación, pero justamente por este motivo es imprescindible establecer un buen método de forma previa para organizarse adecuadamente y ejecutar las distintas tareas de forma eficiente. Es por esta causa que las representativas iteraciones de la metodología espiral resultan tan beneficiosas en este proyecto, permitiendo así valorar y analizar los riesgos de la iteración anterior para fijar las actividades de la siguiente.

Remarcar también el gran aprovechamiento de la tarea de determinación de objetivos, pues a pesar de tener una serie de objetivos establecidos desde el inicio, debido al indeterminismo de los resultados y riesgos, resulta de gran utilidad fijarlos de nuevo conforme va evolucionando el proyecto.

Para poder cumplir con los objetivos especificados y teniendo en cuenta el método de desarrollo establecido, se ha realizado un plan de trabajo completo inicial, el cual se reconsiderará en la fase de planificación de cada iteración del proyecto. Este guión inicial se puede dividir en seis etapas: AWS, dCache, integración de ambos sistemas, gestión de eventos, otras posibles opciones de sistemas híbridos y estudio final. Las tareas de las distintas etapas, así como su duración estimada y los objetivos con los que tienen relación, se encuentran expuestas en la Tabla 3 del anexo A.1.

5 INTRODUCCIÓN A LOS SISTEMAS

A continuación, se expondrán las características principales de los dos sistemas en las que se basa el proyecto, AWS por la parte *cloud* y dCache por la parte física. El proceso de aprendizaje durante la ejecución de las distintas tareas en esta prate del desarrollo ha sido necesario para la comprensión de las necesidades de la interfaz de integración entre ambos sistemas.

5.1. Sistema en la nube: AWS

La amplia colección de AWS cuenta con centenares de servicios. A lo largo del proyecto se ha trabajado con dos tipos de servicios: De computación y almacenamiento. Se ha hecho uso de los servicios de computación Amazon EC2 [19] y AWS Lambda, los cuales ofrecen servidores virtuales en la nube y ejecuciones de código *serverless* respectivamente. Por otro lado, el servicio de almacenamiento que se ha usado es Amazon Simple Storage Service (S3), que ofrece un almacenamiento en la nube mediante objetos en *buckets*.

No obstante el objetivo principal del proyecto se fundamenta en la creación de una interfaz de integración entre el sistema físico dCache y el sistema en nube AWS S3. Es por este motivo que a lo largo del documento este servicio presentará una mayor prioridad frente al resto, los cuales

han sido utilizados para una mejor comprensión de la plataforma y añadir funcionalidades al sistema a desarrollar que faciliten su uso.

Con la finalidad de poder cumplir con los objetivos principales del proyecto se ha profundizado en las opciones de almacenamiento AWS S3 así como sus prestaciones. El servicio ofrece diferentes tipos de almacenamiento en la nube, es decir, diferentes clases de objetos que se almacenan en *buckets* de S3. Concretamente en el proyecto se tratan los de tipo *Standard, Glacier* y *Deep Archive*. Las características principales de estos son las siguientes:

- Amazon S3 *Standard*: Enfocada a objetos a los que se quiere acceder con frecuencia. El almacenamiento de los datos con este tipo ofrece un alto nivel de procesamiento, durabilidad y disponibilidad, obteniendo un acceso rápido y de baja latencia.
- Amazon S3 *Glacier*: Los objetos que se almacenan con este tipo se espera que tengan una mayor duración de retención que con *Standard*, así que la disponibilidad es inferior y es necesario realizar un proceso de recuperación para acceder a ellos. Permite almacenar datos a precios competitivos, proporcionando tres opciones (*"tiers"*) de recuperación que van del minuto a horas en función del seleccionado:
  - *Expedited*: De 1 a 5 minutos.
  - *Standard*: De 3 a 5 horas.
  - *Bulk*: De 5 a 12 horas.
- Amazon S3 *Deep Archive*: Finalmente, esta clase es la más económica, diseñada para la retención a largo plazo de datos, con una estimación de 7 a 10 años o más. Por sus características se muestra como una alternativa a los sistemas de cinta magnética. Al igual que *Glacier*, *Deep Archive* también dispone de diversas opciones de recuperación de los datos:
  - *Standard*: En un margen de 12 horas.
  - *Bulk*: En un margen de 48 horas.

La Tabla 1 expone algunos de los datos mencionados, así como otros de interés para la futura evaluación y comparación entre los sistemas:

TABLA 1: RENDIMIENTO DE LAS CLASES STANDARD, GLACIER Y DEEP ARCHIVE DE S3

	Standard	*Glacier*	*Deep Archive*
Cargo mínimo por capacidad	N/D	40 KB	40 KB
Cargo mínimo por duración	N/D	90 días	180 días
Tiempo de recuperación	milisegundos	minutos / horas	horas

5.2. Sistema físico: dCache

El sistema dCache presenta una alta complejidad, ya que dispone de muchos componentes y una gran cantidad de opciones de configuración. Para poder comprender el funcionamiento de los componentes clave de este sistema de almacenamiento, así como la realización de la posterior configuración necesaria, ha sido precisa la participación de la responsable del servicio de dCache del PIC.

Las primeras ideas sobre la interfaz de integración parten gracias a la documentación de Sistemas de Almacenamiento de Terceros *(TSS)* [20], con la cual se puede llegar a la conclusión de que creando un ejecutable que tenga las funciones *PUT, GET* y *REMOVE* es suficiente para transferir datos desde dCache hasta el sistema TSS especificado, en este caso AWS S3. Esto ofrece la posibilidad de conectar un sistema de almacenamiento externo a la instancia local de dCache sin tener que estar ligados ni a lenguajes ni a metodologías, simplemente a un *input* y un *output*.

El código ya existente de integración de AWS S3 con dCache, desarrollado en *Go* por Tigran Mkrtchyan, presenta limitaciones para el sistema, debido a que no permite trabajar con las diferentes clases de objetos de AWS S3.

Tal y como se ha expuesto durante la explicación del sistema en la nube AWS S3, las clases *Standard, Glacier* y *Deep Archive* tienen características diferentes entre ellas. Gracias a esta diversidad se puede hacer uso de un tipo de clase u otro en función de las necesidades de los niveles de servicio que se pretenden establecer. Por este motivo, así como por diversas ventajas de flexibilidad de integración y facilidad de desarrollo expuestas en el apartado de estado del arte, se ha decidido proseguir con la idea inicial de desarrollar una interfaz de integración propia en *Python*.

No obstante se ha compilado y verificado el correcto funcionamiento del binario mencionado, así como comprobado que constaba de, entre muchas otras, las tres funciones que aparecen en la documentación de TSS de dCache: *PUT, GET* y *REMOVE*. La inversión de tiempo realizada en estas pruebas ha resultado ser muy útil y ha permitido partir la complejidad del trabajo, ya que en caso de futuros problemas, sabiendo que se parte de una base teórica funcional, se debería centrar la atención en resolver los fallos respecto a la integración.

Para ser capaces de empezar con el proceso de integración se ha estudiado el servicio principal del sistema dCache, el cual está compuesto por los siguientes elementos:

- Los *pools*, encargados de guardar los datos. Los *pools* físicos pueden estar formados por múltiples *pools* lógicos.
- La base de datos, que guarda la información de los ficheros; Dónde están, cuándo se han escrito, si se encuentran en disco o en un sistema de almacenamiento externo, etc.
- Las *doors*, encargadas de exportar con diferentes protocolos los datos que tenemos en los *pools*.

Dentro de un *pool* lógico se pueden configurar sistemas de almacenamiento externos, siendo AWS el TSS deseado en el proyecto. Además, cada *pool* tiene su respectivo fichero de *setup*, el cual define detalles relacionados con la conectividad TSS.

Con esta información ya se conoce el esquema básico de dCache. En este punto se puede identificar por dónde se debe enfocar el inicio de la integración por parte del sistema físico, es decir, por la exportación del *pool* lógico en el que se configure AWS como TSS. El esquema en cuestión queda reflejado en la Fig. 1:

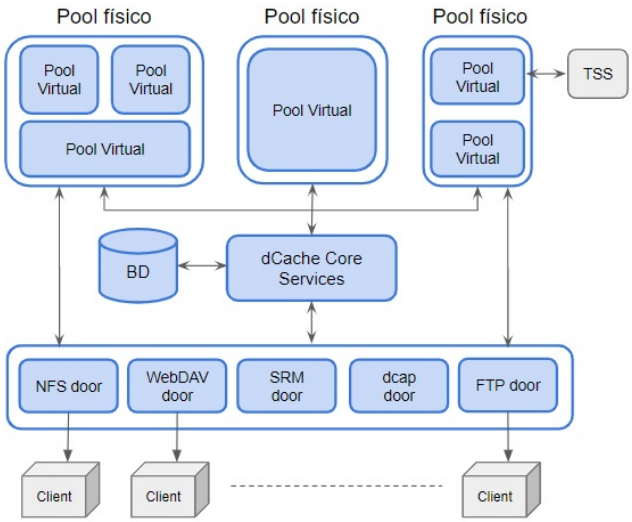

Fig. 1: Esquema básico de dCache en el PIC.

6 Integración de ambos sistemas

Sabiendo que dCache permite exportar su sistema de ficheros en distintos protocolos, entre ellos NFS, el primer paso es configurar la instancia de dCache para que la *door* encargada gestione la exportación del *pool* lógico. De esta forma, se podrá montar el volumen vía NFS en una máquina externa.

Los directorios del sistema tienen una serie de *tags* definidos, como por ejemplo podría ser el que marca el proyecto al que pertenece o el tipo de almacenamiento. Estos *tags* determinan cómo actuarán los ficheros que interaccionen con este. Concretamente en el proyecto se han establecido los *tags* "hsmInstance" y "hsmType" con valor "awss3" en el directorio sobre el cual se quiere realizar la integración con *cloud*, es decir, en aquel que se busca que los archivos que entren se suban al sistema de almacenamiento AWS S3. A continuación, la Fig. 2 muestra todos los *tags* del directorio mencionado:

```
[dteam001@alba01 awss3]$ grep "" $(cat ".(tags)()")
.(tag)(AccessLatency):NEARLINE
.(tag)(hsmInstance):awss3
.(tag)(hsmType):awss3
.(tag)(OSMTemplate):StoreName vo-dteam
.(tag)(RetentionPolicy):CUSTODIAL
.(tag)(sGroup):dteam
.(tag)(storage_group):vo-dteam
```

Fig. 2: Tags del directorio con finalidad de integración.

Seguidamente se ha modificado el fichero de *setup* del *pool* para determinar qué comando y argumentos se deben ejecutar y mandar si llega una interacción con un directorio con los *tags* "hsmInstance" y "hsmType" con valor "awss3". En este también se indica el número máximo de solicitudes permitidas por operación.

```
hsm create awss3 awss3 script -command=/opt/awss3/tfg/s3hsm.py
-c:puts=8 -c:gets=20 -c:removes=5
```

Fig. 3: Configuración del sistema de S3 en el pool.

Como se puede ver en la Fig. 3, la cual muestra la configuración inicial descrita, el fichero responsable de la integración es "s3hsm.py". Es en este en el que se encuentran implementadas las diferentes funciones necesarias para el comportamiento deseado.

Por lo tanto, el primer paso respecto la programación del fichero en Python es la creación de las funciones básicas *PUT, GET* y *REMOVE*. Estas se llaman en las siguientes interacciones con el directorio explicado anteriormente:

- *PUT*: Con una operación de copia hacia este, es decir, cuando se escribe un archivo.
- *GET*: Con una operación de copia desde este, es decir, cuando se lee un archivo.
- *REMOVE*: Con una operación de eliminación de archivo.

6.1. Detalles de ejecutables con soporte TSS

Se ha implementado el fichero "s3hsm.py" a partir de la sintaxis que proporciona la documentación de dCache sobre los ejecutables con soporte de TSS. Gracias a esta se pueden identificar los argumentos que recibe el ejecutable por parte del sistema de almacenamiento físico para cada una de las operaciones básicas, así como las salidas que espera tanto en caso de éxito como de fallida.

```
put <pnfsID>  <filename>  -si=<storage-information>
get <pnfsID>  <filename>  -si=<storage-information> -uri=<storage-uri>
remove -uri=<storage-uri>
```

Fig. 4: Sintaxis de la llamada a un *script* con soporte TSS.

Tal y como indica la Fig. 4 los argumentos que se reciben son los siguientes:

- La operación realizada: *put, get* o *remove*.
- El "pnfsID", que es el identificador interno que proporciona dCache a cada archivo. Este argumento solo se recibe en el caso de las dos primeras operaciones.
- El "filename", que corresponde a la ruta completa del archivo local que se copiará en el TSS, para el caso de *put*, o que se copiará desde el TSS, para el caso de *get*. Este argumento tampoco se recibe con la opción *remove*.
- Finalmente, también se pueden recibir dos opciones con el nombre "si" y "uri". El primero es la información de almacenamiento del archivo y el segundo la URI del mismo. El último mencionado sirve para identificar al archivo en el TSS.

Además, la documentación especifica que se pueden añadir más opciones desde el fichero de *setup* del *pool*, el cual ya se ha modificado tal como muestra la Fig. 2 para el comportamiento básico.

Respecto las salidas esperadas, en caso de éxito con una operación *put* el ejecutable debe devolver una URI de almacenamiento válida. Con el objetivo de mantener la interoperabilidad entre el sistema original y el del proyecto se ha decidido darle el mismo formato [21]. No se especifica ningún tipo de requisito frente a una situación de éxito con las operaciones *get* o *remove*. Por el contrario, en caso de fallida con cualquiera de las tres operaciones se espera un "2" de código de retorno.

A continuación, se expone la implementación de las tres funciones principales del ejecutable. Las explicaciones se basan en un directorio con los *tags* "hsmInstance" y "hsmType" con valor "awss3".

6.2. Función *PUT*

El objetivo de la función *PUT* es almacenar aquellos ficheros introducidos en el directorio. La acción se define dentro de la función "doPUT()".

Cuando un fichero se añade al directorio se llama al ejecutable con operación *put* y mediante el uso de la función *upload_file* de boto3 se transfiere a AWS S3. Este se almacena en el *bucket* de S3 especificado por defecto con clase *Standard*. No obstante se pueden añadir argumentos extra en los parámetros de la función para especificar una clase en concreto.

La Fig. 5 muestra el funcionamiento descrito en la función correspondiente:

```
def doPUT(filePath, bucketName, objectName, hsm_type, hsm_instance):
    try:
        client.upload_file(filePath, bucketName, objectName)
        o = (urlparse(hsm_type + "://" +  hsm_instance + "/" +
             bucketName + "/" + objectName)).geturl()
        print(o)

    except Exception, e:
        print("Upload failed")
        print(e)
        sys.exit(2)
```

Fig. 5: Código de la función de subida a AWS S3.

Además, internamente el sistema físico gestionará la petición para generar un registro en su base de datos, lo que permitirá identificar al objeto y tener acceso a este en el momento que se requiera.

6.3. Función *GET*

La función *GET* pretende descargar los objetos del sistema de almacenamiento *cloud* que se copien desde el directorio. Esta función presenta una mayor complejidad a causa de los requisitos y características de los objetos de AWS S3, los cuales se detallan a lo largo del apartado. La acción se define dentro de la función "doGET()".

En caso de realizar una copia de un fichero del directorio se llama al ejecutable con operación *get*, y mediante el uso de la función *download_file* de boto3 se descarga y copia en local el objeto del *bucket* indicado.

El funcionamiento detallado es el básico para conseguir descargar un objeto que se encuentra almacenado en el sistema *cloud* con clase *Standard*, pero este incrementa su complejidad cuando se pretende descargar un objeto de tipo *Glacier* o *Deep Archive*. Tal y como se ha introducido a lo largo del informe, resulta de interés para el proyecto trabajar con las diferentes clases para ajustarse a los diversos niveles de servicio que se pretende definir, utilizando un tipo de clase u otro en función de los requisitos del nivel. Es por este motivo que la función *get* se debe ajustar a las necesidades del sistema y, por lo tanto, ser funcional para objetos de cualquier clase.

Tal y como se ha detallado en el apartado de introducción al sistema AWS, cuando se quiere realizar una copia de un objeto con una clase diferente a *Standard* se debe efectuar un proceso de restauración previo. Esto implica que es necesario controlar el estado de los objetos antes de mandar una petición de copia, ya que en caso de desear realizar esta operación con datos almacenados de tipo *Glacier* o *Deep Archive* no se podrá completar si este no ha pasado y finalizado el proceso de restauración. Este control y proceso se realiza mediante la función "copyObjet()".

```
def doGET(filePath, uri):
    try:
        bucketName, pnfsID = getURI(uri)
        copyObject(bucketName, pnfsID, filePath)
        client.download_file(bucketName, pnfsID, filePath)
    except Exception, e:
        print("Copy failed: ", e)
        sys.exit(2)
```

Fig. 6: Código de la función de descarga desde AWS S3.

El algoritmo de la función de copia de un dato alojado en AWS S3, es decir, de la función "doGET()" mostrada en la Fig. 6, con la gestión del proceso de restauración a controlar explicado es el siguiente:

1. El primer paso es verificar el *StorageClass* del objeto mediante la función *head_object* de boto3, la cual devuelve un objeto JSON con una serie de parámetros que definen las propiedades del objeto. En caso de que este no sea *Standard* se debe efectuar un proceso de restauración para poder realizar la copia.

2. Si la función que recoge la información del objeto indica que este es *Standard*, se procede a realizar la copia de forma inmediata. En cambio, si se trata de otro tipo de clase se verifica si ya se ha realizado un proceso de restauración.
   La restauración de un objeto implica que pasará a ser de tipo *Standard* durante el tiempo que se especifique, así que si la comprobación anterior es afirmativa significa que se puede proceder a realizar la copia en cuanto este haya acabado el proceso y, por lo tanto, cambie de clase temporalmente.
   Se realiza esta verificación por si simultáneamente el

objeto ha entrado en el proceso de restauración de forma paralela, evitando así realizar otra petición de restauración en un objeto que ya se encuentra en este mismo estado.

3. En caso de que el objeto deseado no presente ningún proceso de restauración existente, se procede a realizar una petición de esta mediante la función de boto3 *restore_object*, indicando durante cuántos días se quiere disponer del objeto restaurado y el *tier* de esta.

4. El siguiente paso se trata de una comprobación periódica del estado de la restauración cada cierto tiempo, que acaba en el momento en el que el proceso de restauración ha finalizado.

5. Finalmente, se procede a copiar el dato deseado.

Una vez el proceso de copia finaliza el objeto queda restaurado durante el tiempo indicado en la petición de restauración. En el diagrama de flujo de la Fig. 7 se puede ver el algoritmo descrito:

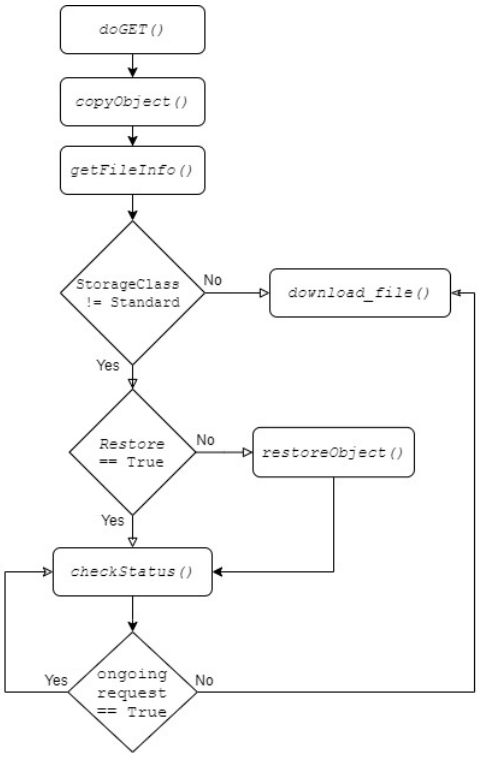

Fig. 7: Diagrama de flujo de la función “doGET()”.

6.4. Función *REMOVE*

Por último, mediante la función *REMOVE* se busca eliminar del sistema de almacenamiento *cloud* aquellos ficheros que se eliminen del directorio. La acción se define dentro de la función “doREMOVE()”.

En caso de eliminar un fichero del directorio se llama al ejecutable con operación *remove* y mediante el uso de la función *delete_object* de boto3 se elimina el objeto del *bucket* y su registro correspondiente en la base de datos de dCache. Como se puede apreciar en la Fig. 8, la cual muestra el funcionamiento descrito en su respectiva función, se destaca la ausencia del control de excepciones y, por lo tanto, del código de retorno en caso de fallida. Esto se debe a la naturaleza de la función utilizada de boto3, la cual independientemente del resultado de la ejecución siempre devuelve una salida con código de éxito. Aún así, debido a que el funcionamiento básico es correcto, se ha decidido ajustar esta función y finalizar el control de la ejecución más adelante y dar preferencia a otras con mayor grado de prioridad.

```
def doREMOVE(uri):
    bucketName, pnfsID = getURI(uri)
    client.delete_object(Bucket = bucketName, Key = pnfsID)
```

Fig. 8: Código de la función de eliminación en AWS S3.

7 GESTIÓN DE EVENTOS

Llegados a este punto, ya se ha desarrollado el sistema básico y, por lo tanto, alcanzado el objetivo principal del proyecto. Aún así la finalidad del proyecto no es únicamente lograr una interfaz de integración entre dCache y AWS S3, sino también dotar al sistema de la capacidad de realizar de forma automatizada el movimiento de datos.

Con el fin de lograr las características mencionadas, se inician las tareas respecto la gestión de eventos. Esta gestión se pretende abordar mediante el sistema de notificaciones de dCache y las funciones AWS Lambda. Como esta parte del desarrollo se ha realizado de forma paralela a la integración entre el sistema en la nube y el físico, se ha empezado trabajando con las funciones AWS Lambda. El motivo principal de esta decisión es debido a que con estas se pueden realizar pruebas totalmente independientes al desarrollo de la tarea principal todavía incompleta.

7.1. Funciones AWS Lambda

Se ha planteado un pequeño proyecto a desarrollar con el objetivo de adquirir conocimientos básicos sobre la gestión de eventos mediante el uso de funciones AWS Lambda. El comportamiento es el siguiente:

A partir de la escritura de ficheros a un *bucket* de S3, se debe crear una copia automatizada mediante AWS Lambda en un segundo *bucket* S3, con *storage class Deep Archive*. Además cuando se especifique que se quiere realizar una lectura de los ficheros tipo *Deep Archive* del *bucket*, se debe notificar vía mail cuando todos los ficheros hayan finalizado su restauración y se encuentren disponibles. El proyecto se puede explicar de forma visual en la Fig. 9:

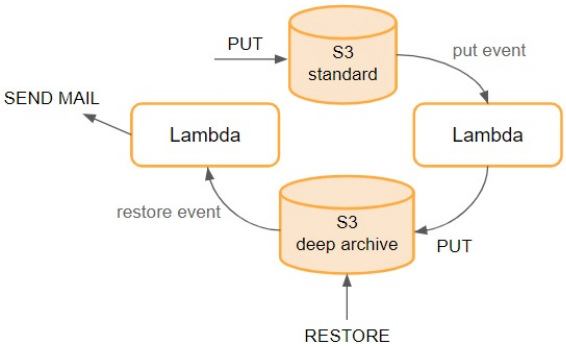

Fig. 9: Esquema del proyecto de eventos.

El primer paso es establecer los eventos que se quieren captar desde la configuración de los distintos *buckets* de trabajo. En este caso se dispone de un primer *bucket* para los eventos *PUT*, y un segundo *bucket* para los eventos *RESTORE*. A continuación, con las funciones Lambda se crean las funciones deseadas para cada situación.

La primera función escucha el *bucket* "pic-standard-tfg" y por cada evento *PUT* captado crea una copia del objeto en el *bucket* "pic-deeparchive-tfg" con clase *Deep Archive*. La segunda escucha el *bucket* "pic-deeparchive-tfg" y por cada evento *RESTORE* captado manda un correo a la dirección establecida. Esta parte del sistema ha quedado incompleta puesto que la finalidad era notificar cuando todos los objetos se encontrasen disponibles, en cambio, tal y como se ha desarrollado, se envía un mensaje por cada operación de *restore*. No se ha podido finalizar debido a que no se ha encontrado la forma de gestionar varias operaciones con una única acción, es decir, no se ha identificado la manera de enviar una única notificación cuando todos los objetos se encuentren disponibles.

Con el proyecto planteado se ha podido alcanzar un nivel básico de aprendizaje sobre la gestión de eventos. Durante el desarrollo de este se ha conocido una nueva herramienta llamada *Batch Operations* [22] la cual parecía simplificar la tarea de administración de los ficheros de un mismo *bucket*, mostrándose como una solución a la problemática encontrada sobre las operaciones masivas.

7.2. Batch Operations

En la gestión de los *buckets* se puede especificar que se cree un inventario diariamente, lo que genera un archivo *manifest*, el cual consta de unos ficheros JSON y CSV. Este lista los ficheros que se encuentran en el *bucket*, así como características especificadas (Tamaño, tipo de almacenamiento, etc). Si se configura un inventario para que genere el *manifest* con el nombre del fichero y el *bucket* al que pertenece, se puede utilizar para crear un *job* desde el servicio de *Batch Operations* que realice cierta operación: Copia, invocación de una función Lambda, reemplazar todos los *tags* de un objeto o restauración.

Se ha probado el uso de las *Batch Operations* con la operación de restauración para la parte de la restauración masiva, pero no se han conseguido los resultados esperados y se ha decidido cerrar esta sección de trabajo en este punto.

8 Resultados

Mediante las diferentes pruebas realizadas se ha podido verificar el correcto funcionamiento de la interfaz de integración de los sistemas.

Name	Last modified	Size	Storage class
00000495CD8 ...	May 18, 2020 3:38:55 PM GMT+0200	4.0 B	Glacier
000064804C6 ...	May 18, 2020 3:39:56 PM GMT+0200	5.0 B	Glacier Deep Archive
0000D6E667D ...	May 18, 2020 1:45:54 PM GMT+0200	100.0 KB	Standard

Fig. 10: Contenido del bucket con tres operaciones de escritura.

Está preparada para trabajar con objetos de AWS S3 de cualquier clase tal y como se puede ver en la Fig. 10, la cual muestra el contenido del *bucket* sobre el cual se han realizado tres operaciones de escritura con las diferentes clases, *Standard, Glacier* y *Deep Archive*, desde el directorio de trabajo local.

También se ha validado el comportamiento de las funciones que gestionan la lectura de un fichero, tanto con objetos *Standard* que no requieren de restauración como con otros que sí la necesitan. Debido a que lo que se quería confirmar era el correcto funcionamiento del ejecutable y a causa del alto tiempo de restauración que implica el tipo *Deep Archive*, estas pruebas se han realizado con objetos *Glacier*.

La Fig. 11 muestra los tiempos de ejecución de dos tareas programadas. El primero conlleva el proceso de escritura de cien objetos de 10 GB con clase *Glacier* y ha finalizado después de aproximadamente cuatro horas. El segundo, que implica el proceso de restauración para la posterior lectura de esos cien objetos, ha finalizado después de más de cuarenta horas.

time_crontabPut.txt

```
real    249m56.075s
user    0m7.437s
sys     42m44.073s
```

time_crontabGet.txt

```
real    2433m37.607s
user    0m18.607s
sys     5m10.643s
```

Fig. 11: Tiempo de ejecución de los Crontabs.

Respecto el trabajo realizado con las funciones Lambda para automatizar el movimiento de datos, las pruebas han sido unitarias ya que no se ha realizado ninguna integración con el sistema principal. En la Fig. 12, se pueden ver los *logs* resultantes de la ejecución de la función de copia:

```
START RequestId: 7709acbf-26a4-4418-ac26-989b85b7e838 Version: $LATEST
PUT file name = test.txt
PUT bucket name = pic-standard-tfg
SOURCE bucket name = pic-deeparchive-tfg
COPY COMPLETED: FILE test.txt FROM BUCKET pic-standard-tfg
COPIED TO BUCKET pic-deeparchive-tfg
END RequestId: 7709acbf-26a4-4418-ac26-989b85b7e838
REPORT RequestId: 7709acbf-26a4-4418-ac26-989b85b7e838
Duration: 1832.80 ms Billed Duration: 1900 ms
```

Fig. 12: Logs de la primera función Lambda.

Así mismo, la Fig. 13 muestra el mensaje recibido gracias a la función de notificación. A pesar de que la implementación de esta función se encuentra incompleta, ya que tal y como se ha expuesto durante el apartado de gestión de eventos no se ha conseguido hacer una correcta gestión de funciones masivas, se ha alcanzado el comportamiento básico deseado.

alba.vendrellm@e-campus.uab.cat a través de amazonses.com
para mí

inglés > español Traducir mensaje

Your operation has been completed!

Fig. 13: Notificación recibida por la segunda función Lambda.

Por lo tanto, el resultado del proyecto es el que expone la Fig. 14. El esquema final varía del inicial planteado, el cual

se encuentra en el anexo A.1 en la Fig. 15, debido a la falta de integración del sistema de eventos con el principal.

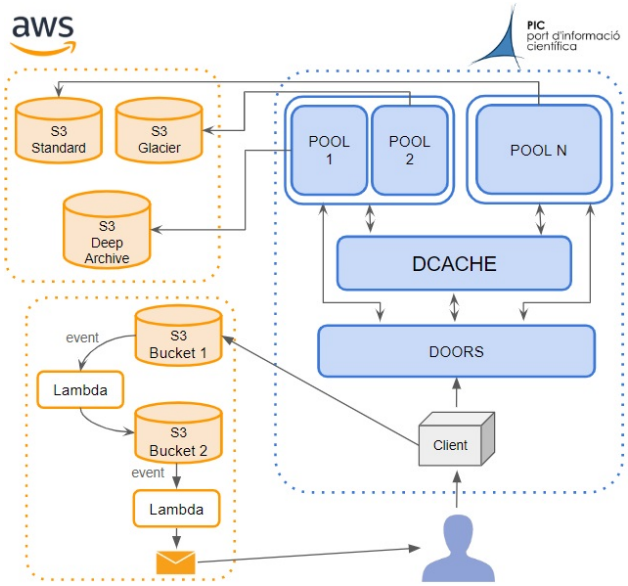

Fig. 14: Esquema final del proyecto.

Los costes del almacenamiento mensual de 1 TB de datos con las diferentes tecnologías que ofrece el centro y AWS S3 son los que se muestran en la Tabla 2. Cabe detallar que los tres cálculos del almacenamiento *cloud* se han realizado con 100 ficheros. Este dato es destacable ya que el total está calculado en función del coste de almacenamiento base y el coste por cantidad de peticiones que se realizan. En este caso al coste base de almacenar 1 TB se le añade el de las 100 operaciones de escritura.

TABLA 2: COSTES DE ALMACENAMIENTO

	PIC		AWS S3		
Almacenamiento	*Disco*	*Cinta*	*Standard*	*Glacier*	*Deep Archive*
1 TB / mes	7,85 €	2,95 €	20,99 €	3,40 €	0,91 €

Conociendo las prestaciones de cada uno, los servicios que compiten entre ellos son el almacenamiento en disco con el tipo *Standard*, y el almacenamiento en cinta con el tipo *Glacier* y *Deep Archive*.

Como se puede ver en la tabla, resulta especialmente beneficioso hacer uso del sistema *cloud* con objetos tipo *Deep Archive* cuando se busca tener segundas copias por un largo periodo de tiempo. Sin embargo, *Glacier* también ofrece un precio realmente competitivo a pesar de ser un 11,25 % más caro que el almacenamiento en cinta con el ejemplo expuesto. Esta es una opción válida que permite ahorrar la dificultad del proceso administrativo que representa comprar cintas, instalarlas en el robot y añadirlas al sistema de Enstore. Al mismo tiempo se debe tener en cuenta la cantidad de operaciones que se desean realizar, ya que este coste extra por operación no está añadido en los servicios que ofrece el PIC, así que con mayor cantidad de peticiones, los valores de 3,40 € y 0,91 € se verían incrementados. Respecto los ficheros que se quieran almacenar con una alta disponibilidad, resulta evidente que actualmente la mejor opción es el uso del disco físico, ya que el precio base de almacenamiento de los objetos *Standard* es muy superior y, además, se encuentra condicionado por la cantidad de lecturas o escrituras realizado.

Por los diversos motivos expuestos, el único *tier* que se ha valorado establecer hasta el momento es el que almacena objetos tipo *Deep Archive*, los cuales se planteen como copias de seguridad y se considere que no requerirán de ninguna lectura en un largo periodo de tiempo. Sin embargo no se descartan el resto de clases y se seguirá analizando el sistema y herramientas para determinar las situaciones donde sus usos sean beneficiosos.

9 CONCLUSIONES

A partir de los cuatro objetivos principales se ha desarrollado el proyecto de forma modularizada, implementando una interfaz de integración entre el sistema de almacenamiento físico dCache y el sistema en la nube AWS S3. Las pruebas realizadas y los resultados obtenidos muestran el cumplimiento del objetivo principal del proyecto, confirmando así la obtención del sistema híbrido de almacenamiento de datos.

Este sistema ofrece una gran ventaja de flexibilidad e inmediatez al centro, ya que se presenta como una alternativa al largo y complejo proceso administrativo cuando se pretende crecer de forma física. Al mismo tiempo, destacar que la implementación del ejecutable, basada en los requisitos del sistema físico, lo vuelve fácilmente compatible con otros sistemas de almacenamiento *cloud* mediante cambios a las llamadas de sus APIs.

No obstante se pueden identificar una serie de objetivos que no han llegado al estado deseado, como se expone en el apartado de líneas de trabajo futuro. Sin embargo, los objetivos principales restantes se han alcanzado mediante dos sistemas independientes; el sistema híbrido y la gestión de eventos. Estos quedan pendientes de unificar y se ha optado por enfocar esta tarea hacia el trabajo pendiente.

A causa de la reducción del mercado de la tecnología en cinta en los últimos años, existe una gran incertidumbre frente al futuro coste de cinta o incluso su existencia y, por lo tanto, un gran riesgo del futuro tecnológico del mercado. Por otro lado, si en algún momento sucediera un desastre en el edificio o el propio robot, sería una situación donde los datos se encontrarían realmente comprometidos. El sistema híbrido logrado ofrece la posibilidad de gestionar los riesgos tanto de migración como de recuperación expuestos.

10 LÍNEAS DE TRABAJO FUTURO

Tal y como se ha ejecutado el proyecto, se identifican un total de tres líneas de futuro trabajo.

Queda pendiente ajustar el ejecutable para el control de excepciones en la operación *remove*. Esto implica verificar antes de realizar la petición de eliminación si el objeto en cuestión existe en el *bucket* especificado. De esta forma se

podrá controlar el código de retorno en caso de fallo.

Por otra parte, también se debe retomar la gestión de eventos, probando las notificaciones de eventos de dCache y perfeccionando el proyecto creado mediante las funciones AWS Lambda. Esta segunda, la cual ha tenido un gran peso en el proyecto, se debe integrar con el sistema híbrido una vez se encuentre finalizada, lo que implicará el alcance del sistema final con una automatización total del movimiento de datos. Para acabar de perfilar esta automatización mediante la popularidad de acceso de los ficheros, se debe volver a valorar y replanificar de forma futura la tarea de consulta de la base de datos de Elasticsearch.

Finalmente, se va a probar el proyecto mediante la aplicación en un caso de uso real con el experimento MAGIC [23], el cual ha identificado un *dataset* muy importante que necesita preservar por un largo periodo de tiempo. Ha pedido al centro realizar dobles copias de seguridad de todo el conjunto de datos, por lo que se plantea ofrecer este servicio mediante el sistema desarrollado con objetos almacenados tipo *Deep Archive*.

AGRADECIMIENTOS

En primer lugar, quiero agradecer al PIC la oportunidad de realizar este proyecto bajo su tutela, con especial mención a mi tutor Jordi Casals por su acompañamiento y apoyo a lo largo del mismo. En segundo lugar agradecer a Elena Planas, Vanessa Acín y Gonzalo Merino su tiempo y cooperación, así como al resto de personas del centro que me han ayudado con el trabajo.

En segundo lugar, a mi tutor académico Eduardo Cesar Galobardes por haberme motivado y brindado sus constantes recomendaciones a lo largo de todo el grado y durante el transcurso del proyecto.

REFERENCIAS

[1] DATA STORAGE MEDIA, *Fujifilms. [En línea]*, https://www.fujifilm.com/products/recording_media/data_storage. (visitado 25-06-2020).

[2] AMAZON WEB SERVICES DOCUMENTATION, *AWS. [En línea]*, https://docs.aws.amazon.com/. (visitado 12-06-2020).

[3] AZURE DOCUMENTATION, *Microsoft Azure. [En línea]*, https://docs.microsoft.com/en-us/azure/?product=featured/. (visitado 26-04-2020).

[4] GET STARTED WITH GOOGLE CLOUD, *Google Cloud Platform. [En línea]*, https://cloud.google.com/docs. (visitado 26-04-2020).

[5] THE SCIENCE CLOUD, *Helix Nebula Science Cloud. [En línea]*, https://www.hnscicloud.eu. (visitado 26-04-2020).

[6] COMPUTING WITH HTCONDOR™, *HTCondor. [En línea]*, https://research.cs.wisc.edu/htcondor. (visitado 26-04-2020).

[7] AWS S3 DOCUMENTATION, *AWS. [En línea]*, https://docs.aws.amazon.com/s3/index.html. (visitado 20-06-2020).

[8] THE DCACHE BOOK, *dCache. [En línea]*, https://www.dcache.org/manuals/book.shtml. (visitado 12-06-2020).

[9] BOTO 3 DOCUMENTATION, *AWS. [En línea]*, https://boto3.amazonaws.com/v1/documentation/api/latest/index.html. (visitado 20-06-2020).

[10] AWS LAMBDA DOCUMENTATION, *AWS. [En línea]*, https://docs.aws.amazon.com/lambda/index.html. (visitado 12-06-2020).

[11] WLCG QOS WORKSHOP, *CERN. [En línea]*, https://indico.cern.ch/event/873367. (visitado 26-04-2020).

[12] MAIN PAGE, *Enstore System. [En línea]*, https://www-stken.fnal.gov/enstore/enstore_system.html. (visitado 26-04-2020).

[13] CEPH DOCUMENTATION, *Ceph File System. [En línea]*, https://ceph.readthedocs.io/en/latest. (visitado 26-04-2020).

[14] NAGIOS DOCUMENTATION, *Nagios. [En línea]*, https://www.nagios.org/documentation. (visitado 26-04-2020).

[15] GRAFANA DOCUMENTATION, *Grafana. [En línea]*, https://grafana.com/docs/grafana/latest. (visitado 26-04-2020).

[16] AMAZON CLOUDWATCH DOCUMENTATION, *AWS. [En línea]*, https://docs.aws.amazon.com/cloudwatch/index.html. (visitado 26-06-2020).

[17] USE S3 AS AN HSM FOR DCACHE, *GitHub kofemann/s3hsm. [En línea]*, https://github.com/kofemann/s3hsm. (visitado 28-05-2020).

[18] JUPYTER NOTEBOOK DOCUMENTATION, *Jupyter. [En línea]*, https://jupyter.org/documentation. (visitado 25-06-2020).

[19] AMAZON ELASTIC COMPUTE CLOUD, *AWS. [En línea]*, https://docs.aws.amazon.com/ec2/index.html. (visitado 26-04-2020).

[20] CHAPTER 8: THE DCACHE TERTIARY STORAGE SYSTEM INTERFACE, *dCache. [En línea]*, https://www.dcache.org/manuals/Book-6.2/config-hsm.shtml. (visitado 25-06-2020).

[21] USE S3 AS AN HSM FOR DCACHE: FLUSH FILE TO S3, *GitHub kofemann/s3hsm. [En línea]*, https://github.com/kofemann/s3hsm#flush-file-to-s3. (visitado 26-04-2020).

[22] S3 BATCH OPERATIONS DOCUMENTATION, *AWS. [En línea]*, https://docs.aws.amazon.com/AmazonS3/latest/user-guide/batch-ops.html. (visitado 12-06-2020).

[23] TELESCOPIO MAGIC, *Wikipedia. [En línea]*, https://es.m.wikipedia.org/wiki/Telescopio_MAGIC. (visitado 28-06-2020).

ANEXO

A.1. Figuras

TABLA 3: PLANIFICACIÓN DEL PROYECTO

Detalle de la tarea	*Duración*	*Objetivo relacionado*
Familiarización con el entorno de AWS.	5h	O.E. 1
Definir los diferentes niveles de servicio de AWS S3 que se querrán aplicar.	10h	O.P. 3
Introducción a dCache e investigación de los posibles sistemas de integración entre el sistema local y S3.	10h	O.E. 2 O.E. 3
Creación de la interfaz de conexión entre dCache y AWS S3.	65h	O.P. 1 O.P. 2
Iniciación al sistema de notificaciones de eventos de dCache.	15h	O.E. 4
Iniciación a la gestión de eventos mediante AWS Lambda.	15h	O.E.4
Realizar copias simples basadas en las notificaciones de dCache y en eventos usando AWS Lambda.	35h	O.P. 2
Pruebas de conexión entre Ceph y S3 usando Rados Gateway.	20h	O.E. 2
Aplicación en un caso de uso real.	10h	O.C. 1
Consultar la BBDD de Elasticsearch y calcular la popularidad de los ficheros para automatizar el proceso de decisión.	10h	O.P. 4
Evaluación del sistema: Realizar pruebas de rendimiento, comparativa de usabilidad, de coste y de sencillez de implementación entre ambos sistemas.	65h	O.C. 1
Cierre del proyecto: Pasar el sistema a producción y documentar.	10h	O.C. 2

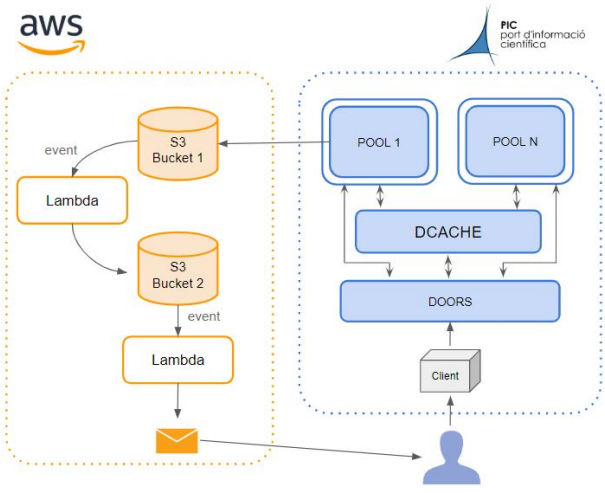

Fig. 15: Esquema inicial del proyecto.