Implementación de un sistema de subida de archivos para Garlanet

José Quiroga Pérez
Máster universitario en Ingeniería Informática

Joan Manuel Marquès Puig y Félix Freitag

Enero 2019

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

A mi mujer Erica, a mi hija Arancha,

Agradecimientos

Tras la elaboración de todo el desarrollo necesario para llevar a buen puerto la realización de este trabajo, del que llevo invirtiendo tiempo desde hace ya más de seis meses, quiero agradecer los esfuerzos del tutor del proyecto Joan Manuel Marquès Puig, por su completa implicación, su asesoramiento, su ayuda directa con algunas partes del proyecto, su disponibilidad para resolver dudas a cualquier hora, sus llamadas telefónicas para explicar los puntos del proyecto, y su buen humor para realizar todas estas tareas, y al trabajo del tutor Félix Freitag para todo lo relacionado con esta memoria y las presentaciones del trabajo, las cuales no serían lo que son sin sus buenos consejos.

¡Gracias sinceras a ambos!

José Quiroga Pérez

Murcia, 5 de enero de 2019.

FICHA DEL TRABAJO FINAL

Título del trabajo:	Implementación de un sistema de subida de archivos para Garlanet
Nombre del autor:	José Quiroga Pérez
Nombre del consultor:	Joan Manuel Marquès Puig / Félix Freitag
Fecha de entrega (mm/aaaa):	01/2019
Área del Trabajo Final:	Sistemas distribuidos
Titulación:	*Máster universitario en ingeniería informática*

Resumen del Trabajo (máximo 250 palabras):

Garlanet es una red social de microblogging distribuida que proporciona privacidad fuerte a los participantes. Dentro de las necesidades de la red social se incluye la que aquí se desarrolla, concretamente la necesidad de añadir a la red social la posibilidad de compartir de una manera análoga a la de los mensajes, archivos de diversa índole, por tanto la finalidad de este Trabajo Final de Máster es la de implementar un sistema de subidas de ficheros, principalmente imágenes, aunque se dotará al proyecto de reutilización para que sea sencillo añadir diversos tipos de ficheros.

En primer lugar vamos a definir los tipos de datos necesarios para procesar los ficheros que se mandan a Garlanet, concretamente hay 2 partes bien diferenciadas en este proyecto; la parte de la aplicación, donde el fichero será seleccionado, fraccionado en bloques, cifrado, y posteriormente subido a través de un microservicio, y dicho microservicio, del que se va a implementar una api que nos va a dar la funcionalidad requerida.

Se va a generar la funcionalidad necesaria para que los ficheros sean replicados en los microservicios de réplica, donde se va a decidir qué política de escritura se va a seguir (aunque aparentemente QUORUM, no obstante se debatirá si debe ser un valor parametrizable o podemos ser más o menos flexibles con esta cuestión).

También vamos a establecer una política de eliminación de los ficheros, los que no se hayan completado de subir, y probablemente otra política de eliminación de ficheros por tiempo.

Abstract (in English, 250 words or less):

Garlanet is a distributed microblogging social network that provides total privacy to the participants. Within the needs of the social network includes the one developed here, specifically the need to add to the social network the possibility of sharing in a similar way to the messages, files of various kinds, therefore the purpose of this TFM is to implement a system for upload files, mainly images, although the project will be reused to make it easy to add different types of files.

First of all we are going to define the types of data necessary to process the files that are sent to Garlanet, specifically there are 2 well differentiated parts in this project; the application side, where the file will be selected, split into blocks, encrypted, and then uploaded through a microservice, and the Microservice side, with the development of a API, that will give us the required functionality.

The necessary functionality will be developed so that the files are replicated in the replication microservices, where the writing policy is going to be decided (although apparently QUORUM, however it will be debated whether it should be a parameterizable value or we can be more or less flexible with this question).

We are also going to establish a policy for the elimination of files, those that have not been completed, and probably another policy for deleting files by time.

Palabras clave (entre 4 y 8):

Garlanet, Social network, Repository, Sharing, Distributed network

Índice

1. Introducción 1
1.1 Descripción del proyecto 1
1.4 Planificación del Trabajo 2
1.4.1 Estructura de desglose de trabajo (EDT) 2
1.4.2 Diagrama de Gantt del proyecto 3
1.5 Breve sumario de productos obtenidos 5
1.6 Breve descripción de los otros capítulos de la memoria 5
2. Implementación de la Metainformación 6
2.1 Análisis de la situación actual y diseño de la solución 7
2.2 Implementación de la solución 9
2.3 Conjunto de pruebas e Integración del código en Garlanet 16
3. Implementación del microservicio Files 21
3.1 Análisis de la situación actual y diseño de la solución 21
3.2 Implementación de la solución 24
3.3 Conjunto de pruebas e Integración del código en Garlanet 32
4. Control de consistencia 40
4.1 Análisis de la situación actual y diseño de la solución 40
4.2 Implementación de la solución 42
4.3 Conjunto de pruebas e Integración del código en Garlanet 45
5. Mejoras pendientes 51
6. Conclusiones Finales 52
7. Glosario 53
8. Bibliografía 56
9. Anexos 57
9.1 Tabla de contenido generado y modificado 57
9.2 MetaInformation.java 58
9.3 MetaInformation.serialize 59
9.4 MetaInformation.deserialize 61
9.5 FileUtils.SplitIntoBlockFile 62
9.6 FileUtils.getBlocks 64
9.7 FileUtils.getMetaInformationFromBlocks 65
9.8 FileUtils.getObjectFromBlocks 65
9.9 FileDescriptionUserInterface 66
9.10 FileDescription 67
9.11 Application.post 68
9.12 Repository.handleFiles 69
9.13 PurgeTimer.timeout 70

Lista de figuras

Figura 1: Estructura del desglose de trabajos 3
Figura 2: Relación de tareas del diagrama de Gantt 4
Figura 3: Diagrama de Gantt del proyecto 4
Figura 4: Bytes desglosados de la metainformación 8
Figura 5: Contenido de la interfaz EnumToByteToEnum 10
Figura 6: Contenido del enumerado EFileType 10
Figura 7: Contenido de EFileSubType 10
Figura 8: GUI de la interfaz de pruebas 15
Figura 9: Subida de un fichero a la GUI de Test 17
Figura 10: Imagen recuperada desde los bloques 18
Figura 11: Imagen de tamaño superior a un bloque... 18
Figura 12: Proyectos que forman Garlanet en GitLab 19
Figura 13: Rama files del proyecto GarlanetCore en Eclipse 19
Figura 14: Resultado del Push de la rama Files con la Fase 1 20
Figura 15: Diagrama de flujo del microservicio Messages 22
Figura 16: Diagrama de flujo de la operación Post... 24
Figura 17: Aspecto gráfico actual de la GUI... 25
Figura 18: Botón para subir ficheros a la GUI 25
Figura 19: Selector de imágenes de la GUI 26
Figura 20: Diagrama de comunicación de la interfaz 27
Figura 21: Manejador Files en la clase Repository... 29
Figura 22: Tabla del servicio FILES 30
Figura 23: Ejemplo de operación de actualización en la tabla. 31
Figura 24: Preview de imagen en la GUI.. 31
Figura 25: Opciones para el fichero 32
Figura 26: Explorador de ficheros guardando una imagen............................... 32
Figura 27: Registro de usuario de pruebas 32
Figura 28: Ejecución detenida en el Repositorio 33
Figura 29: Fichero subido tras forzar un error. 33
Figura 30: Envío concurrente de imágenes de dos usuarios 35
Figura 31: Imagen de gran tamaño para prueba 35
Figura 32: Preview de imagen pesada desde GUI_swing 36
Figura 33: Imagen del punto de ruptura en Application 37
Figura 34: Parada de uno de los repositorios... 38
Figura 35: Imagen recuperada tras la caída de un repositorio 38
Figura 36: Imagen enviada tras eliminar dos repositorios. 39
Figura 37: Actividad registrada en el portal de GitLab de la UOC. 39
Figura 38: Diagrama de flujo de la operación de purgado............................... 41
Figura 39: Diagrama de flujo del control de consistencia 42
Figura 40: Constructor del objeto PurgeItem.. 43
Figura 41: Constructor del PurgeTimer 43
Figura 42: Sesiones de consistencia de los microservicios.............................. 44
Figura 43: Timer que recupera los bloques que faltan 45
Figura 44: Imágenes de distintos tamaños... 45
Figura 45: Prueba de integridad con 2 repositorios.. 47
Figura 46: Ejecución de dos repositorios 47

Figura 47: Contenido en el nuevo repositorio. 48
Figura 48: Nuevas imágenes para el caso 4. 49

1. Introducción

1.1 Descripción del proyecto

Garlanet es una red social de microblogging distribuida que proporciona una privacidad fuerte a los participantes además de hacer partícipe de su red distribuida a todo el usuario que lo desee. Dentro de las necesidades de la red social se incluye la que aquí se desarrolla, concretamente la necesidad de añadir a la red social la posibilidad de compartir de una manera análoga a la de los mensajes, archivos de diversa índole.
En primer lugar vamos a definir los tipos de datos necesarios para procesar los ficheros que se mandan a Garlanet, concretamente hay 3 fases diferenciadas en este proyecto:

1. La parte de la aplicación, donde el fichero será seleccionado, se extraerá la *metainformación* del propio fichero, tras lo que será fraccionado en bloques, cifrado, y posteriormente subido a través de un microservicio.

2. Como segunda parte, como se implementa dicho microservicio, del que se va a desarrollar una api que nos va a dar la funcionalidad requerida. En dicha api el fichero se enviará a los usuarios receptores y se subirá a los nodos réplica de Garlanet.

3. Por último, se debe implementar una purga para los ficheros que no sean subidos de forma correcta, y además se implementará unas sesiones de consistencia para que exista integridad entre los repositorios.

Para la correcta implementación de todo ello se va a seguir el documento de análisis elaborado por el tutor Joan Manuel Marquès Puig, "*Files Microservice",* siendo una importante base del desarrollo del proyecto, en conjunción con la tutorización del mencionado profesor.

1.2 Justificación del proyecto

Dado lo atrayente del proyecto y su envergadura, Garlanet requiere aún de numerosas funcionalidades para que comience su andadura. Una red social que no depende de servidores estancos y que garantiza mediante un cifrado criptográfico que solo el emisor y los receptores autorizados van a ser los conocedores de los mensajes bien justifica hacer un desarrollo que aumente sus posibilidades, en este caso con la implementación del sistema de subida de ficheros.

1.3 Alcance y objetivos del proyecto

Una vez implementada la funcionalidad del proyecto, el alcance se extiende a cualquier persona que quiera hacer uso de Garlanet a modo de usuario, para

poder subir ficheros (en principio ficheros de tipo imagen y ficheros de tipo url) y a cualquier voluntario que quiera instalar un repositorio de Garlanet para que en dicho repositorio se alojen los ficheros subidos, por tanto, un alcance global a todos los usuarios y voluntarios de la red.

Los objetivos del proyecto son la culminación de toda la funcionalidad necesaria para que la subida de ficheros en Garlanet sea implementada, en el caso de que dicha implementación exceda el tiempo total de trabajo de un TFM, un objetivo realista es terminar con las dos fases mencionadas en la descripción del proyecto realizando una batería de test en cada implementación para verificar su correcta implementación, incluyendo un pequeño programa de prueba para verificar el mecanismo que se va a implementar en la GUI para la subida de ficheros.

1.4 Planificación del Trabajo

1.4.1 Estructura de desglose de trabajo (EDT)

Dentro de la metodología de gestión de proyectos del PMBOK [1], la estructura del Desglose del Trabajo (EDT) es una descomposición jerárquica, orientada al producto entregable del trabajo que será ejecutado por el equipo del proyecto, para lograr los objetivos del proyecto y crear los productos entregables requeridos.
El logro de los objetivos del proyecto requiere de una EDT que defina todos los esfuerzos requeridos, la asignación de las responsabilidades a un elemento definido de la organización y que a partir de la EDT se establezca un cronograma y presupuesto adecuado para la realización de los trabajos.
Basándonos en la información recopilada del PMBOK y en la propuesta del índice vamos a establecer la EDT:

Fase	Tarea	Horas	%Proy
0	0.1 Estructura de desglose de trabajo (EDT)	5	1.66%
	0.2 Cronograma del proyecto	5	1.66%
	0.3 Instalación de Garlanet de los repositorios y montaje del entorno	3	1%
1	1.1 Análisis de la solución	10	3.33%
	1.2 Creación de clases auxiliares	10	3.33%
	1.3 Creación del formato de metainformación	12	4%
	1.4 Implementación de la serialización	11	3.66%
	1.5 Implementación de la deserialización	10	3.33%
	1.6 Creación de los bloques de metainformación	25	8.33%
	1.7 Creación de la interfaz de pruebas	8	2.66%
	1.8 Conjunto de pruebas validadoras del desarrollo	10	3.33%
	1.9 Integración del código en Garlanet	8	2.66%
2	2.1 Análisis de la solución	10	3.33%
	2.2 Implementación en la GUI de la subida de ficheros	15	5%
	2.3 Adaptación de la clase GarlanetUserInterface	11	3.66%
	2.4 Creación de la interfaz Files para la subida de	20	6.66%

	ficheros		
	2.5 Coherencia de los repositorios/política de réplica	15	5%
	2.6 Cifrado/Descifrado de la información	15	5%
	2.7 Conjunto de pruebas validadoras del desarrollo	10	3.33%
	2.8 Integración del código en Garlanet	8	2.66%
3	3.1 Análisis de la solución	10	3.33%
	3.2 Elaboración de los métodos para la purga de bloques	10	3.33%
	3.3 Controles de consistencia entre repositorios	10	3.33%
	3.4 Conjunto de test validadores del desarrollo	8	2.66%
	3.5 Integración del código en Garlanet	6	2%
4	4.1 Elaboración de la memoria	25	8.33%
	4.2 Presentación	10	3.33%
TOTAL		300	100%

Figura 1: Estructura del desglose de trabajos

Aunque en esencia estas van a ser con toda seguridad las tareas del proyecto es posible que alguna se termine denominando de otra forma o se aglutinen dentro de la misma tarea. También cabe la posibilidad de que alguna tarea de duración cuantiosa se divida en 2 tareas menos pesadas, aunque de momento la decisión es dejarlas así. Para una mayor claridad nombramos las fases como **fase 0 "Introducción", fase 1 "Implementación de la Metainformación", fase 2 "Implementación del microservicio Files" y fase 3 "Control de consistencia" .**

1.4.2 Diagrama de Gantt del proyecto

En la Figura 1 podemos observar la relación de tareas añadidas al Gantt con sus correspondientes fechas de inicio y de fin. Para su elaboración se ha usado la herramienta GanttProject y se ha tomado como referencia el valor 1 día = 8 horas.

Nombre	Fecha de inicio	Fecha de fin
Fase 0	1/10/18	3/10/18
0.1 Estructura de desglose de trabajo EDT	1/10/18	1/10/18
0.2 Cronograma del proyecto	2/10/18	2/10/18
0.3 Instalación de Garlanet de los repositorios...	3/10/18	3/10/18
Fase 1	4/10/18	29/10/18
1.1 Análisis de la solución	4/10/18	5/10/18
1.2 Creación de clases auxiliares	8/10/18	9/10/18
1.3 Creación del formato de metainformación	10/10/18	11/10/18
1.4 Implementación de la serialización	12/10/18	15/10/18
1.5 Implementación de la deserialización	16/10/18	17/10/18
1.6 Creación de los bloques de metainformac...	18/10/18	23/10/18
1.7 Creación de la interfaz de pruebas	24/10/18	24/10/18
1.8 Conjunto de pruebas validadoras del desa...	25/10/18	26/10/18
1.9 Integración del código en Garlanet	29/10/18	29/10/18
Fase 2	30/10/18	20/11/18
2.1 Análisis de la solución	30/10/18	31/10/18
2.2 Implementación en la GUI de la subida de...	1/11/18	2/11/18
2.3 Adaptación de la clase GarlanetUserInterf...	5/11/18	6/11/18
2.4 Creación de la interfaz Files para API de su...	7/11/18	9/11/18
2.5 Coherencia de los repositorios/política de...	12/11/18	13/11/18
2.6 Cifrado/Descifrado de la información	14/11/18	15/11/18
2.7 Conjunto de pruebas validadoras del desa...	16/11/18	19/11/18
2.8 Integración del código a Garlanet	20/11/18	20/11/18
Fase 3	21/11/18	29/11/18
3.1 Análisis de la solución	21/11/18	22/11/18
3.2 Elaboración de los métodos para la purga ...	23/11/18	26/11/18
3.3 Controles de consistencia entre repositorios	27/11/18	27/11/18
3.4 Conjunto de test validadores del desarrollo	28/11/18	28/11/18
3.5 Integración del código en Garlanet	29/11/18	29/11/18
Fase 4	30/11/18	7/12/18
4.1 Elaboración de la memoria	30/11/18	5/12/18
4.2 Presentación	6/12/18	7/12/18

Figura 2: Relación de tareas del diagrama de Gantt

A continuación podemos ver el diagrama de Gantt. Como solo hay una persona elaborando el proyecto, todas las tareas son secuenciales y una tarea no puede comenzar hasta que se terminen las anteriores.

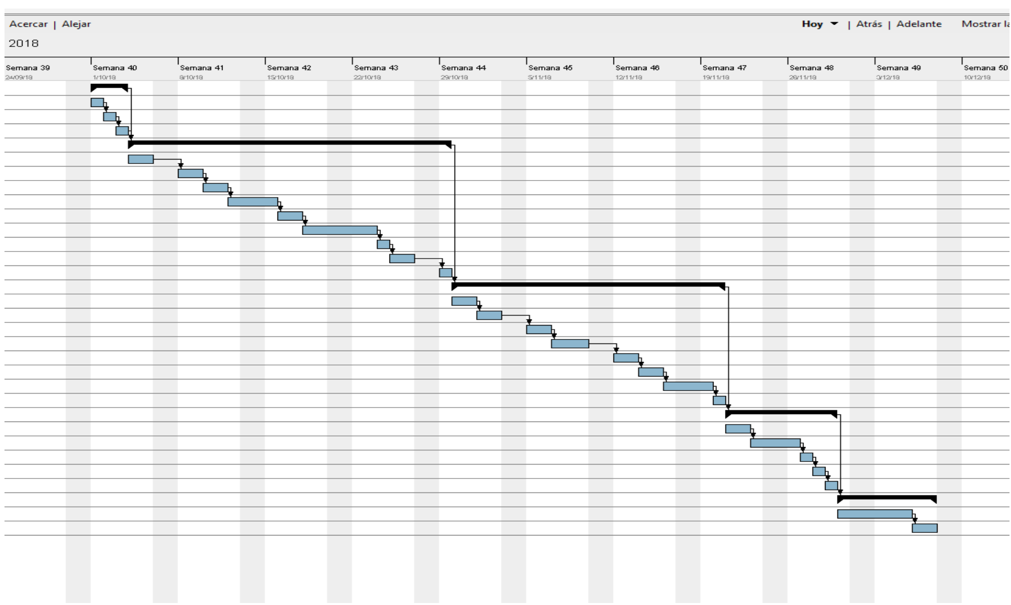


Figura 3: Diagrama de Gantt del proyecto

1.5 Breve sumario de productos obtenidos

Al final del desarrollo de este TFM se va a obtener una nueva versión del código de Garlanet que afecta a varios de los módulos que lo componen, por citar brevemente a alguno de ellos, el módulo *Application*, *GarlanetCore*, *Repository*, o *GUI*_swing, serán modificados para permitir la subida de ficheros de tipo imagen y de tipo enlace. Esta nueva compilación deberá ser fusionada con la rama principal del producto una vez esté verificada y tendrá como resultado el proyecto con la nueva funcionalidad.

1.6 Breve descripción de los otros capítulos de la memoria

A continuación detallaremos en el capítulo 2 y subsiguientes las fases del proyecto, la elaboración del código y la generación de test y de pruebas para validar los resultados obtenidos, para ir quemando etapas hasta la entrega final donde se espera que el proyecto esté claramente finalizado.

2. Implementación de la Metainformación

La primera parte del proyecto consiste en implementar la arquitectura necesaria para generar la metainformación y construir los bloques que se envían a los repositorios.

A vista de pájaro, esta parte del proyecto realiza la extracción de los metadatos de un fichero, y fracciona en bloques la metainformación y los ficheros, para luego volver a recuperarla.

Actualmente Garlanet consta de un sistema de mensajería entre un emisor y distintos receptores, de los cuales el emisor no es consciente, por lo que en teoría un emisor no sabe a cuantos destinatarios les está llegando su mensaje. En parte la privacidad fuerte de Garlanet consiste en esto (y en el cifrado de cada mensaje). Este mecanismo de envío de mensajes debe permanecer en el envío de ficheros, pero además, necesitamos ofuscar la información que se envía de un punto a otro y es manejada por los repositorios. En este caso la información debería ser convertida en bloques de información de igual tamaño, y posteriormente, cifrarla, para que un atacante no pudiera obtener información de estos archivos, y mantener la confidencialidad en la comunicación.

Esta cuestión plantea un problema, ya que una vez divididos los ficheros en bloques de igual tamaño puede ser imposible recuperar el fichero original a tenor de cuestiones como, ¿qué tipo de fichero era? ¿Cuánto ocupaba? ¿Cuál era el nombre original del fichero? ¿Tenía una vista previa?. Es decir, esta forma de procesar la información plantea una serie de cuestiones a la hora de recomponer el fichero y volver a presentarlo tal cual se mandó originalmente para que el receptor pueda recuperarlo de una manera íntegra.

La respuesta a este problema la dan los metadatos asociados al fichero, de aquí en adelante *Metainformación*. Vamos a generar, por cada fichero, unos datos que van a contener aspectos relevantes del mismo y que nos van a permitir restaurarlo con su información original. Dado que no solo consiste esta tarea en generar metadatos a partir de un fichero, si no en extraer dicha información, convertirla en *bytes*, en bloques, reconvertirla en *bytes*, volver a obtener la *Metainformación* y a partir de ella, recuperar los ficheros, es por si sola una tarea con la envergadura suficiente para que sea la primera fase del proyecto.

Además de todo esto, definiremos el tamaño de bloque y definiremos también los tipos de datos que puede manejar Garlanet, para que en un futuro se permitan otros tipos de ficheros y el sistema sea lo suficientemente flexible para que las clases generadas sean reutilizadas.

Una vez realizado el análisis requerido para Garlanet, elaboraré un software de pruebas que servirá para validar el desarrollo realizado y continuar con las siguientes fases.

2.1 Análisis de la situación actual y diseño de la solución

Garlanet carece de la infraestructura necesaria para obtener la metainformación de los archivos que quieran ser subidos, por lo que es necesaria la creación de la estructura que soporte el manejo de la metainformación y el fraccionamiento en bloques de la misma. También es necesario a partir de los bloques, obtener primero la metainformación y a partir de ella, recuperar la imagen original. Esto constituye la primera fase del proyecto.

Tomando como base el documento previo de análisis del tutor Joan Manuel Marquès [2] definimos los campos necesarios para definir la clase MetaInformation y su justificación:

Datos correspondientes al fichero:

- Nombre: Este campo almacenará el nombre original del fichero.

- Tipo: En este campo guardaremos el tipo de dato MIME del archivo. Para el caso especial de un enlace, habrá que definir el tipo especial de Garlanet URL, donde indicaremos que se trata de un enlace.

- Texto alternativo: Este dato contendrá un nombre o descripción del tipo de archivo alternativo al nombre con el que se subió.

- Subtipo: Este campo va a concretar la información aportada por el campo tipo, donde se especificará que tipo de fichero es, por ejemplo en el caso de ser un MIME IMAGE se especificará que es JPEG, PNG, etc…

- Preview: Para el caso de ficheros que permitan tener una pre visualización, en este caso por ejemplo, una imagen reducida o un icono de reducidas dimensiones.

- URL: En el caso de que el tipo de archivo sea un enlace, adjuntaremos el enlace propiamente dicho aquí.

Datos correspondientes a los bloques:

- Timestamp_firstblock: Dados los requisitos de seguridad que debemos mantener, este campo indicará el sello de tiempo al que corresponde el primer bloque del fichero al que hace referencia esta Metainformación. Es decir, una vez que la información esté serializada, separada en bloques de igual tamaño y cifrada, este campo indicará en la lista de bloques, cual es el primer bloque correspondiente al fichero.

- First_byte_first_block: Este campo va a indicar dentro de un bloque, cual es el primer byte que corresponde al fichero propiamente dicho.

- Last_byte_last_block: Este campo indica cual es el último byte que corresponde al fichero.

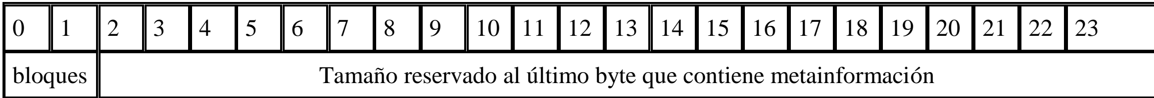
- Num_blocks: Número de bloques que corresponden al fichero.

Estos campos definen aproximadamente como debe ser la clase MetaInformation. Según como definimos el bloque, que es un conjunto de bytes de determinado tamaño, que finalmente será cifrado es posible que no podamos extraer de los bloques la información relativa a la Metainformación, ya que son bytes y hay campos de longitud variable. Además cabe la posibilidad de que la Metainformación ocupe más de un bloque, con lo que tendríamos otro problema asociado, la imposibilidad de recuperar la Metainformación incluye no poder recuperar el fichero.

Para poder extraer la información ocupe uno o varios bloques vamos a definir una estructura que va a ir en los primeros 3 bytes del primer bloque de un listado de bloques que contiene un archivo y su Metainformación asociada.

Los dos primeros bits van a corresponder al número de bloques de la lista de bloques que contienen Metainformación. Por tanto tenemos un número máximo de bloques que puede ocupar la Metainformación: 4.

Los subsiguientes 22 bits restantes que restan de los 3 bytes que reservamos para la recuperación de la Metainformación, van a indicar el último byte que contiene este tipo de dato. Con esta especificación tenemos que el tamaño máximo de bloque con esta limitación no puede exceder 2^{22} = 4,19 Mb.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
bloques		Tamaño reservado al último byte que contiene metainformación																					

Figura 4: Bytes desglosados de la metainformación

Con todo esto, se hace necesario que la clase incluya la interfaz *Serializable* para su conversión en *array* de *bytes*, además, se hace necesaria la operación contraria, *Deserializar*, para recuperar la información transformada en bytes a metainformación. Por tanto, tenemos para esta clase tanto los atributos como dos métodos necesarios para su implementación. Además surgen otras cuestiones como añadir caracteres al final de cada variable de tipo *string* para delimitar de alguna manera su tamaño.

Además, hay que definir el tamaño máximo de cada bloque, que va a definir tanto el tamaño máximo de los bloques de metainformación como el tamaño máximo de los bloques del archivo.

Finalmente, para las pruebas, se hace necesario generar un método *toString* adecuado para mostrar la información que se genera como metainformación y que sufre las transformaciones en array de bytes ya mencionada.

También hay que especificar las clases que van a contener la información sobre los archivos que se están generando. La mejor forma de modelarlo es usar el método de la clase Files (Java7) cuya firma es *Files.probeContentType(Path path);* Este método nos va a proporcionar el tipo mime del archivo, el cual tenemos que separarlo en tipo y subtipo una vez obtenida la respuesta. Para almacenar esta información vamos a emplear dos enumerados generados a la vez que la clase MetaInformation, uno definirá el tipo de archivo y el otro el subtipo al que corresponde, además aprovecharemos la clase tipo para incluir el tipo de archivo URL, sin subtipo.

Ya tenemos definidas la estructura de la clase MetaInformation, y los enumerados *EFileType* y *EFileSubtype*, e incluimos la necesidad de una clase que reciba la metainformación y un fichero y lo transforme a bloques. Esta clase va a tener métodos estáticos que controlen el fraccionamiento de los bloques y que añadan la información necesaria para su recuperación, y que puedan recuperar a partir de bloques los objetos originales.

Vamos a denominar a esta clase como *SplitBlockFileUtility*, para estos dos métodos vamos a usar la siguiente firma: *splitIntoBlockFile(byte[] metaInformation)* que va a convertir un objeto MetaInformation en byte[], (con su correspondiente sobrecarga que va a recibir un objeto en vez de un array de bytes, *getMetaInformationFromBlocks(byte[] blocks)* que realiza la operación inversa, *getBlocks(MetaInformation metaInformation, File file)* realiza la conversión a bloques de un objeto MetaInformation y de su fichero, realizando, dentro del método, la asignación de valores cruciales para la recuperación de ambos objetos como son, el firstbyte_firstblock, num_blocks, lastbyte_lastblock, y tendrá en cuenta el tamaño máximo de bloque que será definido en las variables globales de la aplicación.

Por motivos de espacio no se va a incluir la implementación de los métodos salvo en los casos en los que la explicación lo requiera, además, se van a eliminar los comentarios para la memoria para facilitar la legibilidad del código ya que cada método relevante va a ser explicado de forma pertinente. La implementación del código va a ser realizada bajo el IDE de desarrollo *Eclipse* en su versión *Oxygen*.

2.2 Implementación de la solución

Consultar el anexo 9.1 y sucesivos para una consulta más completa del código fuente.

Enumerados

Como ya definimos en el análisis, tenemos que generar dos enumerados correspondientes a los datos Mime que se recuperan tal y como definimos en el

análisis. Además tenemos que añadir en el tipo principal el tipo URL que no existe como formato Mime. Se añade una interfaz a ambas clases que va a servir para que implementen métodos de conversión a *byte* y viceversa, que vamos a necesitar en las *serializaciones* de los objetos

```
package edu.uoc.garlanet.files.metainformation;

public interface EnumToByteToEnum {

    public byte getByte();

    public EnumToByteToEnum getEnum(byte b);


}
```

Figura 5: Contenido de la interfaz EnumToByteToEnum

Esta clase contiene dos métodos que van a servir para traducir el enumerado a byte y a recuperarlo, de forma que nos servirá para que el objeto MetaInformation tenga menos trabajo a la hora de convertir los tipos y subtipos a byte.

```
public enum EFileType implements EnumToByteToEnum {
    APPLICATION, AUDIO, MULTIPART, TEXT, IMAGE, URL;

    public byte getByte() {

    public EnumToByteToEnum getEnum(byte b) {
    /**
     * Returns a EFileType from a string type description
     */
    public static EFileType getEFileType(String name) {

    public static short size(){
        return Byte.SIZE/Byte.SIZE;
    }
```

Figura 6: Contenido del enumerado EFileType

En este enumerado hemos definido perfectamente la estructura que necesitamos para indicar en un objeto MetaInformation que tipo de archivo contiene, basándonos en el formato Mime. Además incluimos los métodos *getEfileType* y *size* que nos van a servir para recuperar un *EFileType* de su *string* correspondiente, y el tamaño en bytes que va a ocupar dicho enumerado.

```
public enum EFileSubType implements EnumToByteToEnum{
    JAVASCRIPT, JSON,XWWWFORMURLENCODED,XML,ZIP,PDF,SQL,GRAPHQL,LDJSON,
    MSWORD,MPEG,VORBIS,FORMDATA,CSS,HTML, CSV,PLAIN,PNG,JPEG,GIF,VNDOPENXMLFORMATSOFFICEDOCUMENTWORDPROCESSINGMLDOCUMENT,VNDMSEXCEL,
    VNDOPENXMLFORMATSOFFICEDOCUMENTSPREADSHEETMLSHEET,VNDMSPOWERPOINT,VNDOPENXMLFORMATSOFFICEDOCUMENTPRESENTATIONMLPRESENTATION,
    VNDOASISOPENDOCUMENTTEXT;

    public byte getByte() {

    public EnumToByteToEnum getEnum(byte b) {
    /**
     * Retrieves a EFileSubType from a string
     * @param name
     * @return
     */
    public static EFileSubType getEFileSubType(String name) {

    public static short size(){
        return Byte.SIZE/Byte.SIZE;
    }

}
```

Figura 7: Contenido de EFileSubType

Como podemos ver, el contenido de esta clase expande los tipos definidos en el anterior enumerado. Como decisión de implementación se suprimen los "_" y los "-" incluidos en el tipo original Mime, además de los "." que usa el formato

para indicar algunos subtipos. El enumerado contiene los mismos métodos con la misma firma, y su función es análoga a la de los tipos principales.

La metainformación

Según la especificación elaborada en el diseño, tenemos los atributos definidos, vamos a implantarlos tal cual y vamos a seguir la política de usar los métodos *get* y *set* de la clase para establecerlos. Además no vamos a seguir la notación *lowerCamelCase* para las variables (pero sí para los métodos), por lo que seguiremos la *notación C* (solo para variables).

Al finalizar la implementación de la clase tenemos un código bastante completo con diversos constructores, el constructor sin variables inicializa el objeto estableciendo determinados valores a las variables para evitar excepciones posteriores y una correcta inicialización de las mismas.

Los constructores que contienen parámetros son necesarios para establecer la metainformación de forma correcta, si bien se podría haber realizado una implementación basándonos en herencia (para los distintos tipos de metainformación que puede haber, ya que hay distintos tipos de archivos) se ha considerado implementarlo finalmente en un único tipo de objeto.

Una vez decidido esto, se incluyen los atributos del objeto, incluyendo además, como una constante, el delimitador *STRING_DELIMITER* con el valor *'\0'* de fin de cadena. Este atributo nos va a indicar el final de cada campo de tipo *string,* que va a servir para el proceso de *serializado* y *deserializado.*

En cuanto a los métodos auxiliares que contiene la clase, *getLongToBytes, getIntToBytes, getBytesToLong y getBytesToInt* van a servir para el tratamiento de los tipos *long* e *Int* a la hora de la *serialización y deserialización*. En cuanto al método redefinido *toString()* se ha optado por realizarlo de forma manual para darle un formato más adecuado a la hora de representar los datos en consola que el que proporciona el generador automático de código de *Eclipse*, para poder mostrar la información del objeto de una forma más entendible.

La implementación de la serialización

Serializar es convertir en un *array* de *bytes* el objeto de metainformación. El orden en el que se van a ir convirtiendo los atributos de la clase en bytes es crucial para fases posteriores, por lo que se va a tomar la decisión de convertir primero los atributos de tamaño fijo y dejar en las últimas posiciones los campos cuyo tamaño es variable. Esta decisión de diseño va a tener su importancia a la hora de establecer el *timestamp_firstblock* que va a indicar dentro de la *metainformación* el bloque en el que comienza a haber información del fichero propiamente dicha, y que no podemos saber hasta el proceso de convertir en bloques, es decir, información que obtenemos solo cuando ya tenemos los bloques construidos, por lo que para la eficiencia de la aplicación y para no repetir operaciones que pueden resultar en un mal rendimiento de la aplicación, reservamos los 8 primeros bytes del objeto serializado con el campo ya mencionado.

Una vez tomada la decisión de implementación con respecto a la situación de los atributos dentro del objeto serializado, pasamos a su implantación en código. Como la operación puede incluir tipos de ficheros distintos, serializamos primero los campos comunes e incluimos un switch que haga la distinción por tipo de archivo (por ejemplo, si es una URL contendrá un campo que indique que url es, y si es una imagen o cualquier otro tipo de archivo, tendrá un campo subtipo, por lo que esta distinción es necesaria).

Como podemos ver la implementación del método es compleja por lo que vamos a explicar los pasos más relevantes del proceso:

Primero, declaramos como *bytes* todos los campos del objeto, incluyendo los que no vayamos a necesitar, dependiendo de qué tipo de archivo estamos tratando. Estas variables almacenarán el contenido de los valores reales, pero transformados a *array* de *bytes* dependiendo de cómo sea el dato original.

Segundo, hay que ir convirtiendo a *bytes* todos los datos comunes, esto es, el nombre, el texto alternativo, el tipo, el *timestamp*, etc... Para los datos que contienen *strings* se asume una codificación “UTF-8” aunque el fallo está tratado para que si no es así, se convierta igualmente como *byte* sin especificar la codificación de la cadena. En este proceso los atributos específicos de imagen o de URL están tratados de forma específica en un *switch,* como por ejemplo, la url o la *preview* de un archivo.

Tercero, reservamos el *byte array,* que tendrá un tamaño distinto dependiendo del tipo de archivo del que forme parte la metainformación.

Y por último, vamos a ir almacenando los datos en un orden específico dentro del vector de *bytes* para poder almacenar de forma posterior el *timestamp*. Para guardar los valores en el resultado usamos la función de *Java System.arraycopy*. El orden es *timestamp, num_blocks, firstbyte_firstblock, lastbyte_lastblock, num_blocks, name y alternative_text.* Los atributos no comunes se almacenan tras estos y se añaden tras otro *switch.* Una vez

realizada esta operación tenemos un *array* de *bytes* con la información *serializada.*

La implementación de la deserialización

Como operación inversa a la *serialización*, esta operación tiene como finalidad convertir un *array de bytes* en un objeto *MetaInformation* y devolverlo. El orden de las operaciones tiene que ser el mismo que en la operación contraria, por lo que nos encontramos con que los primeros campos a tratar son los primeros campos convertidos en la función de *serializado*.

La recuperación de la información se realiza asignando a un objeto vacío de *MetaInformation* los valores que vamos recuperando, donde podemos hacer dos distinciones relevantes, la recuperación de valores que sabemos que contienen un tamaño determinado, y las variables cuyo contenido es variable.

En los casos en los que las variables tienen un valor determinado aplicamos el tamaño al vector y obtenemos directamente el valor, el cual asignamos al nuevo objeto que estamos construyendo. Para los valores variables usamos el delimitador declarado como constante en la clase *MetaInformation* para recuperar esos campos. Debido a la complejidad de la tarea se han usado índices y variables auxiliares para el correcto tratamiento de la información.

Al ser una función inversa al método *serialize* no es necesario aclarar el orden de las operaciones ni los condicionales, en esencia las variables se recuperan en el mismo orden y los condicionales cumplen la misma función, discriminar por tipo de archivo y recuperar la información pertinente.

Los bloques de metainformación

Como habíamos definido en el análisis, es necesaria la implementación de una clase que gestione la creación de los bloques. Debido a la complejidad inherente a la operación, la construcción de los bloques la he separado en dos operaciones, la construcción de bloques que incluye solo a la *metainformación*, y a la construcción de bloques que incluye *metainformación* y el propio archivo. Además cada operación, al igual que sucede con la *serialización* debe tener su operación inversa, es decir, que a partir de bloques sea capaz de reconstruir los objetos originales.

De toda la fase es junto con la ya mencionada *serialización* la operación más relevante de toda la fase y una de las más determinantes del TFM, así que vamos a ver estas operaciones en detalle comentando los pormenores y los aspectos necesarios para que quede claro.

La clave de esta función son los 3 primeros bytes que recordemos, reservábamos para la operación inversa, por lo que los dos primeros bits van a ocupar el número de bloques que va a ocupar la *metainformación.* Los siguientes 22 bits restantes de los 3 bytes que ocupa la cabecera, es la posición del último byte que contiene *metainformación*. La forma de calcular este último byte es calcular cuántos bloques se van a necesitar para la *metainformación* y recuperar hasta cuánto va a ser necesario del bloque. Esta

operación se debe transformar primero de *int* a *string* (pasando de entero a una cadena que contiene la representación binaria del número), juntar ambas representaciones en una única *string* y posteriormente a *array* de *bytes*, donde ocuparán las 3 primeras posiciones del primer bloque. Una vez hecho esto es ir asignando la *metainformación* a los bloques correspondientes, teniendo en cuenta que los 3 primeros bytes del primer bloque ya están ocupados.

Para la realización de esta operación hemos hecho uso del método privado *completeBit* que completa hasta 2 o 22 bits (dependiendo de si es el bloque o el último *byte*) para la cabecera. Cabe reseñar además, el uso de la constante *ConfigurationGarlanetCore.BLOCK_SIZE* del proyecto *GarlanetCore* que define el tamaño máximo de bloque.

El método *getBlocks* comienza con el establecimiento de varias variables que antes no han podido ser introducidas en la *metainformación* como son, el número de bloques que contienen información del fichero y el primer y último *byte* del fichero. Para el cálculo de esta información necesitamos conocer cuál es el número de bloques que ocupa la *metainformación,* que calculamos al principio (serializando el objeto *MetaInformation* y obteniendo su longitud), de esa manera obtendremos el *firstbyte_firstblock.* Para saber cuántos bloques ocupará el fichero debemos realizar la conversión del fichero a *array* de *bytes,* teniendo en cuenta que en el último bloque de la *metainformación* tenemos también parte del fichero, para obtener el dato *num_blocks*. Por último, calculamos a partir de esos datos el *lastbyte_lastblock.*

Una vez establecidos estos datos en el objeto *MetaInformation*, obtenemos los bloques del mismo con la operación *splitIntoBlockFile* y posteriormente gracias a la información que hemos recuperado, guardamos en los siguientes bloques el fichero. El resultado es un *List<byte []>* es decir, una lista de bloques cuyo tamaño es el definido por la constante *BLOCK_SIZE.* Para completar el último bloque si el fichero no lo llena entero, se rellena con bytes aleatorios.

Las operaciones inversas son *getMetaInformationFromBlocks* y *getObjectFromBlocks y* recuperan la metainformación a partir de los bloques y el fichero a partir de los bloques, haciendo uso la segunda función de la primera para ello.

Creación de una interfaz de pruebas

Los requisitos de la aplicación requieren de la creación de una interfaz donde probar aspectos relevantes del desarrollo, como por ejemplo, la generación de la *metainformación* a partir de ficheros reales, la comprobación de que estos objetos se *serializan* de forma correcta, o recuperar a partir de bloques los objetos originales con los que se generaron.

Esta interfaz de pruebas va a contener dos clases java: *TestFormularioMetaInformation* y *TestMetaInformation,* la primera contiene un pequeño formulario realizado con Java *swing* para el test de los ficheros, y la segunda va a contener un método *main* donde previamente vamos a realizar

pruebas estáticas con objetos *MetaInformation* simples para validar el desarrollo de forma simple.

El propósito de estas pruebas es validar el desarrollo realizado hasta ahora y corregir alguna deficiencia si existiese, lo cual veremos en el siguiente apartado.

El objetivo de esta clase es realizar pequeñas pruebas por consola y generar la que es realmente la interfaz de pruebas de la primera fase del TFM. Contiene un método *main* para ser lanzada como aplicación Java y simplificar el conjunto de pruebas.

El resultado de la interfaz gráfica de este formulario se puede ver en la siguiente figura.

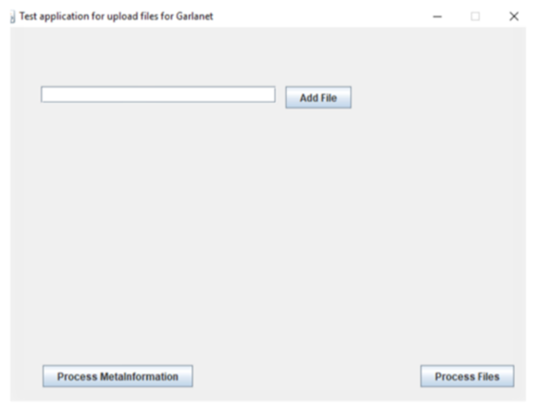


Figura 8: GUI de la interfaz de pruebas

Como comentábamos, tenemos 3 botones en la interfaz, vamos a comentar brevemente la funcionalidad de cada uno de ellos:

- El botón "A*dd File"* añade ficheros a la lista de ficheros que declarábamos en la clase. Es la única funcionalidad que va a tener este botón, por lo que no vamos a desarrollar el comportamiento del mismo.

- El botón "*Process Files"* va a generar de cada uno de los ficheros añadidos mediante el botón anterior, la *metainformación* de cada fichero y la va a añadir a la lista de metainformación. Adicionalmente va a probar además a generar los bloques correspondientes a una *MetaInformación* y a un fichero.

- El botón "*Process MetaInformation*" va a generar pruebas de toda la metainformación *serializándola y deserializandola*, y mostrando el contenido de la misma antes y después de cada operación, para que se muestre la información generada y la obtenida tras la operación y compararlas.

Por tanto, el modo de empleo de la interfaz de pruebas es:

1. Añadir tantos ficheros como se deseen.
2. Procesar los ficheros.
3. Procesar la *Metainformación*.
4. Comprobar los resultados en consola.

Código reseñable del *listener* del botón *Process Files:*

Parte de la información interesante de este método y que posteriormente incluiremos en la GUI de Garlanet es el siguiente:

```
String mime = Files.probeContentType(source);
String[] parts = mime.split("/");
eFileType = EFileType.getEFileType(parts[0]);
eFileSubType = EFileSubType.getEFileSubType(parts[1]);
```

De esta manera obtenemos del fichero el tipo y el subtipo necesarios para generar la *metainformación*. Hay que tener en cuenta que de momento solo se permiten imágenes. Las url se van a generar desde la propia GUI por lo que las probaremos desde consola.

Tras obtener los tipos y los subtipos crearemos objetos *MetaInformation* que vamos a ir añadiendo a la lista que ya mencionamos anteriormente.

2.3 Conjunto de pruebas e Integración del código en Garlanet

Para realizar la batería de pruebas del desarrollo realizado vamos a definir cuáles son los resultados que esperamos obtener y cómo vamos a obtenerlos. Disponemos de la herramienta para pruebas que hemos realizado en el apartado anterior, que nos va a proporcionar información sobre los ficheros que subamos y que nos va a confirmar si el proceso de *serialización* y d*eserialización* y generación de bloques se hace de forma correcta. Por tanto, vamos a concretar pruebas y a mostrar los resultados que se obtienen de ellas. Si el resultado es el esperado, daremos por concluida la fase del desarrollo.

1º Subida y procesamiento de un fichero a través de la GUI de Test.

¿Qué se espera de esta prueba? Se espera que el sistema procese un fichero de imagen, extraiga la metainformación, muestre los datos de la metainformación, serialice el objeto, lo deserialice, y vuelva a mostrar los datos.
Resultados de la prueba:

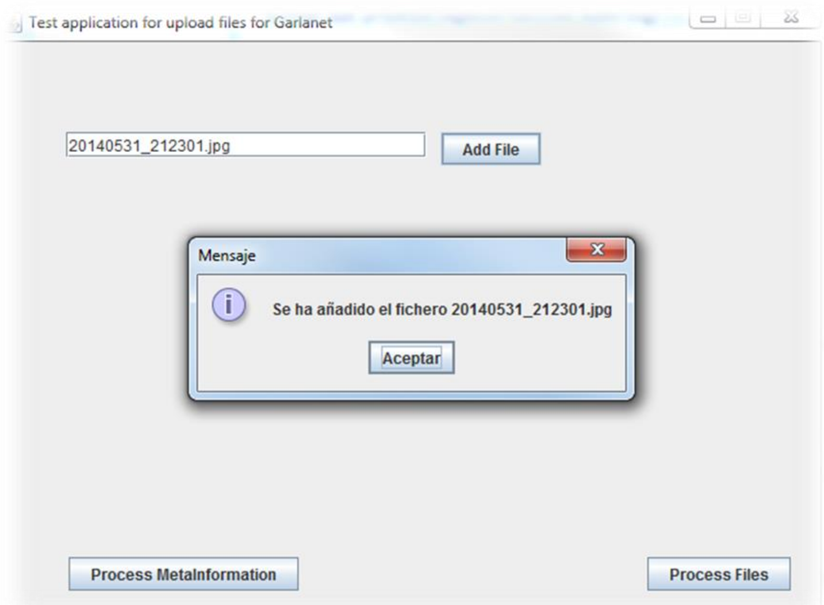


Figura 9: Subida de un fichero a la GUI de Test

Tenemos la metainformación generada a partir del fichero:

MetaInformation: Name: 20140531_212247.jpg
Alternate Text: Imagen jpeg
Timestamp: 11112222
Preview:
Numblocks: 0 Media Type: IMAGE
Media subtype: JPEG First block: 0
Last block: 0

Nota: Los datos de Timestamp y texto alternativo se han establecido en el constructor del objeto.

Información serializada y deserializada:

MetaInformation: Name: 20140531_212247.jpg
Alternate Text: Imagen jpeg
Timestamp: 11112222
Preview:
Numblocks: 0 Media Type: IMAGE
Media subtype: JPEG First block: 0
Last block: 0

Es decir, se recupera la misma información. Ahora vamos a fraccionar el objeto MetaInformation en bloques, junto con el fichero, y vamos a recuperar los objetos originales y vamos a ver el resultado (que vamos a grabar en la unidad C:\ con su nombre original).

MetaInformation: Name: 20140531_212247.jpg Alternate Text: Imagen jpeg
Timestamp: 11112222
Preview:

Numblocks: 1 Media Type: IMAGE
Media subtype: JPEG First block: 59
Last block: 329508

Es decir, se asigna la información al objeto MetaInformation los atributos necesarios para recuperar la imagen, vamos a ver si la imagen se ha recuperado de forma normal.

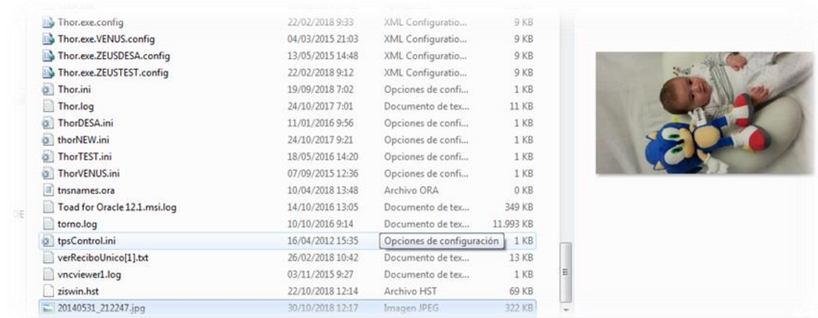


Figura 10: Imagen recuperada desde los bloques

Como podemos ver, la imagen se ha recuperado correctamente desde los bloques y desde el explorador de Windows podemos ver la imagen miniaturizada. Vistos los resultados obtenidos podemos concluir que el sistema es estable y no presenta fallos para un único fichero que ocupa un solo bloque.

2º- Prueba con múltiples ficheros.

En esencia esta prueba es la misma que la anterior pero procesando varios ficheros a la vez. Por ser breves y no ser redundantes, mencionaremos los resultados obtenidos únicamente.

Los resultados para 6 y 20 ficheros añadidos **son exitosos**, se recupera la metainformación de forma correcta.

3º- Prueba con imágenes de tamaño grande.

¿Qué se espera de esta prueba? Con esta prueba buscamos fraccionar el fichero en distintos bloques y ver la estabilidad del sistema en una situación en la que los ficheros ocupan varios bloques.
Resultados de la prueba:

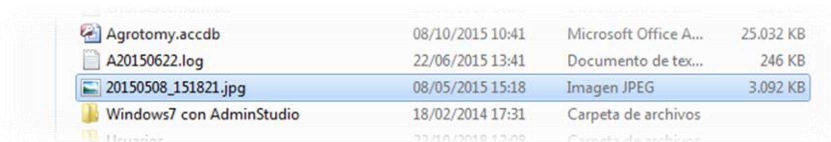


Figura 11: Imagen de tamaño superior a un bloque

La imagen genera la siguiente información tras el serializado y deserializado:

Name: 20150508_151821.jpg Alternate Text: Imagen jpeg
Timestamp: 11112222

Preview:
Numblocks: 7 Media Type: IMAGE
Media subtype: JPEG First block: 59
Last block: 165894

Es decir, genera 7 bloques que después son recuperados. La imagen se recupera de forma estable y se pre visualiza desde el explorador de Windows. Los resultados de la prueba **son satisfactorios.**

Integración del código en Garlanet

Tras el conjunto de pruebas realizadas, y obtenido el visto bueno por parte del tutor del TFM se procede a subir el código a Garlanet, para ello se ha creado en el proyecto, que está alojando en un repositorio *GitLab,* una rama específica para ello, la rama *Files.*

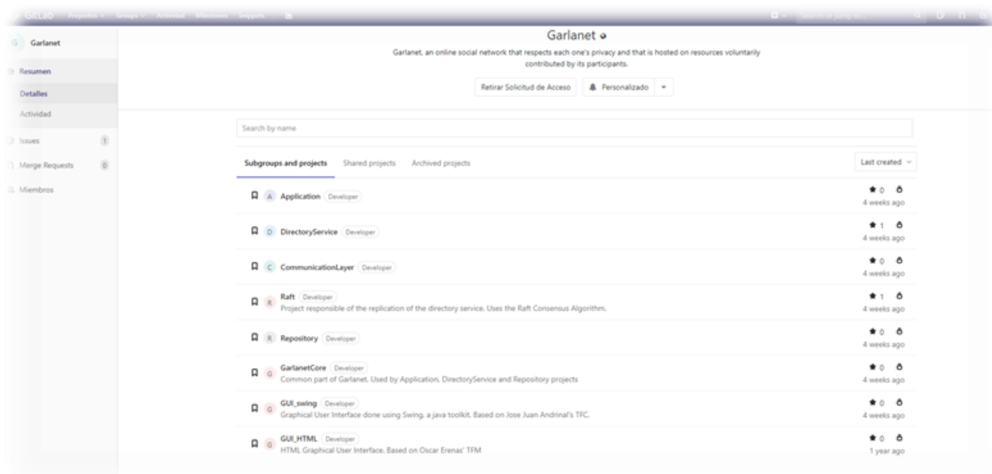


Figura 12: Proyectos que forman Garlanet en GitLab

Cada proyecto que forma Garlanet tiene su propio *tronco* dentro del repositorio, y cada proyecto debe llevar su propia rama, si es necesario modificarlo para ello. En esta fase solo es necesario modificar el código del proyecto *GarlanetCore,* por lo que se solicita la creación de la rama, se clona el proyecto del *tronco* a la rama, y se descarga mediante el *plugin* de Git para *Eclipse.*

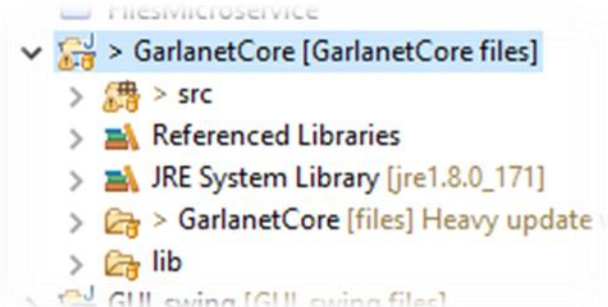


Figura 13: Rama files del proyecto GarlanetCore en Eclipse

Tras la obtención del código del repositorio, creamos dentro de la estructura de carpetas establecida en el proyecto para el código fuente, la carpeta files.*metaInformation,* dentro de src/edu/uoc/Garlanet. Desde ahí copiamos el

código elaborado y realizamos la operación Commit + Push. A partir de ese momento tenemos las modificaciones en nuestro repositorio, dentro de la rama seleccionada.

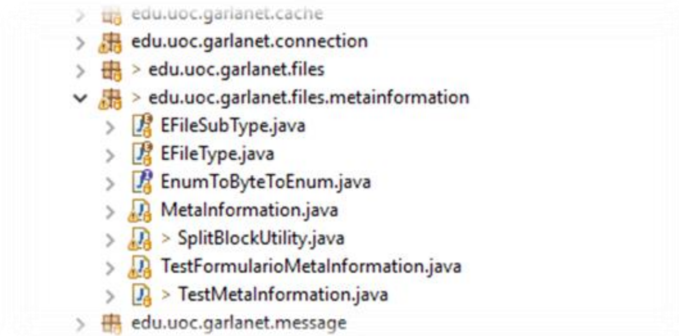


Figura 14: Resultado del Push de la rama Files con la Fase 1

3. Implementación del microservicio Files

A continuación, se debe implementar la lógica necesaria para que desde la interfaz gráfica pueda subirse un fichero que sea procesado en metainformación y en bloques cuando corresponda. Después se distribuya a través de los repositorios y sea recuperado de los mismos. Y por último haga el camino contrario hasta ser presentado a los usuarios a través de la interfaz gráfica y que sea posible su descarga si se requiere.

Mejor explicado, este desarrollo implica poder enviar un archivo desde la GUI a los repositorios, y ser capaz de recuperarlo de los repositorios y que sea presentado al usuario, el cual podrá descargarlo o abrirlo.

Para esta tarea hay que usar los proyectos que conforman Garlanet casi al completo, incluyendo modificaciones en el código de *GarlanetCore, Repository, Application y GUI_swing.*

Parte de la complejidad del desarrollo de esta fase es la integración del nuevo desarrollo con el sistema que ya existe, por lo que hay que reacondicionar las APIs existentes por todo el flujo de los datos para que se adapten a los nuevos métodos que vamos a desarrollar. Además es posible que tengamos que adaptar el paso de mensajes actual para que los ficheros se puedan adjuntar como partes de un mensaje de texto convencional.

Además de todo lo anterior, también hay que establecer algunos detalles como los criterios de réplica en los repositorios (o dejarlo abierto, pero con posibilidad de establecer un dato concreto más adelante), generar las tablas que albergarán los ficheros en los repositorios, su recuperación de los mismos, y su incorporación como vista en la GUI.

En cuanto al peso de la fase, es la fase más importante del proyecto y la consecución de los objetivos depende directamente de esta. Una vez la fase esté terminada se valorará en conjunto si el objetivo del proyecto está cumplido o por el contrario se establecerán unas directrices para terminarlo de forma satisfactoria, indicando los puntos que restan y como concluirlos.

3.1 Análisis de la situación actual y diseño de la solución

Primero vamos a realizar un análisis del flujo de datos actual que ocurre cuando se realiza un envío de mensajes y a que partes del proyecto afecta, que podemos ver en la Figura 15.

Actualmente el dato pasa por la interfaz gráfica (GUI) que es la que recoge el texto a enviar del usuario. Este texto se pasa a la API de comunicación del usuario con la aplicación, denominada *GarlanetUserInterface.* Esta API llama a su vez a la API de aplicación llamada *Application* que es quien invoca al microservicio pertinente, en este caso *Messages,* que consta de dos partes, la parte de la aplicación, y la parte del repositorio. La parte de la aplicación invoca al repositorio, que es quien interpreta el servicio que se está invocando y llama al microservicio para que realice el trabajo. Tanto en el caso de los mensajes

como en el de ficheros hemos de usar persistencia para los datos, por lo que se usa un motor de *SQLite* para ello.

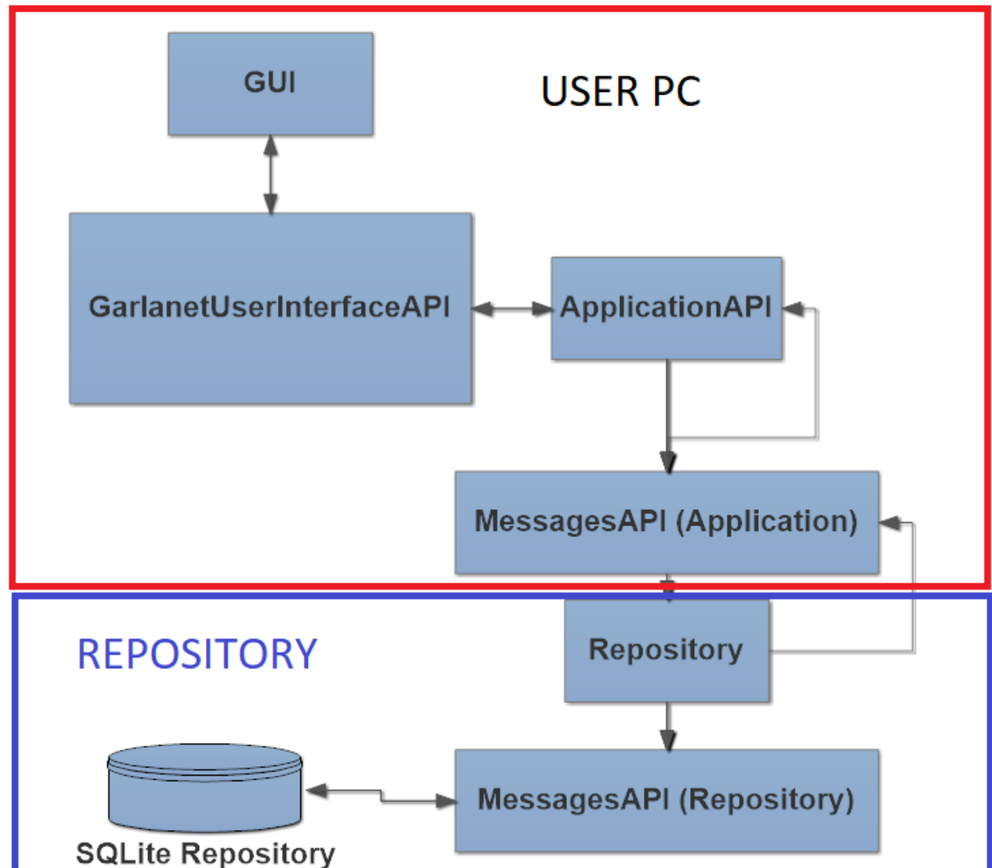


Figura 15: Diagrama de flujo del microservicio Messages

El flujo de datos recorre el camino inverso para mostrar a quien corresponda el mensaje. Como los bloques se guardan en los repositorios, gracias al control de consistencia el contenido se replica en otros repositorios, esta parte la veremos en la tercera fase del proyecto pero es importante señalar su existencia. Además esta persistencia permite tener un control sobre historial de mensajes del usuario y otras ventajas del mismo.

De esta primera vista conceptual extraemos dos partes importantes que van a formar parte del microservicio que va a conformar el envío de ficheros, la API de la aplicación y la API del repositorio. A eso habrá que sumarle el resto de puntos en los que se hace necesaria la adaptación del código para la integración de ambas partes.

Application:

Para esta parte de la API se hacen necesarias cuatro operaciones por las que un fichero va a pasar, ya sea para subida del mismo o para su recepción. La primera operación requiere de una petición de una marca de tiempo para evitar colisiones en los repositorios. Esta operación indicará un código de subida que

solo valdrá para la subida actual, y del número de bloques que se van a enviar. La respuesta debe ser el primer *Timestamp* disponible para ello. Una vez reservadas las filas correspondientes, se hará la reserva en firme de esos bloques en los repositorios. Para ello usaremos una función que hará la reserva y que requerirá del código de subida, del primer *Timestamp* que se había devuelto en la anterior operación y del número de bloques que se van a subir. Esto devolverá un estatus de operación. Para finalizar la operación de subida de ficheros, se requiere de otro método que haga propiamente la subida. Se necesitará de un código de subida, la marca de tiempo, el bloque o bloques, y devolverá un estado de la operación. Todas estas operaciones conformarán el proceso de subida de un fichero.

Para la operación de recuperación de un fichero usaremos un método que solicite el bloque correspondiente a un *Timestamp*. Esto ayudará al anonimato de las operaciones.

Firma de las operaciones en Application:

```
public long putRequest(long upload_id, int num_blocks);

public EStatus putReserve(long upload_id, int num_blocks, long
first_timestamp);

public EStatus putApply(long upload_id, long timestamp, byte [] block);

public byte[] get(long timestamp);
```

Repository:

Cada operación definida en la API *Application* tiene su contrapartida en el repositorio. Para la petición *putRequest* el repositorio comprobará cual es el primer *Timestamp* sin uso del que dispone, y reservará tantos como número de bloques haya en la solicitud, y los marcará con el estado *REQUESTED*. Para la petición *putReserve* comprobará si los bloques anteriormente solicitados están con el estado correspondiente, los establecerá como *RESERVED* y devolverá OK o ERROR según corresponda. Para la función *putApply* guardará los datos en la base de datos y cambiará el estado a *VALID* si procede. Para la función *get* el repositorio devolverá el bloque correspondiente al *Timestamp* pasado como parámetro.

Firma de las operaciones en Repository:

```
public long  putRequest(long upload_id, int num_blocks, String tableName);

public EStatus putReserve(long upload_id, long first_timestamp, int
num_blocks, String tableName);

public EStatus putApply(long upload_id, long timestamp, byte[] block, String
tableName);

public List<byte[]> get(long upload_id, String tableName);
```

Tras este análisis inicial podemos ver que requerimos de una clase que indique el estado de las operaciones, por lo que definimos la clase *EfileStatus* para tal menester, incluyendo *REQUESTED, RESERVED,VALID, y DISCARDED* para los bloques que finalmente tengan que ser purgados.

Además, necesitamos añadir a la clase *Messages* un id que va a contener el *Timestamp* del primer bloque de metainformación de un fichero asociado a dicho mensaje, o el valor 0 si no tiene fichero asociado. Esto va a conllevar más cambios en los proyectos que conforman la aplicación.

Una vez definidos los métodos que conforman la API vamos a establecer cuál va a ser el proceso de una subida de un fichero.

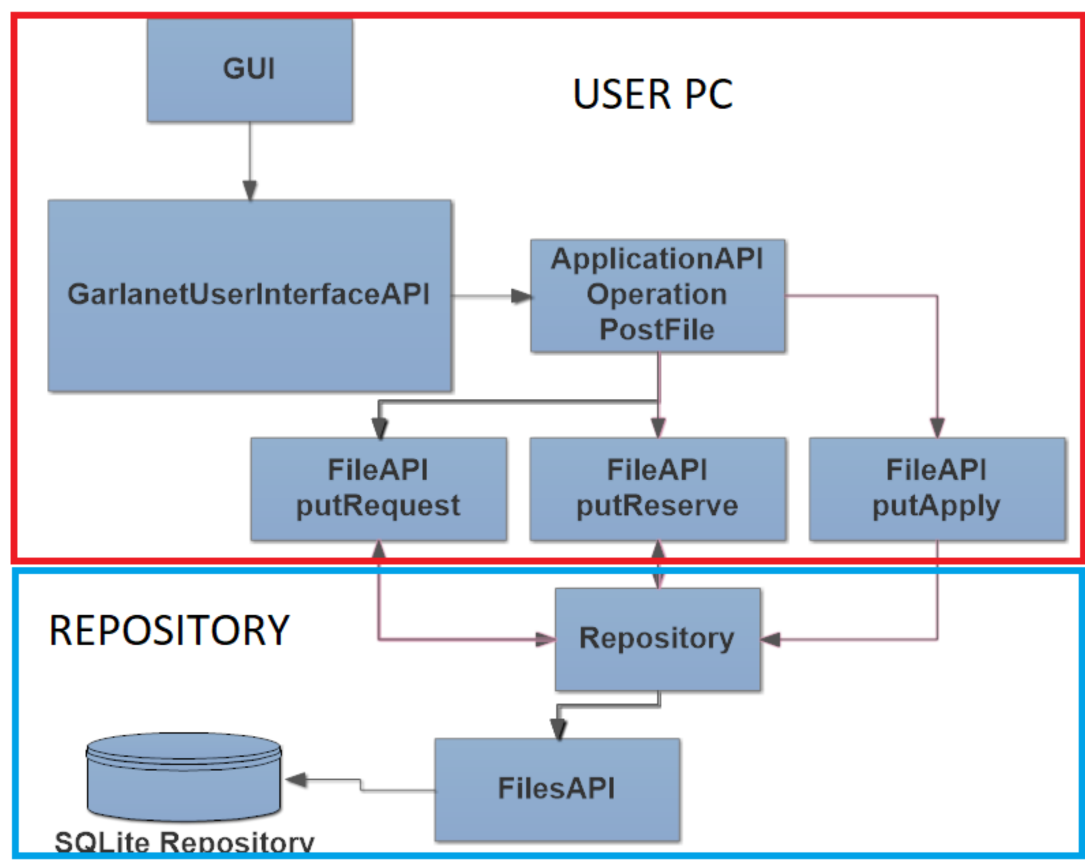


Figura 16: Diagrama de flujo de la operación Post

El proceso de la operación *getFile* va a ser exactamente el mismo, con la salvedad de que es una operación que solo va a usar un método para realizar el proceso.

3.2 Implementación de la solución

Consultar el anexo 9.1 y sucesivos para una consulta más completa del código fuente.

Modificaciones en la GUI

Actualmente en el proyecto GUI_swing de Garlanet existe un mecanismo para el envío de mensajes, que hace uso de las interfaces pertinentes para ello. La GUI está formada por distintos JPanel, cada uno desempeña una funcionalidad concreta, como por ejemplo, la visualización de los mensajes, el envío de los mismos, mostrar la información de los usuarios seguidos, etc…

Para añadir nuestra funcionalidad el primer paso es modificar el panel que contiene el envío de los mensajes en la actualidad. Dicho panel se genera en la clase JPanelPost, uno de los componentes principales de la GUI. Actualmente este panel tiene la misión de recuperar los textos que se introducen y enviarlos si se hace uso del botón "Enviar mensaje".

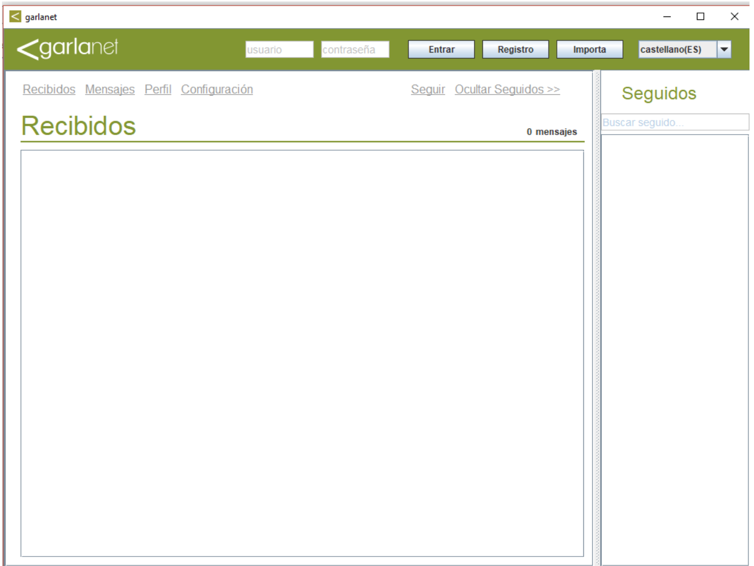


Figura 17: Aspecto gráfico actual de la GUI

Para el envío de los mensajes se hace uso de un *JTextArea* y de un *JButton* donde se recupera el texto y se procesa el envío. Para el envío de los ficheros vamos a hacer uso de otro botón que permita añadir el fichero (donde añadiremos un filtro para que solo seleccione por el momento, imágenes), y reutilizaremos el botón de envío para enviarlo, aprovechando el *JTextArea* para introducir texto que se enviará junto con el archivo.

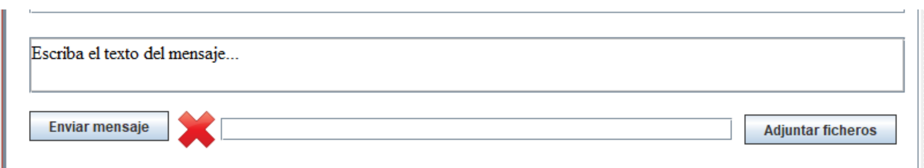


Figura 18: Botón para subir ficheros a la GUI

El botón incluido va a permitir seleccionar un fichero desde el explorador de ficheros, el componente usado es el *JFileChooser.* Una vez pulsado el botón de añadir el fichero, este se incluye en una lista de ficheros, que es vaciada si pulsamos el botón de enviar.

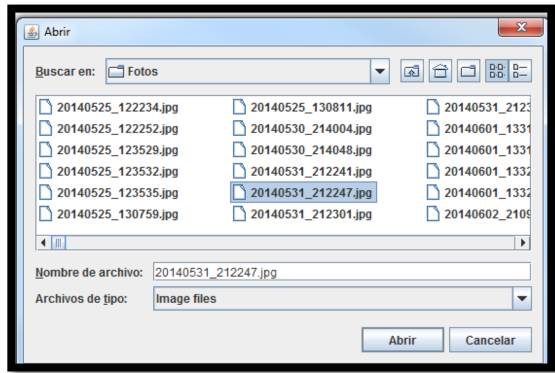


Figura 19: Selector de imágenes de la GUI

Es importante recalcar que no se produce una modificación en el comportamiento anterior del envío de mensajes. Ahora si se ha incluido un fichero y hay texto, el fichero y el texto se enviarán. Para el caso de que no se haya incluido un fichero, se enviará el texto si este existe.

El método empleado para el envío del fichero (si existe) y del texto es el siguiente, y se encuentra en el *ActionListener* del botón de envío:

```
MainFrame.getGarlanet_interface().post(MainFrame.getGarlanet_interface().getM
icrobloggingID(), txtarea_Post.getText().trim(),files);
```

Este método invoca a *GarlanetUserInterface* por lo que vamos a detallarlo a continuación:

GarlanetUserInterface

Hay que tomar *GarlanetUserInterface* como un punto de entrada de la GUI a la aplicación. Las funciones que esta interfaz implementa deben tener su contrapartida en la clase *Application* que es realmente la que ofrece los distintos microservicios de los que se vale el proyecto.

Es por ello que mucha de la funcionalidad que implementa esta interfaz carece de lógica en sus métodos y se limita a llamar a los métodos de la clase *Application*. Los objetos que vamos a transferir de la parte visual a la aplicación no son los objetos *Metainformation* ni los ficheros propiamente dichos, aún no, por lo que es necesario establecer objetos que nos permitan pasar la información necesaria para que el microservicio pueda recuperar lo necesario para generar los bloques y transmitir la información de los ficheros y los ficheros en sí.

GarlanetUserInterface

FilesDescriptionUserInterface Class

FilesDescription Class

Image Class

URL Class

Image Class

URL Class

Application

Figura 20: Diagrama de comunicación de la interfaz

La finalidad de estas clases es pasar la información de un punto a otro, ya que la información solo se puede recuperar a partir de la GUI. Es por ello, que la GUI va a ser la encargada de generar la clase que va a contener la *metainformación* en primera instancia.

La clase en concreto es la *FilesDescriptionUserInterface*, siendo una clase similar a la de *MetaInformation* con algunas salvedades. Esta clase va a tener datos principales del fichero, como el nombre, un nombre temporal, el tipo y el subtipo (si procede). Habrá tras la implementación dos clases que heredarán de esta, que son *FilesDescriptionImageUserInterface* y *FilesDescriptionURLUserInterface*. Estas clases añaden a la clase padre una *preview* y un texto alternativo para las imágenes y una URL para las url.

Estos objetos se construyen al subir un fichero desde la GUI y se llama al método *attachFile* contenido en *GarlanetUserInterface*. Esté método recibe el *Path* del fichero que se está adjuntando, necesario para conocer la ubicación del fichero en el directorio del sistema operativo, y devuelve un objeto *FileDescriptionUserInterface.*

Para obtener este objeto, mueve una copia del fichero al directorio temporal que está establecido en los parámetros de configuración globales de la aplicación, modificando su nombre original por un *Timestamp* para preservar el contenido del fichero, y eliminando la extensión (que preservamos en el objeto devuelto). Para poder relacionar el objeto con la marca de tiempo, relacionamos el *Timestamp* generado con el campo *filename_tmp*. La firma del método es la siguiente:

```
public FileDescriptionUserInterface attachFile(String path) throws
GarlanetException
```

Una vez obtenidos los objetos *FileDescriptionUserInterface*, el usuario tiene la posibilidad de subirlos al hacer *click* en enviar mensaje. En este caso se hará uso del método encontrado en *GarlanetUserInterface* llamado post.

Este método se encarga de la conversión de los objetos *FileDescriptionUserInterface* en *FileDescription*, los objetos de la parte *Application*, y de realizar una llamada al método de esta clase también llamado post. La conversión es sencilla ya que los objetos son similares entre sí, y hemos elaborado un método estático en ambas clases que recibe una lista de objetos de la parte contraria y que devuelve una lista análoga de la propia. En este caso hacemos uso del método contenido en *FileDescription* cuya firma es la siguiente:

```
FileDescription.toListFileDescription(List<FileDescriptionUserInterface>
files));
```

Esta conversión comprueba si los objetos son el tipo base o alguno de los heredados y realiza la conversión en base a ello para recuperar correctamente todos los datos.

Tras la conversión de los datos hacemos la llamada final al método post de la aplicación. Este método recibe un identificador de funcionalidad (aunque actualmente no se usa), un texto (que va a ser incluido como mensaje, y donde servirá para relacionar un mensaje con el envío de ficheros adjuntos) y una lista de objetos *FileDescription*.

El microservicio Files

El microservicio Files consta de dos partes, la parte que implementa el proyecto *Application* y la parte que implementa el proyecto *Repository*, en ambos casos es necesaria dicha implementación para que la comunicación entre la GUI y los repositorios exista.

Para cada proyecto se elabora una clase que va a contener los métodos especificados en el análisis, en ambos casos (tanto para *Application* como para *Repository*) se ha creado una clase llamada *Files*, que hereda de la interfaz *FilesAPI* , aunque cada proyecto tendrá su implementación concreta, dependiendo de la funcionalidad que cada método debe tener en uno u otro caso.

Para la parte situada en *Application* tenemos dos métodos que llaman a la interfaz *Files* y generan la comunicación necesaria entre los proyectos. En concreto en el método *post* y el método *get* de la clase *Application.java*.

El método post de la clase *Application.java* recibe la lista de *FileDescription* asociada a los ficheros que se suben, el id de funcionalidad y el texto asociado a los ficheros. Este método se encarga de recuperar la *Preview* del fichero, generar la MetaInformación y generar los bloques que van a conformar la

subida del fichero. Una vez obtenida esta información se hace uso de la interfaz, ya que tenemos todo lo necesario para poder invocar los métodos para realizar una subida exitosa de un fichero. El método *putRequest* de la interfaz *Files* necesita un id de subida, que es generado de forma aleatoria (es un número que tras la subida, lo eliminaremos para aumentar el nivel de seguridad entre los bloques y el mensaje, haciendo imposible para un atacante saber a priori, que bloques pertenecen a un mismo fichero) y el número de bloques que vamos a subir. Esta parte se comunica con el Repositorio (este método y los posteriores) mediante el uso de la clase *RepositoryMessageObject.* Esta clase permite el envío de los mensajes al repositorio añadiéndole un tipo de mensaje y el método que se quiere efectuar (los cuales, deben haber sido definidos en la clase enumerada *ERepositoryMessageMethod).* La subida de mensajes está condicionada al Quórum predefinido, y devolverá el Timestamp que establecimos en el análisis.

Si todo ha ido bien, tendremos el primer timestamp, y los siguientes *timestamps* reservados hasta llegar al número de bloques que vamos a subir. En el mismo método hacemos uso de la siguiente función definida en la API, *putReserve* que establece los bloques a un estado reservado. Si todo ha ido bien terminamos la operación con *putApply,* donde haciendo uso del identificador generado subimos los bloques. La operación de forma global devuelve a la interfaz gráfica un enumerado de tipo *Estatus* con OK o ERROR, dependiendo del éxito o fracaso de la operación (que recordemos, depende entre otras cosas, del quórum disponible o del éxito individual de cada operación).

En el caso de la operación *get* está relacionada con la descarga de ficheros (a la GUI) y la operación es más sencilla en cuanto a lógica que en el caso de la subida de ficheros. Tenemos un mensaje de texto que tiene un timestamp asociado al fichero (por lo que hemos tenido que añadir este campo a la clase Messages del proyecto *GarlanetCore),* este timestamp se corresponde con el primer bloque que contiene información acerca del fichero, por lo que se pide a través del método mencionado. Una vez obtenido el primer bloque hacemos uso de la funcionalidad que provee *FileUtils* para reconstruir el objeto de MetaInformación y recuperar así el fichero completo, pidiendo tantos timestamps como sean necesarios según los bloques de los que conste el fichero. El método finalmente devuelve la imagen como array de bytes.

La implementación en Repository

Como definíamos al principio de este apartado, para que la operación tenga éxito era necesario definir la API en ambos proyectos, por lo que ahora vamos a ver la clase *Repository.java* del proyecto Repository.

```
case FILES:
    handleFiles(message, tablename);
    break;
```

Figura 21: Manejador Files en la clase Repository

Para ello se incluye una nueva función, *handleFiles* en la clase Repository, donde vamos a añadir los casos posibles de mensajes por parte de *Application,* donde haremos un filtrado dependiendo del ERepositoryMessageMethod correspondiente que haya sido enviado, por lo que, de forma superficial, todos los métodos de la API Files del proyecto Repository van a ser usados dentro de este método.

Para cada ERepositoryMessageMethod enviado se hace uso de su implementación correspondiente en la API, así que para PUT_REQUEST usamos el método *putRequest,* para PUT_RESERVE, tenemos el método *putReserve,* y así sucesivamente.

La persistencia en el repositorio viene definida por una base de datos SQLite que usa como nombre de tabla el identificador del servicio asociado al usuario, de esta manera cada usuario tiene un identificador distinto para cada servicio. En este caso para Files ocurre exactamente eso, por lo que el identificador que viene asociado al mensaje que intercambian Application y Repository es el nombre de la tabla donde se van a guardar los datos.

Para la operación putRequest por ejemplo, requerimos de un timestamp, que es a su vez, clave primaria en la tabla que almacenará el servicio *Files* para un usuario determinado.

TIMESTAMP	UPLOAD_ID	STATUS	ENCDATA	SECUREBLOCK
LONG	LONG	BYTE	BYTE []	BYTE []

Figura 22: Tabla del servicio FILES

El resto de campos vienen definidos en la figura adyacente. Así pues, con la estructura de tabla definida, obtener el primer timestamp disponible del servicio es realizar una *query* buscando el último timestamp almacenado en la tabla e incrementándolo en 1. Posteriormente y dentro de la operación, tenemos que realizar una inserción de filas reservando cada timestamp para los bloques que nos han solicitado. Esto nuevamente requiere de un acceso a la base de datos, estableciendo las filas en el estado REQUESTED como definimos en el análisis.

Si todo ha ido bien, devolvemos el primero de los timestamp reservados. Como podemos ver, cada operación en el repositorio es independiente del resto, al contrario que pasa en *Applicatión* donde hay una operación que aglutina varios métodos de la API para subida y otra para bajada.

Para la operación *putReserve* realizamos el *update* de las filas al siguiente estado (RESERVED). Es esencialmente la misma funcionalidad que para la operación *putApply,* donde únicamente hay que hacer un *update* de la fila cuya *primary key* coincida con la que nos están pasando y tenga el estado correspondiente, no obstante con la operación *putApply* tenemos que actualizar además del estado, el bloque y eliminar el id de subida. Tenemos un ejemplo a continuación.

```
esultSet rs = null;
Status result = EStatus.OK;
ry {
    conn = SQLiteConnector.getConnection(databaseMetadata.getDatabaseName());
    stat = conn.prepareStatement("UPDATE '"+tableName+ "' SET status = ? , encData = ? "
            + "WHERE timestamp = ? AND upload_id = ? ");
    stat.setByte(1, EFileStatus.VALID.getByte());
    stat.setBytes(2, block);
```

Figura 23: Ejemplo de operación de actualización en la tabla.

Por último y de momento para esta parte queda por explicar la operación *get,* que en esencia no es más que una consulta a la fila que contenga el timestamp correspondiente, para recuperar el bloque.

La descarga de ficheros en la GUI

Para poder implementar la descarga de ficheros en la GUI, hemos tenido que modificar la clase *PanelTimelineObj* del proyecto GUI_Swing. Esta clase realiza la función de recuperar los mensajes nuevos pendientes para el usuario y es el sitio indicado, ya que en este punto podemos ver si un mensaje tiene un *attached_file_id* asociado distinto de 0, lo que significará que hay un fichero asociado al mensaje. Una vez analizado y comprobado que el mensaje tiene un fichero, hay que añadir a la vista la *preview* del fichero que contiene y permitir posteriormente su descarga siempre que el usuario lo quiera. Para ello, recuperamos el objeto MetaInformation del fichero y construimos la p*review* mediante el objeto *FileDescriptionImageUserInterface.getPreview()* y la adjuntamos al mensaje como un *ImageIcon* de la librería java.swing. Añadimos la imagen tras el mensaje de texto asociado al fichero dándole unas dimensiones que pueden ser cambiadas en el fichero de configuración general de la aplicación y haciendo uso de la utilidad disponible en *FileUtils.* Como podemos ver en la captura posterior, la *preview* cumple su función e indica que el mensaje contiene un fichero disponible

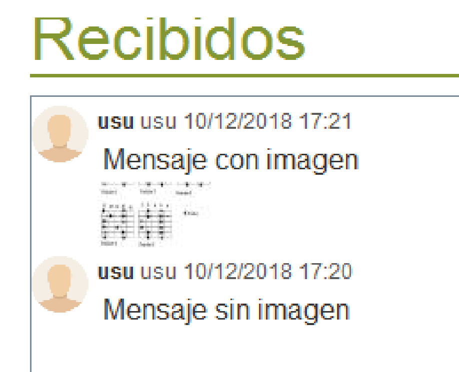


Figura 24: Preview de imagen en la GUI

Ahora lo interesante es añadir la posibilidad de que el usuario pueda abrir la imagen o descargarla si lo desea. Para ello añadimos la funcionalidad requerida en la clase que muestra la *preview*, en este caso *PanelTimelineObj.*
Haciendo uso de un *JOptionPane* y mostrando las opciones de abrir o de guardar, implementamos la funcionalidad de abrir directamente la imagen con el visor por defecto con la clase *Desktop* y pasando el fichero (que debemos recuperar previamente gracias a la funcionalidad implementada para descargarlo) o guardarlo usando un JFileChooser.

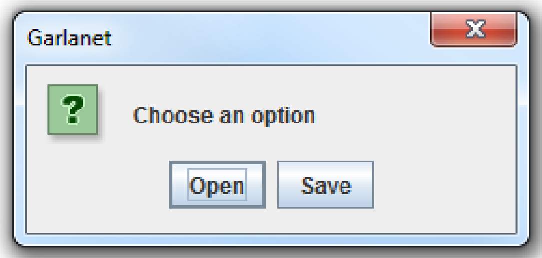


Figura 25: Opciones para el fichero

El nombre del fichero es tomado como el nombre original del fichero al seleccionar la opción de guardar, mediante el explorador de ficheros seleccionamos la ruta deseada y guardamos el fichero.

Figura 26: Explorador de ficheros guardando una imagen

Con esto ya tendríamos implementada la segunda fase del proyecto.

3.3 Conjunto de pruebas e Integración del código en Garlanet

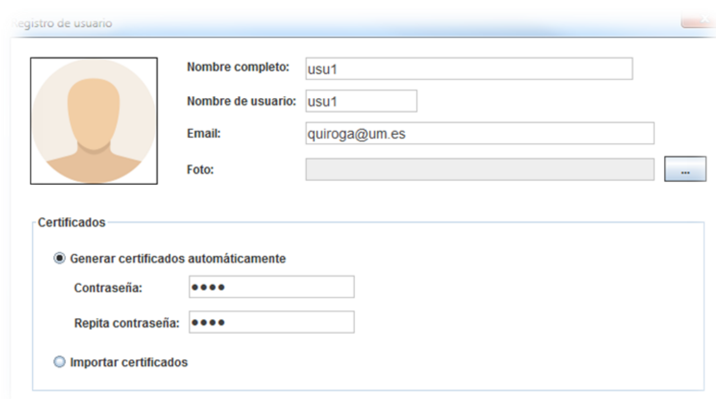


Figura 27: Registro de usuario de pruebas

Para la validación de las pruebas creamos varios usuarios que van a servir para seguir la ejecución del código y ver si el sistema responde y es tolerable a fallos.

Para estos casos de prueba se han realizado las pruebas teniendo en cuenta el siguiente entorno:

Número de réplicas que participan: 1
Número de Quorum: 1
Usuario: usu1

Caso 1: Forzar el fallo en la operación putRequest.

Qué se busca: Con esta operación buscamos que el sistema tras un error en la solicitud de los *Timestamp* sea capaz de recuperarse y funcionar con normalidad posteriormente. Que el sistema sea tolerable a fallos.

Descripción de la prueba: Lo que vamos a realizar para forzar el fallo es provocar un fallo por *timeout* en los comandos que intercambian cliente y repositorio. Este fallo lo que hace es invalidar los comandos que pueda recibir el cliente y reproduce en cierta forma un posible error de comunicación que puede ser muy común. Para el primer paso probamos a enviar un fichero (una imagen cualquiera) e introducimos el texto "ficheroFalloPutRequest".

```
398        List<SecureMessagesMessage> secure_message_list = null;
399        switch(method) {
400        case PUT_REQUEST:
401            if (parameters.size() != 2) {
402                System.out.println("Error HandleMessages: Error in number of parameters");
403                response.add(0);
404            }else {
405                long upload_id = (long) parameters.get(0);
406                int num_blocks = (int) parameters.get(1);
```

Figura 28: Ejecución detenida en el Repositorio

```
The timeout in the WaitReply has been reached!
```

Obteniendo el mensaje "The timeout in the WaitReply has been reached!" a través de la consola de java podemos estar seguros de que la operación va a fallar, lo que vamos a comprobar es, una vez eliminado el punto de ruptura, si el sistema se recupera ante fallos, por lo que deshabilitamos el punto, seleccionamos otra imagen, y le damos el nombre "ficheroTrasFalloPutRequest", y volvemos a enviar.

Figura 29: Fichero subido tras forzar un error.

Como vemos en la figura 20, el fichero se sube y se envía al cliente.

Resultados: Paramos la ejecución en la parte del repositorio (Clase Repository.java) con un punto de interrupción para hacer saltar el *timeout* de comunicación entre los repositorios y el cliente, con lo que conseguimos que se produzca un error forzado. La ejecución continúa devolviendo un objeto *EGarlanetUserInterfaceStatus.ERROR,* se vacía la lista de ficheros y no se produce ningún envío posterior de información. A continuación mandamos otro envío sin provocar ningún fallo en el flujo de ejecución, y el fichero se manda sin problemas. **Es el funcionamiento esperado.**

Caso 2: Forzar el fallo en la operación *putReserve*

Qué se busca: Con esta operación buscamos comprobar que ocurre tras un error en la operación siguiente a *putRequest.* Esta operación es justo la anterior a la subida real de los bloques pero aún así vamos a ver el comportamiento del sistema ante un fallo.

Descripción de la prueba: Al igual que en la prueba anterior, buscamos que la consola de la GUI muestre el siguiente mensaje:

```
The timeout in the WaitReply has been reached!
```

Esto indicará que se ha producido irremediablemente un error en la comunicación, por lo que vamos a recibir un error en la clase Estatus, a partir de aquí esperamos que la aplicación maneje el error y que no provoque ningún estado inconsistente en la aplicación.

Resultados: Con la parada de la ejecución provocamos de nuevo el mismo fallo que en el caso anterior, por lo que el envío de información queda en estado inconsistente y la operación finaliza con un *EGarlanetUserInterfaceStatus.ERROR* igual que ocurría antes. Al intentar mandar de nuevo un fichero sin el error vemos que el comportamiento es el esperado y el fichero se manda correctamente. **Es el resultado esperado.**

Caso 3: Forzar el fallo en la operación *putApply*

Qué se busca: Con este fallo se busca ver la consistencia del sistema ante fallos en la operación de subida. Si el sistema es tolerable a fallos el comportamiento esperado debe ser informar del error pero continuar con su ejecución sin que el usuario sufra ningún inconveniente de la aplicación.

Descripción de la prueba: Buscamos nuevamente parar la ejecución, esta vez en la operación *putApply,* una vez que tengamos el mensaje de error por *timeout* comprobamos el resultado de la variable obtenida en *Estatus.* Si es error hemos conseguido realizar la prueba. Ahora hay que probar a enviar otro fichero y ver si se consigue el objetivo de que se envíe sin problemas, eliminando los puntos de ruptura.

Resultados: **Los resultados nuevamente son los esperados**, el fichero se manda tras provocar el error.

Caso 4: Dos usuarios con un único repositorio

Qué se busca: Con esta prueba se busca comprobar que de forma simultánea dos usuarios hacen uso del mismo repositorio y no se producen fallos. Si bien es una prueba sencilla que tampoco nos va a proporcionar una gran información se busca sobre todo probar la capacidad del repositorio de trabajar de forma concurrente, del microservicio *Files* (ya que asumimos que para el resto de operaciones el repositorio ya ha sido probado).

Descripción de la prueba: Dos usuarios mandan de forma simultánea, una imagen cada uno. Para ello creamos al usuario usu2 y probamos a enviar imágenes de forma simultánea con el usu1.

Resultados: Los resultados de la prueba son exitosos y el comportamiento del repositorio es el esperado, como podemos ver en la siguiente figura.

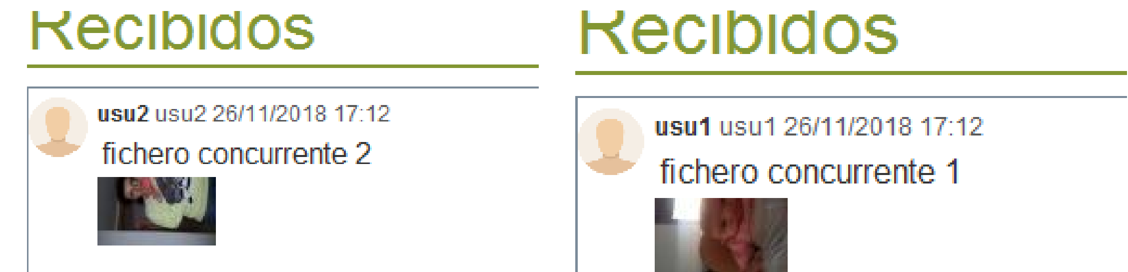


Figura 30: Envío concurrente de imágenes de dos usuarios

Caso 5: Prueba con imágenes pesadas.

Qué se busca: principalmente se busca ver si se suben y se recuperan de forma satisfactoria imágenes que ocupen un número significativo de *megabytes* para obligar a la lógica del proyecto a fraccionar en múltiples bloques y a recomponer la imagen nuevamente a partir de estos. Puede ser una prueba redundante con respecto a las que se hicieron en la Fase 1, no obstante ahora hay más puntos de fallo con respecto a las pruebas que se hacían anteriormente, por lo que tiene interés ver que ocurre en el sistema con imágenes que se componen de muchos bloques.

Descripción de la prueba: Se busca una imagen pesada en el explorador de archivos, encontrando una imagen que supera los 8 MB de información, por lo que debido al tamaño de bloque establecido en Garlanet esta imagen se va a fraccionar en 18 bloques.

20161028_185432.m...	28/10/2016 18:54	Vídeo MP4	13.243 KB
20160401_094707.jpg	01/04/2016 9:47	Imagen JPEG	8.659 KB
20160401_094343.jpg	01/04/2016 9:43	Imagen JPEG	8.657 KB

Figura 31: Imagen de gran tamaño para prueba

Mandamos la prueba con el texto “testX” donde X es el número de imagen que enviamos.

Resultados: **Los resultados son correctos** y la imagen se recupera bien desde la GUI.

Figura 32: Preview de imagen pesada desde GUI_swing

Si bien cabe destacar que la operación tarda en completarse unos segundos. El proceso de subir bloques se puede observar en tiempo real desde consola. En uno de los test se ha producido un error indeterminado al subir uno de los bloques, lo que ha interrumpido la subida de la imagen, no obstante tras un segundo intento la imagen se ha subido sin problemas, lo que indica que el sistema funciona correctamente.

Número de réplicas que participan: 2
Número de Quorum: 1
Usuario: usu3

Caso 6: Prueba de subida de ficheros a múltiples repositorios.

Qué se busca: Buscamos comprobar que las operaciones de asignación de *Timestamp* y de reserva de filas funcionan correctamente para más de un repositorio, además vamos a probar a reproducir errores en alguno de los servidores réplica para ver el comportamiento de las operaciones. Lo que se busca es que si la subida implica un menor número que el Quorum establecido, la operación falle sin afectar a la estabilidad del sistema. Para ello necesitamos otro usuario, usu3.

Descripción de la prueba: Levantamos otro repositorio (el cual obtiene un puerto distinto al primero) y probamos a enviar la imagen con el nuevo usuario. Lo que interesa de esta prueba es ver si la imagen se sube correctamente a cada réplica y se recupera satisfactoriamente. Para ello tenemos que ver en la consola si las operaciones se realizan de forma correcta para cada puerto en el que esté alojado un repositorio.

```
FILES::PUT_REQUEST [to: 1XX.XX.XX.XX:10000]
FILES::PUT_REQUEST [to: 1XX.XX.XX.XX:10001]
FILES::PUT_RESERVE [to: 1XX.XX.XX.XX:10000]
FILES::PUT_RESERVE [to: 1XX.XX.XX.XX:10001]
FILES::PUT_APPLY   [to: 1XX.XX.XX.XX:10001]
FILES::PUT_APPLY   [to: 1XX.XX.XX.XX:10000]
```

Resultados: Como vemos en la traza aportada arriba, se ejecutan los comandos de forma correcta. Cada operación se realiza de forma simultánea en cada repositorio, **además la imagen se recupera en la GUI de forma correcta**, lo que indica que ambos comparten el mismo *Timestamp*. Esto presentará problemas a la hora de realizar pruebas más exhaustivas ya que hasta la Fase 3 no implementaremos el control de consistencia y los

repositorios que no se creen al mismo tiempo que se envíe un fichero es posible que no intercambien información referente a estos, lo que puede dar problemas (ya que los mensajes si se intercambian, y entre la información intercambiada se encuentra el *timestamp* de los ficheros asociados, lo que provocará que la GUI intente descargar el fichero asociado, que puede que no se encuentre en el repositorio que vaya a buscar).

Caso 7: Subir una imagen y forzar el fallo en uno de los repositorios.

Qué se busca: Ver el comportamiento de la GUI cuando uno de los repositorios no termina de subir el fichero. Teóricamente el control de consistencia debería posteriormente replicar el contenido del repositorio que ha subido con éxito la imagen al repositorio que ha fallado, aunque eso no está aún implementado.

Descripción de la prueba: vamos a introducir fallos deliberados en uno de los repositorios en distintos puntos de la subida de una imagen, para provocar errores al igual que hicimos con los casos 1, 2 y 3.
Tras la interrupción de forma sucesiva del comando *putRequest* eliminamos los puntos de ruptura y comprobamos la estabilidad del sistema.

Resultados: La interrupción del comando impide que se suban a ambos repositorios ningún fichero, no obstante cuando recuperamos la ejecución normal el sistema si sube los ficheros. La prueba es **no concluyente** ya que hemos reproducido un caso exacto al caso 1 pero con más repositorios, por lo que hay que realizar otro tipo de prueba para verificar que el sistema es estable con más repositorios.

Caso 8: Uno de los repositorios cae entre comandos.

Qué se busca: Verificar el comportamiento del microservicio ante un fallo grave como puede ser la caída de uno de los repositorios en mitad de la ejecución de comandos de una subida de ficheros.

Descripción de la prueba: Mientras se están ejecutando los comandos *putRequest* y *putReserve* se va a eliminar uno de los dos repositorios. Esto se va a hacer estableciendo en la interfaz *Application* un punto de ruptura para cerrar uno de los repositorios. Se va a mandar una imagen cualquiera y el mensaje “Repositorio caído”.

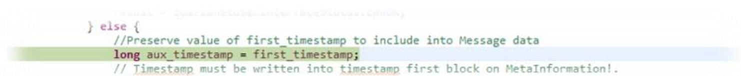


Figura 33: Imagen del punto de ruptura en Application

Una vez parada la ejecución de la GUI, vamos a parar uno de los repositorios que están ejecutándose.

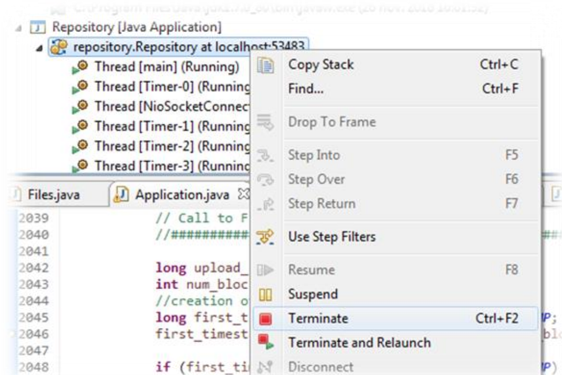


Figura 34: Parada de uno de los repositorios

La imagen se recupera a pesar de que uno de los repositorios en los que originalmente se iba a alojar la imagen ya no está.

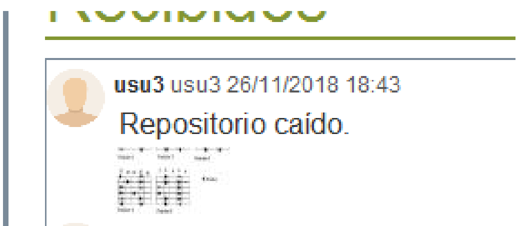


Figura 35: Imagen recuperada tras la caída de un repositorio

Resultados: **La prueba es satisfactoria,** mientras que el quórum se cumpla, el intercambio de ficheros se va a producir.

Número de réplicas que participan: 3
Número de Quorum: 2
Usuario: usu4

Caso 9: Quorum incumplido.

Qué se busca: Probar el sistema si en mitad de una subida de ficheros se incumple el Quorum por caída de repositorios.

Descripción de la prueba: De manera similar al caso anterior, vamos a realizar una prueba que consiste en detener la ejecución entre comandos, esta vez entre *putReserve* y *putApply.* Una vez detenida la ejecución vamos a tirar dos de los 3 repositorios activos, de forma que estaremos incumpliendo el quórum. El resultado esperado es que no se envíe el mensaje pero que posteriores mensajes si se envíen (estableciendo el quórum como un quórum dinámico que se basa en un porcentaje de las réplicas existentes).

Resultado: Al incumplir el quórum en tiempo de ejecución el mensaje con fichero no se envía, tras esto se manda de nuevo el fichero con un quórum que se ajusta al número de repositorios existentes y el fichero se envía y se recibe de forma normal.

Figura 36: Imagen enviada tras eliminar dos repositorios.

Integración en Garlanet

Al igual que hicimos con la primera fase del proyecto, hay que subirlo al repositorio, por lo que tenemos que hacer la operación *Commit* + *push* del control de versiones en cada uno de los proyectos de Garlanet que hemos modificado. En este caso la relación de proyectos modificados es Application, GUI_Swing, GarlanetCore, y Repository.

Como ya hay una rama creada para la subida de todos los proyectos, denominada *Files*, hay que subir los cambios a esa rama. Para no extendernos, los cambios se suben de la misma manera que se hizo en la fase 1, y revisando los cambios en el portal de GitLab asociado al proyecto.

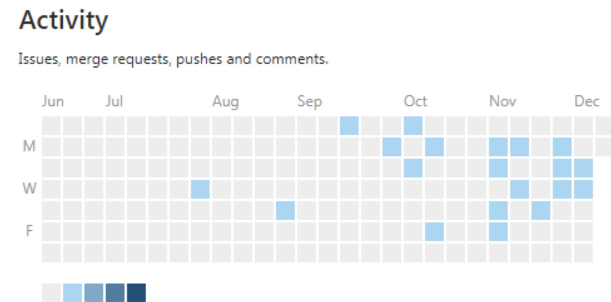


Figura 37: Actividad registrada en el portal de GitLab de la UOC.

4. Control de consistencia

Una vez realizada la implementación de la funcionalidad necesaria para la compartición de ficheros, queda una parte esencial para que el proyecto se complete, el control de consistencia y la purga de bloques.

La purga de bloques es necesaria para mantener una cierta limpieza en bloques que han sido parcialmente subidos o que por ejemplo han provocado por algún error de red (o de otra índole) una respuesta errónea. En estos casos esas filas quedan pendientes de subir y nunca se completan (al expedir por la propia estructura de la API, timestamp superiores. Se debe automatizar la tarea de purga de esos bloques para que mediante operaciones autónomas, sean eliminados de las tablas. Para este caso tomaremos en cuenta el estado en el que se encuentran las filas y por cuanto tiempo están en dicho estado, por tanto la clave del purgado es el campo *status* y el tiempo desde el que está en ese estado.

El control de consistencia es una operación que ya existe en el proyecto Repository para el mantenimiento de otros microservicios como por ejemplo los mensajes. Consiste en comprobar con otros repositorios el contenido que almacenan, y descargarlo si existiese nuevo contenido que no se tiene, o enviar el contenido que tenemos y que el repositorio preguntado no tiene. Con esta operación aseguramos que los repositorios tengan la misma información y que el sistema sea tolerante a la caída de algún servidor. Además sirve para que nuevos repositorios tengan la misma información que los existentes.

En ambos casos el desarrollo va a afectar únicamente al proyecto Repository, donde está toda la funcionalidad que requiere este desarrollo.

4.1 Análisis de la situación actual y diseño de la solución

Aunque ambos forman parte de la última fase del proyecto, se pueden tomar como elaboraciones independientes, si bien, conviven en el mismo proyecto.

Para la operación de purgado debemos tener en cuenta el carácter temporal de la operación, y la independencia de la misma entre los distintos repositorios, y sobre todo su automatismo, lo que nos obliga prácticamente a usar un Timer para su implementación.

En cuanto a la operación de control de consistencia, es un intercambio de información entre repositorios que va a requerir de distintas operaciones de cierta complejidad.

Análisis del purgado

El purgado debe tener un listado de los bloques a purgar, esencialmente nos interesan dos tipos de bloques, aunque en la implementación solo vamos a incluir uno y el otro lo dejaremos como posible mejora. Nos interesa purgar los bloques que permanezcan en un estado RESERVED o REQUESTED y que no tengan el bloque subido, como estos estados pueden ser normales mientras

que los bloques no han terminado de subirse, crearemos una constante en las constantes de la aplicación donde evaluaremos estos estados tras un tiempo considerable. Es decir, los bloques que estén en estos estados durante un tiempo mayor que el definido por la aplicación, serán descartados.

Por tanto, definimos dos operaciones en el purgado, la primera de todas, consultará e incluirá en la lista, los bloques candidatos a la purga, mientras que la segunda operación, marcará los bloques cuyo tiempo sea mayor que el definido como bloques con status DISCARDED. Para la operación de recolección y de marca emplearemos un Timer que se ejecutará de forma autónoma en el proceso principal del Repository y que realizará las dos acciones.

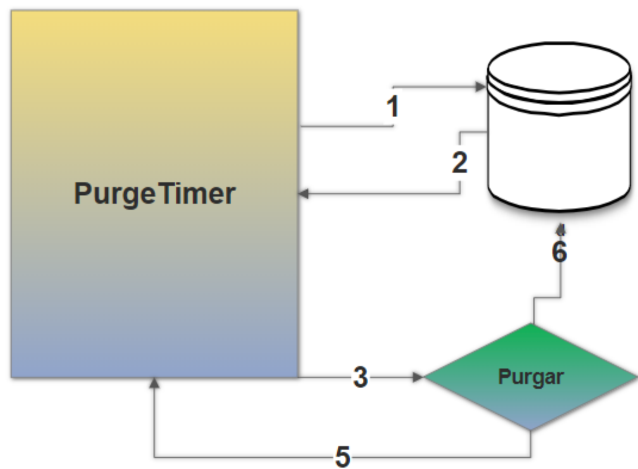


Figura 38: Diagrama de flujo de la operación de purgado

En el diagrama de flujo se puede ver claramente la secuencia que sigue la información, siguiendo la numeración tenemos:

1. El timer realiza una consulta a la base de datos sobre las filas del servicio Files que cumplen la condición.
2. La base de datos devuelve una lista de bloques que cumplen la condición.
3. El timer ejecuta la función de purgar con la lista de bloques recopilada.
4. Operación de purgar.
5. Bloque que no debe ser purgado, se informa al timer de que lo elimine de la lista.
6. Bloque que debe ser purgado, se realiza la actualización de su estado en la base de datos.

Con esto ya tendríamos suficiente para implementar la operación de purgado.

Análisis del Control de consistencia

Esta operación debe contener distintas operaciones de acceso a la base de datos para recuperar, grabar y/o actualizar distintas filas del microservicio Files, pero para simplificar las operaciones entre repositorios, que es la clave de la

funcionalidad, vamos a ver un diagrama de flujo entre dos repositorios y las operaciones entre ellos y después veremos por operación que necesidades va a tener el servicio.

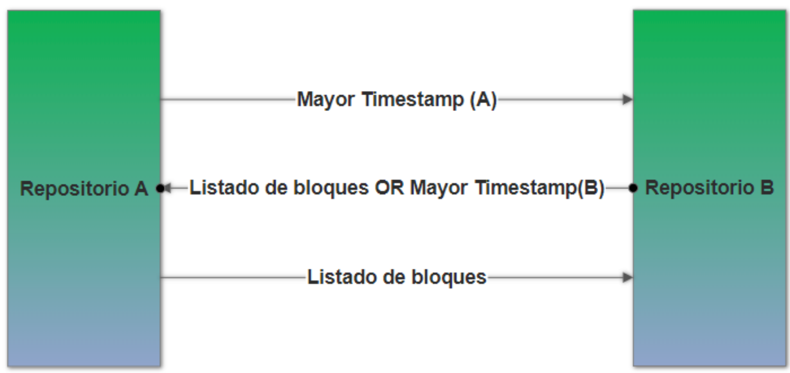


Figura 39: Diagrama de flujo del control de consistencia

Como vemos en el diagrama de flujo, el repositorio A envía el mayor timestamp disponible al repositorio B. Este tiene dos posibles respuestas, enviar el listado de bloques que existen en B y que no existen en A, o por el contrario, si el timestamp de B es menor que el de A, enviar el timestamp a A. El repositorio A puede recibir, el listado de bloques, que en cuyo caso, establecerá los bloques DISCARDED como DISCARDED y los VALID como bloques pendientes de solicitar a otros servidores, o un timestamp, que en este caso, elaborará un listado de los bloques que tiene A y que B no, y los enviará al repositorio B.
Por último, B puede recibir el listado de bloques de A y realizará la misma operación que hubiera hecho A, es decir, completar las filas de su base de datos y añadir a la lista de bloques pendientes de solicitar los bloques válidos para A.

Además esta decisión de diseño nos obliga a realizar un añadido al esquema anterior, tenemos que tener una lista de bloques pendientes de descargar y cuya tarea vamos a automatizar de forma similar a como hemos hecho para la operación de purga: Necesitamos una operación autónoma que pida a los repositorios los bloques que el repositorio no tiene, por lo que programaremos un timer que consulte esa lista de bloques pendientes, haga uso de la operación GET_BLOCK, mandando un timestamp a un repositorio y que ese repositorio devuelva ese bloque.

4.2 Implementación de la solución

Consultar el anexo 9.1 y sucesivos para una consulta más completa del código fuente.

Purga de bloques

Para el purgado de bloques vamos a implementar una nueva función en la API Files, aunque la decisión del método implementado es dejarlo como *public* y meterlo en la interfaz perfectamente se podría haber tomado otra decisión

como excluirlo de la API o dejar el método como *static*. Es un método que solo va a usarse dentro del repositorio y en la ejecución de un *Timer* por lo que también tiene sentido haberlo hecho siguiendo otra lógica. Este método se llama *getPurgeItems* y va a recuperar la información necesaria para posteriormente establecer las filas destinadas a ser purgadas, por lo que necesitamos recuperar el timestamp de la fila y el tiempo desde el que se detecta la anomalía.

Otra decisión de diseño del purgado es establecer el tipo de retorno de la función como un objeto que vamos a denominar *PurgeItem*. Nuevamente es una decisión de diseño que presenta otras alternativas pero que por simplicidad al final se ha optado por ella. La clase se encuentra en el paquete *files* del proyecto *Repository* y contiene dos campos, el timestamp, y la fecha, que se obtiene mediante un System.currentTimeMillis(), lo que nos servirá para comparar los tiempos del *timer* y del objeto creado de forma sencilla.

```
public PurgeItem(long timestamp) {
    super();
    this.timestamp = timestamp;
    this.date = new Date(System.currentTimeMillis());
}
```

Figura 40: Constructor del objeto PurgeItem

Con este objeto y la consulta incluida en el método, que recupera todas las filas cuyo estado sea REQUESTED o RESERVED y que no tenga nada en el campo ENCDATA tenemos un listado de posibles bloques que deban ser purgados.

La clase PurgeTimer se incluye dentro del directorio directoryService del proyecto Repository y se inicializa al mismo tiempo que el resto de *timers* del sistema. En la clase Repository existe una función llamada *initTimers()* cuya funcionalidad es precisamente esa. El objeto PurgeTimer se inicializa en ese momento (al arrancar el Repository) por lo que nos aseguramos de que cualquier repositorio que tenga el código actualizado se ejecutará debidamente. El timer contiene una variable tomada del fichero de configuración general del repositorio, que hemos llamado “time_purge”.

```
public PurgeTimer(DirectoryServiceClient ds, Files files) {
    this.time_purge = Parameters.getIntValue("time_purge");
    this.timer = new Timer();
    //this.time_purge = 5000;
    this.tries = 2;
    this.files = files;
    this.ds = ds;
}
```

Figura 41: Constructor del PurgeTimer

Este constructor toma como parámetros de entrada la clase Files y el DirectoryServiceClient. En el primero tenemos la función con la que vamos a marcar las filas *purgables* y en el otro vamos a recuperar los servicios que el repositorio mantiene y que son de tipo Files.

Una vez recuperado el listado de ítems a purgar gracias a la funcionalidad implementada en la clase Files, tenemos un listado de objetos potencialmente purgables, pero nos falta una nueva función que los marque como filas descartadas. Por lo que implementamos en Files una nueva función que recibe como parámetro un PurgeItem y que vamos a llamar *purgeBlock*. Esta función realiza la consulta de cada PurgeItem y realiza un análisis de casos tal y como hemos definido en el análisis anterior. Establecerá la fila como DISCARDED si reúne las condiciones necesarias para ello (que este el tiempo definido con el bloque de datos vacío) y se eliminará posteriormente de la lista de bloques candidatos a ser purgados.

Controles de consistencia entre repositorios

Las sesiones de consistencia tienen como misión principal mantener la misma información en todos los repositorios disponibles. Estas sesiones ya existen en el proyecto de Garlanet para otros microservicios como los mensajes o el historial de un usuario, se lanzan a través de un Timer existente en la clase *MyServicesTimer*. Por tanto, la implementación de este requerimiento se va a realizar también a través del mismo Timer.

```
switch(serviceType) {
    case PROFILE:
        consistencySessionProfile(service);
        break;
    case MESSAGES:
        consistencySessionMessages(service);
        break;
    case TIMELINE:
        // currently, no consistency sessions for Timeline services
        break;
    case STATUS:
        consistencySessionStatus(service);
        break;
    case FILES:
        consistencySessionFiles(service);
        break;
    default:
```

Figura 42: Sesiones de consistencia de los microservicios

Dentro del método *consistencySessionFiles* se van a enviar las sesiones de consistencia al resto de repositorios. Las sesiones de consistencia van a tener varios pasos dependiendo de la comunicación que se intercambie con otros repositorios. Primero se va a obtener el máximo timestamp que se alberga de un servicio Files. Con este timestamp y el listado de timestamp no recibidos, se realiza un envío a todos los repositorios que contengan este servicio. El repositorio B responde desde la función *handleFiles* a la petición con los timestamp no recibidos y sus estados, el mayor timestamp y el listado de bloques no recibidos para el caso de que sea el repositorio B el que tenga menos información que A, o si ambos repositorios contienen la misma información, terminaríamos la comunicación.

A este primer intercambio le sigue añadir esos timestamp no recibidos a una lista de bloques pendientes de recibir, este listado es procesado por la clase *PendingBlockTimer* que se ejecuta periódicamente y comprueba del listado los bloques que restan por pedir, los cuales se piden al resto de repositorios y se

borran de la lista una vez están descargados, mediante la nueva opción incluida en el procedimiento *handleFiles GET_FILE_BLOCKS.*

```
public PendingBlocksTimer(Files files,ReplicationClientSide replicationClient) {
    //this.time_action = Parameters.getIntValue("time_purge");
    this.timer = new Timer();
    this.time_action = 10000;
    this.tries = 2;
    this.files = files;
    this.replicationClient = replicationClient;
}
```

Figura 43: Timer que recupera los bloques que faltan

4.3 Conjunto de pruebas e Integración del código en Garlanet

Este conjunto de pruebas van a validar el resultado final de todo el proyecto, por lo que van a probar en distintos entornos distintos casos con los que extraer conclusiones sobre el desarrollo realizado, y comprobar su fiabilidad, de forma que el código pueda ser fusionado con la rama principal de todo el proyecto.

Para los siguientes casos de prueba se han tenido en cuenta los siguientes valores:

Caso 1: Subida de imágenes de distintos tamaños para un usuario

Número de réplicas que participan: 1
Número de Quorum: 1
Usuario: usu1

Qué se busca: Esta operación es simplemente una comprobación de que el sistema sigue siendo estable tras la modificación de la clase Repository y del añadido de los métodos de purga y de consistencia. En parte, certifica que el desarrollo anterior sigue siendo válido.

Descripción de la prueba: Simplemente vamos a realizar tres subidas de distintos tamaños de tres imágenes. Si todo va bien el sistema debe de guardarlas y mostrarlas, y que sean recuperables por parte del usuario.

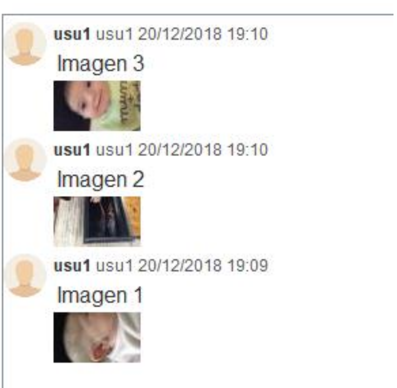


Figura 44: Imágenes de distintos tamaños

Resultados: La prueba es satisfactoria y damos por buenos los resultados, el sistema no se ha visto afectado por el último desarrollo, así que podemos probar distintos casos para descubrir errores.

Caso 2: Dos repositorios

Número de réplicas que participan: **2**
Número de Quorum: 1
Usuario: usu

Qué se busca: En este caso buscamos que el control de consistencia actúe y las imágenes de este usuario se copien en el nuevo repositorio desplegado.

Descripción de la prueba: En esta prueba buscamos que las sesiones de consistencia se ejecuten y que el método que obtiene los bloques faltantes se ejecute en el nuevo repositorio dejando una traza en la consola. Lo único que tenemos que hacer es ejecutar un nuevo repositorio y tras su puesta en marcha hay que ver si las trazas se observan en la consola. (En principio esto solo quiere decir que el método se ejecuta y que se comparten bloques, más adelante revisaremos si esto es efectivo con otras pruebas).

Resultados: La traza esperada se obtiene a través de la consola:

```
FILES::GET_FILE_BLOCKS    [to: 192.168.1.63:10000]
```

Esto quiere decir que se ha solicitado el método GET_FILE_BLOCKS para el repositorio con la ip y el puerto indicados, además hay varias trazas sin errores indicando intercambio de mensajes con el otro repositorio con el método CONSISTENCY_SESSION indicado:

```
FILES::CONSISTENCY_SESSION      [to: 192.168.1.63:10000]
```

A priori, la implementación funciona y hay intercambio de información.

Caso 3: Réplica entre dos repositorios e integridad de los datos.

Número de réplicas que participan: **2**
Número de Quorum: 1
Usuario: usu

Qué se busca: una vez copiados los datos entre repositorios, finalizar la ejecución de uno de los mismos y ver si las imágenes permanecen en el nuevo repositorio y son descargables, en resumen, ver la integridad de los datos con el control de consistencia.

Descripción de la prueba: Esta prueba se va a realizar de la siguiente manera, primero subimos unas imágenes a un solo repositorio, después de subirlas, lanzamos un nuevo repositorio y esperamos a que las sesiones de consistencia se ejecuten para copiar los datos entre los repositorios. Una vez ejecutadas,

eliminamos de la ejecución al primer repositorio, si todo ha ido bien las imágenes y el contenido de los mensajes debe permanecer en el sistema.

Pasos dados y resultados:

Subimos 4 imágenes de gran tamaño y 3 mensajes de texto en el repositorio A, alojado en el puerto 10001, con el usuario usu. Las imágenes ocupan varios bloques cada una de ellas y con un repositorio son descargables sin problemas.

Figura 45: Prueba de integridad con 2 repositorios.

Como vemos en la captura, las imágenes se recuperan bien con un repositorio. El siguiente paso es lanzar un nuevo repositorio (acelerando el Timer que controla las sesiones de consistencia, para no esperar demasiado a que los sistemas intercambien información).

DirectoryServiceServerSide [Java Application]
edu.uoc.directoryService.DirectoryServiceServerSide at localhost:62625
C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22 dic. 2018 18:55:57)
MainFrame [Java Application]
clientApplication.MainFrame at localhost:62685
C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22 dic. 2018 18:56:23)
Repository [Java Application]
repository.Repository at localhost:62989
C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22 dic. 2018 19:03:09)
Repository [Java Application]
repository.Repository at localhost:65491
C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (22 dic. 2018 19:29:46)

Figura 46: Ejecución de dos repositorios

Como vemos en la captura adyacente tenemos dos repositorios en ejecución, tenemos que esperar hasta que las sesiones de consistencia acaben y los datos se copien de un repositorio al otro, y comprobar que las fotos se descargan de forma normal como antes. Una vez esperado ese tiempo (que varía dependiendo de la información que tenga ese repositorio) eliminamos la ejecución del primer repositorio, en este caso el que se ejecuta a las 19:03, dejando el repositorio de las 19:29.

Mandamos un nuevo mensaje de texto y revisamos las imágenes recuperadas por el TIMELINE, la prueba es satisfactoria y tenemos todos los datos de nuevo en el nuevo repositorio.

Figura 47: Contenido en el nuevo repositorio.

Caso 4: Controles de consistencia entre 3 repositorios

Número de réplicas que participan: **3**
Número de Quorum: 1
Usuario: usu2

Qué se busca: La prueba es en esencia la misma que en el caso 3, no obstante añadir un repositorio adicional puede ocasionar algún fallo que no se contemplase en el caso anterior, ahora la información debe fluir entre los 3 repositorios (y hay dos usuarios que permanecen en las bases de datos de dos repositorios por lo que hay una mayor cantidad de información).

Descripción de la prueba: Registramos un usuario (usu2) con un único repositorio activo, subimos 3 fotos de distintos tamaños de bloque y lanzamos dos repositorios más. Cuando las sesiones de consistencia finalicen entre todos los repositorios vamos a ir tirando los repositorios por fecha ascendente (los más antiguos primero) y vamos a comprobar la estabilidad del sistema.

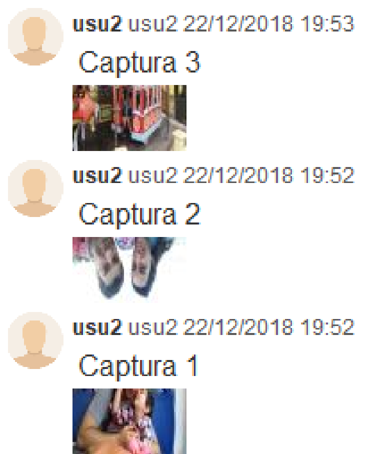


Figura 48: Nuevas imágenes para el caso 4.

Resultados de la prueba: Tenemos 3 repositorios en ejecución, ocupan los puertos 10000 a 10002, con distintas horas de arrancado. El resultado esperado es quedarnos con el repositorio que ocupa el puerto 10002 y que es el último en ser lanzado, y que contenga las capturas que subimos al primer repositorio. Los resultados tras cerrar los dos primeros repositorios son correctos, la información está en el último repositorio.

Caso 5: Bloques con error entre repositorios.

Número de réplicas que participan: 2
Número de Quorum: 1
Usuario: usu2

Qué se busca: ver el comportamiento entre dos repositorios cuando hay bloques de una subida defectuosa.

Descripción de la prueba: Vamos a realizar un proceso de subida de una imagen al repositorio que tenemos activo, pero vamos a interrumpir la subida antes de la operación put_apply, por lo que los bloques no van a subirse, de forma que en la base de datos van a quedar en estado RESERVED. Estos bloques no deben copiarse al repositorio 2, a continuación subiremos una imagen, esta vez de forma completa, de manera que al repositorio 2 deben copiarse los bloques cuyo estado sea únicamente VALID.

Resultados: Paramos la ejecución del repositorio en el método PUT_RESERVE (lo que también sirve para ver la tolerancia ante fallos del resto de la arquitectura), esto provoca un fallo en la subida y el mensaje no se transmite. Subimos otra imagen con el nombre de captura 5. Ahora lanzamos un nuevo repositorio que recordemos, no debe solicitar los timestamp cuyo estado sea

RESERVED. El nuevo repositorio recupera los timestamp 1…7 y 9-10, que pertenecen a las capturas 1, 2, 3 y 5. Es por tanto un resultado correcto. El control de consistencia no ha solicitado las filas cuyo estado no es válido.

Integración del código en Garlanet

Una vez terminado el desarrollo del proyecto hay que fusionar la rama que corresponde al desarrollo actual (rama Files) con la rama principal del proyecto (master). Una vez fusionados los proyectos, este desarrollo estará integrado en Garlanet.

5. Mejoras pendientes

Dentro del trabajo que ha supuesto Garlanet quedan pendientes algunas implementaciones para dar por concluido todo el desarrollo, que no han podido ser incluidas en el presente trabajo por falta de tiempo. A continuación paso a enumerarlas y a proponerlas para una ampliación de este trabajo:

Cifrado de los bloques

Una de las premisas de Garlanet es una fuerte privacidad. Para conseguir esto es necesario encriptar los bloques que se envían desde la GUI hasta los repositorios. Era uno de los objetivos del proyecto pero que finalmente por falta de tiempo no ha podido ser incluido, se propone por tanto, como mejora principal del proyecto, el cifrado de los bloques en el proyecto Application.

Soporte de otros archivos multimedia

Actualmente el microservicio Files no soporta otros tipos de archivos tales como PDF, MP4, doc, etc… Una de las posibles mejoras a proponer es la de incluir usando la infraestructura creada, el soporte a estas extensiones. Al hacer uso de la infraestructura creada debe ser sencillo implementar esta funcionalidad, además de ampliar de forma considerable las posibilidades de Garlanet.

Concatenar las preview de las imágenes tras los bloques de Metainformación

El resultado de incluir las preview de las imágenes como parte de la metainformación es que los bloques de metainformación crecen de tamaño, a mayor imagen, mayor tamaño de preview. En principio esto no es un problema, no obstante lo ideal es que las preview vayan fuera de la metainformación y esta contenga únicamente la información necesaria para delimitar los bloques de preview y poder recuperarlos al igual que se hace con el resto de los ficheros. Además serviría para incluir preview de otro tipo de archivos como vídeos o música.

Adaptar la interfaz gráfica html a la nueva funcionalidad.

Garlanet tiene además de la interfaz gráfica adaptada en java swing, otra interfaz en desarrollo realizada en html. La propuesta sería adaptar esta otra interfaz para que incluya el servicio Files.

6. Conclusiones Finales

En la primera fase, hemos creado la infraestructura necesaria para que se extraiga la metainformación de los archivos, se generen los bloques de igual tamaño y Garlanet sea capaz de a partir de un listado de bloques que conformen metainformación y fichero, recuperar ambos y trabajar con ellos.

En la segunda fase del proyecto, hemos desarrollado el microservicio Files tanto en la parte de los usuarios como en la parte de los repositorios y se ha implementado toda la lógica necesaria para que un fichero sea enviado a los repositorios y recuperado y mostrado finalmente al usuario, el cual puede descargar o abrir el fichero en ese preciso momento.

En la tercera y última parte del proyecto se ha implementado la sesión de consistencia entre repositorios para el servicio Files, y se ha añadido un mecanismo de purga para bloques inconsistentes.

El desarrollo del proyecto ha seguido varios criterios de mi propia cosecha, que pueden ser acertados o no, pero que expongo para que sean tenidos en cuenta para trabajos posteriores: He tratado de ser estanco con el desarrollo para afectar lo menos posible al resto de microservicios. He tratado de ser mimético con respecto a la forma de programar que ya existía en el proyecto. Y sobre todo he tratado de ser ordenado y de no emplear métodos demasiado grandes para favorecer la legibilidad del código, salvo en los casos en los que era estrictamente necesario.

En términos generales el proyecto y su planificación han sido correctos, salvo en el último tramo donde las pruebas de software han revelado algunos fallos cuya solución ha llevado más tiempo del deseado.

Por destacar los aspectos positivos del trabajo, sin duda ha habido un gran aprendizaje de la red social Garlanet, del funcionamiento de los sistemas distribuidos, de implementar microservicios en un proyecto de envergadura, y de recuperar conocimientos de un lenguaje como Java, del que ciertamente se puede decir que nunca se llega a dominar del todo.

Como nota agridulce, destacar que finalmente no se han podido securizar los bloques que se envían a los repositorios por falta de tiempo. Siempre destaco el aprendizaje que puedo obtener de mis proyectos y ha sido una pena no poder aprender esta parte de la privacidad en el presente trabajo.

Finalmente, comentar que el resultado del proyecto es funcional y está preparado para ser incluido en la rama principal del control de versiones de Garlanet.

7. Glosario

Array: En programación se le denomina vector a una zona de almacenamiento contiguo que contiene una serie de elementos del mismo tipo.

Api: *Interfaz de programación de aplicaciones*, es un conjunto de subrutinas, funciones y procedimientos que ofrece cierta biblioteca para ser usada por otro software como una capa de abstracción.

Bit: *Binary digit,* es un dígito del sistema de numeración binario.

Byte: Es la unidad de información de base utilizada en computación y en telecomunicaciones, y que resulta equivalente a un conjunto ordenado de 8 bits.

Clave de cifrado: Es una pieza de información que controla la operación de un algoritmo de criptografía.

Cifrado: El cifrado es un procedimiento que usa un algoritmo de cifrado con cierta clave (clave de cifrado) para transformar un mensaje, sin atender a su estructura lingüística o significado, de tal forma que sea incomprensible.

Commit: En el contexto de las ciencias de la computación, se refiere a confirmar un conjunto de cambios de forma provisional o permanente.

Control de versiones: Se llama control de versiones a los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo.

Deserialización: Proceso inverso a la serialización, obtener a partir de una serie de bytes el objeto original.

Eclipse: Es una plataforma de software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma.

Garlanet: Conjunción de la palabra en catalán *garlar* que significa mantener una conversación intrascendente, con *net*, palabra inglesa que significa *red.*

Git: Es un software de control de versiones diseñado por Linus Torvalds, pensado en la eficiencia y la confiabilidad del mantenimiento de versiones.

Gui: *Interfaz gráfica de usuario,* es un programa informático que actúa de interfaz de usuario, usando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz.

IDE: *Integrated development environment,* es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software.

Internet Protocol: Es un protocolo de comunicación de datos digitales clasificado funcionalmente en la capa de red según el modelo OSI.

Dirección ip: Es una etiqueta numérica asignada a un dispositivo conectado a internet que usa *Internet Protocol* para comunicarse.

Java: Es un lenguaje de programación de propósito general, concurrente, orientado a objetos y que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible.

Metadatos: Son datos que describen otros datos. En general, un grupo de metadatos se refiere a un grupo de datos que describen el contenido informativo de un objeto al que se denomina *recurso.*

Microblogging: Servicio de internet que permite intercambiar mensajes breves de texto.

Mime: *Extensiones multipropósito de correo de internet* son una serie de convenciones o especificaciones dirigidas al intercambio a través de internet de todo tipo de archivos (texto, vídeo, audio, etc…)

Query: En ciencias de la computación, se refiere a realizar una consulta a una base de datos donde existen datos relacionados con una aplicación, y que devuelve un conjunto de filas que cumplen una condición determinada.

Quórum: Del latín *quórum,* es el número de individuos que se necesita para que un cuerpo deliberante o parlamentario trate ciertos asuntos y pueda tomar una determinación válida.

Path: Es la forma de referenciar a un archivo informático o directorio en un sistema de archivos de un sistema operativo determinado.

Preview: Es una muestra o visualización de algo, generalmente en la informática de un vídeo o de una imagen, dando una idea de lo que se puede visualizar al hacer *click* sobre ese contenido.

Purga: Eliminado de registros.

Push: En Git una operación *push* es confirmar los cambios en el repositorio en una operación de actualización de ficheros.

Repositorio: Un repositorio es un espacio centralizado donde se almacena, organiza, mantiene y difunde información digital.

Serialización: En ciencias de la computación, la serialización consiste en un proceso de codificación de un objeto en un medio de almacenamiento con el fin de transmitirlo a través de una conexión en red como una serie de bytes.

Switch: En programación el switch es una estructura de control condicional que permite evaluar una variable dependiendo del valor que tome en un momento dado.

SQLite: Es un sistema de gestión de bases de datos relacional compatible con ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) contenida en una biblioteca escrita en C.

Timeout: En la ingeniería computacional, *timeout* es un evento forzado diseñado para ocurrir al final de un tiempo transcurrido determinado.

Timer: Temporizador, en Java, hace referencia a una tarea que se ejecuta de forma autónoma dado un intervalo de tiempo establecido.

Timestamp: Marca temporal o sello de tiempo. Es una secuencia de caracteres que denotan la hora y fecha en las que ocurrió un evento.

Url: *Localizador Uniforme de recursos*, es un identificador de recursos uniforme cuyos recursos referidos pueden cambiar, esto es, la dirección puede apuntar a recursos variables en el tiempo.

8. Bibliografía

[1] Varios autores (2013). *A Guide to the Project Management Body of Knowledge (PMBOK Guide), Fifth Edition.* PMI Institute. ISBN-10: 1935589679; ISBN-13: 978-1935589679. Capítulo 5. Scope Management.

[2] Joan Manuel Marquès Puig (Rev 2018). *Files Microservice.*

[3] Thierry Groussard (Rev 2013). *Java 7, Bases del lenguaje y de la programación orientada a objetos. Eni-ediciones.* ISBN-10*:* 274607964X ISBN-13: 978-2-7460-7964-9

[4] John Zukowski. (Rev 2011). *The definitive guide to Java Swing. Third Edition.* Springer. ISBN-10: 978-1590594476; ISBN-13: 978-1590594476. Capítulo 4. *Core Swing Components.* Capítulo 5, *Toggle Buttons.* Capítulo 9, *Pop-ups and Choosers.*

[5] Joan Manuel Marquès Puig y Helena Rifà (2018). Garlanet Technical Report. http://dpcs.uoc.edu/projects/garlanet/files/2018-Garlanet-TechnicalReport.pdf

9. Anexos

A continuación se muestran fragmentos del código que son relevantes por las explicaciones que se dan en la memoria y que pueden ser consultados durante la lectura, además de una table con la relación de código y proyectos generados y modificados.

9.1 Tabla de contenido generado y modificado

Nombre	Proyecto	Fases	Original o adaptado
EFileType.java	GarlanetCore	1	Original
EFileSubType.java	GarlanetCore	1	Original
EnumToByteToEnum.java	GarlanetCore	1	Original
FileUtils.java	GarlanetCore	1	Original
MetaInformation.java	GarlanetCore	1	Original
TestMetaInformation.java	GarlanetCore	1	Original
TestFormularioMetaInformation.java	GarlanetCore	1	Original
ConfigurationCommon.java	GarlanetCore	1 y 2	Adaptado
ERepositoryMessageMethod.java	GarlanetCore	2 y 3	Adaptado
Message.java	GarlanetCore	2	Adaptado
Application.java	Application	2 y 3	Adaptado
ApplicationAPI.java	Application	2 y 3	Adaptado
Files.java	Application	2 y 3	Adaptado
FilesAPI.java	Application	2 y 3	Adaptado
FilesDescription.java	Application	2	Adaptado
FilesDescriptionImage.java	Application	2	Adaptado
FilesDescriptionURL.java	Application	2	Original
FileDescriptionUserInterface.java	Application	2	Adaptado
FileDescriptionImageUserInterface.java	Application	2	Adaptado
FileDescriptionURLUserInterface.java	Application	2	Original
Messages.java	Application	2	Adaptado
MessagesUserInterface.java	Application	2	Adaptado
GarlanetUserInterface.java	Application	2	Adaptado
GarlanetUserInterfaceAPI.java	Application	2	Adaptado
MainFrame.java	GUI_swing	2 y 3	Adaptado
JPanelPost.java	GUI_swing	2 y 3	Adaptado
PanelTimelineObj.java	GUI_swing	2 y 3	Adaptado
Resources_XX.properties	GUI_swing	2 y 3	Adaptado
EfileStatus.java	Repository	2	Original
Files.java	Repository	2 y 3	Adaptado
FilesAPI.java	Repository	2 y 3	Adaptado
FilesConsistencyObject.java	Repository	3	Original
PurgeItem.java	Repository	3	Original
PurgeTimer.java	Repository	3	Original
MyServicesTimer.java	Repository	3	Adaptado
PendingBlocksTimer.java	Repository	3	Original

Repository.java	Repository	2 y 3	Adaptado
Parameters.properties	Repository	3	Adaptado

9.2 MetaInformation.java

```
public class MetaInformation {

            private static final char STRING_DELIMITER = '\0';
            private String name;
            private EFileType type;
            private String alternate_text;
            private EFileSubType image_type;
            private String filename_preview;
            private String url;
            private long timestamp_first_block;
            private int num_blocks;
            private int firstbyte_firstblock;
            private int lastbyte_lastblock;
            /**
             * Constructors
             */
            public MetaInformation();

            public MetaInformation(byte [] serializedMetaInformation);

            public MetaInformation(String name,
                         EFileType type,
                         String alternate_text,
                         EFileSubType image_type,
                         String filename_preview,
                         long timestamp_firstblock);

            public MetaInformation(String name,
                         String url,
                         String alternate_text,
                         String filename_preview,
                         long timestamp_firstblock);
            /**
             * Serialize/Deserialize region
             */
            public byte[] serialize();

            public MetaInformation deSerialize(byte [] array;
            /**
             * auxiliary methods
             */
            public  byte[] getLongToBytes(long x);

            public long getBytesToLong(byte[] bytes);

            public byte [] getIntToBytes(int x);

            public int getBytesToInt(byte [] array);

            @Override
            public String toString();
```

```
	//Getters & Setters
	public String getName();

	public void setName(String name);

	public EFileType getType();

	public void setType(EFileType type);

	public String getAlternateText();

	public void setAlternateText(String alternateText);

	public EFileSubType getImageType();

	public void setImageType(EFileSubType image_type);

	public String getFilenamePreview();

	public void setFilenamePreview(String filenamePreview);

	public String getUrl();

	public void setUrl(String url);

	public long getTimestampFirstBlock();

	public void setTimestampFirstBlock(long timestampFirstBlock);

	public int getNumBlocks();

	public void setNumBlocks(int numBlocks);

	public int getFirstByteFirstBlock();

	public void setFirstByteFirstBlock(int firstByteFirstBlock);

	public int getLastByteLastBlock();

	public void setLastByteLastBlock(int lastByteLastBlock);

}
```

9.3 MetaInformation.serialize

```
public byte[] serialize() {
	byte [] result = null;
	byte [] bname = null;
	byte btype = 0;
	byte bimage_type = 0;
	byte [] bfilename_preview = null;
	byte [] balternate_text = null;
	byte [] burl = null;
	byte [] btimestamp = null;
	byte [] bnumblocks = null;
	byte [] bfirstbyte_firstblock = null;
	byte [] blastbyte_lastblock = null;

	try {
```

```
			bname = getName().getBytes("UTF-8");
		} catch (UnsupportedEncodingException e) {
			bname = getName().getBytes();
		}
		btype = getType().getByte();
		try {
			balternate_text = getAlternateText().getBytes("UTF-8");
		} catch (UnsupportedEncodingException e) {
			balternate_text = getAlternateText().getBytes();
		}
		btimestamp = getLongToBytes(getTimestampFirstBlock());
		bnumblocks = getIntToBytes(getNumBlocks());
		bfirstbyte_firstblock = getIntToBytes(getFirstByteFirstBlock());
		blastbyte_lastblock = getIntToBytes(getLastByteLastBlock());
		if (getType() == EFileType.URL) {


			try {
				burl = getUrl().getBytes("UTF-8");
			} catch (UnsupportedEncodingException e) {
				burl = getUrl().getBytes();
			}
			result = new byte[bname.length + EFileType.size() +
balternate_text.length+burl.length+ConfigurationGarlanetCore.SIZE_OF_LONG+bnu
mblocks.length+bfirstbyte_firstblock.length + blastbyte_lastblock.length];
		}
		else if (getType() == EFileType.IMAGE) {
			bimage_type = getImageType().getByte();
			try {
				bfilename_preview = getFilenamePreview().getBytes("UTF-
8");
			}catch (UnsupportedEncodingException e) {
				bfilename_preview = getFilenamePreview().getBytes();
			}
			result = new byte[bname.length + EFileType.size() +
balternate_text.length+EFileSubType.size()+bfilename_preview.length +
ConfigurationGarlanetCore.SIZE_OF_LONG+bnumblocks.length+bfirstbyte_firstbloc
k.length + blastbyte_lastblock.length];
		}
		System.arraycopy(btimestamp, 0, result, 0,
ConfigurationGarlanetCore.SIZE_OF_LONG);
		System.arraycopy(bnumblocks, 0, result,
ConfigurationGarlanetCore.SIZE_OF_LONG, bnumblocks.length);
		System.arraycopy(bfirstbyte_firstblock, 0, result,
ConfigurationGarlanetCore.SIZE_OF_LONG + bnumblocks.length,
bfirstbyte_firstblock.length);
		System.arraycopy(blastbyte_lastblock, 0, result,
ConfigurationGarlanetCore.SIZE_OF_LONG + bnumblocks.length +
bfirstbyte_firstblock.length, blastbyte_lastblock.length);
		System.arraycopy(bname, 0, result,
ConfigurationGarlanetCore.SIZE_OF_LONG + bnumblocks.length +
bfirstbyte_firstblock.length+ blastbyte_lastblock.length, bname.length);
		result[ConfigurationGarlanetCore.SIZE_OF_LONG + bnumblocks.length +
bfirstbyte_firstblock.length+ blastbyte_lastblock.length + bname.length] =
btype;
		System.arraycopy(balternate_text, 0, result,
ConfigurationGarlanetCore.SIZE_OF_LONG + bnumblocks.length +
bfirstbyte_firstblock.length + blastbyte_lastblock.length + bname.length +
EFileType.size(), balternate_text.length);
```

```
        switch (getType())
        {
                case URL:
                        System.arraycopy(burl, 0, result, 
ConfigurationGarlanetCore.SIZE_OF_LONG + bnumblocks.length + 
bfirstbyte_firstblock.length + blastbyte_lastblock.length + bname.length + 
EFileType.size() + balternate_text.length, burl.length);

                break;
                case IMAGE:
                        result[ConfigurationGarlanetCore.SIZE_OF_LONG + 
bnumblocks.length + bfirstbyte_firstblock.length+ blastbyte_lastblock.length 
+ bname.length + EFileType.size() + balternate_text.length] = bimage_type;
                        System.arraycopy(bfilename_preview, 0, 
result,ConfigurationGarlanetCore.SIZE_OF_LONG + bnumblocks.length + 
bfirstbyte_firstblock.length+ blastbyte_lastblock.length + bname.length + 
EFileType.size() + balternate_text.length + EFileSubType.size(), 
bfilename_preview.length);
                        break;
        }
        return result;
        }
```

9.4 MetaInformation.deserialize

```
public MetaInformation deSerialize(byte [] array) {
        MetaInformation result = new MetaInformation();
        int indice = 0;
        int aux = 0;

        result.setTimestampFirstBlock(getBytesToLong(Arrays.copyOfRange(array, 
indice, indice+ConfigurationGarlanetCore.SIZE_OF_LONG)));
        indice=indice+ConfigurationGarlanetCore.SIZE_OF_LONG;

        result.setNumBlocks(getBytesToInt(Arrays.copyOfRange(array, indice, 
indice+ConfigurationGarlanetCore.SIZE_OF_INT)));
        indice=indice+ConfigurationGarlanetCore.SIZE_OF_INT;

        result.setFirstByteFirstBlock(getBytesToInt(Arrays.copyOfRange(array, 
indice, indice+ConfigurationGarlanetCore.SIZE_OF_INT)));
        indice=indice+ConfigurationGarlanetCore.SIZE_OF_INT;

        result.setLastByteLastBlock(getBytesToInt(Arrays.copyOfRange(array, 
indice, indice+ConfigurationGarlanetCore.SIZE_OF_INT)));
        indice=indice+ConfigurationGarlanetCore.SIZE_OF_INT;
        for (int i=indice;i<array.length;i++) {
                if ((char)array[i]==STRING_DELIMITER) {
                        aux = i-indice;
                        break;
                }
        }

        try {
                result.setName(new String(array,indice,aux, "UTF-8"));
        } catch (UnsupportedEncodingException e) {
                result.setName(new String(array,indice,aux));
        }
        indice = indice+aux+1;
```

```
	result.setType((EFileType)EFileType.APPLICATION.getEnum(array[indice])
);
	indice++;
	for (int i=indice;i<array.length;i++) {
		if ((char)array[i]==STRING_DELIMITER) {
			aux = i-indice;
			break;
		}
	}
	try {
		result.setAlternateText(new String(array,indice,aux,"UTF-8"));
	}catch(UnsupportedEncodingException e) {
		result.setAlternateText(new String(array,indice,aux));
	}
	indice = indice+aux+EFileType.size();
	switch (result.getType()) {
		case IMAGE:

	result.setImageType((EFileSubType)EFileSubType.FORMDATA.getEnum(array[
indice]));
		indice++;
		for (int i=indice;i<array.length;i++) {
			if ((char)array[i]==STRING_DELIMITER) {
				aux=i-indice;
				break;
			}
		}
		try {
			result.setFilenamePreview(new
String(array,indice,aux,"UTF-8"));
		}catch(UnsupportedEncodingException e) {
			result.setFilenamePreview(new String(array,indice,aux));
		}
			indice=indice+aux+EFileType.size();

		break;
		case URL:
			for(int i=indice;i<array.length;i++) {
				if ((char)array[i]==STRING_DELIMITER){
					aux=i-indice;
					break;
				}
			}
			try {
				result.setUrl(new String(array,indice,aux,"UTF-
8"));
			}catch(UnsupportedEncodingException e) {
				result.setUrl(new String(array,indice,aux));
			}
			indice=indice+aux+EFileSubType.size();
			break;
	}
	return result;
}
```

9.5 FileUtils.SplitIntoBlockFile

```
public static List<byte[]> splitIntoBlockFile(byte [] metaInformation)
```

```
{
        List<byte[]> blocks = new ArrayList<>();
        short num_block_metainformation = 0;
        int size_of_metainformation = METAINFORMATION_SIZE +
metaInformation.length;
        String head_blocks = "";
        String last_byte_metainformation = "";
        int int_last_byte_meta=0;
        if (size_of_metainformation<=ConfigurationGarlanetCore.BLOCK_SIZE-
METAINFORMATION_SIZE) {
                num_block_metainformation = 0;
                int_last_byte_meta = METAINFORMATION_SIZE +
metaInformation.length;
        }else if (size_of_metainformation>ConfigurationGarlanetCore.BLOCK_SIZE
&& size_of_metainformation<=(ConfigurationGarlanetCore.BLOCK_SIZE*2)) {
                num_block_metainformation = 1;
                int_last_byte_meta = metaInformation.length -
(ConfigurationGarlanetCore.BLOCK_SIZE-METAINFORMATION_SIZE);
        }else if
(size_of_metainformation>(ConfigurationGarlanetCore.BLOCK_SIZE*2) &&
size_of_metainformation<=(ConfigurationGarlanetCore.BLOCK_SIZE*3)) {
                num_block_metainformation = 2;
                int_last_byte_meta = metaInformation.length -
((ConfigurationGarlanetCore.BLOCK_SIZE*2)-METAINFORMATION_SIZE);
        }else if
(size_of_metainformation>(ConfigurationGarlanetCore.BLOCK_SIZE*3) &&
size_of_metainformation<=(ConfigurationGarlanetCore.BLOCK_SIZE*4)) {
                num_block_metainformation = 3;
                int_last_byte_meta = metaInformation.length -
((ConfigurationGarlanetCore.BLOCK_SIZE*3)-METAINFORMATION_SIZE);
        }
        head_blocks =
completeBit(Integer.toBinaryString(num_block_metainformation),2);
        last_byte_metainformation =
completeBit(Integer.toBinaryString(int_last_byte_meta),22);
        String string_three_bytes = head_blocks+last_byte_metainformation;

        byte [] head = binaryStringToByteArray(string_three_bytes);
        int i=0;
        int j=0;
        while (i<=num_block_metainformation) {
                byte [] block = new byte[ConfigurationGarlanetCore.BLOCK_SIZE];
                int index = 0;
                if (i==0) {

                        block[0] = head[0];
                        block[1] = head[1];
                        block[2] = head[2];
                        index = 3;
                }
                System.arraycopy(metaInformation, j, block, index,
(block.length-index<metaInformation.length-j) ? block.length-index :
metaInformation.length-j);
                blocks.add(block);
                j=j+block.length-index;
                i++;
        }
        return blocks;
}
```

9.6 FileUtils.getBlocks

```
public static List<byte []> getBlocks(MetaInformation meta, File file) {

	int firstbyte_firstblock =
((meta.serialize().length+METAINFORMATION_SIZE) %
ConfigurationGarlanetCore.BLOCK_SIZE)+1;

	byte[] byte_file = null;
	try {
		byte_file = Files.readAllBytes(file.toPath());
	} catch (IOException e) {
		e.printStackTrace();
	}
	int num_blocks = (byte_file.length /
ConfigurationGarlanetCore.BLOCK_SIZE) + 1;
	int total = 0;
	if (num_blocks == 1 ) {
		total = byte_file.length+firstbyte_firstblock;
	}else {
		total = ((byte_file.length -
ConfigurationGarlanetCore.BLOCK_SIZE + (firstbyte_firstblock-1)) %
ConfigurationGarlanetCore.BLOCK_SIZE);
	}
	meta.setNumBlocks(num_blocks);
	meta.setFirstByteFirstBlock(firstbyte_firstblock);
	meta.setLastByteLastBlock(total);

	List<byte []> blocks = splitIntoBlockFile(meta);
	int i = 0;
	int src_cont = 0;
	int dst_pos = firstbyte_firstblock;
	int length_pos;
	if ((int)file.length()-( ConfigurationGarlanetCore.BLOCK_SIZE -
firstbyte_firstblock)<=0) {
		length_pos = (int)file.length();
	}else {
		length_pos = ConfigurationGarlanetCore.BLOCK_SIZE -
firstbyte_firstblock;
	}
	while (i < num_blocks) {
		System.arraycopy(byte_file, src_cont,blocks.get(i), dst_pos,
length_pos);
		i++;
		src_cont+=length_pos;
		if (src_cont + ConfigurationGarlanetCore.BLOCK_SIZE >
byte_file.length) {
			length_pos = byte_file.length-src_cont;
		}else {
			length_pos = ConfigurationGarlanetCore.BLOCK_SIZE;
		}
		dst_pos = 0;
		byte [] block = new byte[ConfigurationGarlanetCore.BLOCK_SIZE];
		blocks.add(block);
	}
	blocks.remove(blocks.size()-1);
	byte [] random = new byte[ConfigurationGarlanetCore.BLOCK_SIZE-total];
	new Random().nextBytes(random);
```

```
	System.arraycopy(random, 0, blocks.get(num_blocks-1), total+1, 
random.length-1);
	return blocks;
}
```

9.7 FileUtils.getMetaInformationFromBlocks

```
public static byte[] getMetaInformationFromBlocks(List<byte[]> blocks) {

		byte[] first_block = blocks.get(0);
		byte first_byte = first_block[0];
		byte second_byte = first_block[1];
		byte third_byte = first_block[2];
		String string_first_byte = String.format("%8s", 
Integer.toBinaryString(first_byte & 0xFF)).replace(' ', '0');
		String string_second_byte = String.format("%8s", 
Integer.toBinaryString(second_byte & 0xFF)).replace(' ', '0');
		String string_third_byte = String.format("%8s", 
Integer.toBinaryString(third_byte & 0xFF)).replace(' ', '0');
		String last_byte_with_metainformation = 
string_first_byte.substring(2) + string_second_byte + string_third_byte;
		int num_blocks = Integer.parseInt(string_first_byte.substring(0, 
2), 2);
		int last_byte = Integer.parseInt(last_byte_with_metainformation, 
2);
		byte [] serialized_metainformation = new 
byte[(ConfigurationGarlanetCore.BLOCK_SIZE*(num_blocks))+(last_byte-
METAINFORMATION_SIZE)];
		int i=0;
		int index=3;
		int index_metainformation=0;
		int length=0;
		while (i<=num_blocks) {
			if (i==num_blocks) {
				length=last_byte-index;
			}else {
				length=ConfigurationGarlanetCore.BLOCK_SIZE-index;
			}
			System.arraycopy(blocks.get(i), index, 
serialized_metainformation, index_metainformation, length);
			index_metainformation+=length;
			index=0;
			i++;
		}
		return serialized_metainformation;
	}
```

9.8 FileUtils.getObjectFromBlocks

```
public static void getObjectFromBlocks(List<byte []> blocks) {
	MetaInformation m = new 
MetaInformation(getMetaInformationFromBlocks(blocks));
	byte [] file_bytes = new byte[(ConfigurationGarlanetCore.BLOCK_SIZE-
m.getFirstByteFirstBlock())+m.getLastByteLastBlock()+(ConfigurationGarlanetCo
re.BLOCK_SIZE*(m.getNumBlocks()-2)+1)];
	int src_pos = m.getFirstByteFirstBlock();
	int dst_pos=0;
```

```
        int long_cpy = ConfigurationGarlanetCore.BLOCK_SIZE -
m.getFirstByteFirstBlock();
        if(m.getNumBlocks() == 1) {
                System.arraycopy(blocks.get(0), src_pos, file_bytes, dst_pos,
file_bytes.length);
        }else {
                for (int i=0;i<m.getNumBlocks();i++) {
                System.arraycopy(blocks.get(i), src_pos, file_bytes, dst_pos,
long_cpy);
                dst_pos += long_cpy;
                if (i==m.getNumBlocks()-2) {
                        long_cpy = m.getLastByteLastBlock();
                }else {
                        long_cpy = ConfigurationGarlanetCore.BLOCK_SIZE;
                }
                        src_pos=0;
        }
}
        try {
                Files.write(new File("C:\\"+m.getName()).toPath(), file_bytes);
        } catch (IOException e) {
                e.printStackTrace();
        }
}
```

9.9 FileDescriptionUserInterface

```
public class FileDescriptionUserInterface {

        private long id = Configuration.NULL_FILE_ID;
        private EFileType type = null;
        private EFileSubType subtype = null;
        private String filename = "";
        private String filename_tmp = "";


        // Default constructor
        public FileDescriptionUserInterface(EFileType type, EFileSubType
subtype, String filename, String filename_tmp){
                id = System.currentTimeMillis();
                this.type = type;
                this.subtype = subtype;
                this.filename = filename;
                this.filename_tmp = filename_tmp;
        }

        public FileDescriptionUserInterface(FileDescription file_description){
                this.filename_tmp = file_description.getFilename_tmp();
                this.type = file_description.getType();
                this.subtype = file_description.getSubtype();
                this.filename = file_description.getFilename();
        }

        // --------------
        public static List<FileDescriptionUserInterface>
toListFileDescriptionUserInterface(List<FileDescription> list_files) {
                List<FileDescriptionUserInterface> return_list = new
ArrayList<FileDescriptionUserInterface>();
                for (FileDescription file_description : list_files) {
```

```
                    switch(file_description.getType()){
                    case IMAGE:
                            return_list.add(new
FileDescriptionImageUserInterface((FileDescriptionImage) file_description));
                            break;
                    case URL:
                            return_list.add(new
FileDescriptionURLUserInterface((FileDescriptionURL) file_description));
                            break;
                    }
            }
            return return_list;
    }
```

9.10 FileDescription

```
public class FileDescription {
    private long id = Configuration.NULL_FILE_ID;
    private EFileType type = null;
    private EFileSubType subtype = null;
    private String filename = "";
    private String filename_tmp = "";

    public FileDescription(EFileType type, EFileSubType subtype, String
filename, String filename_tmp){
            this.filename_tmp = filename_tmp;
            this.type = type;
            this.subtype = subtype;
            this.filename = filename;
    }

    public FileDescription(FileDescriptionUserInterface
file_description_user_interface){
            id = System.currentTimeMillis();
            this.filename_tmp =
file_description_user_interface.getFilename_tmp();
            this.type = file_description_user_interface.getType();
            this.subtype = file_description_user_interface.getSubtype();
            this.filename = file_description_user_interface.getFilename();
    }

    // --------------

    public static List<FileDescription>
toListFileDescription(List<FileDescriptionUserInterface> list_files) {
            List<FileDescription> return_list = new
ArrayList<FileDescription>();
            for (FileDescriptionUserInterface file_description : list_files)
{
                    switch(file_description.getType()){
                    case IMAGE:
                            return_list.add(new
FileDescriptionImage((FileDescriptionImageUserInterface) file_description));
                            break;
                    case URL:
                            return_list.add(new
FileDescriptionURL((FileDescriptionURLUserInterface) file_description));
                            break;
```

```
                        }
                }
                return return_list;
        }
```

9.11 Application.post

```
public EGarlanetUserInterfaceStatus post(int functionality_id, String text,
List<FileDescription> files_description) {
                System.err.println("WORK IN PROGRESS");
                EGarlanetUserInterfaceStatus result = null;
                //all the operations must be atomic for FileDescription
                for (FileDescription f : files_description) {

                        List<byte [] > blocks = null;
                        MetaInformation mi = null;
                        File file = null;
                        if (f instanceof FileDescriptionImage) {
                                file = new
File(Configuration.getInstance().getTempPath()+f.getFilename_tmp());
                                byte [] preview = null;
                                try {
                                        preview =
FileUtils.getPreviewFromImage(file,f.getSubtype().toString().toLowerCase(),
Configuration.getInstance().getTempPath());
                                }catch(Exception ex) {
                                        ex.printStackTrace();
                                }
                                mi = new MetaInformation(f.getFilename(),
f.getType(), ((FileDescriptionImage) f).getAlternate_text(), f.getSubtype(),
preview, System.currentTimeMillis());
                                blocks = FileUtils.getBlocks(mi, file);
                        }else if (f instanceof FileDescriptionURL) {
                                mi = new MetaInformation(f.getFilename(),
((FileDescriptionURL) f).getURL(), f.getFilename_tmp(),
"",System.currentTimeMillis());
                                blocks = FileUtils.splitIntoBlockFile(mi);
                        }

                        long upload_id = nextUploadId();
                        int num_blocks = blocks.size();
                        //creation of files_microservice upload

                        long first_timestamp = Configuration.NULL_TIMESTAMP;

                        first_timestamp = files.putRequest(upload_id, num_blocks);

                        if (first_timestamp == Configuration.NULL_TIMESTAMP) {
                                // The function returns error, return ERROR
                                result = EGarlanetUserInterfaceStatus.ERROR;
                        } else {
                                long aux_timestamp = first_timestamp;
                                EStatus status = files.putReserve(upload_id,
num_blocks, first_timestamp);

                                if (status == EStatus.OK) {
                                        // upload blocks, one by one
                                        EStatus aux_status = null;
                                        for (byte[] b : blocks) {
```

```
                                        aux_status = files.putApply(upload_id,
aux_timestamp, b);
                                        aux_timestamp++;
                                        if (aux_status != EStatus.OK) {
                                                break;
                                        }
                                }
                                //finally, send message with first_timestamp
                                if (aux_status == EStatus.OK) {
                                        result = post(functionality_id, text,
first_timestamp);
                                }else {
                                        result =
EGarlanetUserInterfaceStatus.ERROR;
                                }

                        } else {
                                result = EGarlanetUserInterfaceStatus.ERROR;
                        }
                }

                file.delete();
        }
        return result;
}
```

9.12 Repository.handleFiles

```
private void handleFiles(RepositoryMessageObject message, String tablename) {
        ERepositoryMessageMethod method = message.getMethod();
        List<Object> parameters = message.getParameters();
        List<Object> response = new ArrayList<Object>();
        List<SecureMessagesMessage> secure_message_list = null;
        switch(method) {
        case PUT_REQUEST:
                if (parameters.size() != 2) {
                        System.out.println("Error ");
                        response.add(0);
                }else {
                        long upload_id = (long) parameters.get(0);
                        int num_blocks = (int) parameters.get(1);
                        long timestamp = files.putRequest(upload_id,
num_blocks, tablename);
                        response.add(timestamp);
                }
                break;
        case PUT_RESERVE:
                if (parameters.size() != 3) {
                        System.out.println("Error ");
                }else {
                        long upload_id = (long)parameters.get(0);
                        int num_blocks = (int)parameters.get(1);
                        long first_timestamp = (long)parameters.get(2);
                        EStatus status = files.putReserve(upload_id,
first_timestamp, num_blocks, tablename);
                        response.add(status);
                }
                break;
        case PUT_APPLY:
```

```
                    if (parameters.size() != 3) {
                          System.out.println("Error ");
                    }else {
                          long upload_id = (long)parameters.get(0);
                          long timestamp = (long)parameters.get(1);
                          byte [] block = (byte [])parameters.get(2);
                          EStatus status = files.putApply(upload_id,
timestamp, block, tablename);
                          response.add(status);
                    }
              case GET_FILE_BLOCKS:
              case GET_FILE:
                    if (parameters.size() != 1) {
                          System.out.println("Error ");
                    }else {
                          long timestamp = (long)parameters.get(0);
                          List<byte[]> blocks = files.get(timestamp,
tablename);
                          response.add(blocks);
                    }
                    break;
              case CONSISTENCY_SESSION:
              message.setResponse(response);
              this.getListener().answerSession(message.getSessionID(),
message);
      }
```

9.13 PurgeTimer.timeout

```
private void timeout() {
              long startTimeMs = System.currentTimeMillis( );
              try {
                    List<Service> services = ds.myServices();
                    if (services != null){
                          for (Service s : services) {
                                ERepositoryMessageType service_type =
ERepositoryMessageType.valueOf(s.getServiceType().getName());
                                if (service_type ==
ERepositoryMessageType.FILES) {
                                      List<PurgeItem> items = new
ArrayList<PurgeItem>();
                                      items =
files.getPurgeItems(s.getServiceId());
                                      //Purge List set data
                                      if
(list_purge_blocks.containsKey(s.getServiceId())) {

      list_purge_blocks.get(s.getServiceId()).addAll(items);
                                      }else {

      list_purge_blocks.put(s.getServiceId(), new HashSet<PurgeItem>());

      list_purge_blocks.get(s.getServiceId()).addAll(items);
                                      }
                                      //End of set data in list

                                      //Now, check purge blocks and remove
it
```

```
                                        Set<PurgeItem> set_items =
list_purge_blocks.get(s.getServiceId());
                                        for (PurgeItem item: set_items) {
                                              if (files.purgeBlock(item,
s.getServiceId())) {
                                                    set_items.remove(item);
                                              }
                                        }

      list_purge_blocks.put(s.getServiceId(), set_items);
                                  }
                            }

                      }
                      long taskTimeMs  = System.currentTimeMillis( ) -
startTimeMs;
                }catch(Exception e) {
                      e.printStackTrace();
                }
      }
```