



GRADO EN INGENIERÍA TELEMÁTICA



VNIVERSITAT
D VALÈNCIA

TRABAJO DE FIN DE GRADO

APLICACIÓN DISTRIBUIDA MEDIANTE COLAS DE MENSAJES EN EL ÁMBITO DEL PROCESADO DE IMÁGENES

AUTOR:

VÍCTOR IGUALADA CALATRAVA

TUTORÍA:

JUAN GUTIÉRREZ AGUADO

JUNIO 2019

GRADO EN INGENIERÍA TELEMÁTICA

TRABAJO DE FIN DE GRADO

**APLICACIÓN DISTRIBUIDA MEDIANTE COLAS DE
MENSAJES EN EL ÁMBITO DEL PROCESADO DE
IMÁGENES**

AUTOR:

VÍCTOR IGUALADA CALATRAVA

TUTORÍA:

JUAN GUTIÉRREZ AGUADO

TRIBUNAL:

PRESIDENTE/PRESIDENTA:

VOCAL 1:

VOCAL 2:

FECHA DE DEFENSA:

CALIFICACIÓN:

Agradecimientos

Agradecer a Juan Gutiérrez, mi tutor en este proyecto, por inspirarme y hacerme ver cómo quería ser profesional y técnicamente. Alucinando mientras él utilizaba con total dominio y soltura todos los aspectos relacionados con la programación y el desarrollo, me dije a mi mismo que algún día debería llegar a ese nivel.

También me gustaría agradecer a Silvia Rueda y Mariano Pérez, profesores de informática en mi carrera, por hacer que me encantara la programación desde los primeros días del grado hasta años más tarde en asignaturas más complejas. Por enseñarme estrictamente la importancia de las buenas prácticas y explicarme a menudo las cosas varias veces. Otro profesor al que me gustaría agradecer es Francisco Vegara, por ser un profesor dedicado a que sus estudiantes entiendan a la perfección su materia (si estos quieren claro) y por ser tan buena persona y cercano. Siendo las matemáticas lo que me hizo desde joven amar las ciencias, al acceder a la universidad perdí poco a poco el interés. Él consiguió despertar de nuevo ese interés y que mejorara notablemente mis conocimientos.

A mi madre, que ha hecho que esté dónde estoy y a mi padre, que ha hecho que sea como soy.

Por supuesto también agradecer a mis abuelos, que pese a no ser el mejor nieto ni el más atento, me quieren y aceptan.

Y, por último, pero no por ello menos importante, a todos mis amigos y amigas, cuyos nombres no puedo citar porque sería una lista demasiado larga.

Resumen

La computación distribuida representa una tiene como objetivo desplegar una red de computadores capaces de completar un objetivo común y coordinarse mediante el uso de mensajes. Es, sin duda, la solución por excelencia a la hora de programar aplicaciones para computación de alto rendimiento o con un gran número de usuarios. Siendo imposible procesar en tiempo real, por una sola máquina, un número de peticiones del orden de 10^5 en un tiempo aceptable para un usuario humano.

Este paradigma requiere de un sistema de comunicación para asegurar la entrega y el procesado paralelo de las tareas. Las colas de mensajes responden perfectamente a esta problemática. Proveen un protocolo de comunicación asíncrono, eliminando la necesidad del consumidor y del productor de interactuar al mismo tiempo con la cola. Los consumidores pueden acceder al mensaje cuándo esté ocioso y así, asegurar que todos los mensajes se procesan, eventualmente.

El procesado digital de imágenes consiste en un conjunto de técnicas y algoritmos con objetivo de clasificarlas, escalarlas, proyección o reconocimiento de patrones.

La meta de este trabajo es utilizar estos paradigmas y técnicas para detectar rostros humanos en un vídeo enviado por un usuario al sistema y así evaluar el funcionamiento y rendimiento de una arquitectura como la planteada.

Se ha construido una aplicación *web* para recoger las peticiones de los clientes, un *broker* de mensajes que gestionará el sistema de colas, una máquina de almacenamiento y varios *workers*, máquinas encargadas de procesar los mensajes. Para estudiar el funcionamiento y valorar las ventajas de un entorno de estas características, se ha sometido distintas pruebas, con diferentes escenarios, variando el número de *workers* y se han recogido estadísticas del proceso y de los mensajes.

De estos experimentos se han obtenido varias conclusiones. Por un lado, se ha comprobado el correcto funcionamiento del procesado de un vídeo. Por otro, el tiempo de procesado de una tarea para un número N de *workers* es aproximadamente igual al tiempo de procesado de esa misma tarea para un solo *worker* partido por N . Finalmente el tiempo de espera en la cola también ha sido analizado para un número variable de mensajes.

Así se ha estudiado y comprobado la utilidad y eficiencia de un entorno distribuido para el procesado de imagen digital específicamente, pero siendo extrapolable a cualquier otra funcionalidad gracias a la abstracción aportada por el diseño del proyecto.

Palabras clave

Computación distribuida, Colas de mensajes, Procesado de imagen digital, RabbitMQ, OpenCV, Java, FFmpeg, Virtualización.

Tabla de contenido

Resumen.....	1
Palabras clave	1
Tabla de contenido	2
Índice de ilustraciones	5
Índice de tablas.....	7
Relación de acrónimos.....	7
Capítulo 1. Introducción.....	9
1.1. Motivación.....	9
1.2. Objetivos	10
1.3. Metodología	11
1.4. Estructura de la memoria.....	13
Capítulo 2. Estado del arte.....	15
2.1. Virtualización.....	15
2.2. Computación distribuida	18
2.2.1. Definición y objetivos	18
2.2.2. Tipos de sistemas distribuidos.....	18
2.2.3. Comunicación	19
2.3. Colas de mensajes	20
2.3.1. Inicios de las colas de mensajes.....	20
2.3.2. Protocolos para las colas de mensajes.....	22
2.3.3. Message brokers	23
2.4. Lenguajes de desarrollo	29
2.5. Procesamiento de imágenes.....	30
2.5.1. Definición de imagen digital	30
2.5.2. Definición de procesamiento digital de imagen	30
2.5.3. Detección facial.....	31
2.5.4. OpenCV	33
Capítulo 3. Análisis del problema	35
3.1. Análisis de requisitos	35
3.1.1. Requisitos funcionales.....	35
3.1.2. Requisitos no funcionales.....	37

3.2. Análisis de la solución	38
3.2.1. Diagrama de casos de uso	38
3.2.2. Diagramas de Secuencia Generales del Sistema	40
3.3. Análisis de seguridad	41
3.5. Análisis temporal.....	42
3.5.1 Diagrama de Gantt	48
3.6. Estimación de costes	52
3.6.1. Costes de Hardware	52
3.6.2. Costes de Software	52
3.6.3. Costes de Recursos Humanos	53
3.6.3. Costes totales.....	53
3.7. Análisis de riesgos	55
Capítulo 4. Diseño de la solución	59
4.1. Diseño del entorno distribuido	59
4.2. Diseño de las interfaces de usuario	61
4.3. Diseño de colas	62
4.4. Diseño de la comunicación	63
4.4. Arquitectura de software	64
4.4.1. Diagrama de clases	64
4.4.2. Diagramas de interacción de las operaciones del sistema.....	68
Capítulo 5. Implementación	95
5.1. Instalación del entorno de trabajo	95
5.1.1. Instalación de las máquinas virtuales.....	95
5.1.2. Configuración de las interfaces de red	98
5.1.3. Instalación y configuración de software.....	99
5.2. Programación	105
5.2.1. Programación de <i>scripts</i>	105
5.2.2. Programación de la interfaz de usuario	108
5.2.3. Programación de las funcionalidades	109
Capítulo 6. Experimentos y Resultados.....	115
6.1. Descripción de los experimentos	115
6.2. Ejecución de las pruebas.....	116
6.3. Presentación de resultados	119

6.4. Evaluación temporal y presupuestaria	123
Capítulo 7. Conclusiones y trabajos futuros	133
7.1. Conclusiones	133
7.2. Trabajos futuros	134
Referencias	136
Anexo Código Fuente.....	138
Web	138
Storage.....	140
Worker	143
RabbitMQ.....	147

Índice de ilustraciones

Ilustración 1	Estructura de un hipervisor tipo I o nativo	16
Ilustración 2	Estructura de un hipervisor tipo II o hosted	16
Ilustración 3	Ventana de creación de máquina virtual en VirtualBox	17
Ilustración 4	Clonación de una máquina en VirtualBox	18
Ilustración 5	Sistema distribuido organizado como middleware.	19
Ilustración 6	Un ejemplo de alto acoplamiento entre aplicaciones	21
Ilustración 7	Un ejemplo de una distribución basada en message-oriented middleware	22
Ilustración 8	Ejemplo de un broker de mensajes.....	24
Ilustración 9	Cola de mensajes simple	26
Ilustración 10	Cola de Workers	27
Ilustración 11	Publicación de mensaje en dos colas mediante un exchange	27
Ilustración 12	Interfaz de administración de RabbitMQ	28
Ilustración 13	Detección de escamas del algoritmo Viola-Jones para la región ojos-mejillas	32
Ilustración 14	Detección de escamas del algoritmo Viola-Jones para la región ojos-nariz	32
Ilustración 15	Diagrama de flujo del algoritmo de detección facial	33
Ilustración 16	Diagrama de casos de uso de la aplicación	38
Ilustración 17	Diagrama de Secuencia General del Sistema del caso de uso Subir Vídeo.....	40
Ilustración 18	Diagrama de Secuencia General del Sistema del caso de uso Consultar Resultado	40
Ilustración 19	Diagrama de GANTT estimado p.1.....	49
Ilustración 20	Diagrama de GANTT estimado p.2	50
Ilustración 21	Diagrama de GANTT estimado p.3	51
Ilustración 22	Red del entorno distribuido	60
Ilustración 23	Diagrama de flujo de información en la topología	60
Ilustración 24	Interfaz de usuario para la subida de archivos	61
Ilustración 25	Interfaz de Usuario subida correcta	61
Ilustración 26	Interfaz de Ususario del resultado	61
Ilustración 27	Proceso de producción y consumición de tareas	63
Ilustración 28	Proceso de producción y consumición de logs	63
Ilustración 29	Diagrama de paquetes del sistema	65
Ilustración 30	Diagrama de clases de aplicación en Web	66
Ilustración 31	Diagrama de clases de aplicación en Storage	66
Ilustración 32	Diagrama de clases de aplicación en Worker	67
Ilustración 33	Diagrama de clases de paquete Common	67
Ilustración 34	Diagrama de clase de la máquina RabbitMQ	68
Ilustración 35	DIOS main RabbitMQ.....	69
Ilustración 36	DIOS createChannel.....	70

Ilustración 37	DIOS doPost	71
Ilustración 38	DIOS configureUploadFile	73
Ilustración 39	DIOS createUploadDirectory	73
Ilustración 40	DIOS createStatisticsFile.....	74
Ilustración 41	DIOS saveUploadedFile	74
Ilustración 42	DIOS processVideo	75
Ilustración 43	DIOS run.....	76
Ilustración 44	DIOS main ReceiveLog.....	77
Ilustración 45	DIOS createConsumer	78
Ilustración 46	DIOS handleDelivery Storage.....	79
Ilustración 47	DIOS convertFromJson	79
Ilustración 48	DIOS generateStatistics	80
Ilustración 49	DIOS processRequest ServelImagesServlet.....	81
Ilustración 50	DIOS getSortedFiles.....	82
Ilustración 51	DIOS isPngFile	82
Ilustración 52	DIOS constructImageBean.....	83
Ilustración 53	DIOS main Storage	84
Ilustración 54	DIOS createMessage	85
Ilustración 55	DIOS main ReceiveLog.....	86
Ilustración 56	DIOS handleDelivery Storage.....	86
Ilustración 57	DIOS main Worker	88
Ilustración 58	DIOS handleDelivery Worker.....	88
Ilustración 59	DIOS doWork	90
Ilustración 60	DIOS checkDir	91
Ilustración 61	DIOS detectFaces	91
Ilustración 62	DIOS imageToMat	92
Ilustración 63	DIOS copyProcessedFile	92
Ilustración 64	DIOS copyFileToWeb.....	93
Ilustración 65	DIOS logWebAndStorage.....	93
Ilustración 66	DIOS deleteDir	94
Ilustración 67	Configuración inicial máquina virtual en VirtualBox.....	96
Ilustración 68	Configuración del disco virtual en VirtualBox	97
Ilustración 69	Selección de la imagen a montar en VirtualBox	97
Ilustración 70	Configuración de red de una máquina en VirtualBox	98
Ilustración 71	Interfaz del plugin RabbitMQ Web Management	104
Ilustración 72	Estado de las colas en la interfaz de gestión de RabbitMQ	117
Ilustración 73	Interfaz de usuario mientras se sube un archivo	118
Ilustración 74	Interfaz de usuario de una subida correcta	118
Ilustración 75	Interfaz de usuario del resultado del procesamiento.....	118
Ilustración 76	Tiempo de respuesta del sistema	120
Ilustración 77	Función de densidad del tiempo de respuesta	121
Ilustración 78	Tiempo de respuesta para cada frame con cara o no	121
Ilustración 79	Tiempo de espera de un paquete por número de workers	122
Ilustración 80	Comprobación de detección facial	122

Ilustración 82	Diagrama de Gantt real p.1.....	129
Ilustración 83	Diagrama de Gantt real p.2.....	130
Ilustración 84	Diagrama de Gantt real p.3.....	131
Ilustración 85	Diagrama de Gantt real p.4.....	132

Índice de tablas

Tabla 1	Descomposición de tareas del proyecto	44
Tabla 2	Secuencia de las actividades el proyecto	47
Tabla 3	Costes de hardware.....	52
Tabla 4	Costes de software	53
Tabla 5	Costes de recursos humanos con sueldos base de 2017	53
Tabla 6	Costes totales estimados del proyecto	54
Tabla 7	Asignación IP de máquinas del sistema.....	59
Tabla 8	Comparación estimación temporal con real	125
Tabla 9	Costes de hardware reales	126
Tabla 10	Costes de software reales.....	127
Tabla 11	Costes de recursos humanos reales	128
Tabla 12	Costes totales reales.....	128

Relación de acrónimos

AMQP:	Advanced Message Queuing Protocol
APT:	Advanced Packaging Tool
BSD:	Berkeley Software Distribution
CSS:	Cascade Style Sheet
CURL:	Create Update Remove List
DIOS:	Diagrama de Interacción de las Operaciones del Sistema
DSGS:	Diagrama de Secuencia General del Sistema
DMZ:	Demilitarized Zone
DTO:	Data Transfer Object
GB:	Giga Bytes
HTML:	HyperText Markup Language
HTTP:	Hypertext Transfer Protocol
IDE:	Integrated Development Environment
IEC:	Electronic International Comission
IP:	Internet Protocol
ISO:	International Organization for Standardization
IVA:	Impuesto sobre el Valor Añadido

- JDK:** Java Development Kit
- JMS:** Java Message System
- JSON:** JavaScript Object Notation
- JSP:** JavaServer Pages
- JVM:** Java Virtual Machine
- LAN:** Local Area Network
- MD5:** Message-Digest Algorithm 5
- MVC:** Model View Controller
- MOM:** Message-Oriented Middleware
- MQTT:** Message Queue Telemetry Transport
- PHP:** Hypertext Preprocessor
- POJO:** Plain Old Java Object
- PPA:** Personal Package Archive
- RAM:** Random Access Memory
- RMI:** Remote Method Invocation
- RPC:** Remote Procedure Call
- RSA:** Rivest, Shamir y Adleman
- SCP:** Secure Copy
- SHA:** Secure Hash Algorithm
- SOLID:** Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion
- SSH:** Secure Shell
- STOMP:** Simple Text Oriented Message Protocol
- TCP:** Transport
- UUID:** Unique Universal Identifier
- URL:** Uniform Resource Locator
- WAN:** Wide Area Network

Capítulo 1. Introducción

1.1. Motivación

Sistemas de computación distribuidos, a menudo, grandes desconocidos para el usuario, sin embargo, cercanos a los dispositivos y aplicaciones con los que interactúan.

Debido a la abstracción de estos sistemas resulta difícil percibir cuando se está tratando con uno, y este es su objetivo: una red de decenas, centenas o miles de computadores que se comportan como uno solo. No es necesario ir muy lejos para encontrar estos sistemas en el día a día. Grandes redes sociales como Facebook o buscadores como Google aprovechan las ventajas de un modelo distribuido para gestionar y analizar los datos de usuarios y páginas web en decenas de centros de datos por todo el mundo.

En el procesamiento de imágenes, del mismo modo que en otras áreas como los cálculos biomédicos, o el procesamiento de datos meteorológicos, es necesaria gran capacidad computacional, incurriendo costes muy altos para ser procesados por un único computador. Es aquí cuando los sistemas distribuidos triunfan, en lugar de requerir únicamente un supercomputador capaz de realizar el cómputo en tiempos muy pequeños, la división de la tarea entre muchos ordenadores no tan potentes reduce significativamente el tiempo de procesado, acercando el coste temporal al de un supercomputador. Esto permite ofrecer un alto rendimiento a un precio relativamente asequible en comparación con súper ordenadores capaces de realizar estos trabajos de alta carga computacional.

Así, la división y disposición de una tarea entre diferentes máquinas es la raíz de la problemática en entornos distribuidos, debiendo asegurar que todas las partes se computan correctamente para finalmente obtener el trabajo completado con éxito. Para ello es necesario un mecanismo capaz de gestionar las subtareas haciendo hincapié en la integridad de los datos y la confiabilidad de su entrega.

Las colas de mensajes, o *message queues* en inglés, satisfacen la necesidad de gestión de subtareas mediante un sistema de colas capaz de crear una comunicación asíncrona confiable entre computadores. Una vez una tarea ha sido dividida, simplemente se debe encolar el mensaje que hace referencia a cada subtarea, mientras los nodos de la red distribuida, denominados *Workers*, consumen dichos mensajes para su procesamiento, produciendo un nuevo mensaje en otra cola con la tarea finalizada.

El mundo de la informática se rige por una premisa: las tecnologías, marcos de trabajo, técnicas y metodologías cambian, se expanden y se quedan obsoletos día a día. Cualquier solución que se implemente debe ir en acuerdo con esto y en la computación distribuida la extensibilidad y escalabilidad de un sistema es primordial. En este trabajo se intentará respetar dichos principios contando con las facilidades de la virtualización.

Para ilustrar esta situación de cambio y mejora constante qué mejor que un contraejemplo histórico de una de las grandes corporaciones informáticas del mundo:
“Windows NT will be able to address 2GB of RAM, which is more than any application will ever need” Microsoft team, 1992.

1.2. Objetivos

El objetivo de este trabajo es desarrollar una arquitectura distribuida basada en colas de mensajes capaz de distribuir tareas de procesado de imágenes entre diferentes nodos. Este modelo busca, emulando una *IaaS (Infrastructure as a Service)*, ofrecer un servicio de computación distribuida al usuario sin que se preocupe o sea consciente de cómo se está procesando la tarea que ha requerido.

Más específicamente se pretende que mediante un servicio web, un usuario (humano o máquina) sea capaz de enviar un vídeo, que éste sea dividido en *frames*, que por cada frame se crea un mensaje que encapsula la petición de un trabajo, y que sean publicados en una cola para que diferentes nodos o *Workers* los procesen y detecten si aparece algún rostro en ellos. Finalmente, el usuario puede recuperar las imágenes en las que se ha detectado una cara. De todo esto se recogerá datos temporales para analizar el funcionamiento del sistema.

A continuación, se presentan los objetivos desglosados:

- Desplegar una red de computación distribuida (*IaaS*)
- Permitir la subida de un video mediante un servicio web
- Desarrollar una aplicación distribuida para la publicación y consumición de mensajes en una cola.
- Desarrollar una aplicación para el procesado de las imágenes
- Permitir al usuario recuperar las imágenes en las que el proceso produce un resultado positivo
- Probar la eficiencia de un sistema distribuido

Para estudiar el funcionamiento de la infraestructura se realizarán pruebas para 1 a N trabajos con 1 a N *Workers*. Las métricas por analizar serán el tiempo de respuesta del sistema para cada trabajo, y los tiempos de procesado y espera en la cola de un mensaje.

1.3. Metodología

Como en cualquier desarrollo de software, la metodología es crucial y debe estar estructurada y planificada desde el primer momento.

Desde el punto de vista técnico, todo el desarrollo y entorno distribuido se construirá sobre un ordenador portátil con procesador *Intel Core i7 7300k* con 8GB de memoria RAM. La red por diseñar se compone de 6 máquinas virtuales, todas con sistema operativo *Ubuntu 14.04*, y serán virtualizadas mediante el software *Oracle VirtualBox*, que permite interconectarlas entre sí de manera sencilla, creando así una red interna. Estas máquinas son: un servidor Web que actuará de *Front-End* para permitir al cliente subir un archivo de vídeo a la red y de *Mid-End* para comunicarse con el resto del sistema, una máquina de almacenamiento temporal de los datos para su correcto procesado, la máquina que actuará de *Broker* de mensajes (cuyo funcionamiento se detallará más adelante), y finalmente, tres *Workers*. Para el desarrollo de las aplicaciones Java se utilizará el entorno de desarrollo NetBeans IDE, dado el profundo conocimiento obtenido en la carrera. Además, se pretende respetar los principales patrones de diseño de software entre ellos: patrones *SOLID* y *MVC*.

Con respecto a las tecnologías a utilizar, *RabbitMQ* para los procesos de encolado y *OpenCV* para el procesado de imágenes, en este caso de estudio, la detección facial, se explicarán en profundidad en los siguientes capítulos.

A continuación, se describen las diferentes fases que componen el proyecto.

RECOPILACIÓN Y ESTUDIO DE INFORMACIÓN

Del 14/09/2017 al 24/09/2017

La primera fase del proyecto es encontrar, entender y desechar las tecnologías que se utilizarán o no en el proyecto. Es primordial conocer en profundidad los materiales seleccionados para poder comenzar la implementación de los objetivos sin dudar de si la elección ha sido correcta. Determinará el avance y éxito del proyecto, por lo que se hará de manera rigurosa y selectiva.

INSTALACIÓN Y LANZAMIENTO DEL ENTORNO DE TRABAJO

Del 25/09/2017 al 09/10/2017

En esta etapa el objetivo es crear todas las máquinas virtuales, establecer las vías de comunicación entre ellas e instalar todo el software y dependencias necesarios para que sean capaces de llevar a cabo las tareas.

Se deberá instalar el *JDK*, en este caso el 1.8, en todas las máquinas, clientes y servidores *SSH*, un servidor web *Apache Tomcat*, el *broker* de mensajes *RabbitMQ* y las librerías *OpenCV* y *avconv*.

DESARROLLO DE LAS APLICACIONES PRODUCTORAS Y CONSUMIDORAS DE MENSAJES

Del 10/10/2017 al 30/10/2017

Durante este período se desarrollarán las aplicaciones Java encargadas de, a partir de una imagen, encolar desde la máquina *Storage* un mensaje que le haga referencia a ésta en el *Broker*, en forma de una cadena de texto en formato *JSON*; y otra, alojada en los *Workers*, capaz de consumir estos mensajes para su posterior procesado.

DESARROLLO DE LA APLICACIÓN DE PROCESADO DE IMÁGENES

Del 31/10/2017 al 11/11/2017

De nuevo en Java, se desarrollará una aplicación capaz de detectar caras humanas en una imagen. Éste es el trabajo principal de los *Workers* una vez hayan consumido los mensajes.

DESARROLLO DEL PORTAL WEB Y LÓGICA DE SUBIDA DE IMÁGENES

Del 12/11/2017 al 24/11/2017

Aquí se trata de crear un portal web simple que permita a un usuario subir un archivo de vídeo y que el servidor lo almacene y lance un proceso de copia y ejecución de la tarea. Se implementará el portal web con *HTML*, *JavaScript* y *CSS* complementado con la librería *Bootstrap de Twitter* para la parte frontal, es decir el cliente, y con *Java Servlets* para la parte del servidor.

EVALUACIÓN DEL RENDIMIENTO Y PRUEBAS MEDIANTE CARGAS DE TRABAJO

Del 25/11/2017 al 10/12/2017

Puede que la parte más importante del proyecto no sea crear dicha infraestructura distribuida, sino comprobar su funcionamiento y viabilidad. Contrastar los resultados, tiempos de ejecución y de espera, para concluir finalmente que dicho sistema distribuido satisface una necesidad de mayor carga computacional para un computador.

1.4. Estructura de la memoria

En este apartado se pretende guiar al lector a través de las principales fases del proyecto y ofrecer una visión general de éste.

- **Estado del arte:** Investigación y planteamiento de las diferentes tecnologías existentes en el mercado y que podrían emplearse para la realización del proyecto.
- **Análisis:** Definición de los requisitos, estudio de las posibles soluciones para conseguir los objetivos de la manera más eficiente y segura posible y planteamiento de los costes del proyecto.
- **Diseño:** A partir de los resultados del análisis, será posible definir la arquitectura de las aplicaciones, así como los detalles técnicos de la realización.
- **Implementación:** Desarrollo de las aplicaciones y scripts siguiendo el diseño planteado.
- **Pruebas y experimentos:** Engloba las pruebas de usabilidad y de rendimiento del proyecto.
- **Conclusiones y trabajos futuros:** Se plantea el trabajo que no ha sido posible realizar o funcionalidades que serían interesantes añadir en un futuro.



UNIVERSITAT
POLIÈCNICA
DE VALÈNCIA

[] Escola Tècnica
Superior d'Enginyeria

Capítulo 2. Estado del arte

En este capítulo se llevará a cabo un análisis en profundidad de las principales tecnologías que cubren las necesidades del proyecto. Contemplando diferentes alternativas, se justificará el uso de las elegidas para este proyecto.

Se pretende realizar una explicación detallada sobre los principales temas que atañen el proyecto. En primer lugar, se explicará brevemente la virtualización de sistemas informáticos, más tarde, la computación distribuida y los tipos de sistemas distribuidos, luego las colas de mensajes y en concreto la librería *RabbitMQ*, y por último el procesamiento de imagen digital y su vertiente de detección facial. Todo esto justificando la elección de las tecnologías escogidas.

Así, mediante este estudio se pretende recorrer los temas más importantes y exponer cuál es el estado actual del mercado en el ámbito del proyecto.

2.1. Virtualización

La virtualización hace referencia al hecho de crear a través de un *software* una versión virtual de un recurso *hardware* de una máquina o red. El objetivo principal es reducir los costes e inversión en equipos informáticos mientras se obtiene un aprovechamiento de los recursos disponibles.

Expresado de manera diferente, se obtiene la abstracción de recursos de un computador llamada *Virtual Machine Monitor* o *Hypervisor*. Ésta es una capa intermedia entre la máquina física y la virtual, permitiendo la división de los recursos hardware entre anfitrión y huésped¹.

Existen dos tipos de *hipervisores* [1]: por un lado, los de tipo I representados en la Ilustración 1, denominados también nativos, siendo este *software* que se ejecuta directamente sobre los recursos físicos de la máquina, algo poco convencional en ordenadores personales y en pequeñas y medianas empresas. Por otro lado, los de tipo II representados en la Ilustración 2, conocidos como *hosted*, es una aplicación que se ejecuta sobre un sistema operativo en lugar de directamente sobre el *hardware*. Es el más común y con el que se trabajará en este proyecto.

En el caso de este proyecto es imprescindible utilizar la virtualización dado el elevado coste que supondría adquirir equipos informáticos para suplir las 6 máquinas que conforman el sistema distribuido en cuestión. De esta manera con un sólo equipo es posible implementar toda la red y ahorrar en coste económicos, temporales y de implementación ya que la interconexión de las máquinas se llevará a cabo con las herramientas que proporcionan los programas de virtualización y no mediante equipos de red como *routers* o *switches*.

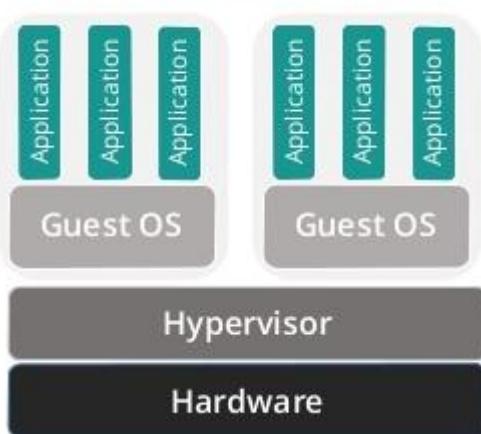


Ilustración 1 Estructura de un hipervisor tipo I o nativo

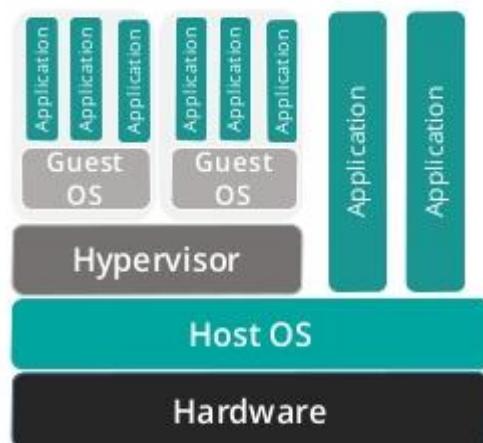


Ilustración 2 Estructura de un hipervisor tipo II o hosted

A continuación, se llevará a cabo un análisis sobre las herramientas disponibles y sobre cuáles se utilizarán finalmente en el desarrollo.

Hoy en día existen muchas alternativas para la virtualización, sin embargo, únicamente se valorarán las dos más utilizadas y gratuitas.

VMWare Workstation Player

VMWare Workstation Player es la versión gratuita de *VMWare Workstation Pro* que no tiene todas las características que trae la versión completa de pago. Entre las características que no están disponibles en la versión gratuita se encuentra la posibilidad de tomar *Snapshots* de una máquina o de ejecutar más de una máquina virtual simultáneamente. Siendo esto último un impedimento para la realización del proyecto. Este producto es uno de los que mejor rendimiento tienen del mercado según diferentes comparativas que se pueden encontrar en la red, sin embargo, las limitaciones que posee a nivel de configuración y ejecución lo descartan totalmente para su uso.

VirtualBox

VirtualBox de *Oracle* es la solución más utilizada para la virtualización de sistemas operativos. Cuenta con la mayoría de las características que posee la versión de pago de *VMWare*, pero de manera totalmente gratuita. Entre ellas la posibilidad de tomar instantáneas de una máquina, de ejecutar tantas máquinas como los recursos del anfitrión puedan soportar, de configurar los dispositivos hardware y de qué manera se van a virtualizar, de crear una red interna de máquinas de manera que sean capaces de comunicarse mediante un *switch* y otras características igual de importantes.

Ya que *VirtualBox* es de uso libre y gratuito, y posee muchas más configuraciones posibles que *VMWare* en su versión gratis, como por ejemplo la posibilidad de construir una red o de ejecutar varias máquinas simultáneamente (tareas indispensables para la realización del proyecto) se elegirá el primero como software para la virtualización [2].

VirtualBox ofrece una manera simple de crear una máquina virtual, especificando nombre, sistema operativo y memoria *RAM* que se le desea asignar. Esto desencadena un proceso guiado donde se especificará el tipo de archivo de almacenamiento (*VirtualBox Disk Image*, *Virtual Hard Disk* o *Virtual Machine Disk*) y el tamaño del almacenamiento.

El proceso de instalación del sistema operativo comienza inmediatamente. En la Ilustración 3 se puede ver el asistente de instalación de *VirtualBox*.

Además, ofrece la posibilidad de clonar máquinas. Esto permite no tener que repetir este proceso con cada máquina, sino que una vez instalada y configurada sólo es necesario clonarla y se obtendrá una máquina virtual idéntica, pero con distinto nombre. Luego sólo será necesario realizar algunos cambios desde el nuevo sistema operativo como la dirección *IP* o el *hostname* de la máquina y listo. En la Ilustración 4 se presenta cómo realizar la clonación.

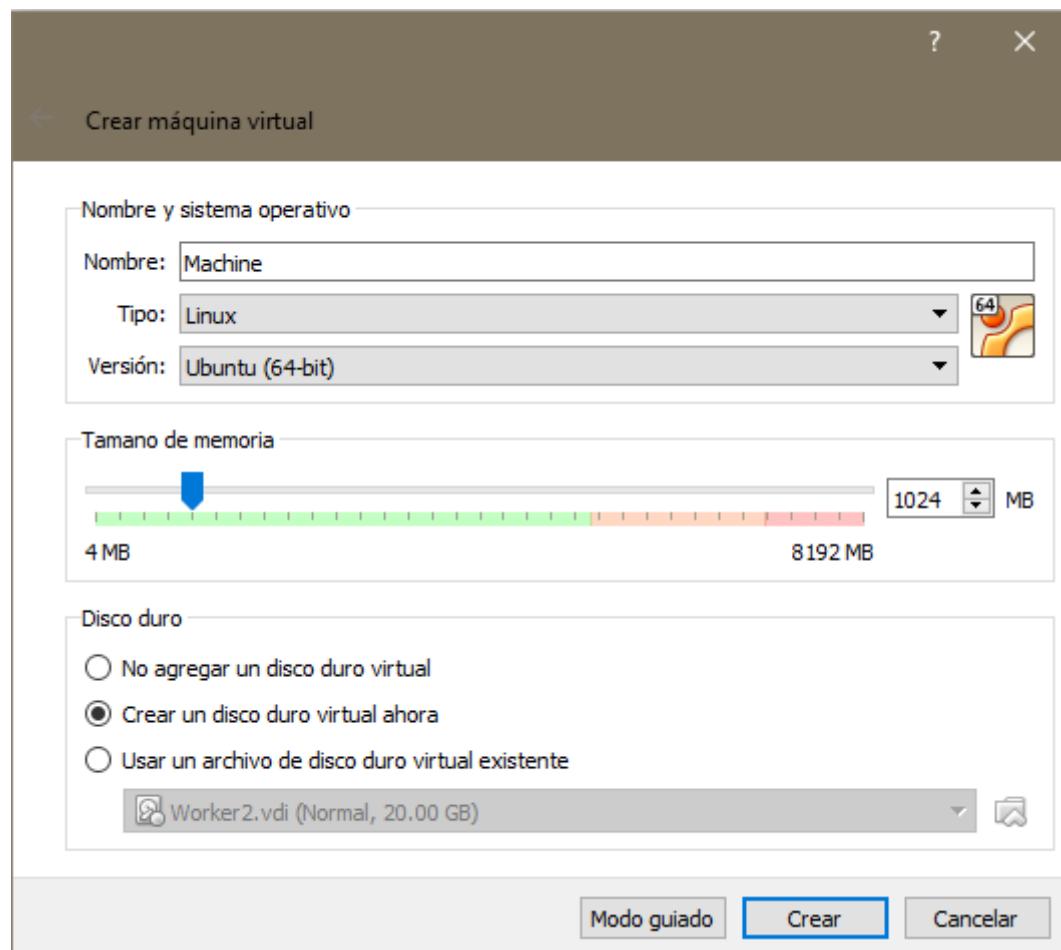


Ilustración 3 Ventana de creación de máquina virtual en *VirtualBox*

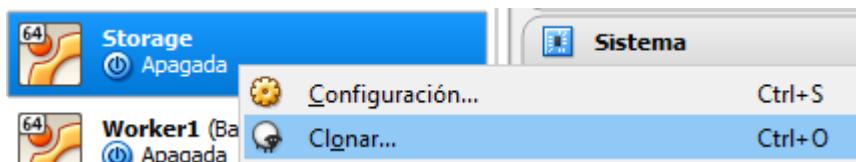


Ilustración 4 Clonación de una máquina en VirtualBox

2.2. Computación distribuida

Desde la invención de las computadoras, su crecimiento y mejora ha sido exponencial, pasando de ser máquinas muy caras y con poca capacidad de procesamiento, a ser accesibles a todos los usuarios y millones de veces más potentes. De máquinas que costaban diez millones de dólares y podían procesar algunos miles de operaciones por segundo [3], a máquinas que cuestan menos de 1000 y son capaces de realizar diez mil millones de operaciones por segundo. Este avance sumado al desarrollo de las redes como LAN o WAN permiten crear redes de computadores muy potentes que bien podrían plantarle cara a los supercomputadores actuales.

2.2.1. Definición y objetivos

Según Andrew S. Tanenbaum y Maarten Van Steen, “un sistema distribuido es una colección de computadores independientes que aparentan ser un único sistema coherente para el usuario.” [4] Esta definición, según señalan los autores, tiene varios aspectos importantes. El primero de [4]ellos es que un sistema distribuido está compuesto por diferentes nodos que son autónomos. El segundo es que los usuarios piensan que tratan con un único sistema. Y esto implica que los diferentes componentes autónomos de la red han de ser capaces de colaborar. Existen diferentes maneras de conseguir esa colaboración, pero todas pasan a través del mencionado anteriormente *broker* o *middleware*. En la Ilustración 5 se ilustra un sistema distribuido simple, utilizando una capa de *middleware* para la comunicación de las aplicaciones distribuidas en diferentes máquinas.

Un aspecto clave de los sistemas distribuidos es su escalabilidad. Es decir, la capacidad de estos de integrar nuevos nodos a la red y que su funcionamiento no se vea alterado negativamente.

2.2.2. Tipos de sistemas distribuidos

Diferentes tipos de computación distribuida se han planteado en las últimas décadas, como la organizada en *clusters* o grupos, o la organizada en *grid* o malla.

En los sistemas de computación en grupo, cuyo objetivo principal es el alto rendimiento de computación, todos los nodos del grupo comparten especificaciones en *software* y generalmente en *hardware*, como sistema operativo, aplicaciones y componentes físicos [4]. Integrando este grupo en una red de alta velocidad es posible obtener un tipo de supercomputación aprovechando la capacidad de las máquinas de

computar de manera paralela. Para ello debe existir una máquina o mecanismo que se comporte como maestro de las demás y sea capaz de distribuir las tareas de manera efectiva y con tolerancia a fallos. La manera más simple es mediante una cola de mensajes.

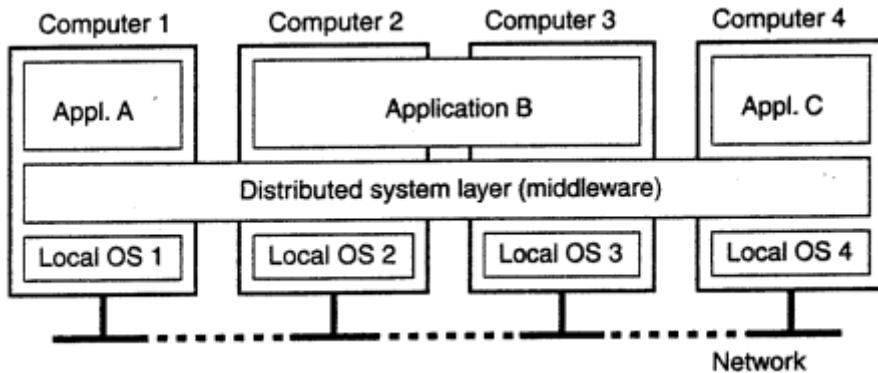


Ilustración 5 Sistema distribuido organizado como middleware.

Por otro lado, los sistemas de computación en malla no cuentan con la homogeneidad como su principal atributo. Éstos se componen por distintas agrupaciones

de máquinas y componentes que no guardan relación en sus especificaciones: pueden tener sistemas operativos y hardware diferentes, estar conectados por redes dispares y ser gestionados en diferentes dominios. Su objetivo es juntar los recursos de diferentes organizaciones para permitir la colaboración de un grupo de usuarios o instituciones [4].

2.2.3. Comunicación

La comunicación en los entornos distribuidos juega un papel esencial para poder obtener un sistema flexible y potente. Existen diversos métodos para conseguirlo dependiendo de los requisitos del sistema a implementar.

Remote Procedure Call

Basado en la ejecución de código en una máquina remota abstrayendo las comunicaciones entre ellas. Un cliente inicia el proceso conociendo únicamente el prototipo del procedimiento, es decir, su nombre y sus parámetros. La implementación en el lado del cliente denominada *stub* se encarga de encapsular los parámetros indicados por él y enviarlos al servidor. Éste último, llama al procedimiento de manera local y reenvía los resultados en forma de mensaje, permitiendo al *stub* recuperar los datos y devolverlos a la aplicación que realizó la llamada.

RPC ofrece un mecanismo de comunicación síncrono que bloquea al cliente hasta recibir la respuesta del servidor, por lo que puede resultar inadecuado para muchas aplicaciones en las que es necesario enviar simultáneos mensajes o en los cuales el procedimiento remoto puede tardar más de unos milisegundos (contando con la latencia de la red).

Modelo orientado a mensajes

En un sistema dónde la persistencia de los datos y asincronismo de las máquinas son un requisito, los modelos orientados a mensajes ofrecen una solución fiable y sólida. Aquí, de manera que se asegure la persistencia de un mensaje, éste queda almacenado por el sistema de comunicación hasta su entrega. Esto significa que ni el emisor ni el receptor deben estar ejecutándose simultáneamente para que la comunicación tenga lugar [4].

Modelo orientado a la transmisión

En este tipo de comunicación la problemática reside en la dependencia entre dos mensajes sucesivos a nivel temporal. Dicha dependencia basada en un retraso extremo a extremo máximo fijado previamente se utiliza comúnmente en transmisión continua de vídeo o audio.

Modelo multicast

El último modelo es utilizado cuando un mensaje debe ser enviado a muchos receptores. Es un modelo que implica severas configuraciones y cuya funcionalidad puede ser emulada por una cola a la que están suscritos varios consumidores.

2.3. Colas de mensajes

Las colas de mensajes son un mecanismo existente en todos los sistemas operativos, permitiendo el transcurso de información entre aplicaciones internas. Es una solución imprescindible para el correcto funcionamiento de la mayoría de los sistemas informáticos de hoy día: desde computadores personales, supercomputadoras, *smartphones*, servidores, así como multitud de aplicaciones e infraestructuras que serían inconcebibles sin este tipo de comunicación.

Los sistemas operativos UNIX poseen un mecanismo de colas mediante *arrays* que permite a través de diferentes funciones publicar y consumir mensajes existentes desde sus inicios, sin embargo, es propio y funciona con funciones que sólo UNIX posee.

2.3.1. Inicios de las colas de mensajes

Antes del siglo XXI, las tecnologías orientadas a mensajes sufrían de algo conocido como *tightly coupled*, es decir, estrechamente acoplado; implicando una fuerte relación entre ambos extremos de la comunicación. Tecnologías como redes *TCP*, *sockets* o *Remote Method Invocation* de Java, hacían imposible desacoplar al receptor del emisor y viceversa, teniendo que ser éstos capaces de comunicarse mediante un mismo protocolo o lenguaje de programación.

Remote Procedure Call (RPC) es un ejemplo de alto acoplamiento de comunicación. Permitiendo ejecutar subrutinas en un espacio de direcciones distinto, generalmente en otro computador o red, como si se hiciera de manera local, sin necesidad de codificar los detalles de la interacción remota. Evidentemente, entre dos máquinas es una buena

opción para su comunicación, pero en entorno distribuido y posiblemente asíncrono, genera una dependencia entre máquinas indeseada. En la Ilustración 6 se puede observar dicho acoplamiento entre una red de diferentes nodos o aplicaciones.

En el año 2000, aparecieron unos primeros intentos de solventar este fuerte acoplamiento a la hora de comunicar aplicaciones y sistemas distintos entre sí: *Java Message Service*, *IBM Integration Bus* y otros. Éstos se basan en una infraestructura de programación que permite envío y recepción de mensajes entre diferentes computadores conocido como *Message-oriented middleware (MOM)*. Así para la comunicación entre nodos o aplicaciones distribuidos ya no es necesario que cada uno esté *conectado* al otro y funcionen bajo el mismo sistema operativo y usen mismas interfaces de red para ser capaces de comunicarse, sino que existe un software intermediario que se encarga de estandarizar la comunicación abstrayéndola para el desarrollador y la máquina, al fin y al cabo.

“La forma más fácil de integrar componentes heterogéneos no es recrearlos como elementos heterogéneos en sí sino proveer una capa que les permita comunicarse a pesar de sus diferencias” [5]. Así si dos aplicaciones necesitan comunicarse entre sí, lo correcto no es desarrollarlas de tal modo que sean capaces de comunicarse entre sí sino utilizar un mecanismo intermedio que se encargue de ello. En la Ilustración 7 se presenta un ejemplo de *MOM* donde diferentes aplicaciones se comunican a través de un software intermedio en lugar de entre sí. En este contexto se destaca la importancia de la capacidad de las colas de mensajes para facilitar la información de manera asíncrona, en un entorno en el que no se sabe cuántas máquinas producen mensajes, cuántas consumen y si están disponibles en un instante de tiempo determinado.

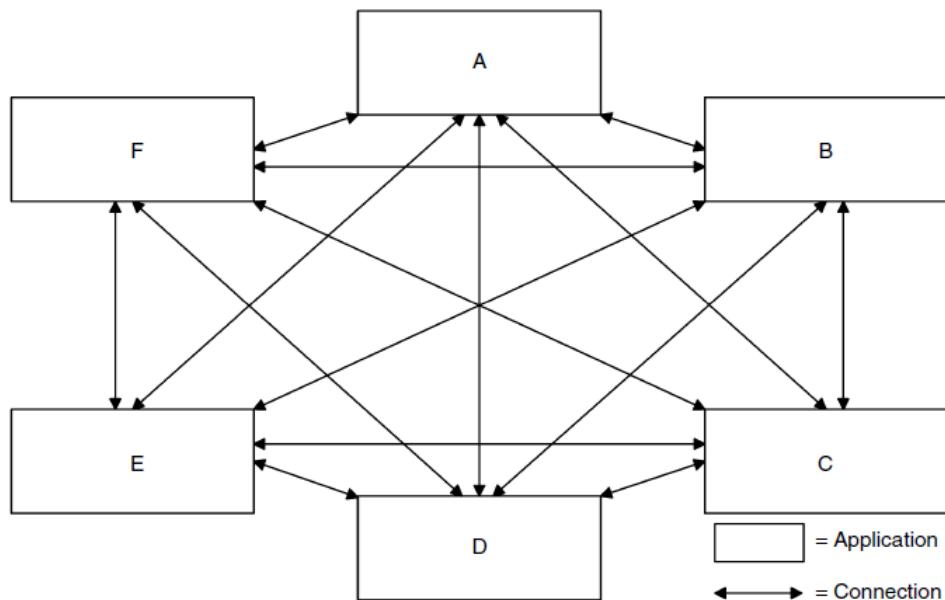


Ilustración 6 Un ejemplo de alto acoplamiento entre aplicaciones

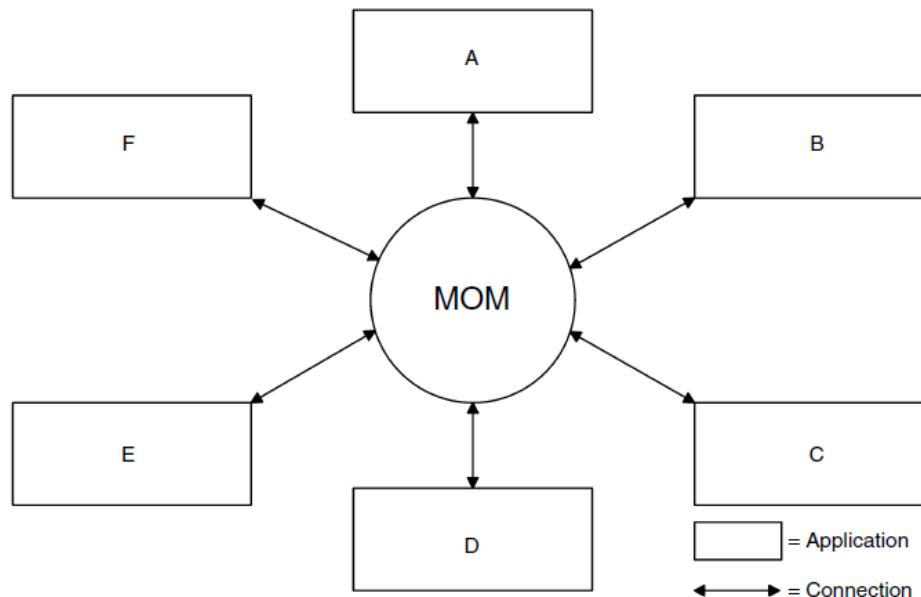


Ilustración 7 Un ejemplo de una distribución basada en message-oriented middleware

2.3.2. Protocolos para las colas de mensajes

Así, un *message-oriented middleware* requiere de un mecanismo de comunicación basado en mensajes, que abstraiga el proceso de interacción entre aplicaciones. Como por ejemplo la orientación de mensajes, el *queuing* o encolar mensajes, el enrutamiento, la escalabilidad la confiabilidad y la seguridad. Estas necesidades básicas dan lugar al desarrollo de diferentes protocolos que centralicen estas tareas. A continuación se repasan los 3 más importantes.

Message Queue Telemetry Transport o MQTT

Un protocolo desarrollado por IBM orientado principalmente a compañías. Es un protocolo que empezó siendo de pago, pero cuya versión 3.1.1 fue declarada como estándar OASIS en 2014, agrandando su comunidad de usuarios y desarrolladores, siendo actualmente utilizado por aplicaciones como Facebook. Cómo señalan en su página web se centra en ser muy ligero y en requerir muy poca información del sistema [6] y ocupar muy poco ancho de banda. Es ideal para comunicaciones vía satélite o uso móvil dado su bajo consumo de energía, paquetes de datos mínimos y distribución eficiente de la información entre los receptores. Es un protocolo que funciona sin colas, únicamente publica y suscribe, no asegura la persistencia de los datos ni su durabilidad y no posee un control de flujo.

Advanced Message Queuing Protocol o AMQP

Es un protocolo de código abierto desde sus inicios en 2004 reconocido en 2014 como estándar ISO/IEC 19464. Orientado también al mundo empresarial presenta sus características como: seguridad, confiabilidad, interoperabilidad, estándar y código abierto; además de sus capacidades clave como: “organizaciones (aplicaciones en diferentes organizaciones), tecnologías (aplicaciones en diferentes plataformas), tiempo (los sistemas no deben estar disponibles simultáneamente), espacio (operaciones confiables a distancia o a través de redes pobres)” [7]. Además, cuenta con colas, algo imprescindible para el desarrollo de este proyecto, persistencia y durabilidad de los datos, compatibilidad con servicios como *JMS* mencionado anteriormente, control de flujo y reconocimiento de entrega de mensajes, multiplexación para atravesar firewalls y otras características menos relevantes. Y una de sus características más importantes es que es *language-agnostic* es decir que un *broker* desarrollado en un lenguaje puede recibir mensajes de un productor desarrollado en otro totalmente diferente. Es un protocolo utilizado por diversas compañías internacionales como *Cisco*, *Microsoft*, *vmware* y *redhat*, además de instituciones gubernamentales y grandes bancos como *BankofAmerica* y *Barclays*.

Simple Text Oriented Message Protocol o STOMP

Es un protocolo basado en sólo texto similar a *HTTP*. La comunicación se realiza a través de *frames* de igual manera que lo hace *AMQP*, pero no cuenta con una cola. Es altamente interoperable pero no asegura la durabilidad ni persistencia de los datos. Utiliza *verbos* igual que *HTTP*, permitiendo enviar a una dirección IP (*SEND*) y que un consumidor se suscriba a ese destino (*SUBSCRIBE*) [8]. Es un protocolo simple y ligero que permite su uso hasta a través de una conexión *Telnet*, pero que no reúne las características necesarias para llevar a cabo las tareas requeridas.

2.3.3. Message brokers

Broker es una palabra mencionada anteriormente, pero cuya función y significado no se han clarificado en profundidad.

Un *message broker* en su raíz es un programa intermediario que traduce mensajes enviados por un productor mediante un protocolo, o codificados en un lenguaje, a otros protocolos o lenguajes utilizados por un consumidor. Se considera que es un patrón de diseño de arquitectura para la validación, transformación y enrutamiento de mensajes [9].

Media la comunicación entre aplicaciones reduciendo la conciencia que éstas deberían tener entre ellas para ser capaces de comunicarse, es decir llevar a cabo un desacoplamiento. En la Ilustración 8 se representa la estructura básica de comunicación de un *message broker*.

Las principales acciones que va a implementar un *broker* son:

- Enrutar mensajes hacia uno o varios destinos
- Agregar, descomponer y finalmente recomponer mensajes

- Interactuar con un repositorio externo para almacenar un mensaje
- Invocar servicios web para recuperar datos
- Responder a eventos o errores
- Asegurar el uso del patrón publicar-suscribir

Poseen diferentes maneras de transferir mensajes agrupando las tareas para que más tarde puedan ser filtradas según diferentes criterios como por ejemplo el canal dónde se van a publicar o el contenido que van a representar. Generalmente se usará únicamente el filtro de canal, con el nombre de una cola determinado.

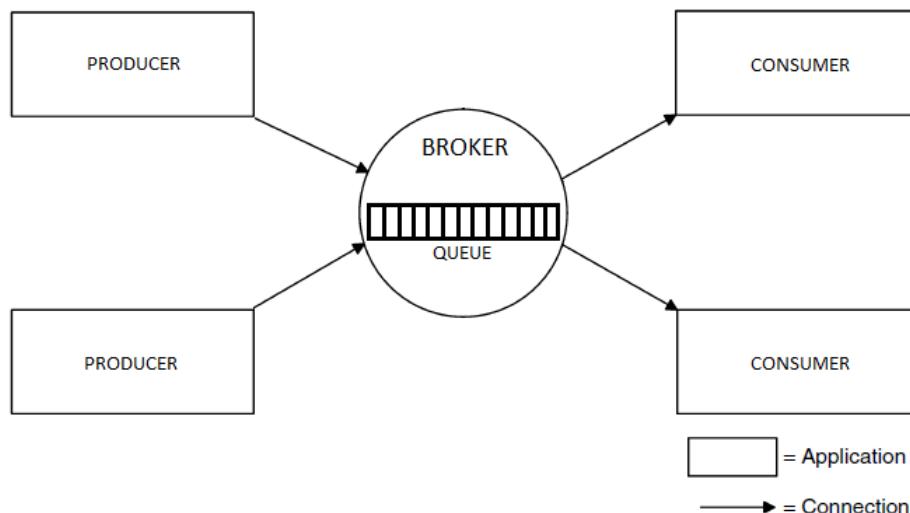


Ilustración 8 Ejemplo de un broker de mensajes

Actualmente existen decenas de aplicaciones *message broker* y resultaría imposible considerarlas todas. Por lo tanto, de todas ellas, y en particular las que son *open-source* se considerarán las siguientes.

RabbitMQ

Puede que el más extendido y usado de todos ellos, es una solución simple, compatible con todos los protocolos mencionados anteriormente y con una gran cantidad de documentación y libros sobre su uso. Está escrito en *Erlang* un lenguaje poco conocido pero muy adecuado para su cometido según los desarrolladores. No representa esto ningún problema para su uso ya que es posible implementar la solución en una decena de lenguajes diferentes como *Java*, *PHP*, *Python*, *Ruby*, *C#* o *Javascript* entre otros. Permite gran variedad de configuraciones mediante la edición de un único archivo además de contar con numerosos *plugins* como una interfaz web de administración [10]. Es capaz de trabajar con colas simples y complejas, en las que hay uno o más productores y uno o más consumidores. Además, posee un sistema de intercambios que, mediante la identificación de estos por una clave, permite publicar mensajes en el *broker* sin conocer (ni si quiera sin importar) en qué cola debe publicarlos. Así permite desacoplar la lógica de la aplicación productora del nombre o tipo de una cola y por tanto, la existencia o

número de colas. Esto, en un sistema en el que hay decenas de colas, productores y consumidores permite programar sin problemas de acoplamiento y escalabilidad.

Kafka

Kafka fue diseñado originalmente por el equipo de *LinkedIn* y mantenido actualmente por *Apache*. Es una solución eficiente capaz de publicar gran cantidad de mensajes en un único hilo sin agotar la memoria de una máquina. Utiliza entradas en archivos de texto plano generando un tipo de base de datos, lo que lo hace rápido pero no es capaz de llevar un registro de qué mensajes han sido consumidos y por quién, por lo que podría resultar útil en aplicaciones de tiempo real pero no para aplicaciones que requieran de confiabilidad y entrega de los datos. Apache provee servicios a parte para llevar a cabo estas tareas como *ZooKeeper* [11]. En la misma línea, no posee ninguna herramienta de administración o visualización de los datos del *broker*. Es una aplicación rápida y ligera pero que no atiende a muchas de las especificaciones esperadas de un *broker*. Como punto positivo también se debe añadir que posee clientes para casi una veintena de lenguajes diferentes.

ActiveMQ

Apache ActiveMQ es una opción potente y más completa como *RabbitMQ*. Desarrollada en *Java* cuenta numerosas características mencionadas anteriormente y funciona también sobre todos los protocolos mencionados [12]. Pudiendo así utilizarse en casi cualquier lenguaje de programación. Igual que el anterior depende del servicio *ZooKeeper* para la confiabilidad de los datos.

Dada su flexibilidad y compatibilidad con diferentes lenguajes de programación. Así como su gran documentación y su uso extendido a nivel profesional. También por su estructuración y abstracción del sistema de encolado que para este proyecto no es realmente una necesidad, pero pensando en la escalabilidad del mismo permitiría un entorno limpio y desacoplado. La elección indiscutible para trabajar como *message broker* es evidentemente *RabbitMQ*.

Ahora, una vez elegido *RabbitMQ* como *broker* de mensajes, se verá su funcionamiento en detalle.

La estructura básica de una cola de mensajes puede verse en la Ilustración 9. En ella como ya se ha citado en el capítulo anterior existe un Productor, un Consumidor y la Cola gestionada por el *broker*. Para su funcionamiento, el Productor debe crear un canal de comunicación con el *broker* mediante una conexión. La conexión se consigue mediante una abstracción de un *socket*. El siguiente código *Java* crea una conexión y un canal.

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
```

ConnectionFactory es una clase *Java* que permite la abstracción de la conexión remota a otra máquina con *IP*, en este caso, *localhost* pero que podría ser una *IP* cualquiera. La

última sentencia crea un canal que permitirá el flujo de información entre el Productor y el *broker*.

El siguiente paso es declarar la cola en la que se publicarán los mensajes y publicarlos. El siguiente fragmento de código realiza dicha acción.

```
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
String message = "Hello World!";
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
```

El método *queueDeclare* ofrecido por el protocolo *AMQP* permite crear dicha cola con parámetros como el nombre de la cola, parámetros booleanos declarados a *false* que establecen la durabilidad, exclusividad y auto eliminación del mensaje en la cola, y como último parámetro otros argumentos de construcción de la cola como el tipo de mensajes que almacenará.

Como paso final para publicar debe ejecutarse el método *basicPublish*. En él se especifica en el siguiente orden: en qué intercambio se debe publicar, cual es la *routing-key* con la que se publicará, es decir, el identificador de la cola, propiedades del mensaje a publicar (*null* en este caso) y finalmente el mensaje.



Ilustración 9 Cola de mensajes simple

Hasta ahora se ha completado el proceso de publicación. Queda entender cómo se consumen los mensajes de la misma cola en la que se han publicado.

De igual forma que a la hora de publicar, se debe crear una conexión, un canal y una cola. Una vez hecho esto, se puede especificar al consumidor que debe consumir mensajes. Para ello se sobrescribe el método *handleDelivery* proporcionado por la clase *DefaultConsumer*, en el caso de *Java*, que permite escuchar en un determinado canal y obtener los mensajes que en este se encuentren. Así ejecutando la siguiente línea de código sobre un canal, se llama al método *handleDelivery* del objeto *consumer* de tipo *DefaultConsumer* en el cual se implementa la lógica sobre el mensaje.

```
channel.basicConsume(QUEUE_NAME, true, consumer);
```

Como se puede observar, el método recibe el nombre de la cola donde escuchar, un booleano que representa si el mensaje debe enviar un mensaje de *acknowledge* de vuelta al *broker* y el *DefaultConsumer* definido previamente.

De esta manera se puede publicar y consumir un mensaje en una cola. Existen diferentes métodos de hacerlo, también es posible establecer varios productores o consumidores a

una misma cola. En la Ilustración 10 se representa una cola con dos consumidores que se implementaría de manera similar.

También es posible que un Productor publique en dos colas diferentes. Esto es posible gracias a los *exchanges* o intercambios mencionados anteriormente. En lugar de especificar el nombre de la cola en la que se publicará, se deja el campo como una cadena vacía y se especifica el nombre del *exchange*, así el método *basicPublish* publicará el mensaje en todas aquellas colas que tengan como *exchange* la cadena coincidente con la especificada. En la Ilustración 11 se representa lo citado.

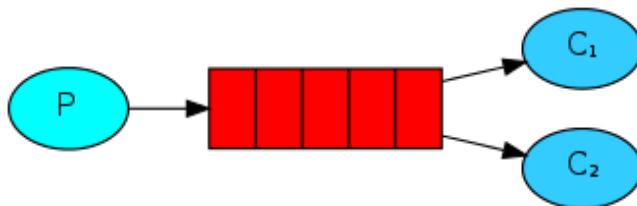


Ilustración 10 Cola de Workers

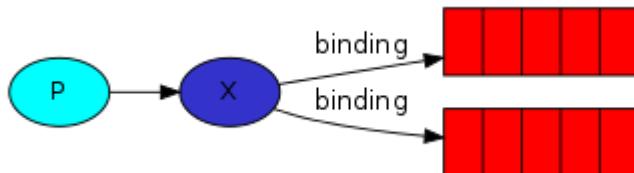


Ilustración 11 Publicación de mensaje en dos colas mediante un exchange

The screenshot shows two screenshots of the RabbitMQ Management interface. The top screenshot is the 'Overview' page, which includes a chart of queued messages over time, statistics for connections, channels, exchanges, queues, and consumers, and a message rates chart. The bottom screenshot is the 'Queues' page, which lists all queues with columns for name, status, and message counts.

Name	Exclusive	Parameters	Policy	Status	Ready	Unacked	Total	incoming	deliver / get	ack
task_queue	D			Idle	0	0	0	0	0	0

Ilustración 12 Interfaz de administración de RabbitMQ

Finalmente, como elemento extra mencionado anteriormente, *RabbitMQ* cuenta con una interfaz de administración instalado como *plug-in* que permite ver, publicar y consumir mensajes de todas las colas, *exchanges* y canales que gestione el *broker*. Esta interfaz es accesible vía web en el puerto 15672 de la interfaz de loopback (<http://localhost:15672>) y mediante usuario y contraseña por defecto (*user: guest, pw: guest*) o los especificados en los archivos de configuración por el propio usuario. En la Ilustración 12 se puede observar dicha interfaz y sus menús y configuraciones. Esta interfaz puede ser muy útil para entender cómo funciona realmente el *broker* y ser capaz de depurar errores de programación o de lógica.

2.4. Lenguajes de desarrollo

Las diferentes exigencias del proyecto y el deseo de homogeneizar el desarrollo mediante el uso de un único lenguaje para su desarrollo acotan bastante la búsqueda.

Los lenguajes orientados a objetos proporcionan las herramientas necesarias para cumplir los objetivos. Cuyas principales características son: definir nuevas clases o tipos de objetos, los cuales poseen diferentes operaciones asociadas que permiten el manejo de éstos para tratar los datos. A grandes rasgos se pueden dividir en dos tipos: basados en clases y basados en prototipos. Los basados en clases proporcionan una estructura sólida poco mutable, es decir, se definen rigurosamente las clases, sobre qué tipos operan y cuáles son sus atributos y métodos pudiendo ser instanciadas en el momento de su uso. Mediante las clases se define una jerarquía de clases y subclases cuyas propiedades pueden ser heredadas conforme se desciende en ella. Entre las opciones existentes está *C#, C++* o *Java*. Por otro lado, los basados en prototipos ofrecen una programación más laxa, donde cualquier objeto puede heredar de otro creando una jerarquía de manera diferente mediante el prototipado y dónde las propiedades de un objeto pueden variar dinámicamente en tiempo de ejecución. Entre los diferentes lenguajes orientados a prototipos están *JavaScript*, *Ruby* o *Python*.

Así, dados el tamaño y complejidad del desarrollo, teniendo que ser estructurado y distribuido a la vez que flexible e interconectado, la elección es obvia: se utilizará un lenguaje orientado a objetos basado en clases. La siguiente cuestión es ¿qué lenguaje cumple los requisitos para el desarrollo?

En el proyecto son necesarios distintos elementos: un servidor web que gestione las peticiones de los usuarios, unas máquinas que sean capaces de trabajar con software de procesamiento de imágenes y otras que sean capaces de producir y consumir mensajes mediante una cola de mensajes.

Para el servidor web la opción más obvia y simple sería utilizar *PHP* además de que también es compatible con el protocolo *AMQP* y con *RabbitMQ*. Sin embargo el procesamiento de imágenes es algo que escapa a sus capacidades por lo que no será la opción elegida.

C++ y *C#* son buenas opciones ya que es capaz de implementar un servidor web, es compatible con *RabbitMQ* y la mayoría de librerías de procesado de imágenes están desarrolladas en *C*. Por lo que serían una de las mejores opciones para el desarrollo.

Por otro lado, *Java* ofrece los *Java Servlets* ideales para el desarrollo del *Back-End* de un servidor web, también es compatible con *RabbitMQ* y pese a que las librerías de procesado de imágenes como *FFMPEG* y *OpenCV* no son nativas para *Java*, se pueden generar paquetes *.jar* contenido las librerías y *.so* con las dependencias y *dlls* necesarias para trabajar con ellas. Además, *Java* es multiplataforma y puede correr en cualquier máquina con cualquier sistema operativo de manera simple y homogénea.

Finalmente, de entre todas las opciones, debido a su flexibilidad y al profundo conocimiento obtenido durante la carrera además de la cobertura para todos los requisitos planteados, el lenguaje de desarrollo para el proyecto será *Java*.

2.5. Procesamiento de imágenes

2.5.1. Definición de imagen digital

Una imagen en el campo de la óptica es, según la R.A.E., “la reproducción de la figura de un objeto por la combinación de los rayos de luz que proceden de él”.

Desgraciadamente, esta definición no es entendible por un computador, por lo que es necesario buscar una que se adecúe más a su entendimiento. En el libro *Apuntes de procesamiento digital de imágenes*, José Ramón Mejía Vilet ofrece una definición mucho más acertada [13]. Aquí se define una imagen matemáticamente como una función con dos dimensiones:

$$f(x, y)$$

en la que x e y son coordenadas espaciales en un plano y la función f es el nivel o intensidad de gris para unas coordenadas concretas. De esta manera un computador puede generar o entender una imagen según sus componentes espaciales. A cada coordenada se le denomina *píxel*.

2.5.2. Definición de procesamiento digital de imagen

El procesamiento digital de imágenes es un conjunto de técnicas con objeto de mejorar la calidad o facilitar la búsqueda de información en imágenes digitales.

Resulta difícil acotar el ámbito de ejecución del procesamiento de imágenes dado que para un computador una imagen no se reduce al rango visible del espectro Electro Magnético. Por ejemplo, se considera procesado de imagen digital una tomografía axial computarizada o T.A.C. pese a que se realiza en términos de Rayos X donde la frecuencia es muy superior a las frecuencias visibles (hasta 50000 veces).

Ya que en este trabajo sólo se trabajará con imágenes visibles para el ojo humano no se considerarán aplicaciones de procesamiento más allá del rango visible. Es decir, aquel procesamiento cuya entrada y salida sean imágenes visibles. Dicho esto, se pueden considerar 3 tipos de procesos [13]:

- Procesos de bajo nivel:
Se caracterizan por el uso de operaciones como el preprocesamiento, para añadir filtros o variar el contraste de una imagen.
- Procesos de nivel medio:
Utilizan operaciones de segmentación y clasificación ofreciendo como salida un segmento o característica de la imagen de entrada. En el caso de estudio de la detección facial, éste proceso de nivel medio será el que se utilizará.

- Procesos de alto nivel:
Su objetivo es obtener algún tipo de significado semántico de las imágenes procesadas como sería el reconocimiento facial o de objetos.

2.5.3. Detección facial

La detección facial tiene como objetivo localizar la posición de la imagen en la que se encuentra un rostro humano. A lo largo de los años se han desarrollado diferentes técnicas y algoritmos que cuajaron finalmente en la década de los 90 gracias al aumento de la potencia computacional y de las redes.

Actualmente existen técnicas varias para la detección facial, algunas más precisas que otras. Algunos se basan en referencias como la textura o la tonalidad de la piel, otros en características físicas como tamaños de la cabeza o número de ojos, otros en las formas geométricas de los objetos y otros en experiencia previa de otras imágenes. Los que mejor resultado dan, es decir mayor número de verdaderos positivos y menor número de falsos positivos son los dos últimos: los basados en plantillas y los basados en reconocimiento [14].

Merecen una mención especial los basados en reconocimiento. Utilizan un fichero cascada, denominado así por su gran cantidad de datos almacenados secuencialmente. Este fichero contiene valores poco entendibles para el humano, pero muy característicos para el computador que le permiten hallar características de un rostro en una imagen. Estos ficheros pueden ser entrenados por el usuario, o pueden utilizarse otros predefinidos. En particular hay un algoritmo que utiliza estas cascadas para detectar rostros y es el más utilizado actualmente por la mayor parte de las tecnologías.

Algoritmo Viola & Jones

Es un algoritmo con bajo coste computacional y que ofrece una tasa de verdaderos positivos cercana al 100% y de falsos positivos de alrededor del 3% [14]. Es tan rápido que puede usarse incluso en tiempo real y es el utilizado por las cámaras fotográficas digitales.

Se basa en dividir la imagen en subregiones y detectar en ellas un rostro. Si por algún casual en alguna de ellas no hay ningún rasgo facial potencial comparándolo con el clasificador en cascada, dicha región o ventana no se analizará en profundidad. Esto hace que no sea necesario analizar todos los *píxeles* de la imagen como con otros algoritmos. Existen diferentes clasificadores: rasgos de borde, rasgos de línea o rasgos de 4-rectángulos. En las Ilustraciones 13 y 14 se pueden ver los de primer y segundo tipo respectivamente.

Por otro lado, es un algoritmo que convierte imágenes a escala de grises para su detección, por lo que no se basa en colores si no en tonalidades, haciéndolo más eficiente.

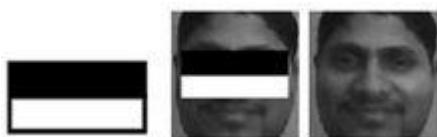


Ilustración 13 Detección de escalas del algoritmo Viola-Jones para la región ojos-mejillas



Ilustración 14 Detección de escalas del algoritmo Viola-Jones para la región ojos-nariz

Los rostros humanos comparten diferentes características que este algoritmo usa para su detección (en escala de grises):

- La región de los ojos es más oscura que la parte superior de las mejillas (Figura 2.5.6 (a))
- La región del puente nasal es más brillante que los ojos (Figura 2.5.6 (b))
- Composición del rostro (ojos, nariz, boca y orejas)

Inclinación de la intensidad de los *píxeles* El algoritmo sigue los siguientes pasos [15]:

- Selección de rasgos Haar
- Creación de una integral de la imagen
- Entrenamiento del algoritmo AdaBoost
- Clasificadores en cascada

Para no entrar mucho en detalles, *Haar* fue un matemático húngaro de principios del siglo XX que definió un modelo matemático compuesto por un conjunto de funciones para un ejemplo de un sistema ortonormal contable en la recta real.

AdaBoost es un algoritmo de aprendizaje máquina desarrollado en 2003 cuyo objetivo es seleccionar entre todas las muestras aquellas que mejoren significativamente la potencia de predicción del modelo.

Dentro de este algoritmo y de la mayoría, el proceso de detección facial es el siguiente: calcular la integral de la imagen (a escala de grises), posicionar una ventana o región, posicionar una sub-ventana y extraer sus características, comparar estas características con la cascada, comprobar si es un rostro y en caso afirmativo añadir la sub-ventana a la lista de rostros detectados. En la Ilustración 15 se describe el algoritmo mediante un diagrama de flujo. Así, mientras el algoritmo sea capaz de posicionar una ventana, es decir aún quedan regiones de la imagen por analizar, se repite en bucle y su salida es una lista con la lista de rostros detectados, pudiendo estar vacía.

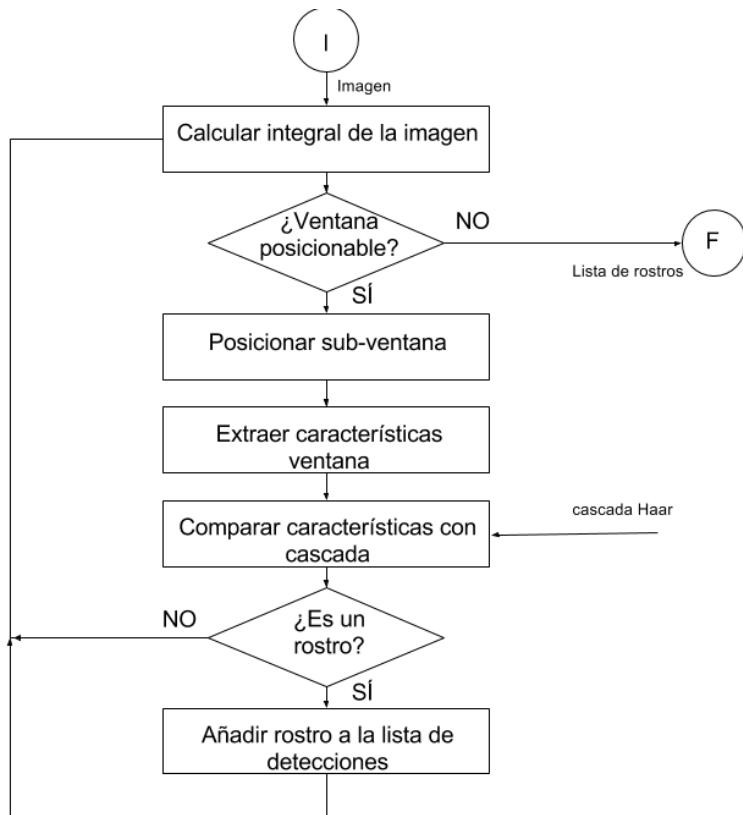


Ilustración 15 Diagrama de flujo del algoritmo de detección facial

2.5.4. OpenCV

OpenCV u *Open Source Computer Vision Library*, es una librería de visión artificial desarrollada inicialmente por *Intel* y que cuenta con una licencia BSD de uso comercial e investigación. Su desarrollo se basa en eficiencia computacional y es lo que la distingue principalmente de otras herramientas similares, ofreciendo soporte para aplicaciones en tiempo real. El código de la biblioteca está escrito en *C* y *C++* optimizado [16] pero posee interfaces para desarrollo en *C*, *C++*, *Python* y *Java* y es soportado por los principales sistemas operativos como *Windows*, *Mac OSX* y *Linux*. Además es capaz de utilizar el sistema integrado de primitivas de *Intel*, que son rutinas de bajo nivel de los procesadores *Intel* permitiendo alcanzar altos niveles de rendimiento.

Para el procesado de imágenes digital es una solución adoptado por las grandes compañías de *Internet* como *Google*, *Microsoft*, *Intel* o *IBM* y de compañías dispares como *Honda* o *Toyota*. Es usado también por organizaciones gubernamentales como China para la supervisión del equipo de minas, o en Europa para detección de ahogamientos en piscinas y playas.

Para resolver la problemática propuesta, *OpenCV* representa la mejor opción dada la gran cantidad de funciones, clases y algoritmos que provee.



UNIVERSITAT
POLÈTICA
DE VALÈNCIA

[] Escola Tècnica
Superior d'Enginyeria

Capítulo 3. Análisis del problema

A continuación, se expondrán las fases necesarias para analizar el problema en profundidad. Dado que el proyecto se compone de diferentes aplicaciones distribuidas en varias máquinas y que éstas se comunican entre sí a través de diferentes protocolos, el análisis puede no parecer tan centralizado y estructurado como lo sería para una aplicación que funciona en una única máquina.

3.1. Análisis de requisitos

Los requisitos son la raíz del análisis de un problema y de dónde parten todos los demás subpuntos, por lo que deben estar bien definidos y ser claros y concisos. Representando estos una condición o necesidad de un usuario o del sistema para realizar una tarea o cumplir un objetivo o una condición o capacidad que debe existir en un sistema para satisfacer un contrato, especificación u otro documento formal.

En los requisitos, cuando se denomina *Web*, *Storage*, *RabbitMQ* o *Workers* se hace referencia a las máquinas virtuales homónimas.

3.1.1. Requisitos funcionales

- RF.1 El sistema debe permitir al usuario subir un archivo de vídeo a un portal web y que este se almacene en *Web*.
- RF.2 El sistema debe almacenar un vídeo en un directorio cuyo nombre sea un *UUID*.
- RF.3 El sistema debe permitir que un vídeo almacenado en *Web* sea copiado a *Storage* a un directorio con mismo nombre.
- RF.4 El sistema debe permitir a *Storage* dividir un vídeo en frames y almacenarlos en el directorio en el que se encuentra el vídeo.
- RF.5 El sistema debe permitir serializar los objetos mensaje con información relativa a una imagen.
- RF.6 El sistema debe permitir a *Storage* encolar, es decir publicar en una cierta cola (“task_queue”), un mensaje en la máquina *RabbitMQ*.
- RF.7 El sistema debe permitir a *Storage* establecer el instante de tiempo en el que se ha publicado el mensaje.
- RF.8 El sistema debe permitir a los *Workers* suscribirse a una cierta cola y consumir mensajes de ésta.
- RF.9 El sistema debe permitir a los *Workers* establecer el tiempo en que un mensaje ha sido consumido.

- RF.10 El sistema debe permitir a los *Workers* de-serializar un mensaje y recuperar la imagen de la máquina *Storage*.
- RF.11 El sistema debe permitir a los *Workers* procesar una imagen para detectar si aparece una cara.
- RF.12 El sistema debe permitir a los *Workers* contabilizar el tiempo que ha tardado en procesarse la imagen y establecer ese tiempo en el mensaje.
- RF.13 El sistema debe permitir a los *Workers* serializar en un objeto mensaje la información de las imágenes en las que se ha detectado una cara.
- RF.14 El sistema debe permitir a los *Workers* copiar una imagen en la que ha sido detectada una cara a *Web* en la carpeta en la que se subió el vídeo inicialmente de modo que en esa carpeta sólo estén las imágenes con caras.
- RF.15 El sistema debe permitir a los *Workers* publicar un mensaje en una cola (“log”).
- RF.16 El sistema debe permitir a los *Workers* eliminar una imagen una vez ha sido procesada.
- RF.17 El sistema debe permitir a *Storage* suscribirse y consumir mensajes de la cola “log”.
- RF.18 El sistema debe permitir a *Storage* contabilizar los mensajes que se han recibido y una vez todos recibidos eliminar el contenido del directorio que contiene dichos mensajes.
- RF.19 El sistema debe permitir a *Storage* calcular el tiempo total de respuesta, desde que se publican todos los mensajes pertenecientes a un vídeo hasta que se reciben y escribir el tiempo en un archivo.
- RF.20 El sistema debe permitir a *Web* suscribirse y consumir mensajes de la cola “log”.
- RF.21 El sistema debe permitir a *Web* de-serializar los mensajes consumidos.
- RF.22 El sistema debe permitir a *Web* escribir en un fichero las estadísticas de procesado de un vídeo. Siendo éstas tiempo de espera, tiempo de respuesta y *worker* para cada *frame*.
- RF.23 El sistema debe permitir al usuario recuperar las imágenes en las que hay una cara accediendo a una *URL* que contiene el *UUID* generado inicialmente.

3.1.2. Requisitos no funcionales

Los requisitos no funcionales hacen referencia a aquellos elementos que no representan una funcionalidad directa del sistema pero que lo definen y restringen según las herramientas a utilizar y otros factores.

- RNF.1 La codificación del portal web se realizará en HTML5 y CSS3 con uso de la librería Bootstrap de Twitter.
- RNF.2 La subida y recuperación de archivos se hará mediante Java Servlets.
- RNF.3 Las copias de archivos entre máquinas virtuales se realizarán utilizando el protocolo SCP.
- RNF.4 Las comunicaciones deben estar cifradas mediante clave pública y privada RSA de 256 bits.
- RNF.5 La división de un vídeo en *frames* debe realizarse con la librería *avconv*.
- RNF.6 La serialización y de-serialización debe tener el formato *JSON* y se llevará a cabo usando la librería *GSON*.
- RNF.7 El broker de mensajes utilizará la aplicación *RabbitMQ* y el protocolo *AMQP*.
- RNF.8 Las aplicaciones para la publicación y suscripción a las colas se implementará en *Java*.
- RNF.9 La detección facial se llevará a cabo usando la librería *OpenCV* y codificado en *Java*.

3.2. Análisis de la solución

3.2.1. Diagrama de casos de uso

El usuario interactúa con el sistema para poder obtener un resultado y lo hace mediante los siguientes casos de uso:

- Subir Video: Permite al usuario subir un vídeo al sistema a través del portal web.
- Consultar Resultado: Permite al usuario acceder al resultado generado por el sistema

El siguiente diagrama expresa los casos de uso.

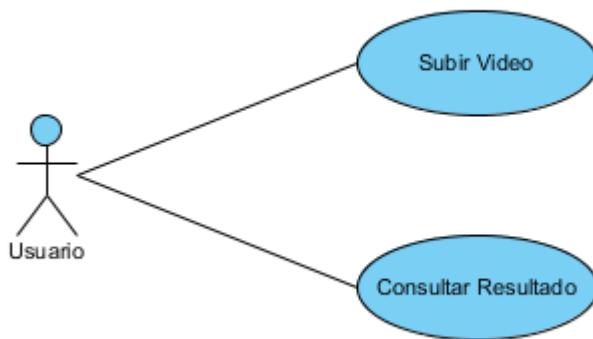


Ilustración 16 Diagrama de casos de uso de la aplicación

Pese a que el caso de uso Consultar Resultado podría considerarse como una extensión de Subir Vídeo, el proceso de consulta no se realiza a la finalización de la subida del vídeo, pudiendo el usuario guardar el enlace proporcionado y acceder días más tarde o desde otro sistema; tratándose del procesado de imagen, el tiempo desde que se sube hasta que se procesa podría llegar a tardar horas dependiendo del tamaño del vídeo. Por lo tanto, se ha dividido en dos casos de uso diferentes.

A continuación se presenta la especificación en formato extendido de los casos de uso.

Caso de Uso:

Subir Video

Actores:

Usuario

Propósito:

Permitir a un usuario subir un video al sistema.

Resumen:

Un usuario selecciona un archivo de su máquina y lo envía al sistema.

Tipo:

Primario

Referencias:

RF.1 hasta RF.22

Pre-condiciones:**Post-condiciones:**

Se han debido almacenar en el servidor *Web* todas las imágenes en las que los *Workers* hayan detectado una cara y sus estadísticas.

Flujo de eventos principal:

1. El caso de uso comienza cuando el usuario desea subir un archivo de vídeo.
2. El sistema carga la página principal de subida.
3. El usuario selecciona el archivo de vídeo de su máquina y elige la opción Upload Video
4. El sistema procesa la solicitud, genera una *URL* para el usuario y redirige a la página de espera.

Caso de Uso:

Consultar Resultado

Actores:

Usuario

Propósito:

Permitir a un usuario obtener las imágenes procesadas.

Resumen:

Un usuario hace *click* en el enlace proporcionado y consulta las imágenes procesadas.

Tipo:

Primario

Referencias:

RF.23

Pre-condiciones:

El caso de uso Subir Video ha sido completado sin errores.

Post-condiciones:

Flujo de eventos principal:

1. El caso de uso comienza cuando el usuario desea consultar las imágenes procesadas haciendo *click* en el enlace.
2. El sistema muestra todas las imágenes que han sido procesadas.
3. El cliente cierra la aplicación.

Flujo de eventos alternativo (No hay imágenes)

1. El flujo alternativo comienza en el punto 1.
2. El sistema redirige a una página que lo indica.
3. El cliente cierra la aplicación

3.2.2. Diagramas de Secuencia Generales del Sistema

Dado que la interacción del usuario con el sistema se realiza a través de *Servlets Java* los diagramas de secuencia son casi idénticos siendo la única diferencia el ‘verbo’ HTTP que utilizan a la hora de realizar la petición al servidor. A continuación se presentan ambos diagramas.

DSGS Subir Vídeo



Ilustración 17 Diagrama de Secuencia General del Sistema del caso de uso Subir Vídeo

DSGS Consultar Resultado



Ilustración 18 Diagrama de Secuencia General del Sistema del caso de uso Consultar Resultado

3.3. Análisis de seguridad

En toda aplicación informática la seguridad es uno de los puntos más importantes a tener en cuenta. Sin ella cualquier desarrollo podría suponer un riesgo para la información y para los componentes *hardware*.

No debería obviarse en ningún caso a nivel profesional, pese a que hoy en día la abstracción de los lenguajes de programación, del software en general y las comunicaciones ofrezcan mecanismos de seguridad transparentes para el usuario. Estos siguen sin ser suficientes y es necesario tomar medidas extra para asegurar los sistemas.

Sin embargo, en este proyecto la seguridad no era el tema por desarrollar y aunque se han aplicado algunas medidas básicas, hay muchos puntos por los que la seguridad del sistema podría verse comprometida. A continuación se repasan las medidas aplicadas.

En el desarrollo del software se ha intentado respetar los principales patrones que aportan consistencia y solidez al sistema.

Otra medida de seguridad tomada viene de la comunicación entre máquinas, que como se ha comentado anteriormente deben compartir imágenes entre sí. Para ello utilizan el protocolo *SCP* o *secure-copy* que permite copiar archivos o directorios de manera remota basándose en el protocolo *SSH*. Para ello se generaron un par de claves pública-privada de 256 *bits* entre cada máquina de manera que la comunicación queda cifrada.

Estas medidas son pobres en cuanto a seguridad se refiere, y podrían aplicarse muchas otras medidas que quedan fuera del marco del proyecto. Entre ellas y las más importantes serían:

- Utilizar un cortafuegos que gestione las conexiones entrantes y salientes de la red interna.
- Utilizar una red *DMZ* o desmilitarizada para crear una capa de seguridad entre equipos accesibles públicamente y la red interna.
- Control de acceso a la información mediante mecanismos tipo listas de acceso.
- Mecanismos para asegurar que la recepción de los datos no haya sido alterada. Como *hashes SHA* o *MD5*.
- Criptografiar la información almacenada.
- Utilizar sesiones y autenticaciones en el servidor web.

Si todas estas medidas fuesen implementadas se obtendría un sistema robusto y seguro que poco podría ser atacado por intrusos, sin embargo, como ya se ha mencionado, la seguridad queda fuera del marco del proyecto y no es un punto principal del desarrollo.

3.5. Análisis temporal

En la realización de proyecto es de vital importancia definir detalladamente los tiempos necesarios para el desarrollo de cada tarea de manera que se pueda controlar el avance de éste de manera precisa y tomar medidas de contingencia en caso de retrasos en las entregas. En la siguiente tabla se presentan las actividades con los tiempos estimados para cada una de ellas.

TAREAS		ESTIMACIÓN
1	GESTIÓN DEL PROYECTO	9,5
1.1	Inicio	1
1.2	Planificación	3,5
1.3	Ejecución y Control	4
1.4	Cierre	1
1.5	Proyecto Aprobado	
2	FASE DE ANÁLISIS	6
2.1	Obtención de requisitos	2
2.2	Plan de Proyecto	4
2.3	Entrega de análisis y plan de proyecto	
3	ESTUDIO	14
3.1	Estudio de herramientas de virtualización	1
3.2	Estudio del paradigma de computación distribuida	2
3.3	Estudio de RabbitMQ	4
3.4	Estudio de OpenCV	2
3.5	Estudio de la topología de red	1
3.6	Instalación y configuración de herramientas	2
3.7	Evaluación y prueba de herramientas	2
3.8	Entrega del estudio	
4	DISEÑO	12
4.1	Diseño de la aplicación en máquina Web	5

4.1.1	Diseño de Servlet de subida de vídeo	1
4.1.2	Diseño de Servlet de carga de imágenes	1
4.1.3	Diseño de componente de transferencia de datos	1
4.1.4	Diseño de componente de recolección de estadísticas	2
4.2	Diseño de la aplicación en máquina Storage	2,5
4.2.1	Diseño de componente de división de vídeo	0,5
4.2.2	Diseño de componente de publicación de mensajes	1,5
4.2.3	Diseño de componente de recolección de estadísticas	0,5
4.3	Diseño de aplicación en máquinas Worker	4,5
4.3.1	Diseño de componente de consumición de mensajes	0,5
4.3.2	Diseño de componente de detección facial	1,5
4.3.3	Diseño de componente de publicación de mensajes	0,5
4.3.4	Diseño de componente de transferencia de datos	0,5
4.4	Diseño de script en OCTAVE para presentación de estadísticas	1,5
4.5	Entrega Diseño	
5	IMPLEMENTACIÓN	28,5
5.1	Implementación de la aplicación en máquina Web	6,5
5.1.1	Implementación de Servlet de subida de vídeo	1,5
5.1.2	Implementación de Servlet de carga de imágenes	2
5.1.3	Implementación de componente de transferencia de datos	1

5.1.4	Implementación de componente de recolección de estadísticas	2
5.2	Implementación de la aplicación en máquina Storage	5,5
5.2.1	Implementación de componente de división de vídeo	1
5.2.2	Implementación de componente de publicación de mensajes	2
5.2.3	Implementación de componente de recolección de estadísticas	2,5
5.3	Implementación de aplicación en máquinas Worker	16,5
5.3.1	Implementación de componente de consumición de mensajes	5
5.3.2	Implementación de componente de detección facial	7
5.3.3	Implementación de componente de publicación de mensajes	1
5.3.4	Implementación de componente de transferencia de datos	0,5
5.4	Implementación de script en OCTAVE para presentación de estadísticas	3
5.5	Entrega Implementación	
6	PRUEBAS	8,5
6.1	Pruebas de funcionamiento de la aplicación	3,5
6.2	Pruebas de rendimiento	5
6.3	Entrega Pruebas	
7	ELABORACIÓN DE LA MEMORIA	29
7.1	Entrega Documento	

Tabla 1 Descomposición de tareas del proyecto

La planificación presentada en al Tabla 1 recoge la división del proyecto en tareas, las cuales se plasman temporalmente en el diagrama de Gantt de las Ilustraciones 19 a 21.

Según estos datos, el proyecto tendría una duración 107,5 días. Asumiendo que se trabajará en régimen de media jornada, de lunes a viernes, y descansando fines de semana y festivos, y, si el proyecto empieza el día 16 de Octubre de 2017 se estima que finalizaría el día 10 de Marzo de 2018.

Se establece un orden y unas dependencias para las tareas en la Tabla 2.

TAREAS		PREDECESORA
1	GESTIÓN DEL PROYECTO	-
1.1	Inicio	-
1.2	Planificación	1.1
1.3	Ejecución y Control	1.2
1.4	Cierre	1.3
1.5	Proyecto Aprobado	1.4
2	FASE DE ANÁLISIS	-
2.1	Obtención de requisitos	1.5
2.2	Plan de Proyecto	2.1
2.3	Entrega de análisis y plan de proyecto	2.2
3	ESTUDIO	-
3.1	Estudio de herramientas de virtualización	2.3
3.2	Estudio del paradigma de computación distribuida	3.1
3.3	Estudio de RabbitMQ	3.2
3.4	Estudio de OpenCV	3.3
3.5	Estudio de la topología de red	3.4
3.6	Instalación y configuración de herramientas	3.5
3.7	Evaluación y prueba de herramientas	3.6
3.8	Entrega del estudio	3.7
4	DISEÑO	-
4.1	Diseño de la aplicación en máquina Web	3.8
4.1.1	Diseño de Servlet de subida de vídeo	3.8

4.1.2	Diseño de Servlet de carga de imágenes	4.1.1
4.1.3	Diseño de componente de transferencia de datos	4.1.2
4.1.4	Diseño de componente de recolección de estadísticas	4.1.3
4.2	Diseño de la componente en máquina Storage	4.1.4
4.2.1	Diseño de componente de división de vídeo	4.1.4
4.2.2	Diseño de componente de publicación de mensajes	4.2.1
4.2.3	Diseño de aplicación de recolección de estadísticas	4.2.2
4.3	Diseño de aplicación en máquinas Worker	4.2.3
4.3.1	Diseño de aplicación de consumición de mensajes	4.2.3
4.3.2	Diseño de aplicación de detección facial	4.3.1
4.3.3	Diseño de aplicación de publicación de mensajes	4.3.2
4.3.4	Diseño de aplicación de transferencia de datos	4.3.3
4.4	Diseño de programa en OCTAVE para presentación de estadísticas	4.3.4
4.5	Entrega Diseño	4.4
5	IMPLEMENTACIÓN	
5.1	Implementación de la aplicación en máquina Web	4.5
5.1.1	Implementación de Servlet de subida de vídeo	4.5
5.1.2	Implementación de Servlet de carga de imágenes	5.1.1
5.1.3	Implementación de aplicación de transferencia de datos	5.1.2

5.1.4	Implementación de aplicación de recolección de estadísticas	5.1.3
5.2	Implementación de la aplicación en máquina Storage	5.1.4
5.2.1	Implementación de aplicación de división de vídeo	5.1.4
5.2.2	Implementación de aplicación de publicación de mensajes	5.2.1
5.2.3	Implementación de aplicación de recolección de estadísticas	5.2.2
5.3	Implementación de aplicación en máquinas Worker	5.2.3
5.3.1	Implementación de aplicación de consumición de mensajes	5.2.3
5.3.2	Implementación de aplicación de detección facial	5.3.1
5.3.3	Implementación de aplicación de publicación de mensajes	5.3.2
5.3.4	Implementación de aplicación de transferencia de datos	5.3.3
5.4	Implementación de programa en OCTAVE para presentación de estadísticas	5.3.4
5.5	Entrega Implementación	5.4
6	PRUEBAS	-
6.1	Pruebas de funcionamiento de la aplicación	5.5
6.2	Pruebas de rendimiento	6.1
6.3	Entrega Pruebas	6.2
7	ELABORACIÓN DE LA MEMORIA	-
7.1	Entrega Documento	6.3

Tabla 2 Secuencia de las actividades el proyecto

3.5.1 Diagrama de Gantt

El diagrama de Gantt es una de las técnicas más empleadas en planificación de proyectos. Expone las tareas de un proyecto, marcando su inicio y fin en un gráfico temporal. En las siguientes ilustraciones se expone la planificación del proyecto desarrollado

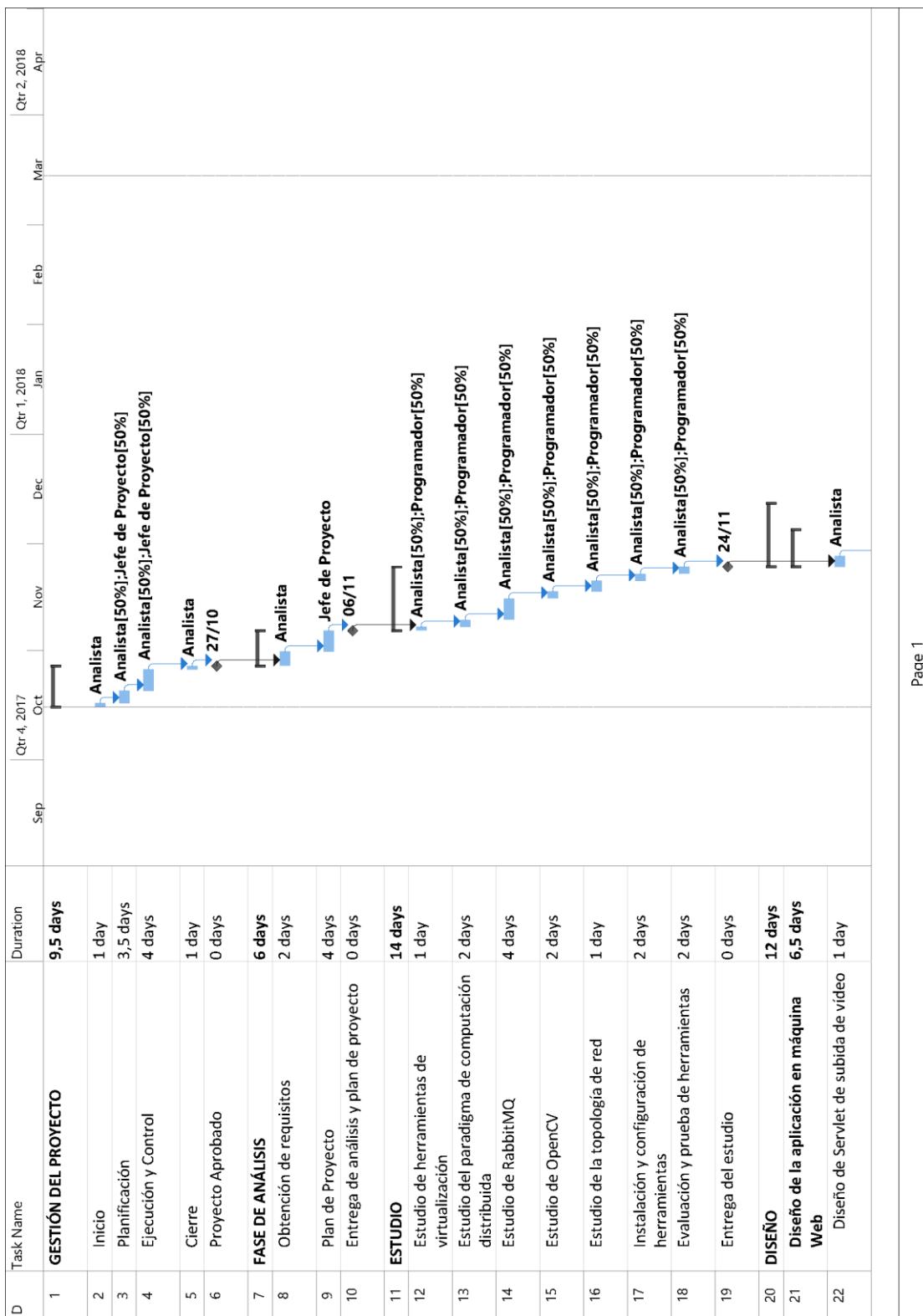
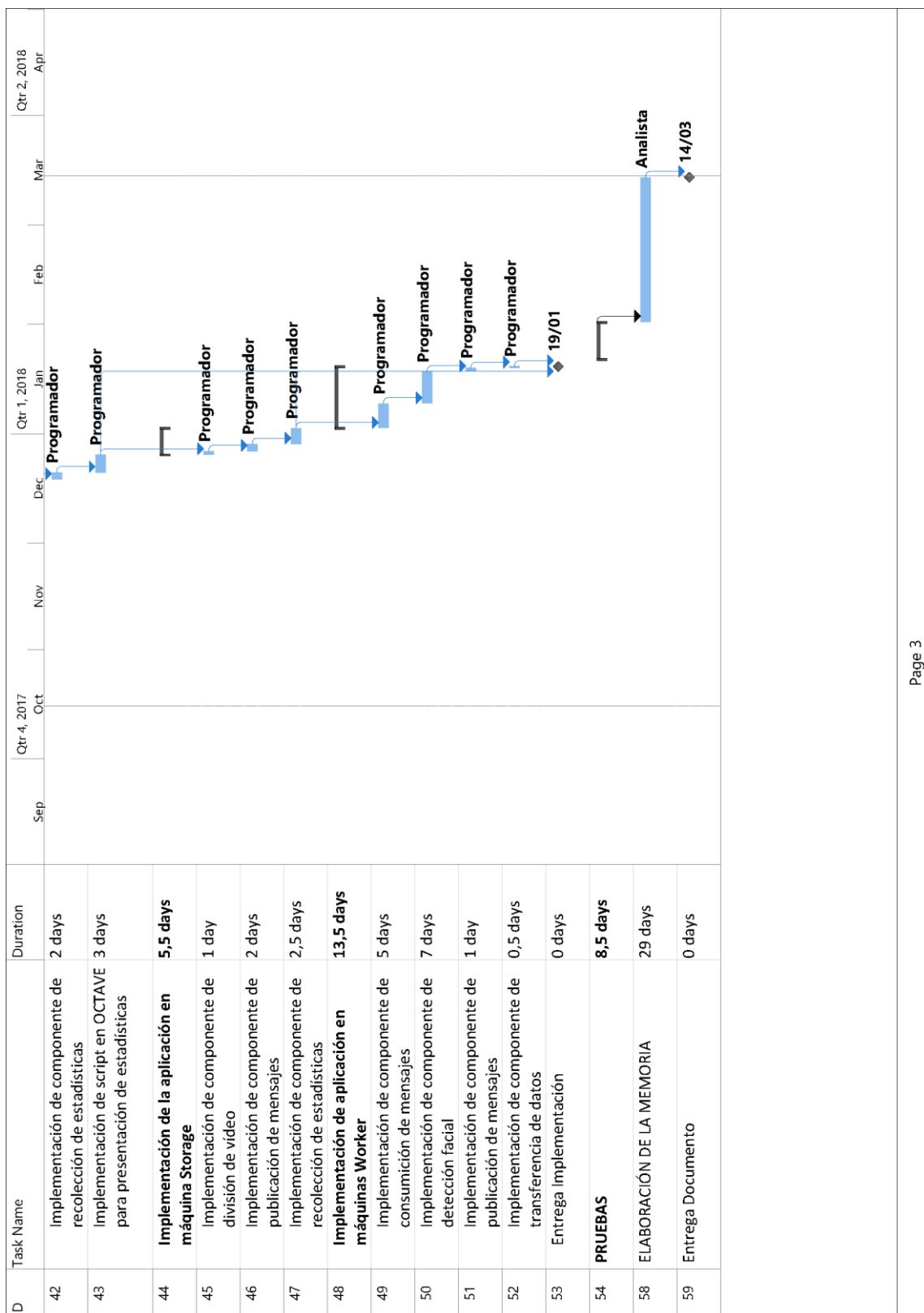


Ilustración 19 Diagrama de GANTT estimado p.1

ID	Task Name	Duration	Sep			Oct			Nov			Dec			Jan			Feb			Mar			Apr		
			Qtr 4, 2017			Qtr 1, 2018			Qtr 2, 2018			Qtr 1, 2018			Qtr 2, 2018			Qtr 1, 2018			Qtr 2, 2018			Qtr 1, 2018		
23	Diseño de Servlet de carga de imágenes	1 day																								
24	Diseño de componente de transferencia de datos	1 day																								
25	Diseño de componente de recolección de estadísticas	2 days																								
26	Diseño de script en OCTAVE para presentación de estadísticas	1,5 days																								
27	Diseño de la aplicación en máquina Storage	2,5 days																								
28	Diseño de componente de división de video	0,5 days																								
29	Diseño de componente de publicación de mensajes	1,5 days																								
30	Diseño de componente de recolección de estadísticas	0,5 days																								
31	Diseño de aplicación en máquinas Worker	3 days																								
32	Diseño de componente de consumo de mensajes	0,5 days																								
33	Diseño de componente de detección facial	1,5 days																								
34	Diseño de componente de publicación de mensajes	0,5 days																								
35	Diseño de componente de transferencia de datos	0,5 days																								
36	Entrega Diseño	0 days																								
37	IMPLEMENTACIÓN	28,5 days																								
38	Implementación de la aplicación en máquina Web	9,5 days																								
39	Implementación de Servlet de subida de vídeo	1,5 days																								
40	Implementación de Servlet de carga de imágenes	2 days																								
41	Implementación de componente de transferencia de datos	1 day																								

Page 2



3.6. Estimación de costes

El análisis de los costes del desarrollo de un proyecto es importante para una organización para determinar si dicho proyecto es viable y si se adapta a los requerimientos presupuestarios de ésta. Comúnmente se desglosan en tres partes: los costes de hardware, los costes de software y los costes de recursos humanos.

3.6.1. Costes de Hardware

Los costes de *hardware* del proyecto se reducen al equipo portátil utilizado. Un *Lenovo IdeaPad U330T*. Para este tipo de dispositivos se suele considerar una amortización de 3 años y que según la duración del proyecto supone un 12%.

Hardware	Unidades	Precio	Amortización	Coste final
Lenovo IdeaPad U330T	1	699,00 €	12%	83,90 €
TOTAL (+ IVA 21%)				101.5 €

Tabla 3 Costes de hardware

3.6.2. Costes de Software

Por otra parte, los costes de *software* se componen por un sistema operativo y por 3 programas dado que el resto de aplicaciones y herramientas son soluciones o bien *open-source* o bien de uso gratuito. Normalmente para los productos *software* se establece un periodo de amortización de 6 años, lo que supone 7% para este proyecto. A continuación se resumen los programas utilizados y sus costes.

Software	Descripción	Precio	Amortización	Precio final
Windows 10 Ultimate	Sistema Operativo	135,00 €	7%	9,45 €
Ubuntu 14.04	Sistema Operativo	0,00 €	7%	0,00 €
VirtualBox 5.0.1	Software Virtualización	0,00 €	7%	0,00 €
RabbitMQ	Broker de mensajes	0,00 €	7%	0,00 €
OpenCV	Librería procesado imág.	0,00 €	7%	0,00 €
AvConv	Librería procesado imág.	0,00 €	7%	0,00 €
NetBeans 13	Entorno de desarrollo	0,00 €	7%	0,00 €

Visual Paradigm 14.1	Documentación Ing. Soft.	86,00 €	7%	6,02 €
Microsoft Office 2016	Herramienta ofimática	127,99 €	7%	8,95 €
Microsoft Projects 2016	Administración proyectos	119,00 €	7%	8,33 €
Google Chrome v61	Navegador web	0,00 €	7%	0,00 €
Mozilla Firefox 45	Navegador web	0,00 €	7%	0,00 €
TOTAL (+ IVA 21%)				32,75 €

Tabla 4 Costes de software

3.6.3. Costes de Recursos Humanos

En cuanto al desarrollo de un proyecto de este tipo, diferentes perfiles son necesarios para su consecución. Se considera necesario un Analista de Datos, un Programador y un Jefe de Proyecto. Pese a que la realización la lleva a cabo una única persona, para obtener una estimación realista se tendrán en cuenta los 3 perfiles con sus respectivos salarios base y una cotización a la seguridad social del 28%.

Los sueldos base se puede encontrar publicados en el Boletín Oficial del Estado para jornadas completas de 8 horas, que se dividirá por la mitad para obtener la media jornada que es el tiempo dedicado al proyecto.

Personal	Sueldo anual	Tasa S.S.	Sueldo Anual Total	Cantidad de trabajo	Coste total 5 meses
Analista de Datos	14 329,54 €	4 055,26 €	18 384,80 €	55%	4 213,2 €
Programador	13 508,57 €	3 822,92 €	17 331,49 €	33%	2 383,1 €
Jefe Proyecto	15 150,50 €	4 287,59 €	19 438,09 €	12%	971, 9 €
Total					7 568,2 €
Total media jornada					3 784,1 €

Tabla 5 Costes de recursos humanos con sueldos base de 2017

3.6.3. Costes totales

Los costes totales se derivan de la suma de los anteriormente presentados más un 20% de costes indirectos del proyecto.

Tipo de coste	Coste Directo	Coste Indirecto (20% coste directo)	Coste total
Hardware	101,5 €	20,3 €	121,8 €
Software	32,75 €	6,55 €	39,3 €
Recursos Humanos	3 784,1 €	756,82 €	4 540,9 €
Total			4 702,1 €

Tabla 6 Costes totales estimados del proyecto

3.7. Análisis de riesgos

En esta sección se analizarán los riesgos principales detectados para el desarrollo proyecto y se propondrán medidas para reducir el impacto de estos al mínimo posible.

En la Tabla 7 se calcula la exposición de cada riesgo multiplicando la probabilidad de que suceda por el impacto que causaría en el proyecto. Así se obtienen los riesgos más críticos y a los que habría que prestar mayor atención.

Cód.	Riesgo	Probabilidad	Impacto	Riesgo
A	Elaboración de la planificación		Exposición	12
A.1	Planificación desajustada u optimista	60%	20	12
A.2	Retrasos en tareas producen retrasos en las tareas dependientes	40%	15	6
A.3	Se presentan problemas al abordar las tareas	40%	15	6
B	Organización y gestión		Exposición	4
B.1	Falta de motivación al inicio del proyecto	10%	10	1
B.2	La planificación desborda al autor del proyecto	5%	20	4
B.3	El proyecto se retrasa por motivos laborales	5%	30	1,5
C	Ambiente e infraestructura de desarrollo		Exposición	7
C.1	La máquina de desarrollo no es suficientemente potente para soportar toda la infraestructura	70%	10	7
C.2	El ambiente de trabajo no es adecuado	10%	5	0,5
C.3	Las tecnologías escogidas no cumplen con lo esperado	5%	30	1,5
D	Requisitos		Exposición	7,5
D.1	La especificación de los requisitos no es precisa	30%	20	6
D.2	Aparecen nuevos requisitos no planificados	50%	15	7,5
E	Proyecto		Exposición	10
E.1	Los conocimientos requeridos sobrepasan a los del autor	40%	15	6
E.2	La calidad de la solución no supera los criterios de aceptación	20%	20	4
E.3	Diseño inadecuado	30%	30	10
F	Personal		Exposición	15

F.1	Aparecen problemas personales	10%	20	2
F.2	Desmotivación o desencanto reducen la productividad	40%	30	15

Tabla 7 Posibles riesgos del proyecto

Teniendo en cuenta los riesgos más significativos, se diseñan unas medidas de mitigación y contingencia para minimizar el impacto que se recogen en las Tablas 8 a 13.

A.1.	Planificación desajustada u optimista	12
Medidas de mitigación		
1	Consulta a profesores o profesionales del sector con experiencia en estimación	
2	Basar en estimaciones de proyectos de software ya realizados	
Medidas de contingencia		
1	Aumentar el esfuerzo de trabajo	
2	Replanificar y priorizar tareas	

Tabla 8 Medidas contra el riesgo A.1

B.2	La planificación desborda al autor del proyecto	4
Medidas de mitigación		
1	Abordar las tareas con máxima atención	
2	Realización de horas suplementarias	
Medidas de contingencia		
1	Replanificar y priorizar tareas	
2	Realización de horas suplementarias	

Tabla 9 Medidas contra el riesgo B.2

C.1	La máquina de desarrollo no es suficientemente potente para soportar toda la infraestructura	7
Medidas de mitigación		
1	Diseñar e implementar algoritmos y código eficientes	
2	Utilizar la máquina únicamente para el desarrollo y prescindir de aplicaciones y servicios innecesarios para este	

Medidas de contingencia	
1	Transferir el desarrollo a otra máquina

Tabla 10 Medidas contra el riesgo C.1

D.2	Aparecen nuevos requisitos no planificados	7,5
Medidas de mitigación		
1	Elicitar minuciosamente los requisitos	
2	Realizar controles periódicos del avance del proyecto	
Medidas de contingencia		
1	Realización de horas suplementarias	
2	Acordar un retraso de los plazos de entrega	

Tabla 11 Medidas contra el riesgo D.2

E.3	Diseño inadecuado	10
Medidas de mitigación		
1	Diseñar minuciosamente los especificado por los requisitos	
2	Realizar reuniones con el tutor para corroborar el diseño	
Medidas de contingencia		
1	Realización de horas suplementarias.	
2	Acordar un retraso de los plazos de entrega	

Tabla 12 Medidas contra el riesgo E.3

F.2	Desmotivación o desencanto reducen la productividad	15
Medidas de mitigación		
1	Adquirir conocimientos e intereses sobre el marco del trabajo	
2	Diseñar tareas simples que hagan visible el avance del proyecto semanalmente	
Medidas de contingencia		
1	Vacaciones o tiempo de descanso	
2	Motivaciones externas	

Tabla 13 Medidas contra el riesgo F.2

Capítulo 4. Diseño de la solución

4.1. Diseño del entorno distribuido

El sistema distribuido en red es el centro del desarrollo y lo que permitirá realizar las tareas y optimizar el procesado de imágenes. Sabiendo esto se ha realizado de manera minuciosa para ser consistente y respetar los patrones de diseño de redes.

El sistema cuenta con un número mínimo de cuatro máquinas: el servidor web *Web*, la máquina de almacenamiento *Storage*, el *broker* de mensajes *RabbitMQ* y un *Worker*. A medida de los requerimientos y la carga computacional se añadirían más *workers*. En este estudio se ejecutarán hasta tres simultáneamente dados los recursos limitados del ordenador donde se ejecuta.

Todas las máquinas se interconectan a través del *switch* ofrecido por *VirtualBox* y que mediante la interfaz *host-only* permite conmutar las conexiones de todas las máquinas virtuales. A cada máquina se le asigna una dirección *IP* para poder comunicarse. La asignación se presenta en la tabla de la Ilustración 22. La topología puede observarse en la Ilustración 23.

Es importante entender también cuál es el flujo de información en el sistema. A continuación, se describe el proceso haciendo referencia a los pasos de la Tabla 8. Desde que el usuario sube el vídeo (1), el servidor *Web* lo recoge y almacena en la máquina *Storage* (2) que lo divide en *frames* y encola en el *broker* (3); cada *worker* entonces consume un mensaje de la cola (4) procesa el trabajo y publica un nuevo mensaje en otra cola para indicar que ha sido finalizado el trabajo (5) que finalmente es consumido por *Web* para almacenar dicho *frame* y por *Storage* para eliminarlo (6). En este diagrama las flechas que apuntan a una cola implican que la máquina ha publicado un mensaje en ellas y las que salen de la cola implican que la máquina ha consumido un mensaje de ellas.

Máquina	Hostname	Dirección IP	Puerta de enlace
Web	web	192.168.10.10	192.168.10.1
Storage	storage	192.168.10.20	192.168.10.1
RabbitMQ	rabbitmq	192.168.10.30	192.168.10.1
WorkerN	workerN	192.168.10.50/28	192.168.10.1

Tabla 14 Asignación IP de máquinas del sistema

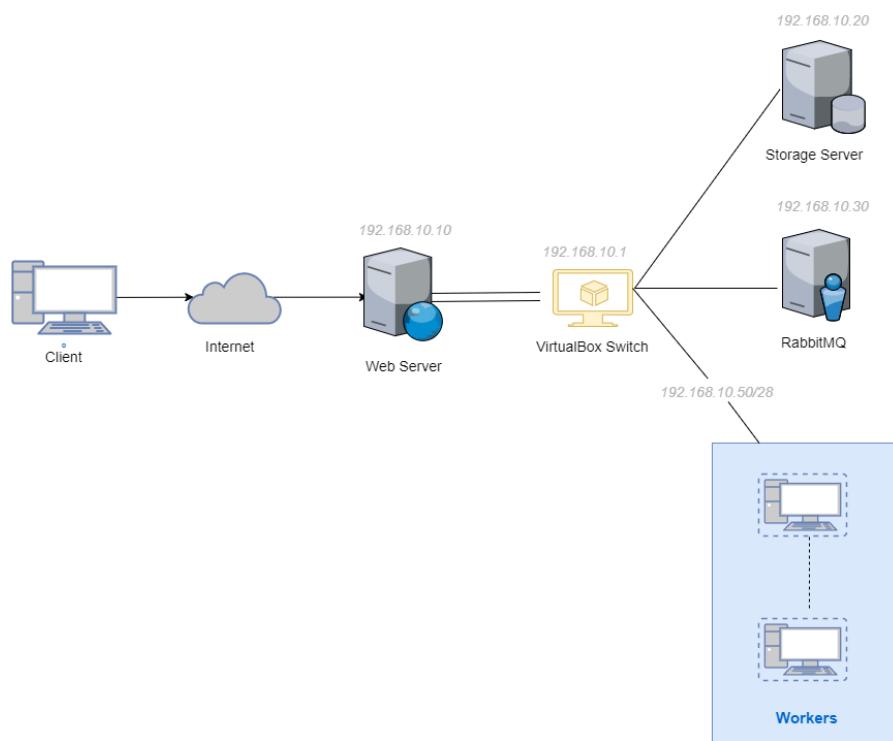


Ilustración 22 Red del entorno distribuido

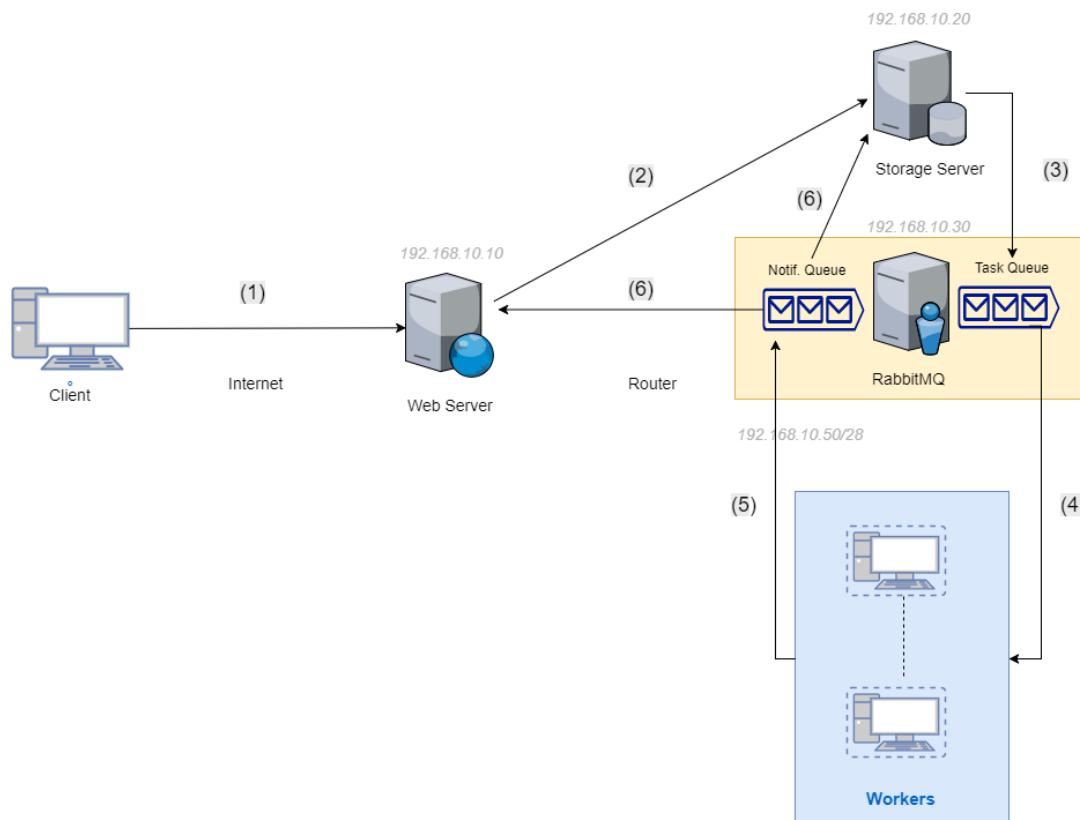


Ilustración 23 Diagrama de flujo de información en la topología

4.2. Diseño de las interfaces de usuario

Las interfaces de usuario son muy simples y no representan mucho trabajo. A continuación, se puede observar en la Ilustración 24 la página principal *index.jsp*. En la Ilustración 25, la página a la que el usuario es redirigido una vez subido el vídeo, indicando el enlace donde podrá encontrar el resultado del procesamiento que queda representada en la Ilustración 26.

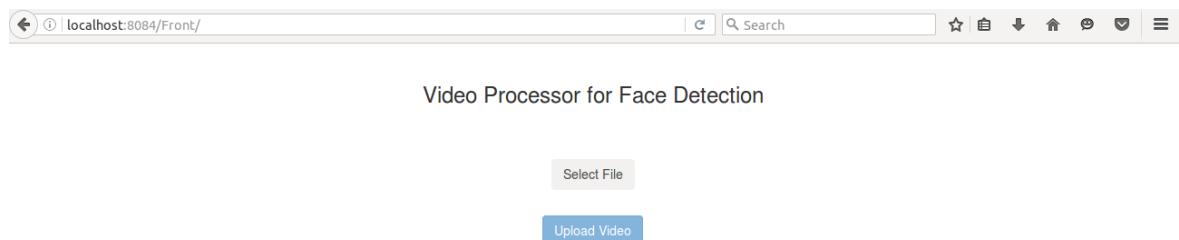


Ilustración 24 Interfaz de usuario para la subida de archivos

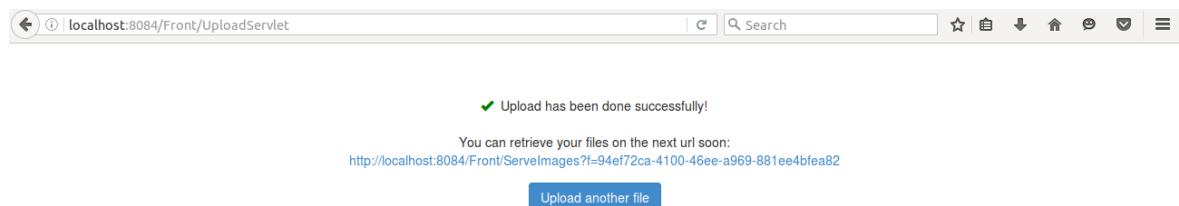


Ilustración 25 Interfaz de Usuario subida correcta

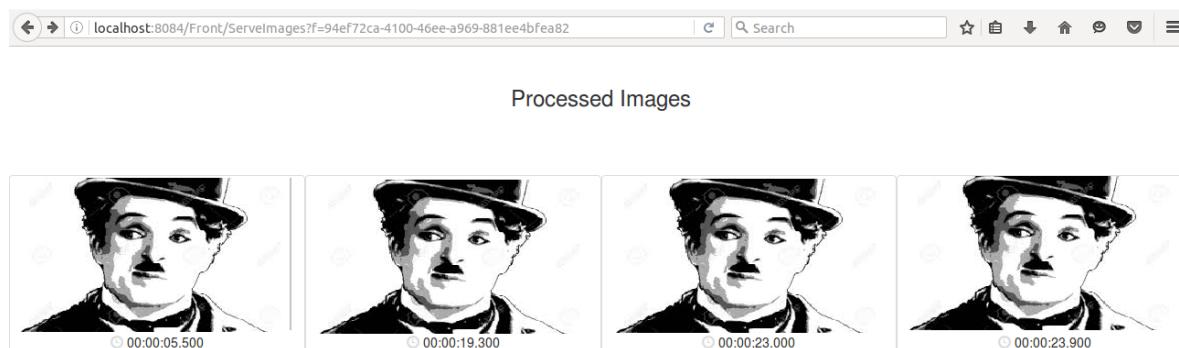


Ilustración 26 Interfaz de Ususario del resultado

4.3. Diseño de colas

En la sección 4.1. Diseño del entorno distribuido se ha ofrecido una pincelada sobre las colas y cómo funcionan, sin embargo, aquí se intenta profundizar y precisar más su funcionamiento. El título del capítulo es *Diseño de colas* denotando más de una. Pese a que para producir tareas y consumirlas sólo es necesaria una cola (aunque podría utilizarse cuántas se deseen) se ha optado por utilizar dos colas más para los mensajes de registro o *logs*. Aunque no llega a ser totalmente necesario, dado que *RabbitMQ* ofrece la opción de encolar mensajes con *routing keys* que los distinguen, es preferible desarrollar un sistema escalable y adaptable a nuevos cambios.

Así, se utilizarán tres colas: *task_queue* será la cola que contienen las tareas de procesado, *storelog_queue* y *weblog_queue* las que reciban los mensajes para las estadísticas. Pese a que todas las estadísticas de cada imagen están contenidas en el mensaje que las representa (recogidas por *Web*) es la máquina *Storage* quien inicia el proceso y por tanto la única máquina que puede determinar el tiempo en el sistema de un vídeo completo (referenciado por un *UUID*).

Con respecto a las colas de *log*, dado que es el *Worker* quién publica el mensaje y debe hacerlo en dos colas distintas, se utilizará la funcionalidad de *RabbitMQ exchanges* que como ya se ha explicado permiten abstraer a los productos de las colas. Para ello se utilizará un *exchange* de tipo *fan out*, o disperso en castellano, que permite publicar los mensajes que reciba en todas las colas con las que tenga un enlace, que se denominará *logs*.

Por coherencia y escalabilidad, para la cola de tareas se procederá a hacer los mismo y se creará también un *exchange*, pero esta vez de tipo *direct* que publica un mensaje en una y sólo una cola con nombre *img_process*.

En la Ilustración 27 se representa el procesado de las tareas. La máquina *Storage* publica un mensaje en el *exchange* *img_process* y éste, al estar *atado* a la cola *task_queue*, lo publica en esta última. Por último, los *workers* están a la espera de que haya mensajes en la cola y consumirlos.

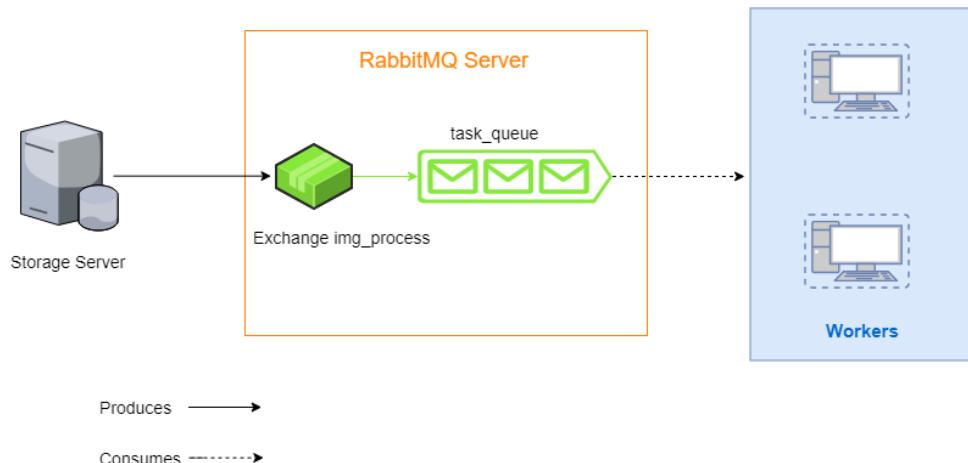


Ilustración 27 Proceso de producción y consumición de tareas

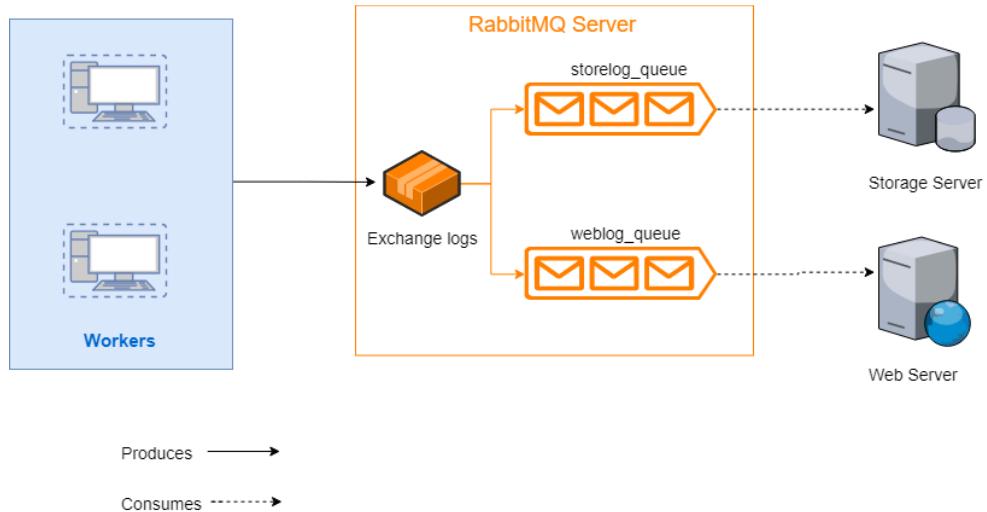


Ilustración 28 Proceso de producción y consumición de logs

Una vez las tareas han sido procesadas, los *Workers* completan el mensaje con la información de procesado y lo publican en el *exchange logs* que a su vez publica cada mensaje en todas las colas con las que esté *ligado*. En la Ilustración 28 se representa el proceso.

Cómo medida para futuras adaptaciones del sistema también se ha establecido *routing keys* para los mensajes. Esto permitiría, si hubiera diferentes tipos de trabajos a realizar por los *workers*, consumir únicamente los mensajes que estén destinados específicamente a ese tipo de trabajo.

4.4. Diseño de la comunicación

Se ha hablado bastante sobre mensajes, estadísticas y tiempos, pero no se ha explicado qué contienen los mensajes. Para empezar, el mensaje seguirá el siguiente objeto *JSON*:

```
{
  "sourceDir": string,
  "sourceIP": string,
  "destinationDir": string,
  "destinationIP": string,
  "workerID": string,
  "fileName": string,
  "operation": string,
  "processTime": number,
  "queueInTime": number,
  "queueOutTime": number
}
```

La propiedad *source* hace referencia a la máquina o directorio en la que se almacena la imagen (*Storage*) y *destination* a donde deberá enviarse (*Web*). Del campo *operation* no se hará uso en este proyecto, pero en caso de que se quisieran aplicar otros procesados a la imagen, se podría referenciar mediante este parámetro. El campo *processTime* indica el tiempo que ha tardado en procesarse la imagen y los referenciados por *queue* representan el tiempo de entrada y salida de la cola, el primero establecido antes de publicar el mensaje en el *exchange img_process* por *Storage* y el segundo por el *Worker* una vez consumido de la cola *task_queue*.

4.4. Arquitectura de software

La arquitectura software de la solución es sin duda el punto más delicado del proyecto dada su naturaleza. Al ser un entorno distribuido donde un número significativo de máquinas interconectadas deben cooperar, el diseño de este proceso marcará el éxito del proyecto en primera instancia, y después la usabilidad y escalabilidad del mismo en el futuro.

4.4.1. Diagrama de clases

El diagrama de clases es una herramienta indispensable a la hora de diseñar una aplicación o un conjunto de ellas. Expresa la relación entre clases como la herencia o agregación de éstas y los atributos que poseen y operaciones que son capaces de llevar a cabo.

En este proyecto, se tiene un sistema distribuido en varias máquinas, cada cual con sus clases propias. Para expresar la relación entre las máquinas se presenta el diagrama de paquetes de la Ilustración 29. Los paquetes se relacionan, representando cada uno de éstos un diagrama de clases propio. Cada paquete se denomina como la máquina que representa y todos comparten (excepto el servidor *RabbitMQ*) un paquete *Common* que

contiene una clase común a todos ellos, el mensaje *Message* definido en la sección 4.4. Diseño de la comunicación, y que utilizan en los procesos de comunicación.

En las figuras siguientes se pueden observar los diagramas de clases de cada máquina, con sus atributos, métodos y relaciones y la relación entre paquetes. Cabe destacar que las flechas de unión representan únicamente una relación entre los paquetes, ya sea comunicativa o agregativa.

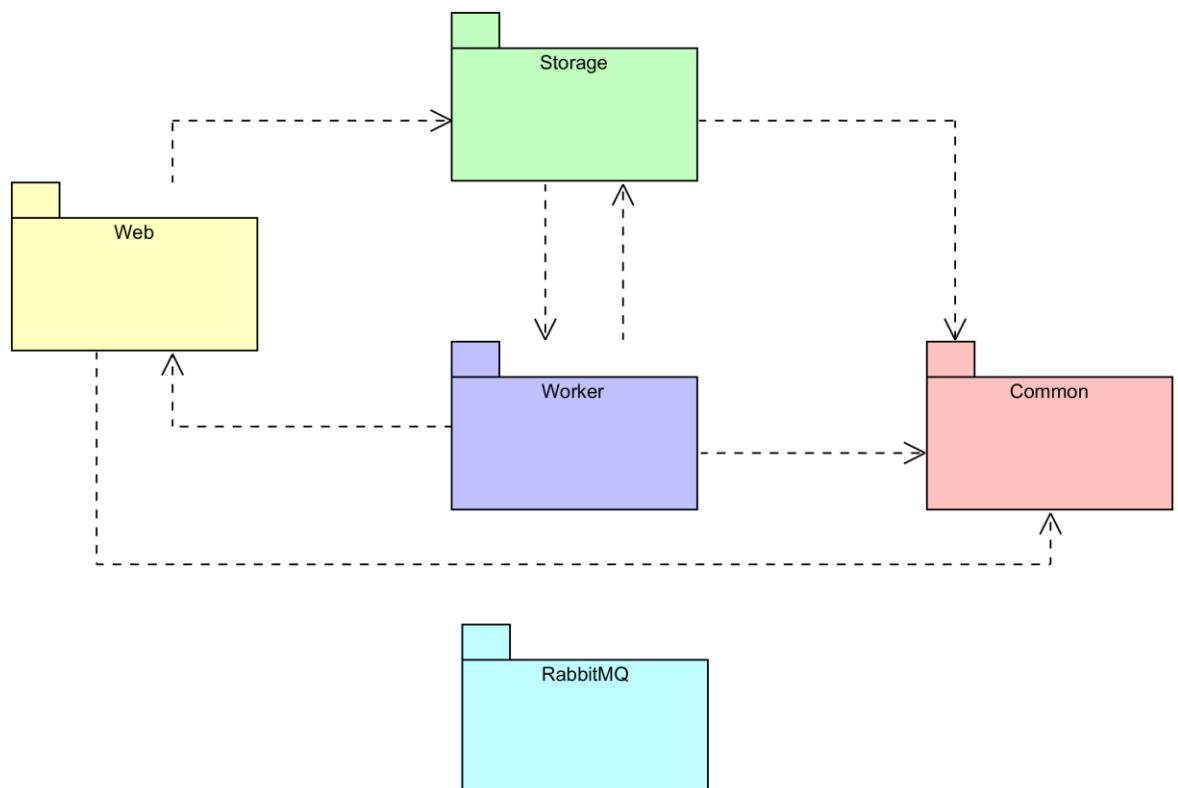


Ilustración 29 Diagrama de paquetes del sistema

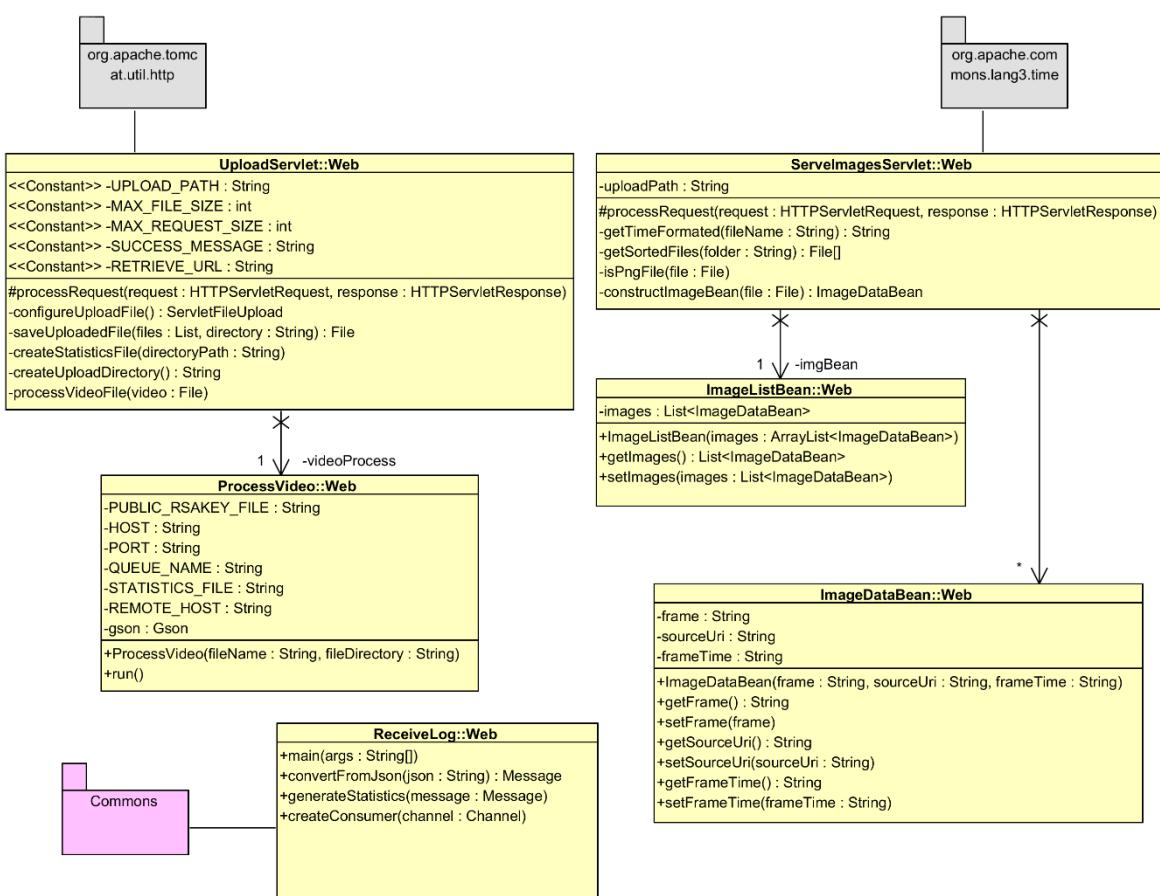


Ilustración 30 Diagrama de clases de aplicación en Web

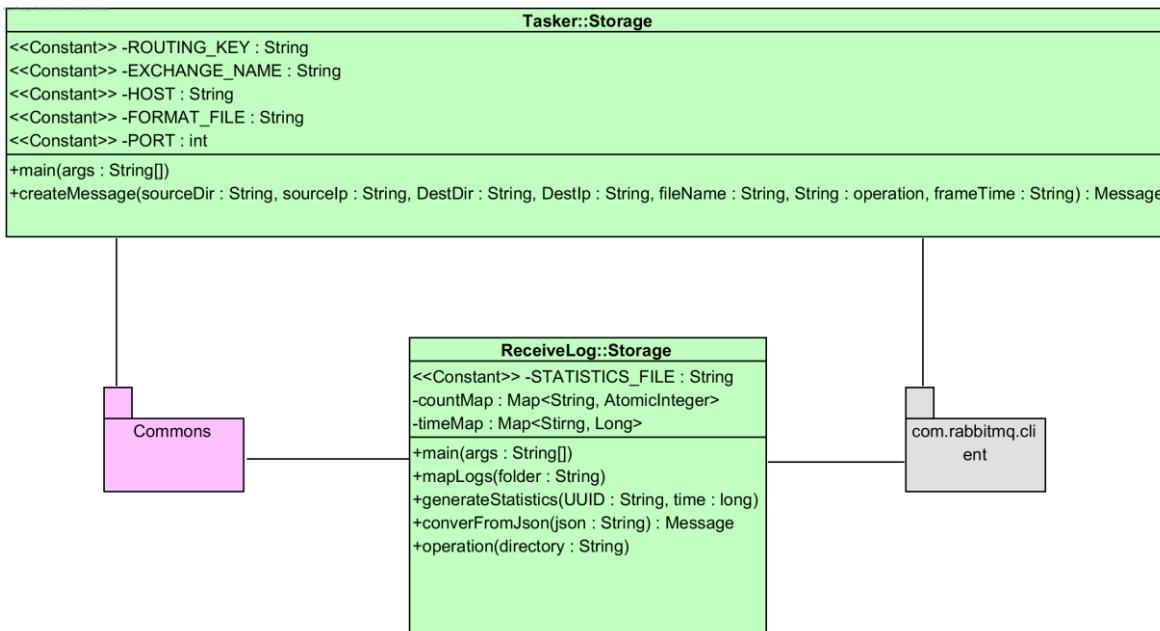


Ilustración 31 Diagrama de clases de aplicación en Storage

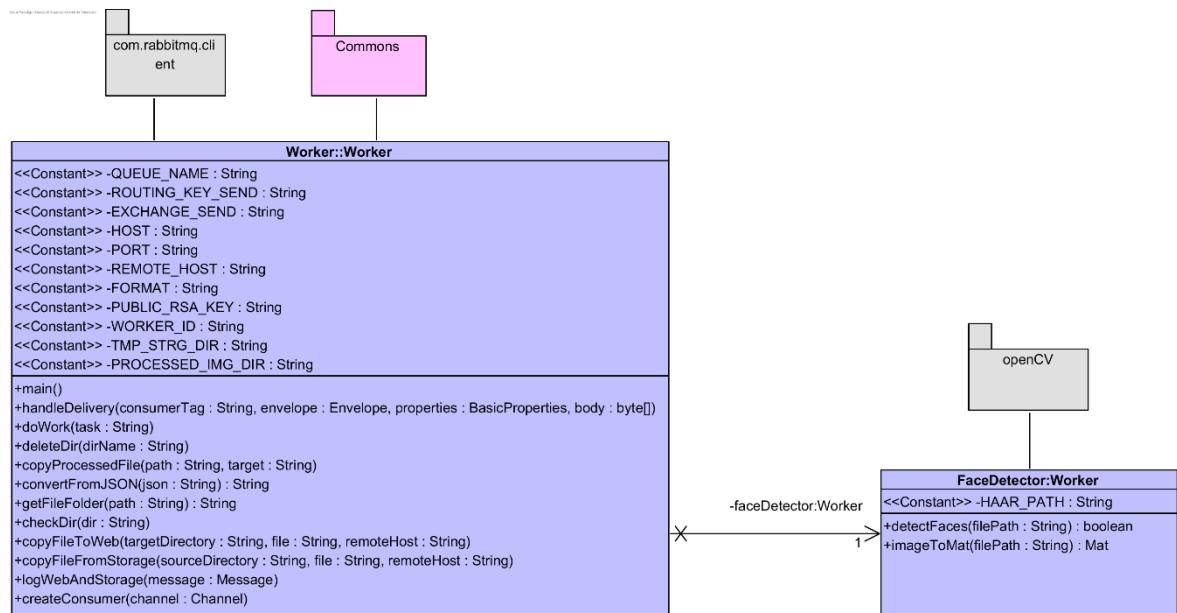


Ilustración 32 Diagrama de clases de aplicación en Worker

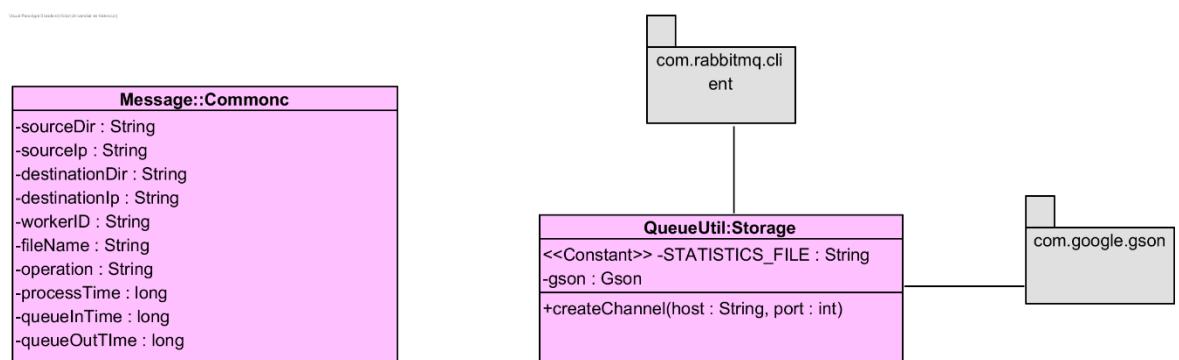


Ilustración 33 Diagrama de clases de paquete Common

QueueInitializer
<<Constant>> -TASK_QUEUE_NAME : String
<<Constant>> -WEBLOG_QUEUE_NAME : String
<<Constant>> -STORELOG_QUEUE_NAME : Str...
<<Constant>> -HOST : String
<<Constant>> -PORT : int
<<Constant>> -QOS : int
<<Constant>> -EXCHANGE_LOG : String
<<Constant>> -EXCHANGE_TASK : String
<<Constant>> -ROUTING_KEY_LOG : String
<<Constant>> -ROUTING_KEY_TASK : String
+main(argv : String[])

Ilustración 34 Diagrama de clase de la máquina RabbitMQ

4.4.2. Diagramas de interacción de las operaciones del sistema

Los diagramas de interacción expresan detalladamente las acciones a llevar a cabo para implementar el código y deben seguirse minuciosamente. Deben respetar los patrones de diseño convencionales y ser claros y concisos. Se dividen a continuación los diagramas según la máquina a la que pertenecen.

Algunos diagramas han sido omitidos para evitar redundancias dado que representan procesos casi idénticos en otras máquinas.

RabbitMQ

DS main: Este Diagrama representa la inicialización de las colas. En él sólo se ha representado la creación de un *exchange*, una cola y la relación de estos. Sin embargo, es necesario repetir los tres últimos mensajes también para las dos colas de *logging*.

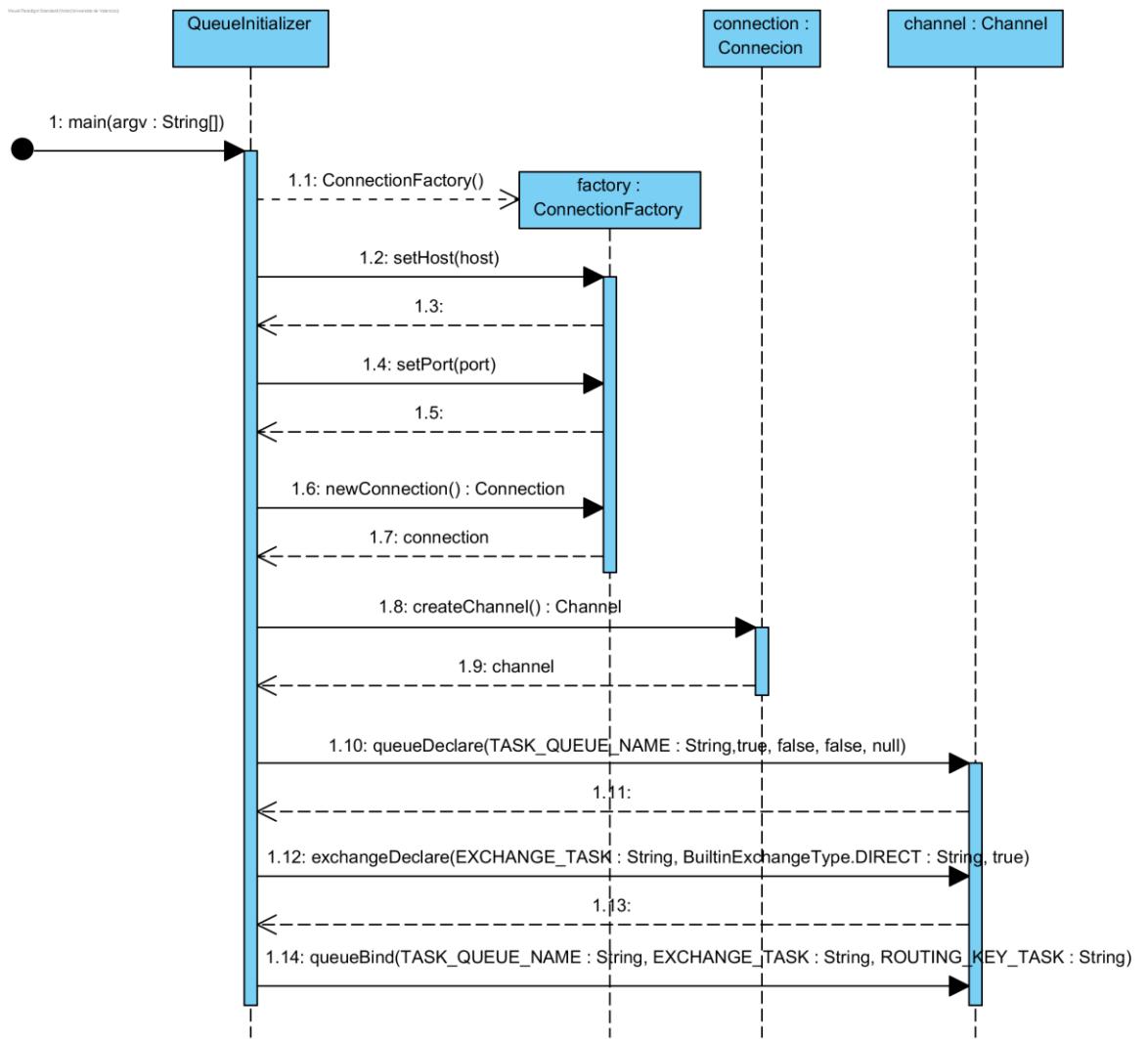


Ilustración 35 DIOS main RabbitMQ

Common

DS createChannel: Este Diagrama representa la creación de un canal, especificando un *host* y un puerto.

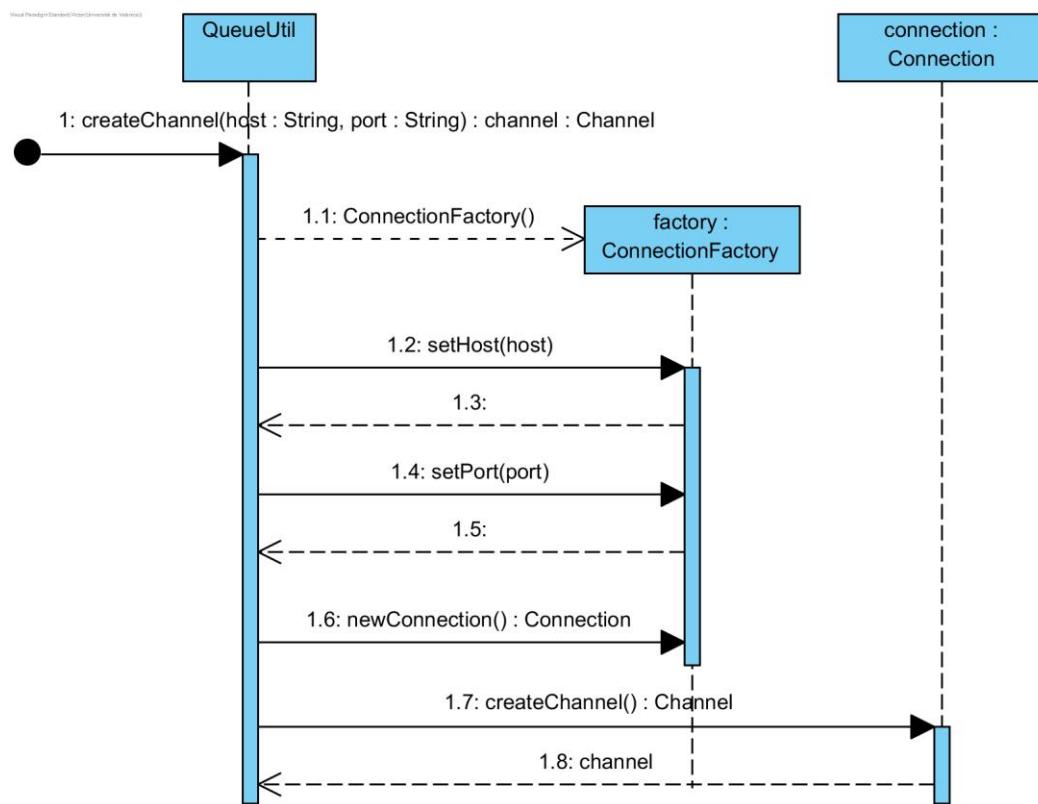


Ilustración 36 DIOS createChannel

Web

DS processRequest: Este Diagrama representa la subida del vídeo al servidor.

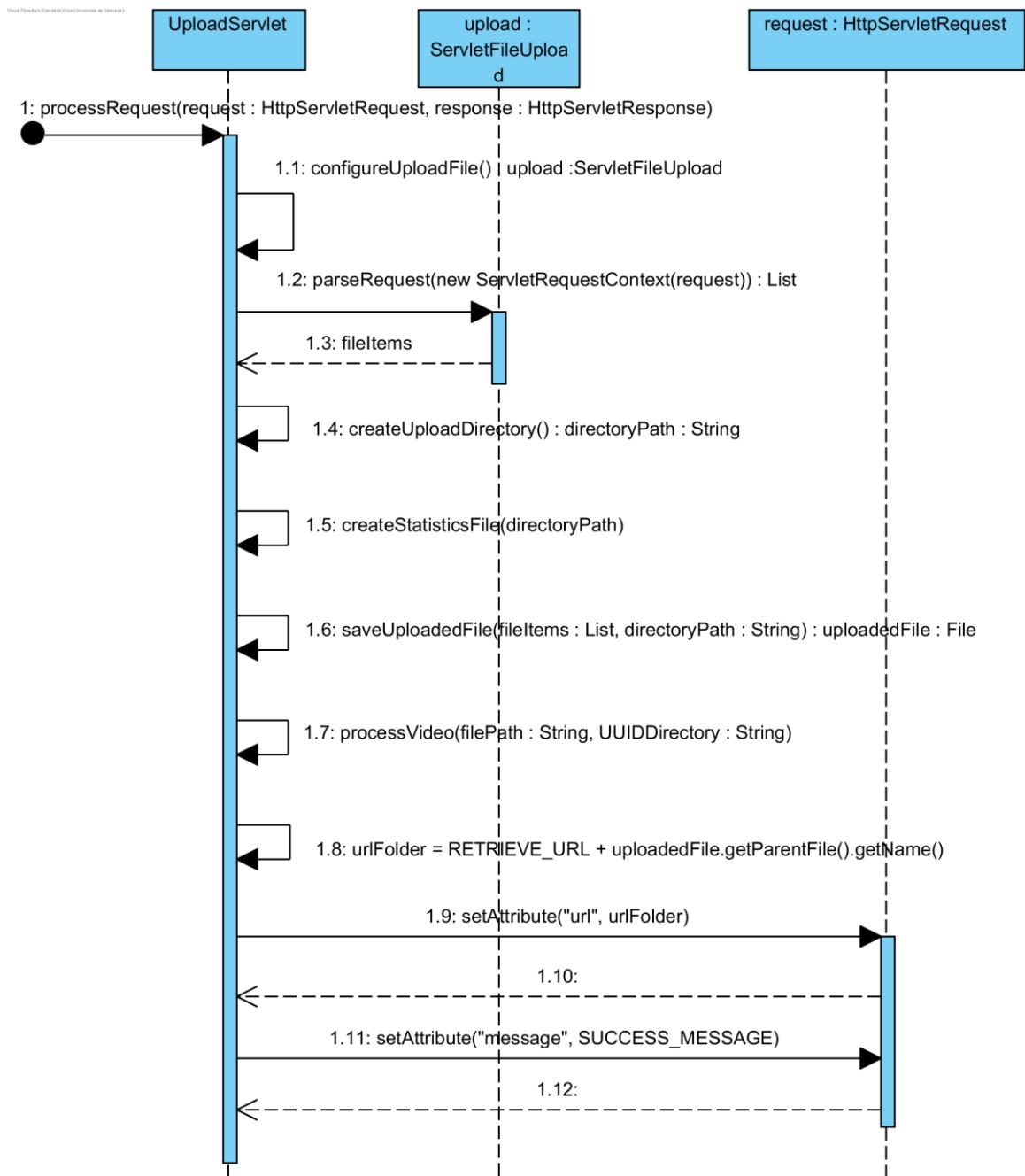


Ilustración 37 DIOS doPost



UNIVERSITAT
POLÈTICA
DE VALÈNCIA

[] Escola Tècnica
Superior d'Enginyeria

DS configureUploadFile: Este Diagrama representa la creación de un objeto y sus parámetros para recoger un archivo de la petición HTTP.

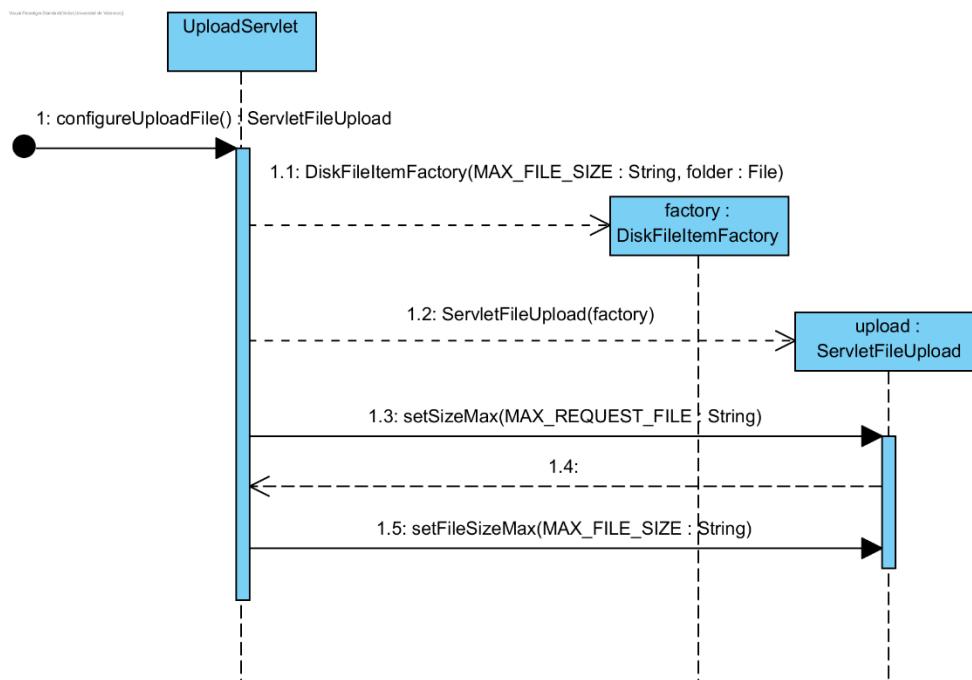


Ilustración 38 DIOS configureUploadFile

DS createUploadDirectory: Este diagrama representa la creación del directorio dónde se subirá el archivo.

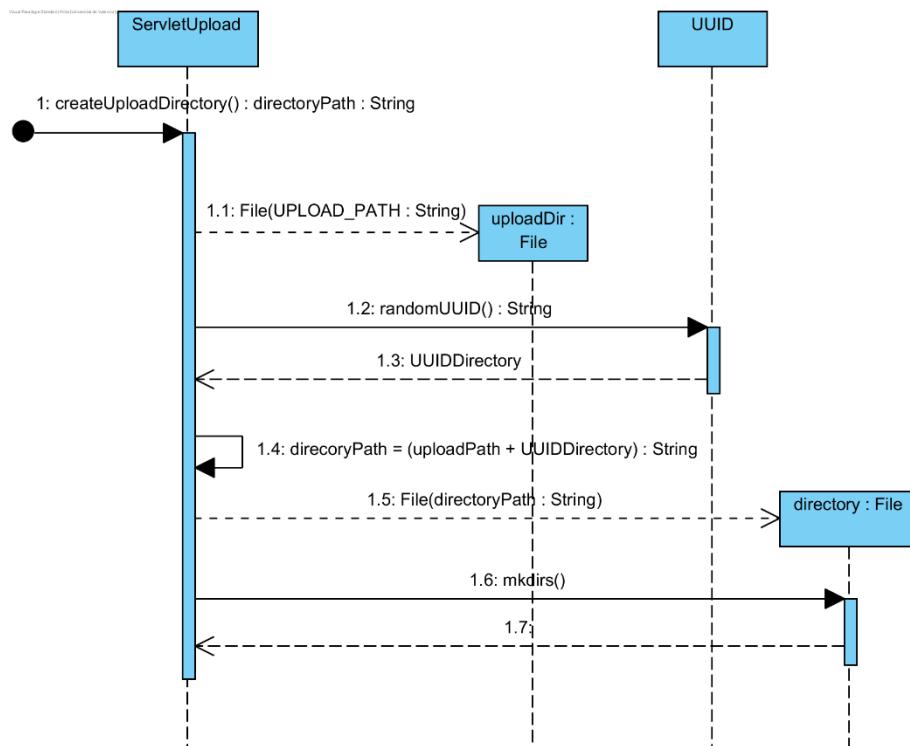


Ilustración 39 DIOS createUploadDirectory

DS createStatisticsFile: Este diagrama representa la creación del archivo de estadísticas en la carpeta de cada archivo.

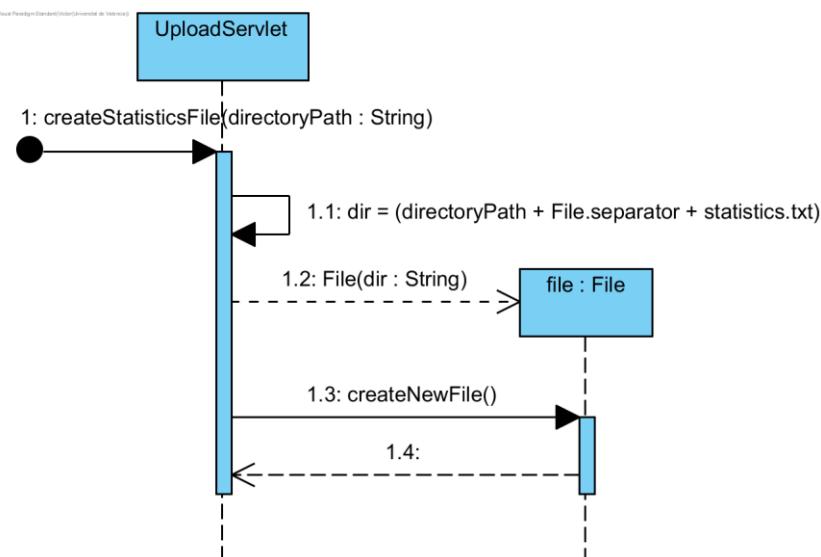


Ilustración 40 DIOS createStatisticsFile

DS saveUploadedFile: Este Diagrama representa el guardado en disco del archivo subido.

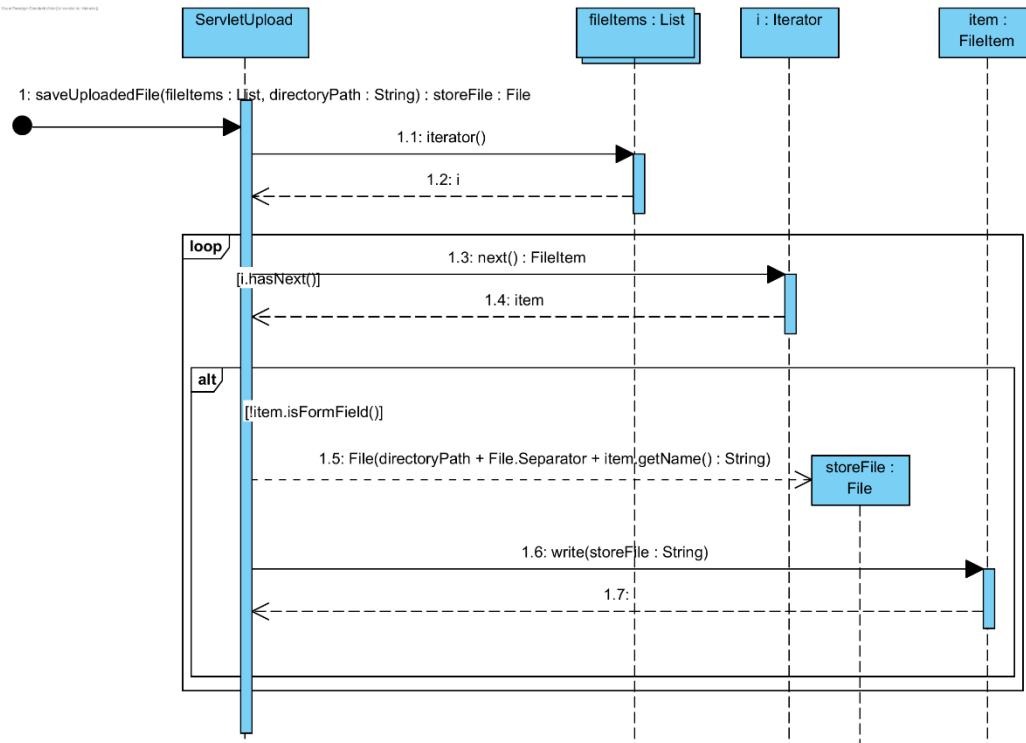


Ilustración 41 DIOS saveUploadedFile

DS processVideo: Este Diagrama representa el proceso de lanzado en un hilo del proceso de copia, encolado y procesado del vídeo. Es la función que desencadena toda la funcionalidad del sistema

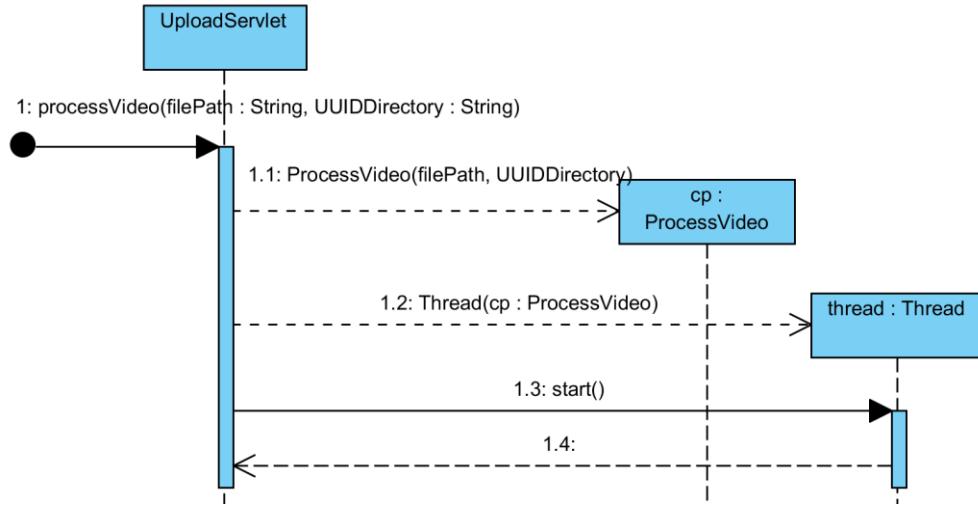


Ilustración 42 DIOS processVideo

DS run: Este Diagrama representa la ejecución del hilo (*Thread*) del Diagrama anterior mediante el método *start()*, en el que se ejecutan los procesos de copia a la máquina Storage y de ejecución del proceso de detección facial.

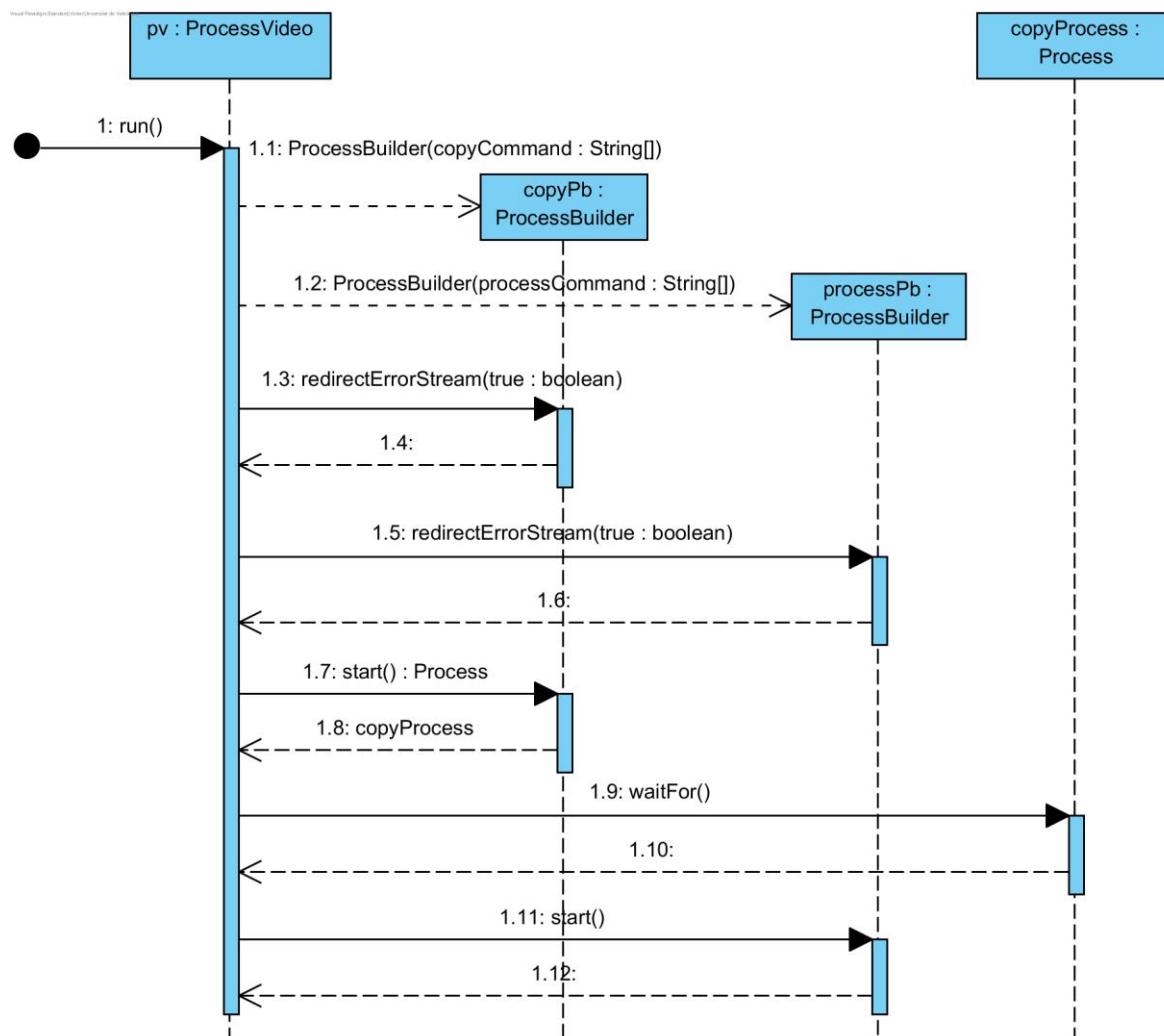


Ilustración 43 DIOS run

DS main ReceiveLog: Este diagrama representa la creación de un consumidor y su puesta en marcha.

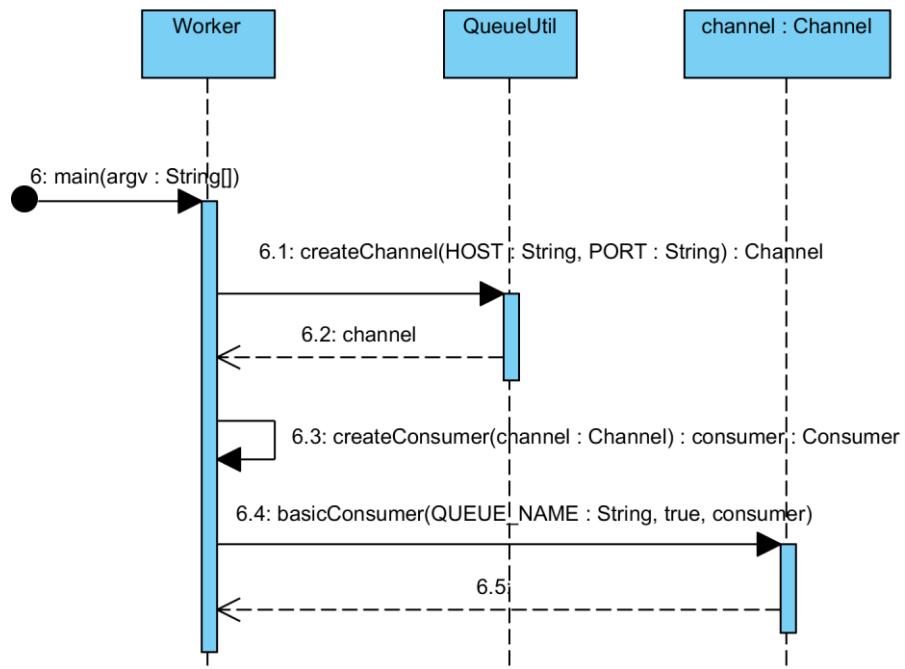


Ilustración 44 DIOS main ReceiveLog

DS createConsumer: Este Diagrama representa la creación de un nuevo consumidor. La clase `DefaultConsumer` es abstracta y exige así sobrescribir el método `handleDelivery`. El cual se representa en el Diagrama de la Ilustración 46.

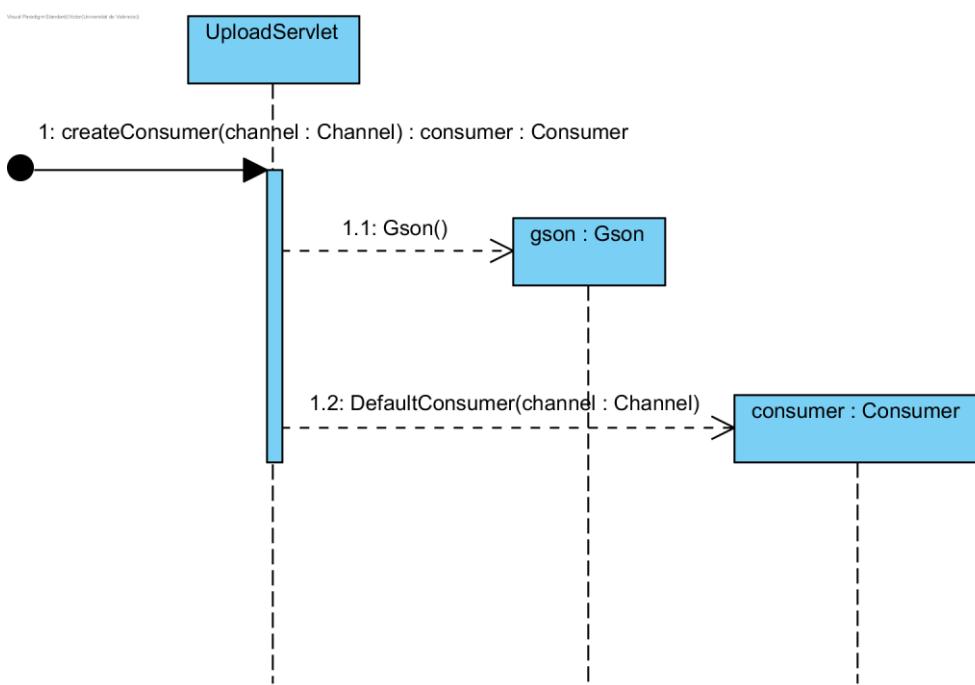


Ilustración 45 DIOS `createConsumer`

DS handleDelivery: Este diagrama representa el método que se ejecuta cuando un mensaje es consumido de la cola.

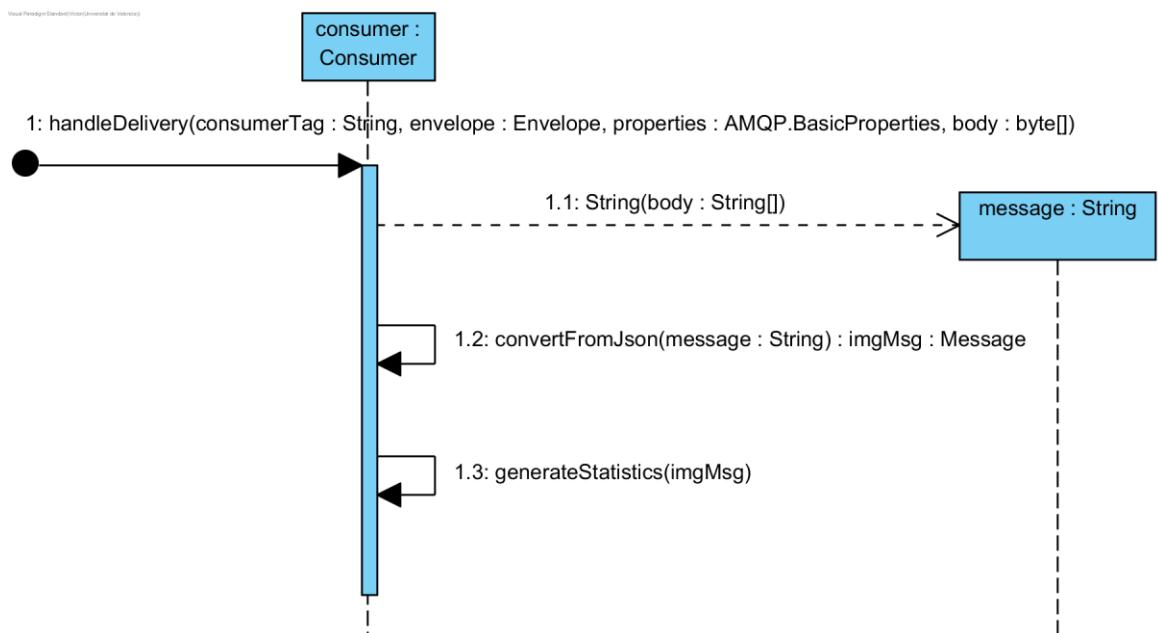


Ilustración 46 DIOS handleDelivery Storage

DS convertFromJson: Este Diagrama representa la deserialización del mensaje.

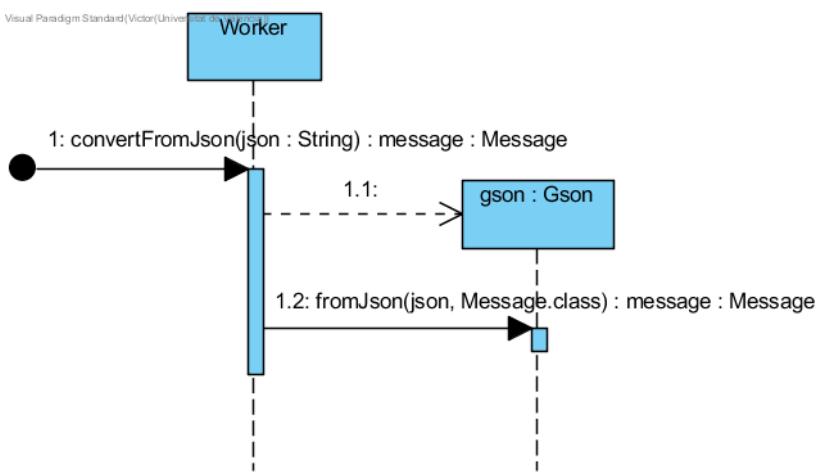


Ilustración 47 DIOS convertFromJson

DS generateStatistics: Este Diagrama representa la función encargada de generar las estadísticas para cada mensaje recibido.

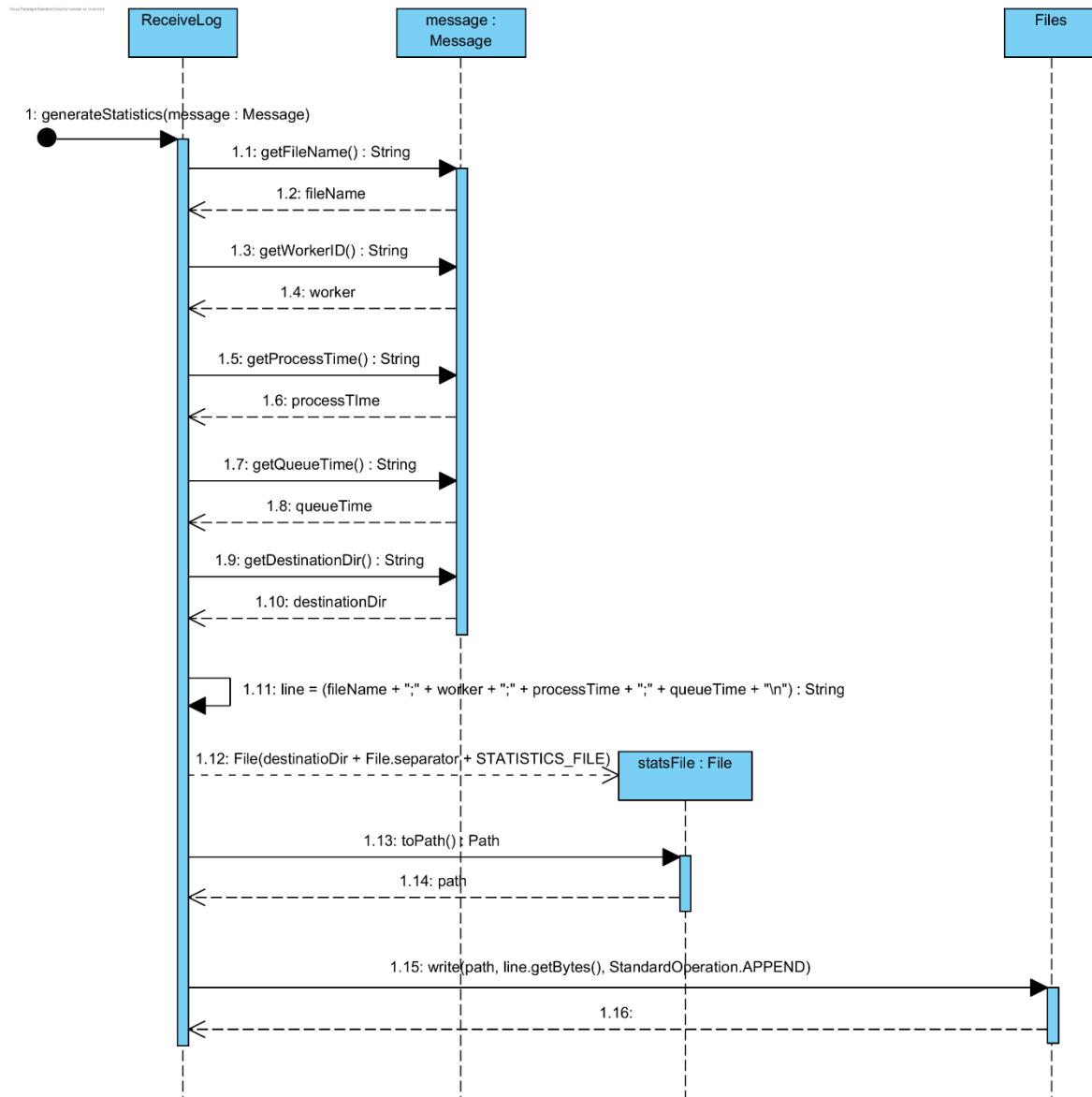


Ilustración 48 DIOS generateStatistics

DS processRequest ServelImagesServlet: Este Diagrama representa el procesado de la petición HTTP para listar el resultado de la operación del sistema, es decir, las imágenes en las que se ha detectado una cara.

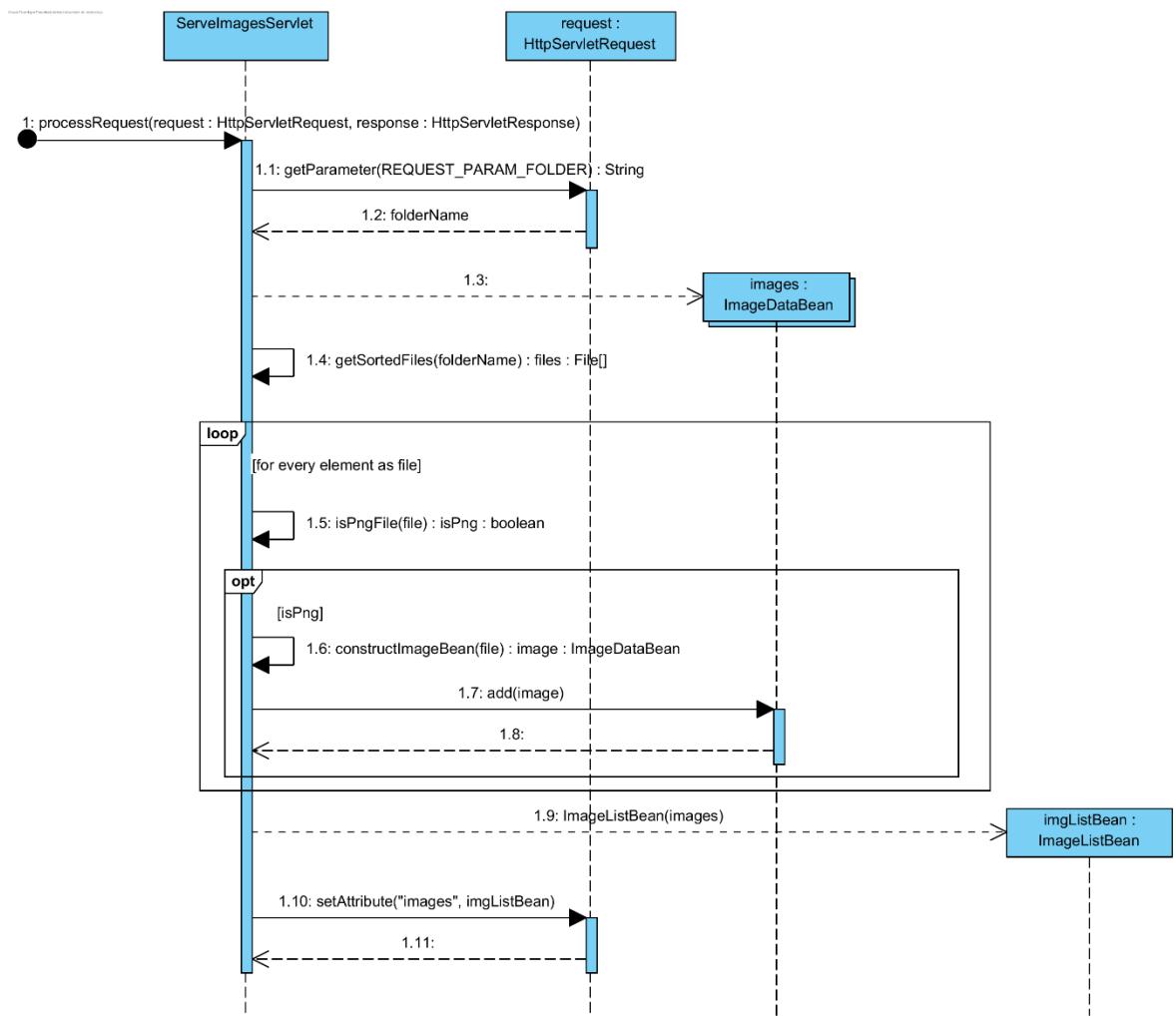


Ilustración 49 DIOS processRequest ServelImagesServlet

DS getSortedFiles: Este Diagrama representa la función de ordenar alfabéticamente y así cronológicamente las imágenes.

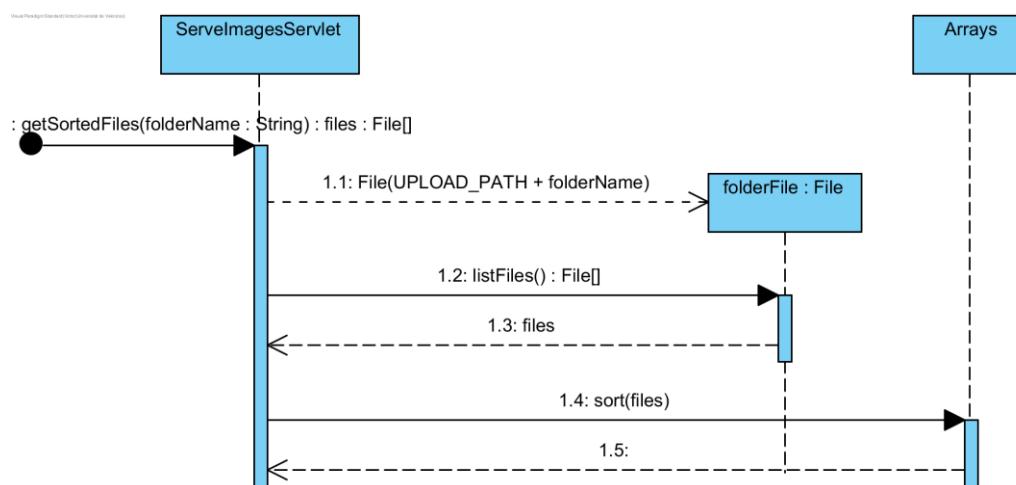


Ilustración 50 DIOS getSortedFiles

DS isPngFile: Este Diagrama representa una función para comprobar si un archivo tiene la extensión .jpg.

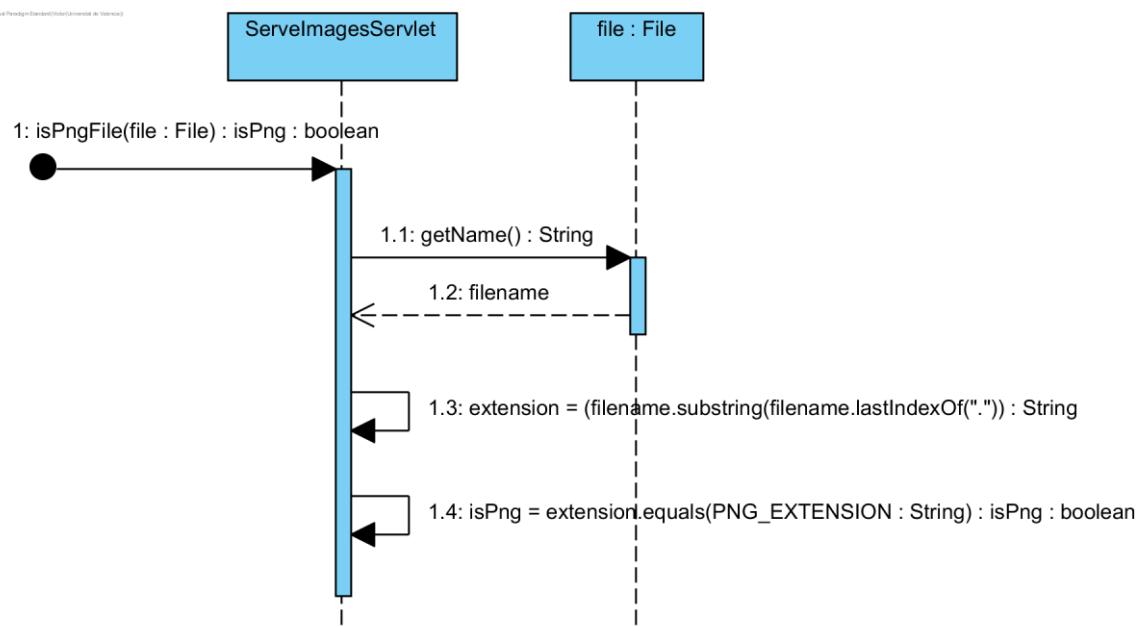


Ilustración 51 DIOS isPngFile

DS constructImageBean: Este Diagrama representa la construcción del bean que representa una imagen.

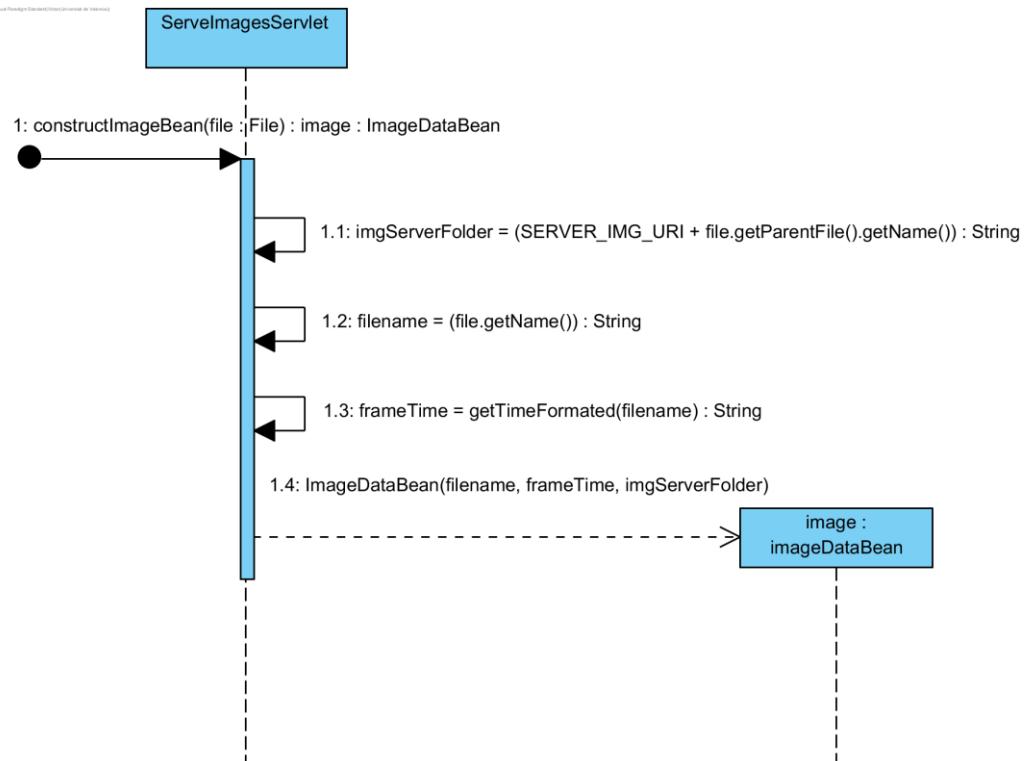


Ilustración 52 DIOS `constructImageBean`

Storage

DS main Tasker: Este Diagrama representa el método principal que publica los mensajes a partir de los *frames*.

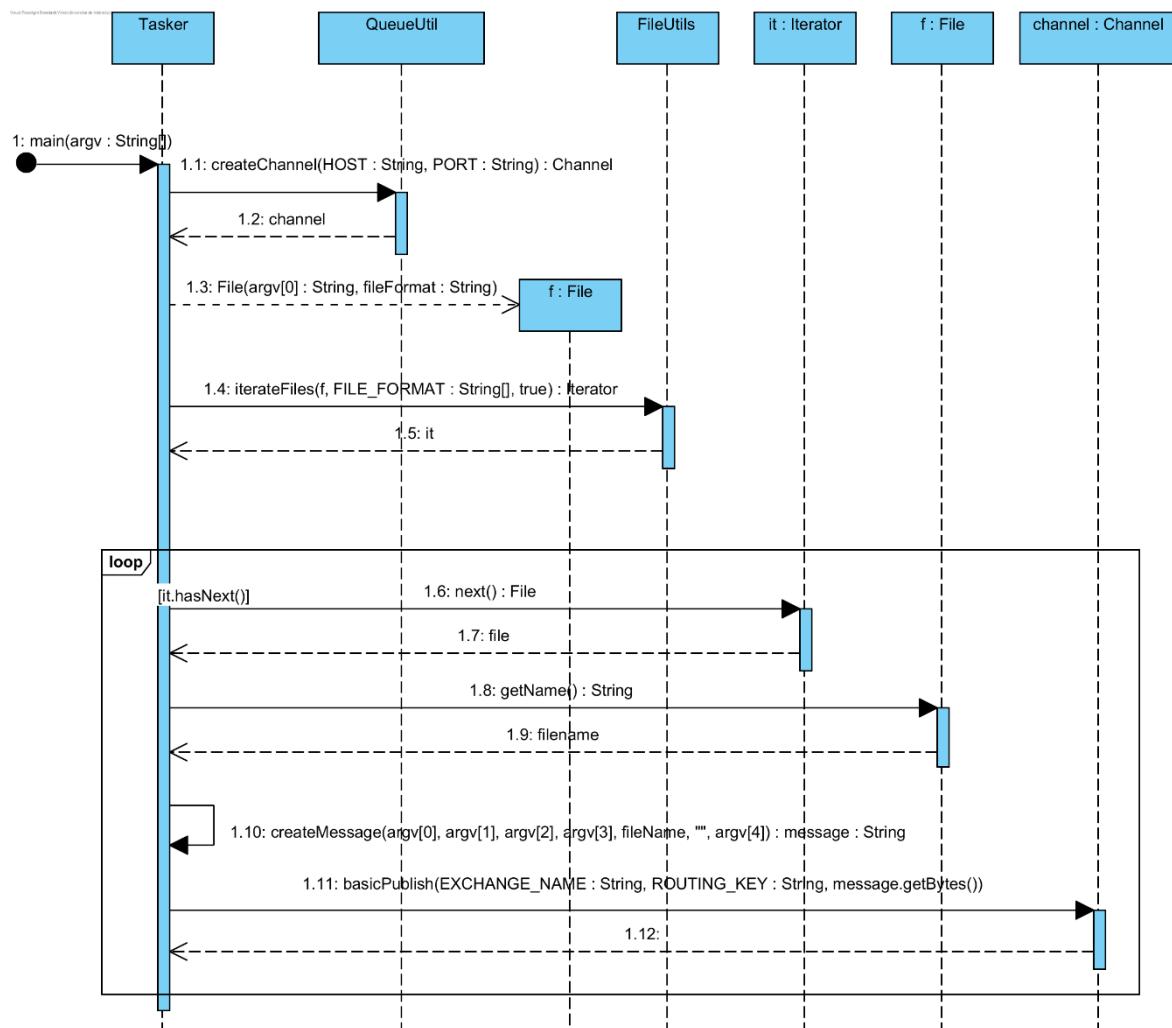


Ilustración 53 DIOS main Storage

DS createMessage: Este Diagrama representa la producción de mensajes a partir de los *frames* del vídeo guardado.

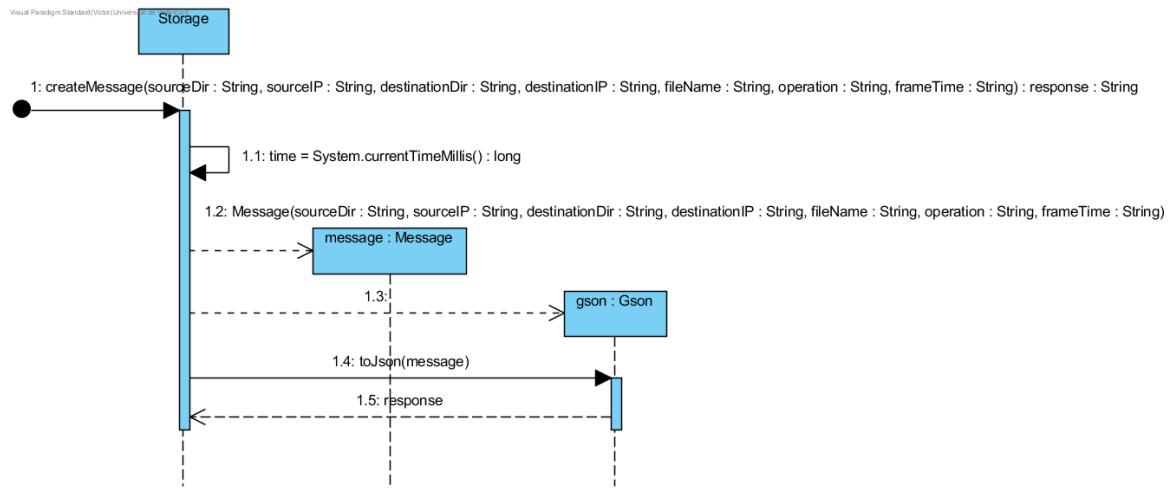


Ilustración 54 DIOS createMessage

DS main ReceiveLog: Este Diagrama representa la consumición de los mensajes que contienen los *logs*

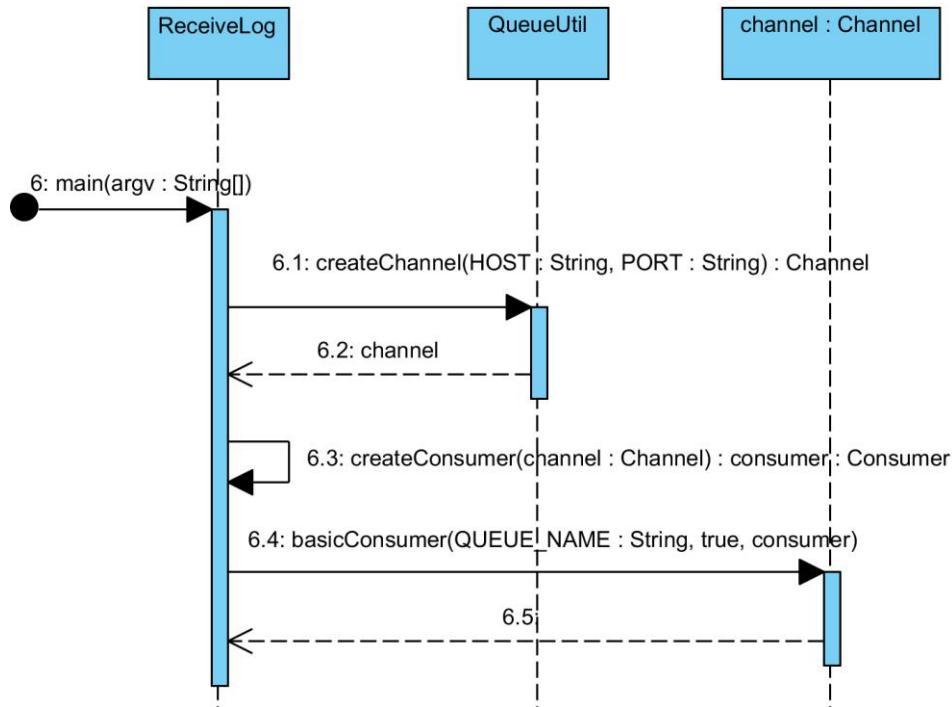


Ilustración 55 DIOS main ReceiveLog

DS handleDelivery: Del mismo modo que antes, se crea un consumidor con el mismo método que previamente *createConsumer* y se le pasa al método *basicConsume*. Que ejecutará el método *handleDelivery* sobrescrito en el consumidor.

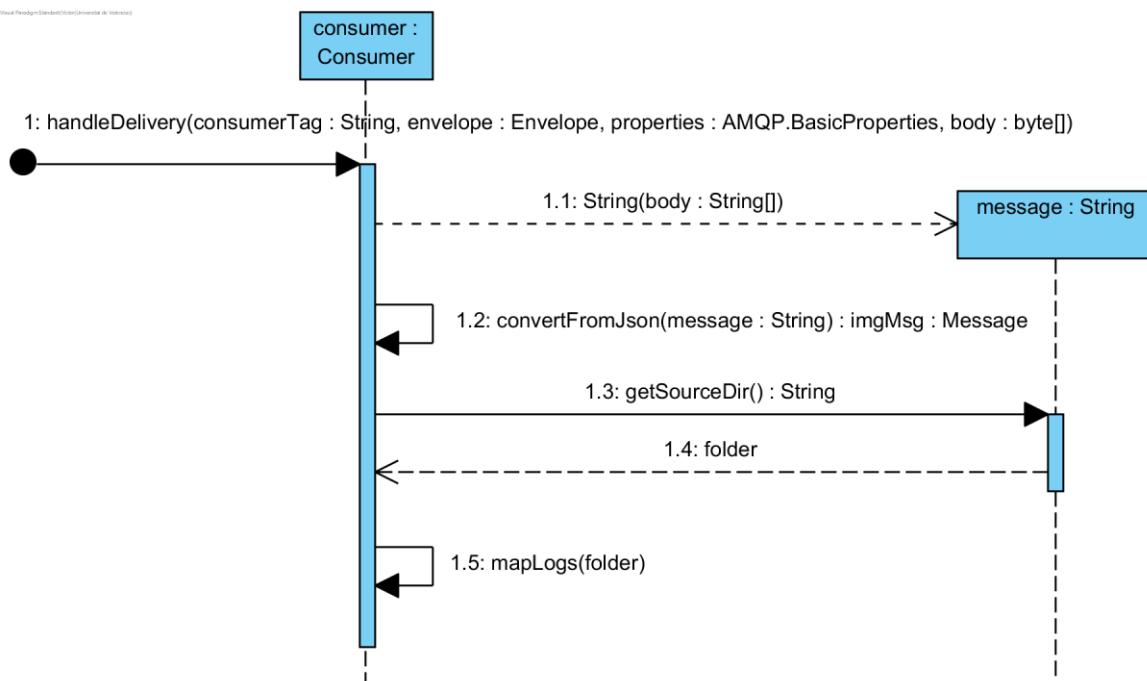
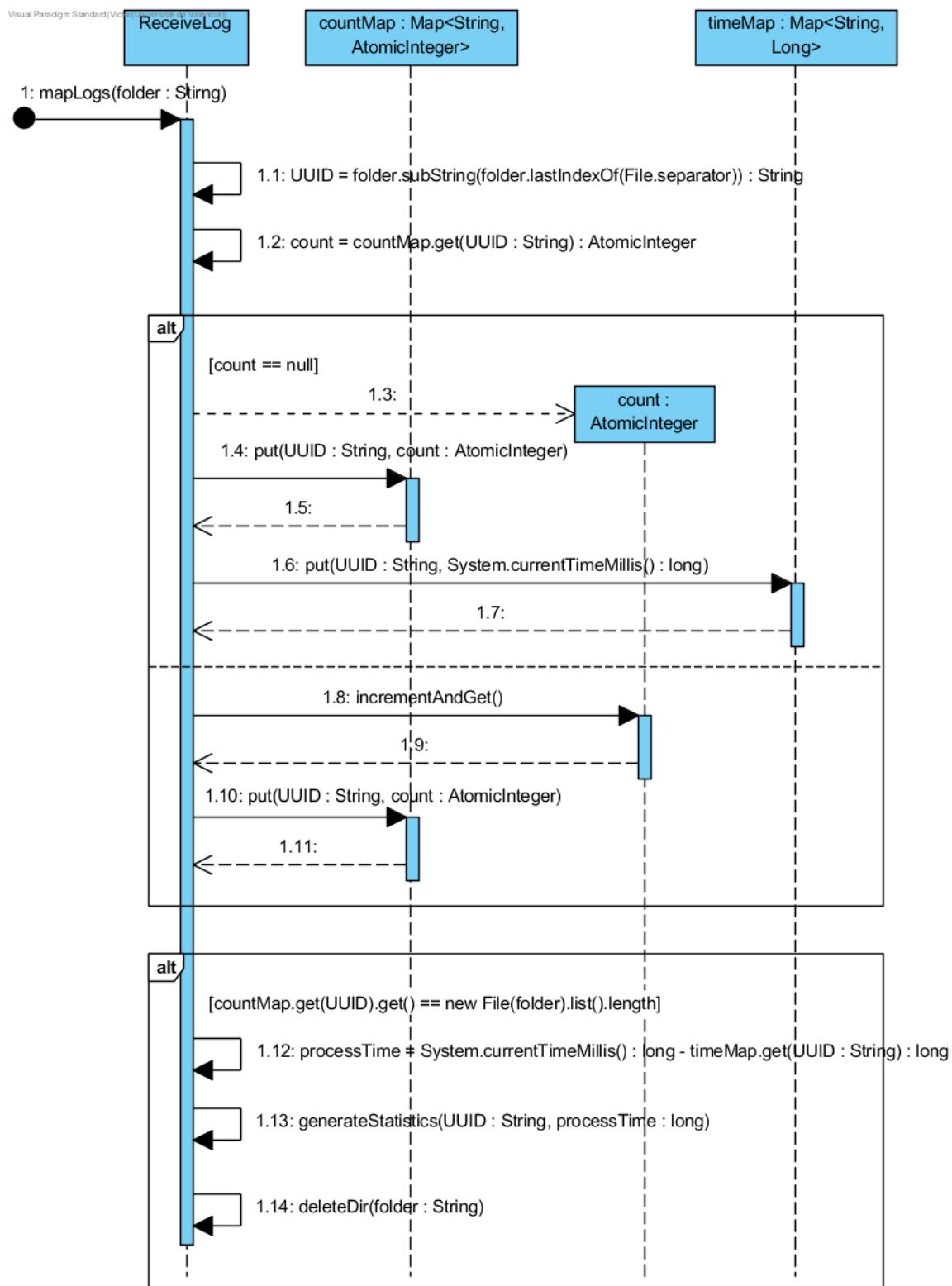


Ilustración 56 DIOS handleDelivery Storage

DS mapLogs: Este Diagrama representa el proceso de cálculo de estadísticas, es decir el cálculo del tiempo de procesamiento de una tarea.



Worker

DS main: Este Diagrama representa el método principal encargado de consumir mensajes de una cola. Del mismo modo que en los anteriores casos, el método *handleDelivery* es sobrescrito para definir las acciones a llevar a cabo cuando un mensaje es consumido de la cola.

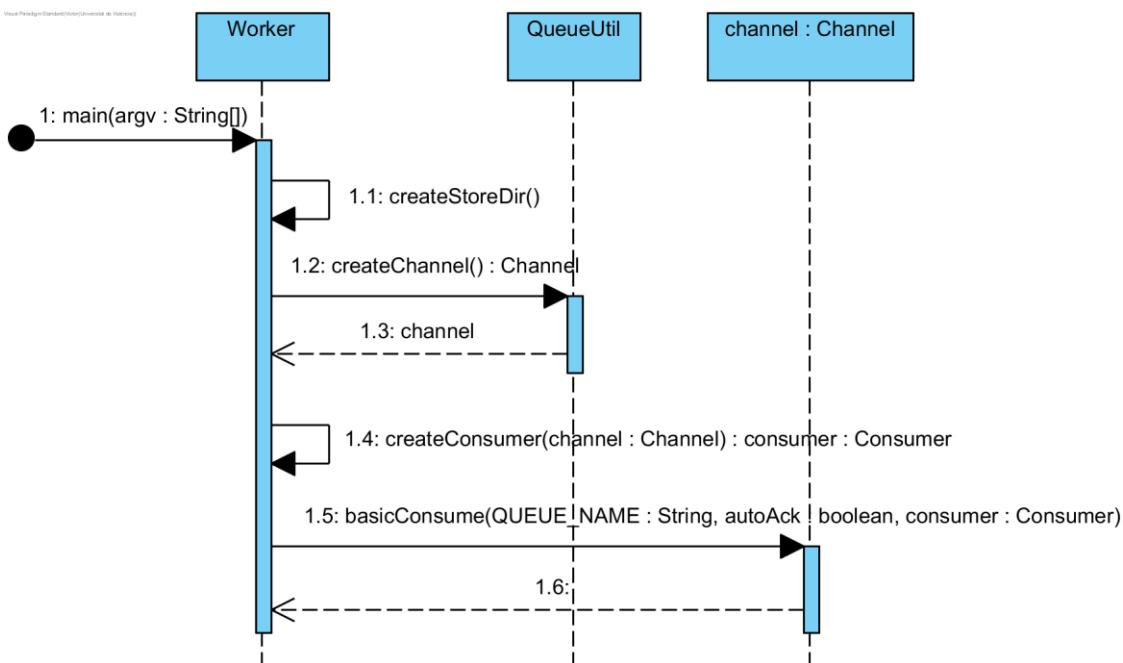


Ilustración 57 DIOS main Worker

DS handleDelivery:

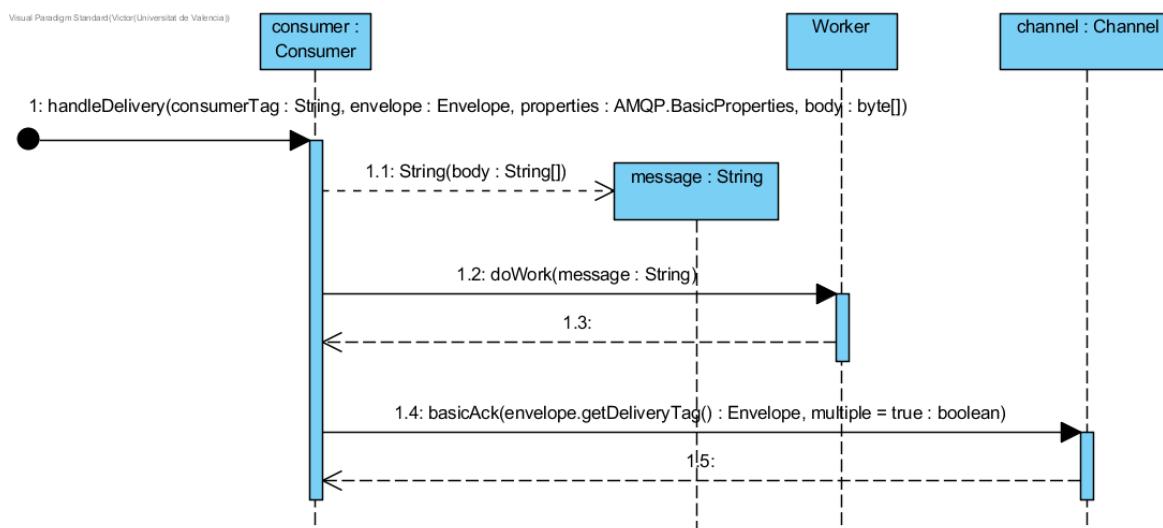


Ilustración 58 DIOS handleDelivery Worker



DS doWork: Este Diagrama representa el método más importante, que procesa los trabajos, envía las imágenes con positivos en detección y envía los logs.

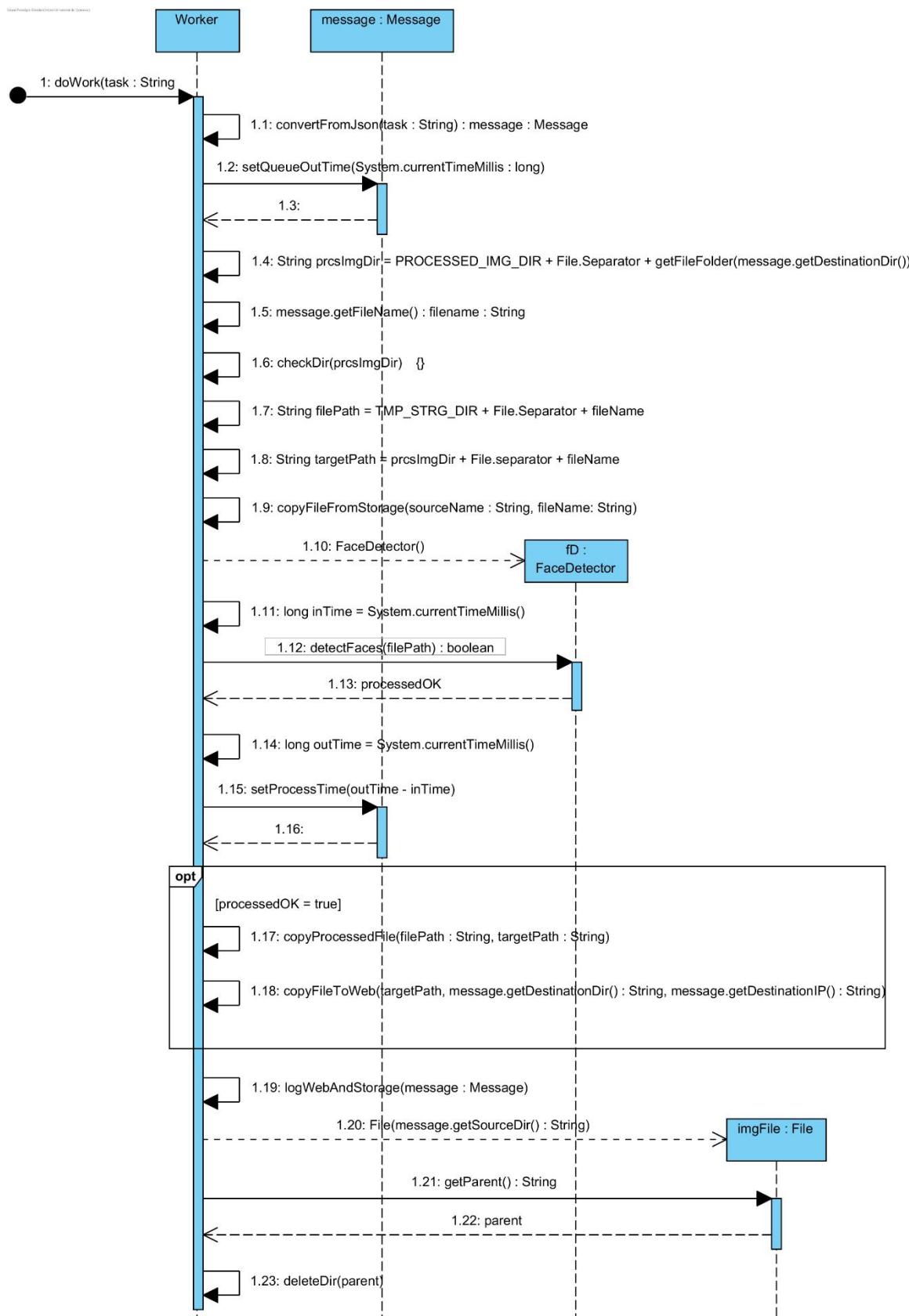


Ilustración 59 DIOS doWork

DS checkDir: Este Diagrama representa la funcionalidad de comprobar si el directorio donde almacenar las imágenes existe y en caso negativo crearlo.

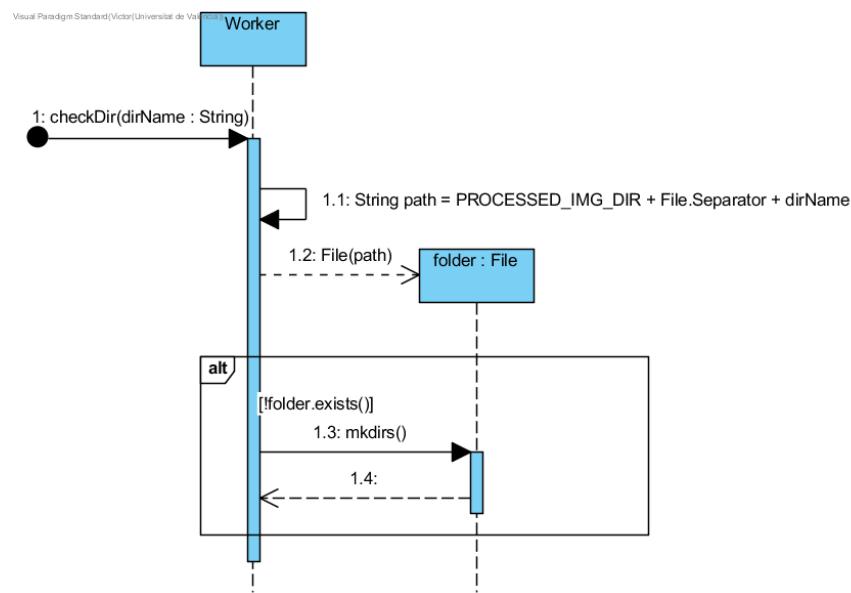


Ilustración 60 DIOS checkDir

DS detectFaces: Este Diagrama representa el proceso de detección facial mediante OpenCV.

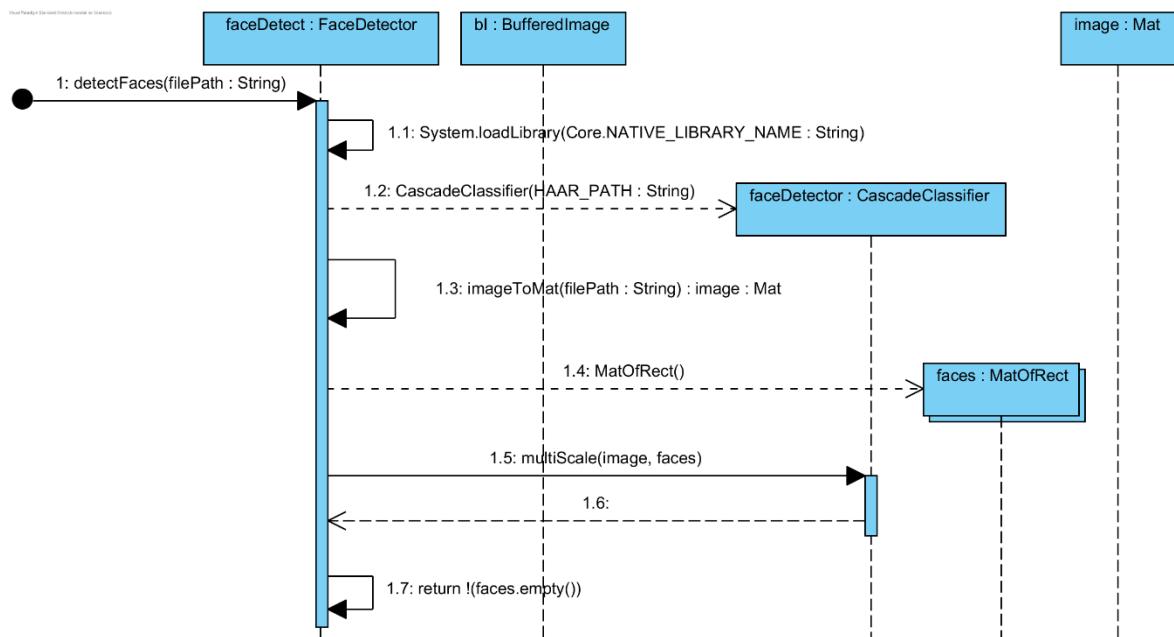


Ilustración 61 DIOS detectFaces

DS imageToMat: Este Diagrama representa la función para generar una matriz de rectángulos sobre una imagen.

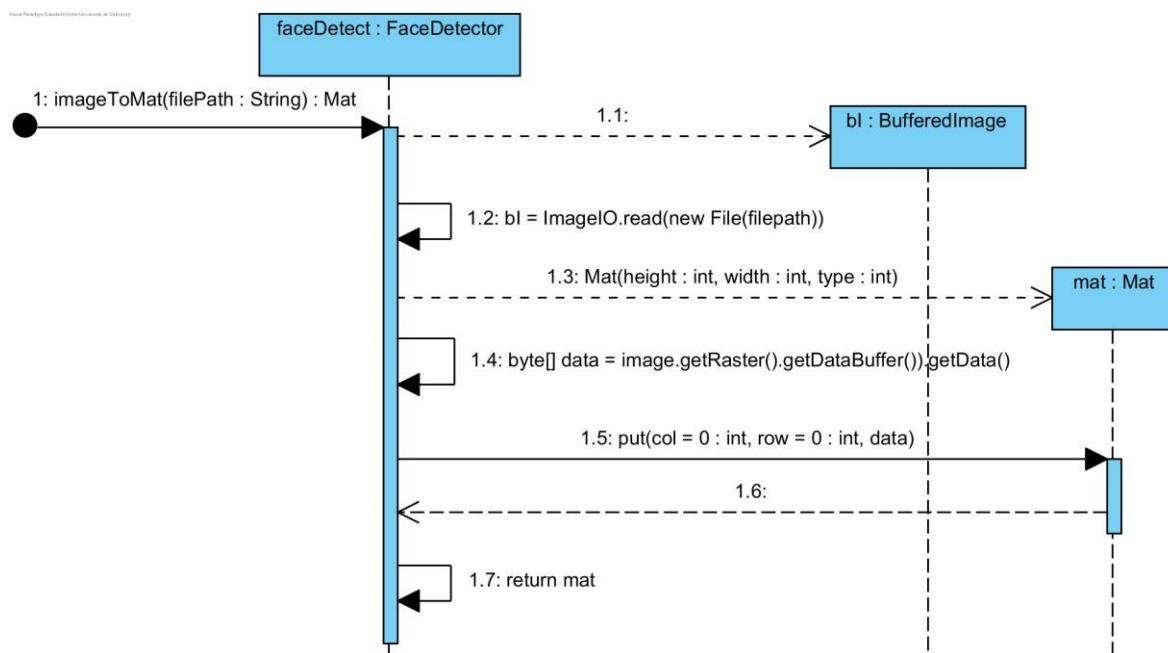


Ilustración 62 DIOS `imageToMat`

DS copyProcessedFile: Este Diagrama representa la copia o almacenamiento de una imagen que contiene una cara (detección positiva)

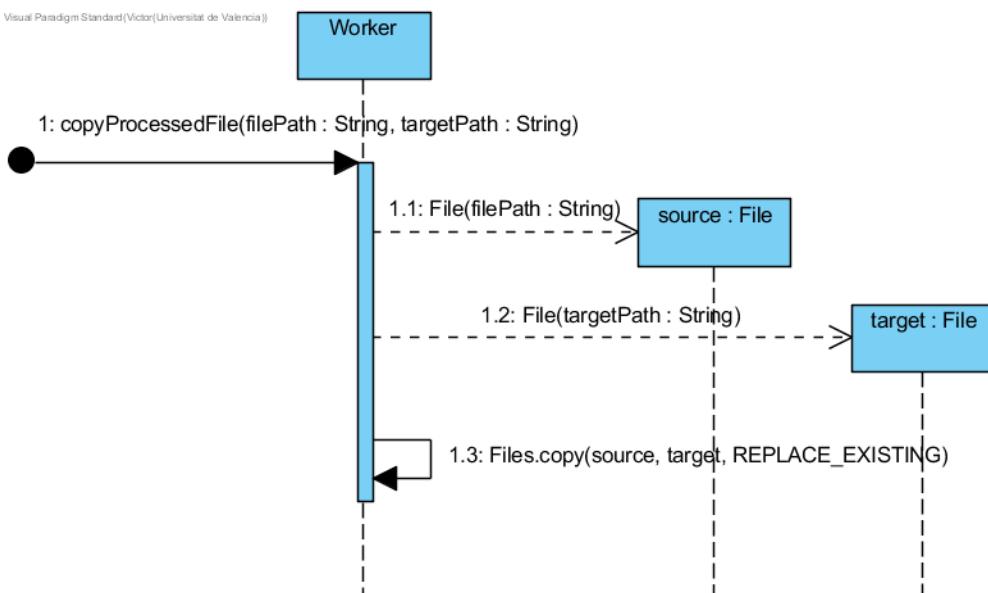


Ilustración 63 DIOS `copyProcessedFile`

DS copyFileToWeb:

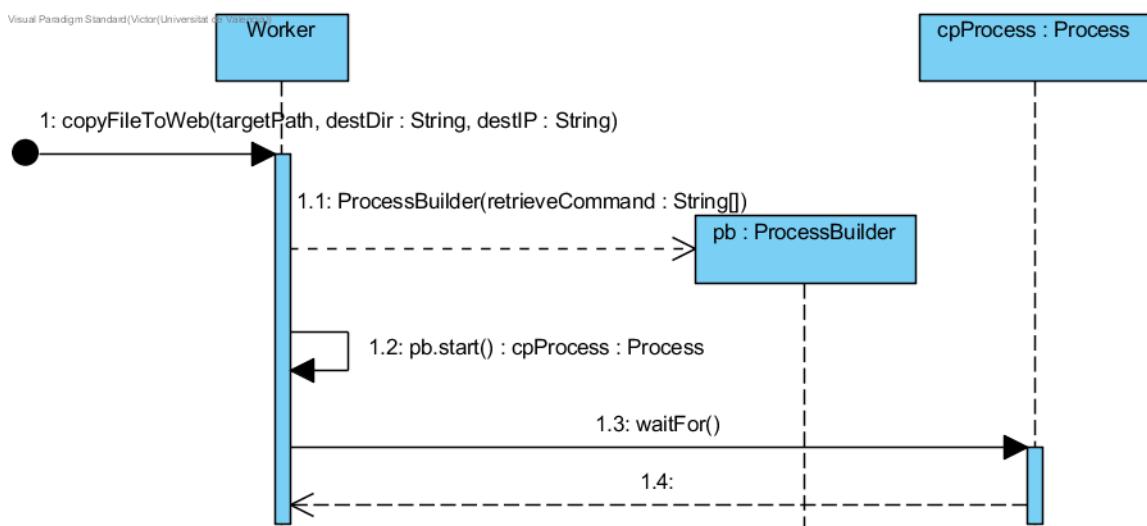


Ilustración 64 DIOS copyFileToWeb

DS logWebAndStorage: Este Diagrama representa el proceso de publicación en la cola de las estadísticas.

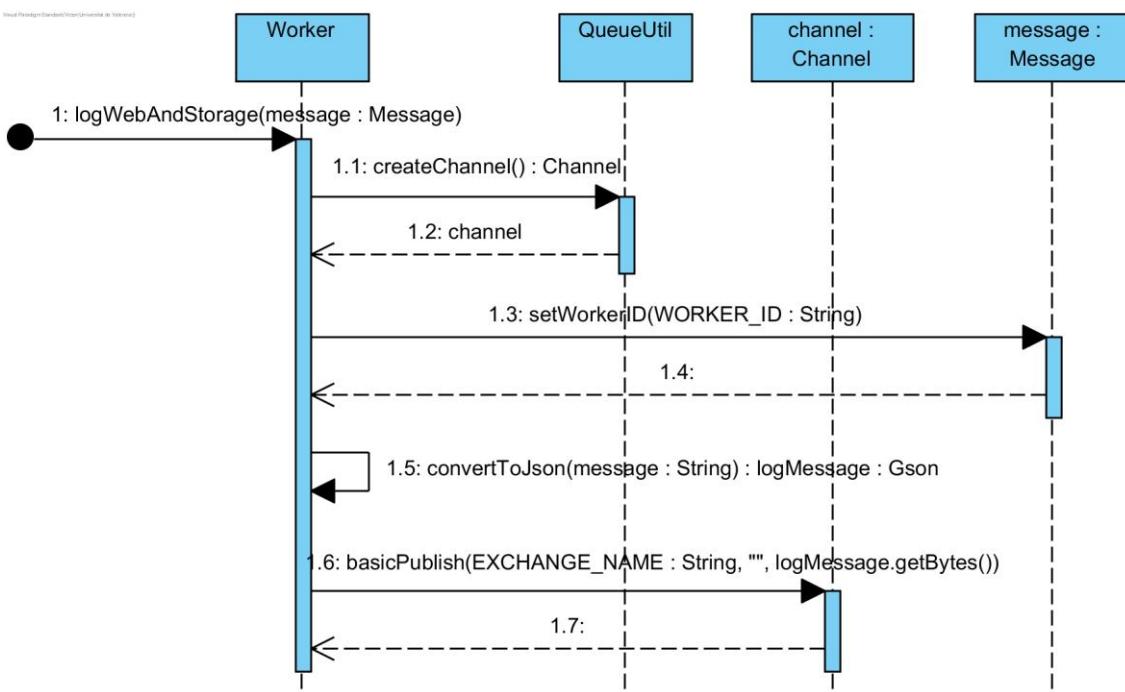


Ilustración 65 DIOS logWebAndStorage

DS deleteDir:

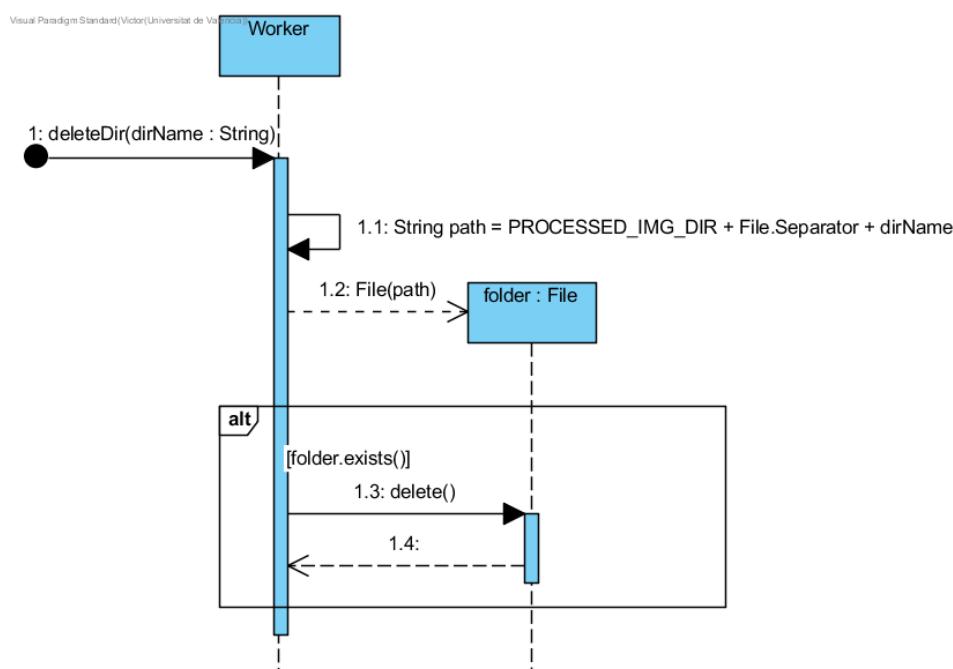


Ilustración 66 DIOS deleteDir

Capítulo 5. Implementación

Una vez analizado el problema y tomado las decisiones de diseño se abordará la implementación de la solución. Constará de dos fases: instalación del entorno de trabajo y codificación.

5.1. Instalación del entorno de trabajo

En este apartado se tratarán todas las instalaciones de software y configuraciones necesarias para el desarrollo del proyecto. El primer paso, es instalar *VirtualBox* en la máquina anfitriona (PC Windows) mediante el instalador que se puede encontrar en su página web <https://www.virtualbox.org/wiki/Downloads>

5.1.1. Instalación de las máquinas virtuales

El primer paso es instalar las máquinas virtuales necesarias. Entre ellas, como se ha explicado previamente, están: el servidor (Web), el *broker* de mensajes (*RabbitMQ*), el servidor de almacenamiento (*Storage*) y los *workers* (*Worker<N>*). Para ello se creará una máquina virtual a partir de una imagen *ISO* Ubuntu en *VirtualBox*.

Una vez arrancado *VirtualBox* se debe seleccionar la opción *Nueva* para crear una nueva máquina. Lo que abrirá una nueva ventana donde empezar la configuración. Se deberá elegir el tipo de sistema operativo, Linux y Ubuntu 64 bits en este caso, así como la memoria RAM que se dedicará a dicha máquina, en total 1024Mb. En la Ilustración 67 se puede observar la configuración.

A continuación, se deberá elegir el tipo de disco virtual que se desea utilizar, se ha elegido *VirtualBox Disk Image* ya que no se busca compatibilidad con otros *software* de virtualización y es el formato nativo de *VirtualBox* y el tamaño de este, 20Gb, Ilustración 68.

Una vez la máquina ha sido creada, es necesario iniciarla para indicarle la ubicación de la imagen ISO y comenzar con la instalación del sistema operativo. En la Ilustración 69 se puede observar la ventana de selección del archivo de instalación. Al hacer click en el botón Iniciar, comenzará la instalación guiada que culminará con una máquina funcional.

Cuando la instalación haya finalizado, con objetivo de no repetir este proceso seis veces se procederá a Clonar la primera máquina. En *VirtualBox* tan solo es necesario hacer click derecho en la máquina virtual y seleccionar la opción Clonar, que iniciará un asistente y producirá una nueva máquina idéntica a la recién creada.

Como último paso de configuración en *VirtualBox* resta configurar la virtualización de las interfaces para que las máquinas puedan comunicarse entre ellas. Para ello se debe abrir la configuración, acceder a la pestaña de Red cada máquina virtual y activar una

interfaz de red en modo Host-Only¹. En la Ilustración 70 se expone dicha ventana de configuración.

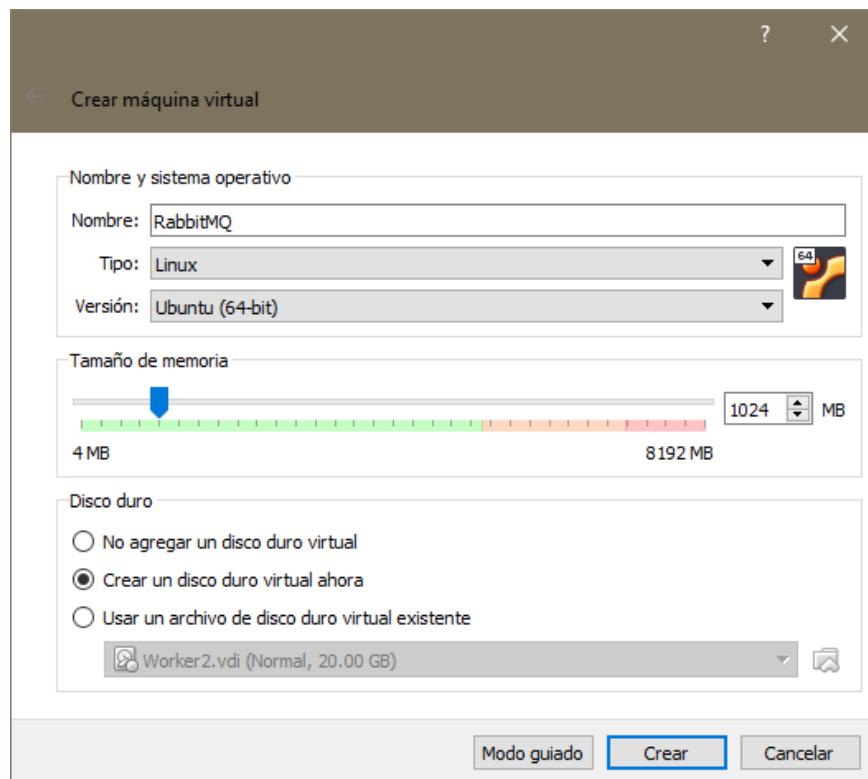


Ilustración 67 Configuración inicial máquina virtual en VirtualBox

¹ “El modo host-only provee una conexión de red entre las máquinas virtuales utilizando un adaptador de red virtual Ethernet visible para el sistema operativo anfitrión.” Fuente: virtualbox.org/manual

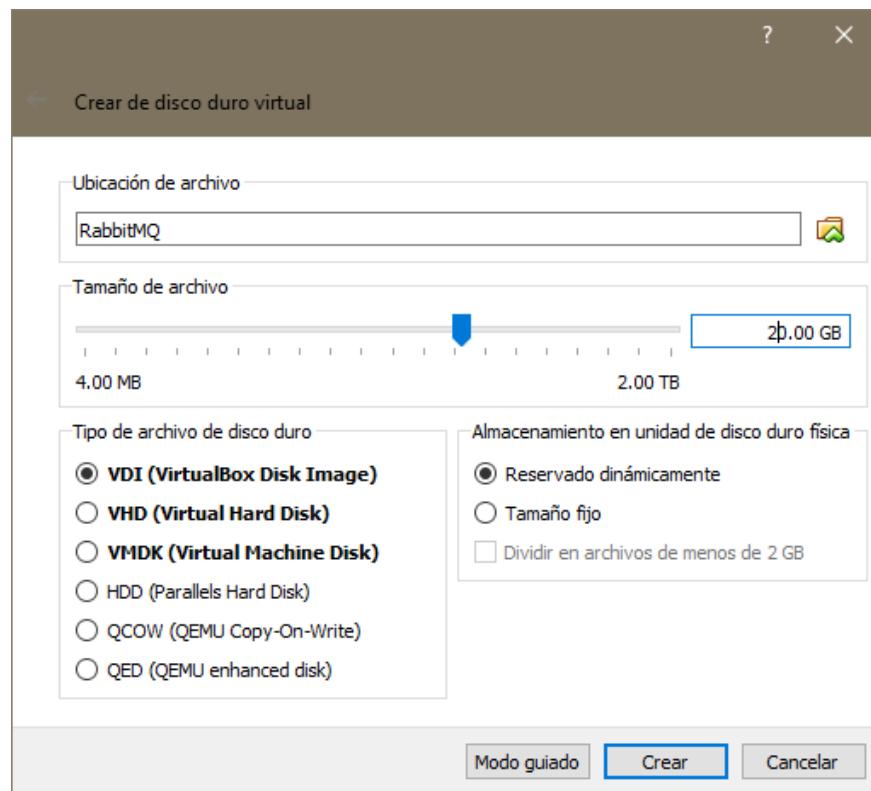


Ilustración 68 Configuración del disco virtual en VirtualBox

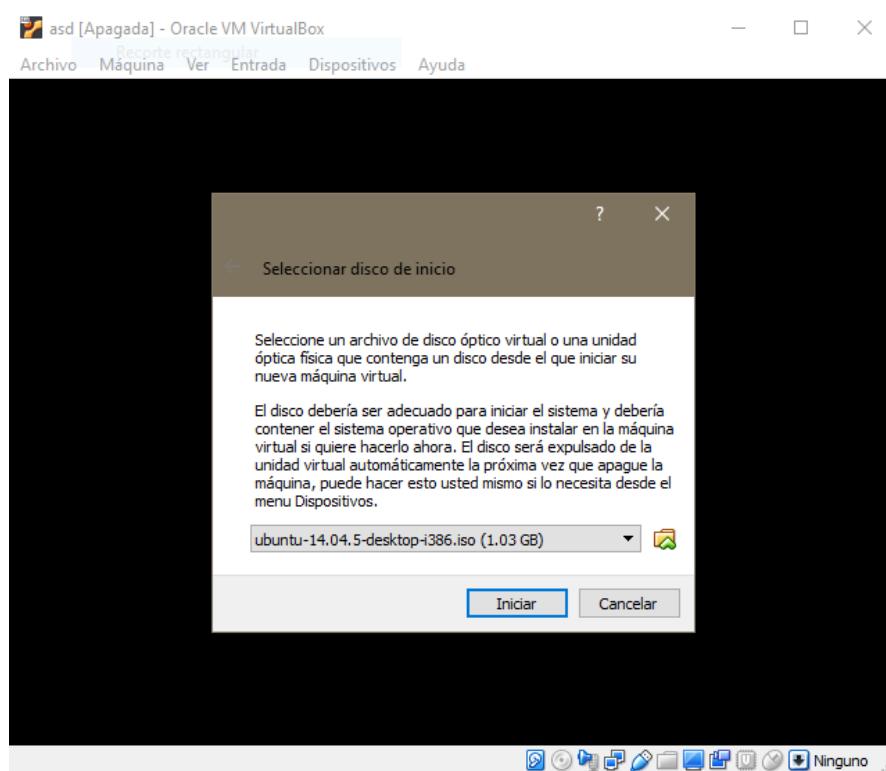


Ilustración 69 Selección de la imagen a montar en VirtualBox

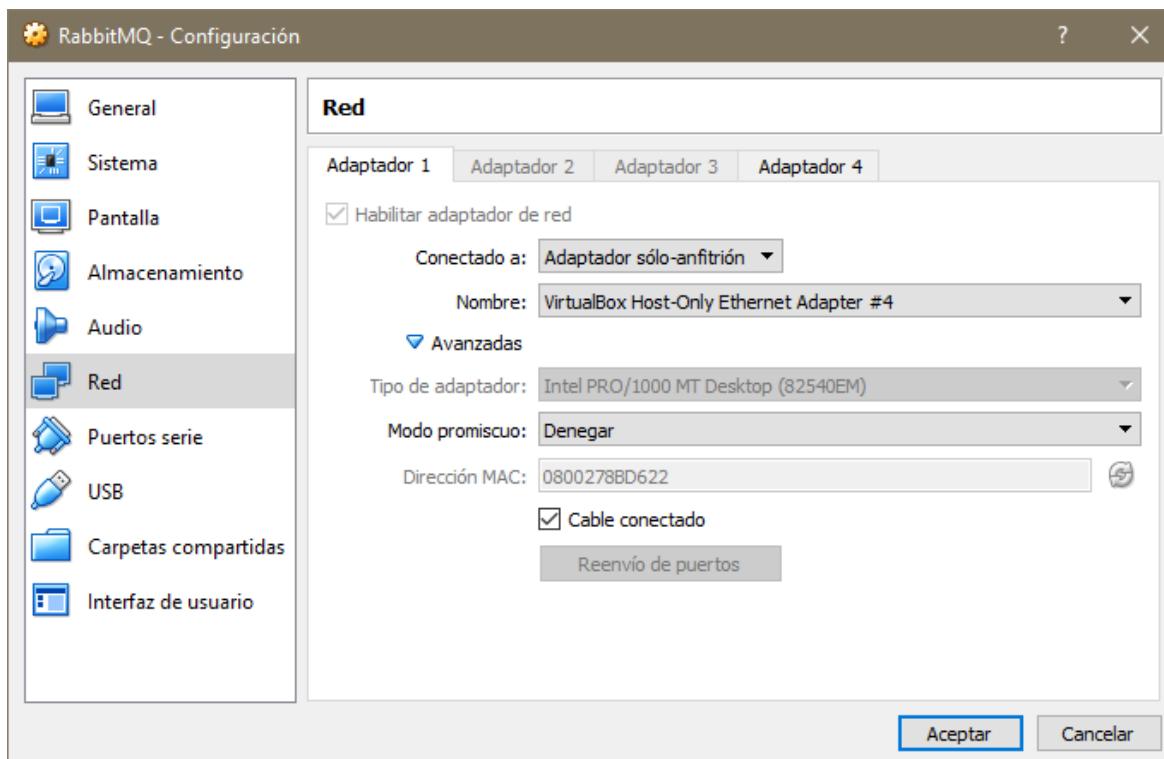


Ilustración 70 Configuración de red de una máquina en VirtualBox

5.1.2. Configuración de las interfaces de red

El primer paso consiste en determinar el nombre de cada host mediante la edición del fichero `/etc/hostname`. El siguiente comando revela la configuración para la máquina RabbitMQ en la que se establece el nombre siguiendo los valores de la Tabla 8 del apartado 4.1. Diseño del entorno distribuido.

```
root@rabbitmq:~# cat /etc/hostname
rabbitmq
```

Pudiendo identificar cada máquina según su nombre sólo queda configurar las interfaces de red mediante el fichero `/etc/network/interfaces` donde se especificarán los valores indicados en la Ilustración 29. En las siguientes líneas se presenta la configuración elegida:

```
root@rabbitmq:~# cat /etc/network/interfaces
# interfaces(5) file used by ifup(8) and ifdown(8)
```

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
    address 192.168.10.30
    netmask 255.255.255.0
    network 192.168.10.0
    broadcast 192.168.10.255
```

Aquí se ha configurado la interfaz *eth0* que corresponde a la interfaz configurada previamente en VirtualBox con asignación de IP estática y se han configurado el resto de valores conforme a lo explicado en el capítulo anterior.

Será necesario configurar todas las máquinas del mismo modo, pero con los valores correspondientes a cada una.

5.1.3. Instalación y configuración de software

Se procederá a la instalación de diferentes distribuciones de software y librerías. Cada máquina requiere programas específicos que se especifican a continuación.

Instalaciones comunes

Los siguientes programas se instalarán en todas las máquinas menos en RabbitMQ excepto el *JDK de Java* que sí se instalará.

Java JDK 8 Oracle distribution mediante los siguientes comandos:

```
sudo add-apt-repository ppa:webupd8team/java  
sudo apt-get update
```

para añadir el repositorio de *Oracle* a *APT* y actualizar el repositorio de paquetes. A continuación, sólo queda instalarlo:

```
sudo apt-get install oracle-java8-installer
```

Para asegurar el buen funcionamiento de todas las aplicaciones y servidores que utilicen *Java* se definirá la variable de entorno '*JAVA_HOME*'. Se debe acceder al fichero */etc/environment* y añadir la siguiente línea de modo que todo el sistema conozca dónde se halla la instalación del *JDK*.

```
JAVA_HOME="/usr/lib/jvm/java-8-oracle"
```

Netbeans IDE 8.2 mediante la descarga desde su página web

www.netbeans.org/downloads para la implementación del diseño en Java. Se elegirá la versión *Java EE* dado que incluye soporte para *HTML/CSS* y los servidores embebidos *GlassFish* y *Tomcat* que se utilizarán para lanzar el servidor. Este proceso se puede realizar antes del clonado para no repetirlo en cada máquina, aunque las características citadas sólo son necesarias en la máquina *Web*.

Open-ssh mediante los siguientes comandos:

```
sudo apt-get install openssh-client y  
sudo apt-get install openssh-server,
```

para permitir la copia de imágenes y ficheros entre las máquinas de manera segura y sin necesidad de identificarse para cada copia. Para ello es necesario generar también un par de claves Privada/Pública para cifrar y descifrar las transferencias. Para ello se utiliza el comando

```
ssh-keygen -t rsa -b 4096
```

especificando que se genere una clave de tipo RSA y con 4096 bits. Lo que arrancará un proceso en la consola para crear la clave pudiendo elegir el nombre y ruta del fichero donde se almacenará, y contraseña. A continuación, se presenta una porción de salida de la consola de dicho proceso.

```
victor@worker1:~$ ssh-keygen -t rsa -b 4096  
Generating public/private rsa key pair.  
Enter file in which to save the key  
(/home/victor/.ssh/id_rsa):  
/home/victor/.ssh/id_rsa already exists.  
Enter passphrase (empty for no passphrase):
```

Your identification has been saved in
/home/victor/.ssh/id_rsa.
Your public key has been saved in
/home/victor/.ssh/id_rsa.pub.
The key fingerprint is:
9b:1a:38:0a:6a:ad:cf:d3:72:bf:de:d8:2e:53:62:23
victor@worker1
The key's randomart image is:

A continuación, una vez disponemos de las claves en cada máquina, es necesario copiar la clave pública de la máquina que quiera establecer una conexión SSH (ubicada en el archivo indicado durante la creación, generalmente en `~/.ssh/id_rsa.pub`) en el fichero de hosts autorizados de la máquina que recibe la petición (este fichero se encuentra en la misma ubicación `~/.ssh/authorized_keys`).

Así, como la máquina *Web* necesita copiar imágenes hacia la máquina *Storage* se deberá copiar la clave pública de la primera a la lista de claves autorizadas de la segunda. Se realizará lo mismo a la inversa, entre los *Workers* y *Storage* en ambos sentidos y finalmente en *Web* será necesario autorizar la clave de los *Workers* para que puedan transmitir los resultados del procesamiento.

RabbitMQ

Esta máquina sólo actuará como broker, por lo que únicamente se instalará el servidor *rabbitmq* para que actúe como tal. La instalación requiere varios pasos.

Instalar *Erlang* en primer lugar como requisito indispensable para el funcionamiento de *rabbitmq* dado que es el lenguaje que utiliza para codificar y gestionar las colas. Los siguientes comandos permiten su instalación:

```
wget https://packages.erlang-solutions.com/erlang-solutions_1.0_all.deb  
sudo dpkg -i erlang-solutions_1.0_all.deb
```

El siguiente paso es añadir el repositorio *PPA* de *rabbitmq* a *APT* de la siguiente manera:

```
echo 'deb http://www.rabbitmq.com/debian/ testing main' | sudo tee /etc/apt/sources.list.d/rabbitmq.list
```

Añadir también su clave pública a la lista de hosts autorizados:

```
wget -O- https://www.rabbitmq.com/rabbitmq-release-signing-key.asc | sudo apt-key add -
```

Y finalmente actualizar los repositorios e instalar *rabbitmq-server*:

```
sudo apt-get update  
sudo apt-get install rabbitmq-server
```

Ahora se puede habilitar y lanzar el servicio mediante los siguientes comandos:

```
sudo systemctl enable rabbitmq-server  
sudo systemctl start rabbitmq-server
```

Pese a que el servidor *rabbitmq* ofrece un usuario por defecto (usuario: “*guest*”, contraseña: “*guest*”) es preferible configurar un usuario propio y concederle todos los permisos para poder controlar y monitorizar las pruebas sin trabas. Así:

```
sudo rabbitmqctl add_user admin admin
sudo rabbitmqctl set_user_tags admin administrator
sudo rabbitmqctl set_permissions -p / admin ".*" ".*" ".*"
```

Por último y como utilidad para mejorar la observación y depurar las pruebas, se procederá a habilitar el plugin *web management* que permite visualizar, editar y controlar las colas en tiempo real. Con el siguiente comando se activa dicha consola:

```
sudo rabbitmq-plugins enable rabbitmq_management
```

Ahora es posible acceder mediante la *URL* <http://<192.168.10.30/localhost>:15672/api/> a la interfaz, identificándose con el usuario y contraseña creado previamente. En la Ilustración 71 se puede ver la interfaz en su sección de colas con una cola llamada *task_queue* que en ese momento se encontraba ociosa.

Workers

Los workers necesitan de *OpenCV* para procesar las imágenes. Esta librería está basada en el lenguaje de programación *C* y es necesario instalar diferentes programas para poder extraer el código fuente y exponer su interfaz *Java*.

Primero es necesario instalar *g++* para permitir compilar y ejecutar el código fuente, *cmake* para construir y empaquetar software y *ant* con la misma función, pero desarrollado para *Java*:

```
sudo apt-get install g++
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:george-edison55/cmake-3.x
sudo apt-get update
sudo apt-get install cmake
sudo apt-get install ant
```

Segundo se descargará y extraerá *OpenCV*, donde x.x.x indica la versión:

```
get https://github.com/opencv/opencv/archive/x.x.x.zip -O
opencv-x.x.x.zip
unzip opencv-x.x.x.zip
```

Name	Exclusive	Parameters	Policy	Status	Ready	Unacked	Total	incoming	deliver / get	ack
task_queue	D			idle	4	0	4			

Ilustración 71 Interfaz del plugin RabbitMQ Web Management

Y accediendo a la carpeta extraída, se creará un nuevo repertorio con nombre *build/* desde el cuál se ejecutará *CMake* para que compile las librerías y gracias a *Ant* crear un paquete *.so* que será más tarde importado en los proyectos de codificación.

```
cmake -D BUILD_SHARED_LIBS=OFF ..
make -j4
```

Por último, para instalar las librerías en el sistema se debe ejecutar el siguiente comando desde la misma carpeta de extracción.

```
sudo make install
```

Storage

En la máquina de almacenamiento será necesario instalar la librería *avconv* incluida en el paquete *libav-tools* y que utiliza herramientas incluidas en el paquete *ffmpeg*. Es necesaria para poder dividir el vídeo, recibido por el servidor *web*, en frames y así poder crear un mensaje para cada uno. Con el instalador de paquetes de *Ubuntu* se puede instalar con el siguiente comando:

```
sudo apt-get install libav-tools ffmpeg
```

5.2. Programación

En este apartado se estudiará la programación del código más relevante y que representa la verdadera esencia de lo diseñado previamente. Otras partes relevantes del código pueden consultarse en el Anexo Código fuente.

5.2.1. Programación de *scripts*

Para la división del vídeo se ha elegido la opción de implementarla mediante un *script* en *bash* dado que se realiza con una única línea y programarlo en *Java* conllevaría mucho más trabajo para obtener el mismo resultado. A continuación se presenta el código.

```
#!/bin/bash
# Parameters
FILE=$1 # Name of the file
SOURCE_IP=$2 # Source IP that be inserted in the message
DEST_IP=$3 # Destination IP that be inserted in the message
SOURCE_DIR=$4 # Directory containing the video source
DEST_DIR=$(dirname $FILE) # Directory in which store the frames
NAME=$(basename $FILE) # Name of the video file
FRAME_RATE=10 # Frame Rate to divide the Video

# Classpath from Libraries and project
CLASSPATH=/home/victor/libraries/amqp-client-
5.0.0.jar:/home/victor/libraries/slf4j-api-1.8.0-
alpha2.jar:/home/victor/libraries/slf4j-simple-1.8.0-
alpha2.jar:/home/victor/libraries/gson-
2.8.2.jar:/home/victor/libraries/commons-io-
2.4.jar:/home/victor/Proyectos/VideoProcessor/build/classes

avconv -i $SOURCE_DIR/$NAME -r $FRAME_RATE -f image2
$SOURCE_DIR/%05d.png
# Remove the video file
rm $SOURCE_DIR/*.mp4

java -cp $CLASSPATH app.Tasker $SOURCE_DIR $SOURCE_IP $DEST_IP
$DEST_DIR $FRAME_RATE
```

Cómo se puede ver, el script espera una serie de parámetros en su ejecución que determinan las opciones de división y de envío. Después se especifica el *CLASSPATH* donde se encuentran las librerías necesarias para la ejecución del *Tasker*, clase encargada en crear las tareas o mensajes y encolarlos en el servidor *RabbitMQ*. A continuación, se ejecuta el programa *avconv* para dividir el vídeo y cuyos parámetros son:

- *-i*: archivo origen
- *-r*: frame rate

- *image 2* : especifica a *avconv* que se trata de una conversión de imagen a vídeo o viceversa dependiendo de si el valor que recibe es un vídeo o una imagen.

El valor que recibe *-f image2* es una imagen por lo tanto se trata de una conversión de vídeo a imagen. Se guardará en un directorio y la expresión *%05d* ofrece la posibilidad de guardar cada imagen con un nombre de 5 dígitos.

Una vez la ejecución de *avconv* ha finalizado, se ejecuta la clase *Java* encargada de crear las tareas y que recibe como parámetros los datos de las imágenes para poder hallarlas en el sistema, identificarlas y enviarlas.

Además, para automatizar la puesta en marcha del sistema distribuido, se crean scripts para lanzar automáticamente las clases *Java* encargadas de gestionar la información. Se ejecutan mediante la opción *-cp* del comando *java*, o ejecutando directamente el archivo *.jar* en el caso de los *Workers*:

```
java -jar worker.jar -Djava.library.path="src/app/dll" >  
/home/victor/logs/worker_log.txt
```

En el que mediante la opción *-Djava.library.path* se especifica la ruta donde se encuentra la librería de *OpenCV* generada como explicado en el apartado 5.1.3. Instalación y configuración de software, y se redirige la salida a un fichero de texto.

Estos *scripts* se lanzarán al inicio del sistema añadiéndolos al directorio */etc/init.d* y dándoles permisos de ejecución. Basta con actualizar los enlaces de *scripts* de inicialización mediante la siguiente orden:

```
sudo update-rc.d script.sh defaults 98 02
```

donde mediante *defaults* se elige la configuración por defecto y mediante los valores *98 02* que presuntamente el servicio no será requerido por ningún otro. Esto se realizará para las máquinas *Web*, *Storage* y *Worker* para que lancen el servidor, inicien la recepción de mensajes en *Storage* y lancen la aplicación de detección en cada *worker*.

Por último, se ha implementado varios pequeños *scripts* en el lenguaje *M*. Propio a *Matlab* pero desarrollado mediante el *IDE* de código abierto *Octave*.

Como ejemplo se presenta uno de ellos, el de mayor complejidad. Se encarga de leer las estadísticas de procesado almacenadas en sus respectivos repertorios. En estos se almacena, como ya se ha explicado, los tiempos de procesado y de espera de cada imagen. Primero se calcula el tiempo medio de espera hallando la media de las imágenes para cada trabajo y finalmente obteniendo la media de los trabajos. Y más tarde se obtiene el tiempo de respuesta de cada trabajo. A continuación, se presenta el *script*:

```

# Variables
path = '/home/victor/apache-tomcat-8.0.27/webapps/data/';
stats_file = "/statistics.txt";

## Read file paths
paths = readdir(path);
# Remove linux . and .. folder
paths(2) = [];
paths(1) = [];
# Set number of works equal to the number of folders
works = size(paths)(1);

files = paths;
for i=1:works
    # Map statistics file in each folder
    files(i, 1) = [path paths{i} stats_file];
endfor

## Read statistics file bytes
data = files;
for i=1:works
    # Divide each of the four values separated by semicolon (;)
    data(i, 1) = dlmread(files{i}, ';');
endfor

## Calculate statistics
# Waiting time
waiting_time = files;
mean_wait_time = 1:works;
for i=1:works
    # For every line in the file, read the fourth parameter
    # and convert to seconds
    waiting_time(i) = data(i, 1){1}( :, 4)/1000;
endfor
  
```

Esto produciría una salida con el tiempo de espera para un número de trabajos y un número de *workers*. Si por ejemplo se han obtenido los datos para uno, dos y tres *workers*, sólo quedaría generar un gráfico mediante el siguiente código:

```

meanMeanW = [meanOneWorker meanTwoWorkers meanThreeWorkers]

figure(1)
h = bar(meanMeanW);
grid on;
title("Mean Waiting time by number of workers and 3 jobs")
xlabel("Number of workers (workers)")
ylabel("Mean waiting time of the process (s)")
  
```

Por último, y sólo para automatizar la subida de archivos de modo que las pruebas y resultados se asemejen lo máximo posible a un entorno real donde diferentes peticiones llegan al mismo tiempo, se programará un *script* en *bash* utilizando el comando *curl* que permita ejecutar una orden HTTP *POST* para dicha subida. Así el código necesario es el siguiente:

```
#!/bin/bash
for i in {1..5}
do
    curl -X POST -H "Content-Type: multipart/form-data" -F
    "data=@test.mp4" http://localhost:8080/UploadServlet/
done
```

Así se repetirá la subida de un archivo en bucle el número de veces deseado. Con respecto a la orden *curl*: la opción *-X* especifica el método *HTTP* que se utilizará, la opción *-H* se utiliza para insertar las cabeceras y *-F* para añadir un archivo a la petición.

5.2.2. Programación de la interfaz de usuario

Para permitir la subida de vídeos y observación del resultado, se programará una interfaz web simple mediante el uso de *HTML5*, *CSS3 Bootstrap* para la presentación y *JSP* para la creación dinámica de elementos. Lo que cabe destacar en este punto, es cómo se generan dinámicamente contenedores con las imágenes procesadas. En el siguiente bloque de código se ha resaltado en negrita las porciones del código que corresponden a líneas *JSP*. Las cuales crean el contenido recogiendo de la petición *HTTP* la lista con toda la información de las imágenes mediante la función *getAttribute(String attribute)* de la clase *HttpServletRequest* que es un objeto serializado a *JSON*. Así, una vez se tiene la lista de las imágenes sólo es necesario iterar sobre ellas creando un contenedor *<div>* con un atributo ** para cada una.

Las clases *ImageListBean* y *ImageDataBean* son *Beans* o *DTO (Data Transfer Object)* que representan la información de la imagen, como ya se ha visto en el Capítulo 4 Diseño de la Solución.

```
<div class='fluid-container'>
    <div class="row">
        <div class="title text-center">Processed Images</div>
    </div>
    <% ImageListBean ilB =
    (ImageListBean) request.getAttribute("images");
    List<ImageDataBean> iList = ilB.getImages();%>
```

```

    int cont = 0;
    for (ImageDataBean img : iList) {
%>
    <div class="img-thumbnail caption col-md-3 text-center
image-container">
        " />
        <div class='caption-time'>
            <span class='glyphicon glyphicon-time' style='color:
gainsboro; margin-right: 4px;'></span>
            <span class="caption text-center<%
img.getFrameTime()%></span>
        </div>
    </div>
<% } %>
</div>
  
```

Por otro lado, con respecto a la presentación de contenidos, las clases añadidas a las etiquetas *HTML* pertenecen a la librería *Bootstrap* y son utilizadas para estructurar de manera sencilla y estable los resultados.

5.2.3. Programación de las funcionalidades

En este apartado sólo se pretende explicar las partes cuya implementación ha supuesto un mayor grado de dificultad o investigación, o simplemente son interesantes para la comprensión del funcionamiento.

Se explicará las funcionalidades clave intentando seguir el orden del flujo de información; aunque en algunos momentos se omitirán, por irrelevancia o repetición, algunos puntos del desarrollo.

Dadas las similitudes entre las clases consumidoras y productoras presentes en casi todas las máquinas, se expondrá una explicación general sobre su funcionalidad. Para declarar cualquier actor de la comunicación, es necesario declarar un canal de conexión mediante el cual se enviarán o recibirán los mensajes. Especificando un *host* de destino y el puerto en el que escucha el servidor *RabbitMQ* como ya se ha explicado en el Capítulo 2 Estado del arte.

Una vez se dispone del canal, es posible declarar la cola en la que se quiere producir o de la que se quiere consumir. Sin embargo, dónde reside el potencial de *RabbitMQ* es en esto exactamente: no es necesario indicar en qué cola se va a producir el mensaje, sino que, mediante un *exchange* o medio de intercambio, se libera a los productores de

tener que conocer la lógica detrás del *broker*. De este modo, el servidor *RabbitMQ* declarará y relacionará a su inicio colas y *exchanges*:

```
channel.exchangeDeclare(EXCHANGE_NAME,  
BuiltinExchangeType.FANOUT);  
String queueName = channel.queueDeclare().getQueue();  
channel.queueBind(queueName, EXCHANGE_NAME, ROUTING_KEY);
```

Por último, para encolar o consumir un mensaje se realiza mediante las funciones *basicConsume()* o *basicProduce()* ofrecidas por la librería *amqp*. En este caso para consumir se utiliza la siguiente llamada:

```
channel.basicConsume(QUEUE_NAME, true, consumer);
```

donde se especifica: el nombre de la cola, que se desea acuse de recibo o *acknowledge* y el consumidor (objeto que queda a la espera de recibir mensajes y consumirlos) cuya implementación se explicará más adelante en el apartado de los *workers*. Y para producir:

```
channel.basicPublish(EXCHANGE, ROUTING_KEY,  
MessageProperties.PERSISTENT_TEXT_PLAIN,  
message.getBytes());
```

donde se puede observar como no se especifica ninguna cola, sino únicamente el *exchange* y la *routing_key* que identifican el tipo de intercambio. También recibe como parámetros el tipo de contenido del mensaje y el mensaje en sí que es una serialización a *JSON* de un objeto de la clase diseñada *Message* -un *POJO* que sigue la estructura mencionada en el apartado 4.4. Diseño de la comunicación.

Web

El *Servlet*, a través de la petición del usuario (mensaje (2) de la Ilustración 23), es el encargado de iniciar todo el proceso. Aunque su implementación no conlleva mayor dificultad que la de cualquier otro *Servlet*, se explicará cómo comienza el flujo de los datos.

Mediante un formulario *HTML* que genera una petición *POST* con contenido *multipart/form-data* se recoge el vídeo gracias a la clase *ServletFileUpload* provista en el paquete *http* de *Apache*. Se almacena en el disco en un directorio con nombre *UUID* y por lo tanto único. Finalmente, se lanza el procesado mediante la ejecución de la clase *ProcessVideo* que extiende a *Runnable*. Permitiendo la ejecución de cada subida en un hilo diferente, para no bloquear el servidor con cada petición. En la siguiente porción de código se puede observar dicha llamada.

```
ProcessVideo cp = new ProcessVideo(filePath);
Thread thread = new Thread(cp);
thread.start();
```

El constructor de *ProcessVideo* recibe el camino absoluto hasta el archivo de vídeo a procesar, y se ejecuta en un hilo nuevo.

ProcessVideo cuenta con dos parámetros que contienen los comandos para copiar a la máquina *Storage* un archivo mediante *scp* y para ejecutar también en *Storage* el *script* que divide el vídeo y encola las imágenes. Dichos comandos se construyen en un *String* y se ejecutan en un *ProcessBuilder* para ser interpretados como comandos *bash*. A continuación, se escriben los comandos y la ejecución de éstos, dónde el primero es el encargado de la copia, mediante una conexión *ssh* para la creación del directorio (que ha de ser único) y asegurar su existencia, y otra *scp* que utiliza el mismo protocolo para transferir archivos. Y el segundo, también una conexión *ssh* para ejecutar de forma remota el *script* explicado en la sección 5.2.1.

```
copyCommand = new String[]{
    "/bin/bash",
    "-c",
    "ssh " + remoteHost + " mkdir -p " + destDirectory +
    " && " +
    "scp -i " + publicRSAKeyFile + " " + filePath + " " +
    remoteHost + ":" + destDirectory
};

processCommand = new String[]{
    "/bin/bash",
    "-c",
    "ssh " + remoteHost + " sh " + script + " " + filePath +
    " " + sourceIP + " " + destIP + " " + destDirectory,
};
```

Debido a que la clase implementa a *Runnable* es necesario especificar el método *run()* gracias al cual se permite su ejecución mediante la clase *Thread*. Una implementación simple que ejecutara los comandos consistiría en instanciar un objeto de la clase *ProcessBuilder* pasando como parámetro a su constructor el comando anterior. Seguido, ejecutar el comando de copiado, mantener el proceso a la espera de finalización y por último ejecutar el *script* en *Storage*.

A continuación, se puede ver dicha implementación. Deberían tratarse las posibles excepciones de la ejecución de la función *start()* pero por simplicidad aquí no se ha hecho.

```

public void run() {

    ProcessBuilder copyPb = new ProcessBuilder(copyCommand);
    ProcessBuilder processPb = new
    ProcessBuilder(processCommand);

    Process copyProcess = copyPb.start();
    // Thread waits until the copy is done
    copyProcess.waitFor();
    Process processProcess = processPb.start();
}

```

Por otro lado, la máquina *Web* ha de quedar a la espera y consumir mensajes de la cola de procesamiento una vez los *Workers* han empezado a producir mensajes. De ello se encarga la clase *ReceiveLog* y se trata de un consumidor cuya implementación requiere de la instanciación de la interfaz *Consumer*. Así, se sobrescribe el método *handleDelivery* para ejecutar el código específico a la aplicación y tratar los mensajes. En este caso se recibe el mensaje en el cuerpo o *body*, se *deserializa* el objeto desde el formato *JSON* y se generan las estadísticas de procesado. En el siguiente bloque de código se puede ver dicha función.

```

Consumer consumer = new DefaultConsumer(channel)
    public void handleDelivery(String consumerTag, Envelope
envelope,
        AMQP.BasicProperties properties, byte[] body) throws
        IOException
    {
        String message = new String(body, "UTF-8");
        System.out.println(" [x] Received '" + message + "'");

        Message imgMsg = convertFromJson(message);
        generateStatistics(imgMsg);
    }
};

```

Este mismo consumidor es el que recibe la llamada a *basicConsume()* explicada anteriormente:

```
channel.basicConsume(QUEUE_NAME, true, consumer);
```

Storage

Esta máquina encola los mensajes a partir de las imágenes. Para ello utiliza una clase que se encarga, según lo explicado al principio de la sección, debe crear un mensaje con la información de una imagen y añadirlo a la cola. Esta clase, *Tasker*, es la ejecutada en el script *processVideo.sh*.

También se encarga de calcular el tiempo total de procesado de una tarea recogiendo del *exchange* adecuado los mensajes con sus estadísticas. La implementación de la clase es prácticamente la misma excepto por el método *generateStatistics()* cuyo código difiere en cómo se calculan las estadísticas. Su código completo y el de la clase *Tasker* se pueden consultar en el Anexo.

Worker

El *worker* procesará una imagen por cada mensaje que reciba. Por lo que editando el código de recepción de mensajes *handleDelivery()* se trata cada mensaje para recoger la imagen, procesarla para la detección facial, calcular el tiempo de procesado y en caso de detección enviarla al servidor web.

El código que ejecuta se explica a continuación. Primero una vez se tiene el mensaje se extrae el directorio de la máquina *Storage* dónde está almacenada la imagen, luego se copia a un directorio temporal y finalmente se procesa. El proceso de copiado se ejecuta mediante un *ProcessBuilder* como explicado en el apartado anterior *Web*.

```
copyFileFromStorage(message.getSourceDir(),  
message.getFileName());  
  
FaceDetector fd = new FaceDetector();  
boolean faceDetected;  
  
long inTime = System.currentTimeMillis();  
faceDetected = fd.detectFaces(filePath);  
long outTime = System.currentTimeMillis();  
  
message.setProcessTime(outTime - inTime);  
message.setWorkerFile(targetPath);
```

Una vez se ha procesado la imagen y establecido las estadísticas del mensaje se comprueba si la variable *faceDetected* es verdadera, lo que implica que ha detectado una cara, y copiar la imagen al servidor *Web*.

El código detrás de la función *detectFaces()* es un código tipo para la detección facial y que no requiere mucha explicación. Únicamente que, se genera una matriz de rectángulos sobre la imagen a partir de los *bytes* de ésta, mediante un objeto de la clase *Mat* de *OpenCV*, y a partir de esta y de la función *detectMultiscale()* de *CascadeClasifier* se obtiene una lista de todos los rectángulos en los que se ha hallado una cara. El clasificador en cascada es una clase que provee funciones que permiten la detección de objetos mediante el algoritmo *Viola-Jones* y las cascadas de *Haar*. Su código completo y el del resto de clases de los *Workers* se puede encontrar en el Anexo.



UNIVERSITAT
POLÈTICA
DE VALÈNCIA

[] Escola Tècnica
Superior d'Enginyeria

Capítulo 6. Experimentos y Resultados

Como culminación del trabajo, se mostrará primero el sistema en funcionamiento para más tarde analizar los resultados obtenidos.

Se realizarán pruebas con diferentes escenarios para analizar la efectividad del sistema, con diferentes videos y de diferentes tamaños para poder también llevar a cabo pruebas de carga.

6.1. Descripción de los experimentos

Para poder estudiar el sistema en profundidad y desde los ángulos deseados se definen diferentes tipos de experimentos. En un entorno como el desarrollado, lo interesante es estudiar cómo se comporta el sistema de colas de mensaje y cómo trabaja *OpenCV*.

Experimento 1:

Para analizar los tiempos de respuesta del sistema y de espera para un mensaje, se lanzarán un número definido de tareas constante, pero variando el número de *workers* que intervienen en el proceso.

Lanzando así tres trabajos para uno, dos y tres *workers* se genera un conjunto de datos que, mediante los *scripts* de *R*, se agrupan y analizan. Estos datos son: el tiempo de respuesta general del sistema, el tiempo de espera medio en la cola para una tarea y el tiempo de procesado de cada imagen.

Se obtiene una aproximación del tiempo de respuesta total del sistema sustrayendo el tiempo en que se detecta el final del trabajo al tiempo en el momento en que se recibe un video.

$$t_{response} = t_{endjob} - t_{videoin}$$

El tiempo de espera en la cola de cada mensaje se ha obtenido sustrayendo el tiempo en que este es consumido por un *worker*, al tiempo en que dicho mensaje se ha publicado en la cola:

$$t_{waiting} = t_{qout} - t_{qin}$$

Experimento 2:

Por otro lado, para estudiar el funcionamiento de la librería de procesamiento de imágenes, el número de *workers* o el número de tareas son independientes, por lo que se lanza una tarea con un número definido nodos.

Así se recoge el tiempo de procesado de cada mensaje individualmente. Este tiempo es la diferencia entre el tiempo proporciona por el Reloj en Tiempo Real, o *RTC* por sus siglas en inglés, de la máquina virtual antes de ser procesado por *OpenCV*, es decir al ser consumido el mensaje, y justo al terminar:

$$t_{process} = t_{consumed} - t_{finished}$$

Experimento 3:

Para probar el funcionamiento de *OpenCV* de manera controlada, se genera un vídeo en el que se conoce en qué *frame* hay una cara y en cuáles no. Así se elige una imagen de internet en la que aparece un rostro y se integra con otras en las que no hay mediante la herramienta *avconv*, de manera similar a cómo se hizo para dividirlo. El resultado sería un vídeo de 120 segundos en el que en los segundos cincuenta y setenta y cinco contendrán un rostro.

6.2. Ejecución de las pruebas

Primero se llevará a cabo una prueba para comparar la eficiencia del sistema con un número variable de *workers*. Se enviará el mismo vídeo tres veces con uno, dos y tres *workers* en funcionamiento.

Así, con un número de 3 trabajos simultáneos, se utiliza el script especificado previamente para la subida de los archivos. Mediante el comando siguiente se lanza la ejecución:

```
./uploadVideo.sh
```

Lo que ejecutará la orden *CURL* y devolverá la página *HTML* de respuesta donde se encuentra la *URL* para recuperar el resultado.

Para realizar un seguimiento del proceso, se puede acceder a la interfaz gráfica de *RabbitMQ* desde la máquina web y así ver cómo los mensajes son publicados y consumidos.

Introduciendo la *IP* de la máquina *rabbitmq* (192.168.10.30) con el puerto en el que se ejecuta el servicio (15672) se consulta la interfaz. Ver Ilustración 72

Name	Exclusive	Parameters	Policy	Status	Ready	Unacked	Total	incoming	deliver / get	ack
storelog_queue		D		Active	0	0	0	0.00/s	0.00/s	
task_queue		D		Active	0	50	50	0.00/s	0.00/s	0.00/s
weblog_queue		D		Active	43	0	43	0.00/s		

Ilustración 72 Estado de las colas en la interfaz de gestión de RabbitMQ

Se pueden observar las tres colas (la de tareas y las de *logging*) en funcionamiento.

Alternativamente, desde la consola, es posible realizar una conexión *SSH* hacia las máquinas para visualizar los registros de las aplicaciones. Así con el comando siguiente:

```
ssh victor@192.168.10.xx
```

Una vez establecida la conexión se accede al archivo de *logs* y mediante la orden *tail* con la opción *-f* se puede mantener un flujo continuo del archivo.

Una vez el proceso ha finalizado, se pueden consultar las imágenes mediante la *URL* proporcionada o accediendo a la carpeta de almacenamiento del servidor.

Para analizar los resultados, sólo es necesario ejecutar el *script octave* citado en el quinto capítulo que generará las gráficas para el tiempo de espera en la cola y el tiempo de respuesta general del sistema.

El proceso de subida para un usuario, utilizando la interfaz *web* se presenta a continuación.

El usuario accederá al sitio web aterrizando en la página principal para la subida (Ilustración 73) pudiendo seleccionar un archivo mediante el explorador de archivos y realizar la subida.

Será redirigido automáticamente hacia la página de subida correcta dónde verá el enlace para recuperar las imágenes procesadas (Ilustración 74)

Esta página de resultados ofrece una vista en cuadrícula de las imágenes (Ilustración 75)

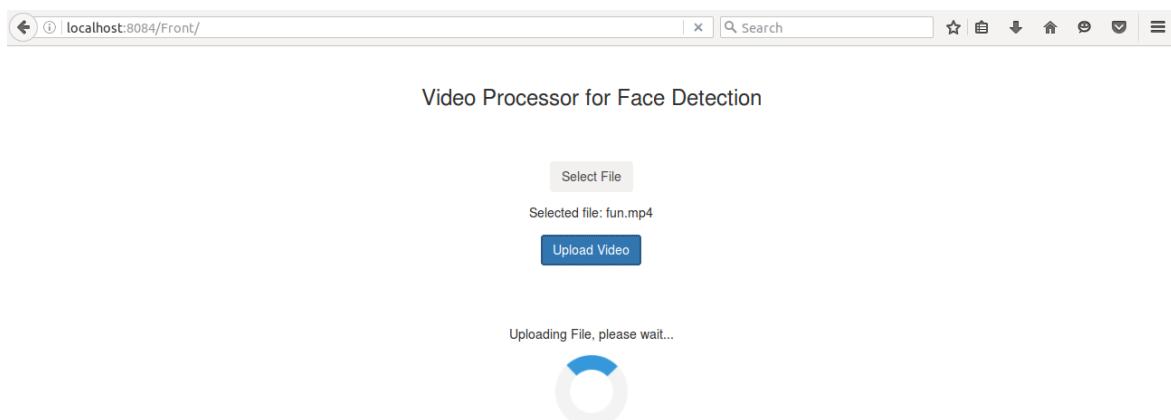


Ilustración 73 Interfaz de usuario mientras se sube un archivo

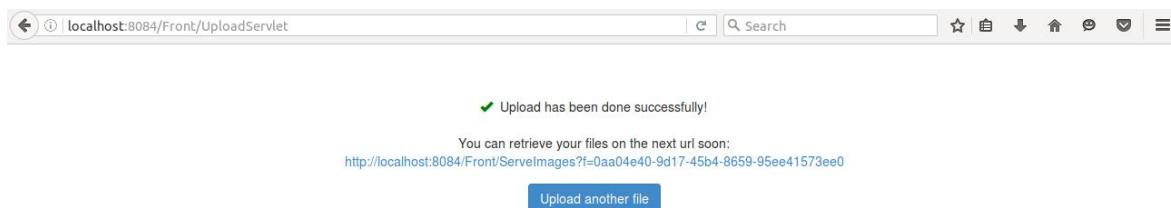


Ilustración 74 Interfaz de usuario de una subida correcta

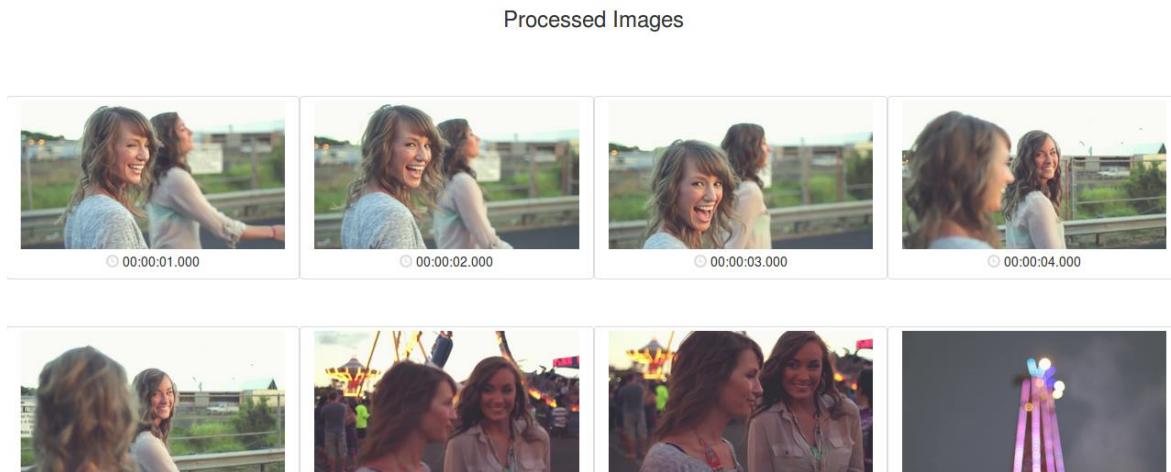


Ilustración 75 Interfaz de usuario del resultado del procesamiento

Como se puede observar, la imagen inferior derecha no es un rostro. Esto es un falso positivo y es algo bastante común cuando se utilizan cascadas de *Haar* por defecto. Pero que puede paliarse entrenándolas con los resultados obtenidos de los propios análisis.

6.3. Presentación de resultados

De los resultados del primer experimento se puede obtener una conclusión evidente y que no habría requerido ningún estudio: a mayor número de *workers*, menor es el tiempo de respuesta. Pero cabe preguntar si esta diferencia de tiempo en el sistema desarrollado se corresponde con lo esperado: el tiempo de respuesta del sistema debería ser la mitad utilizando el doble de *workers* y un tercio utilizando el triple. Es decir que el tiempo de procesado para un número N de *workers* sería, siendo $p(t)$ el tiempo que tarda en procesar el trabajo un único *worker*:

$$p(t)_N = p(t)/N$$

En la Ilustración 76 se representan los datos.

Ejecutar las tareas con un único *worker* produce un tiempo de respuesta de unos 4000 segundos; un tiempo bastante alto. Al ejecutar la tarea con dos *workers*, el tiempo se reduce prácticamente a la mitad. Al hacerlo con tres, el tiempo se reduce a aproximadamente un tercio del tiempo inicial.

Esto prueba el éxito del sistema, donde gracias al paralelismo en el procesamiento de los trabajos, los tiempos de respuesta disminuyen razonablemente.

El sistema está muy limitado desde el punto de vista de *hardware* y sobre todo en el aspecto de la memoria, tanto en *RAM* como en almacenamiento. Por ello, el estudio no se puede realizar para un tiempo demasiado largo –es decir para un número muy alto de imágenes- ni definido, ya que eventualmente, la máquina virtual de Java *JVM* termina por consumir toda la memoria y una excepción del tipo *OutOfMemoryError* es lanzada por *OpenCV*. Esto ya se había tenido en cuenta a la hora de analizar los riesgos del proyecto, sin embargo ha sido imposible contenerlo debido a que el presupuesto es limitado.

Con total certeza, con un entorno construido sobre contenedores *Docker* estos problemas no habrían aparecido. Además todos los procesos de ejecución y construcción de las máquinas habría sido más rápido y estable; también la modificación y compilación del código fuente.

Es necesario añadir que el tiempo de respuesta (el tiempo que no es de espera) incluye muchas variables sujetas al hardware, como el tiempo de copia de imágenes entre máquinas, la eficiencia de la librería *OpenCV*, la asincronía entre los relojes de cada máquina virtual o posibles fallos y excepciones.

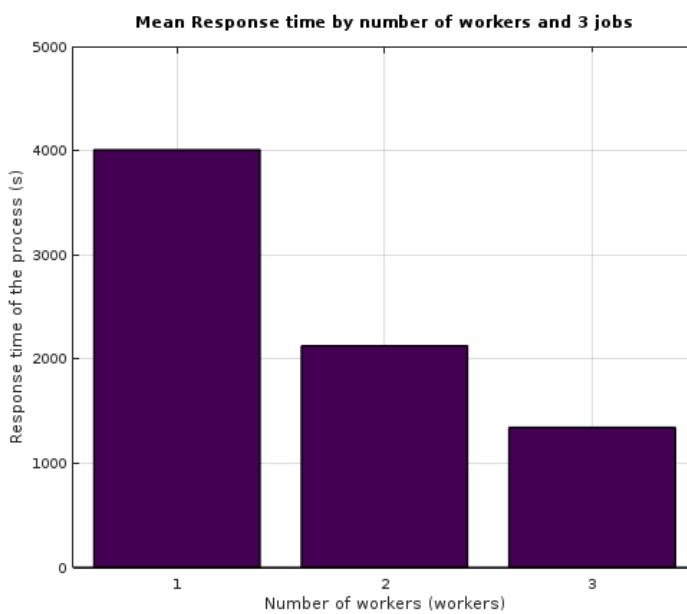


Ilustración 76 Tiempo de respuesta del sistema

El segundo experimento arroja información sobre la eficiencia de *OpenCV* y sobre su funcionamiento. En la Ilustración 77 se representa el histograma de la distribución del tiempo de procesado para los mensajes de una tarea, superpuesta con su función de densidad en rojo. Se puede identificar la media de la distribución mediante la recta de color naranja, con un valor de aproximadamente 1,6 segundos. En amarillo, la desviación típica, de aproximadamente 0,6 segundos, sumado o restado a la media. Una desviación no muy alta con respecto a la media indica que los mensajes tardan todos aproximadamente lo mismo con una variación de 0,6 segundos.

Quedaría saber si el tiempo que tarda *OpenCV* en procesar una imagen depende de la consecución del trabajo que desempeña o no. Esto es, si el tiempo depende de que se detecte una cara, o, por el contrario, es totalmente arbitrario. Para ello se recogen los tiempos de procesado de 100 mensajes utilizando 3 *workers*. De estos datos deriva el gráfico de la Ilustración 78 en la que se representa en naranja las imágenes donde se ha detectado una cara, y en azul en los que no se ha detectado ninguna.

Como se ve, el tiempo que tarda en procesarse una imagen con cara no es ni mayor ni menor que el tiempo que tarda en procesarse una en la que no hay cara. Esto se debe a que, como explicado previamente, el cuadrado (o Matriz como es denominado por *OpenCV*) que se desplaza sobre la imagen, la recorre de inicio a fin, por lo que para un tamaño fijo de imagen con un número de *píxeles* dado, el tiempo de procesado de una imagen es independiente de los resultados del trabajo.

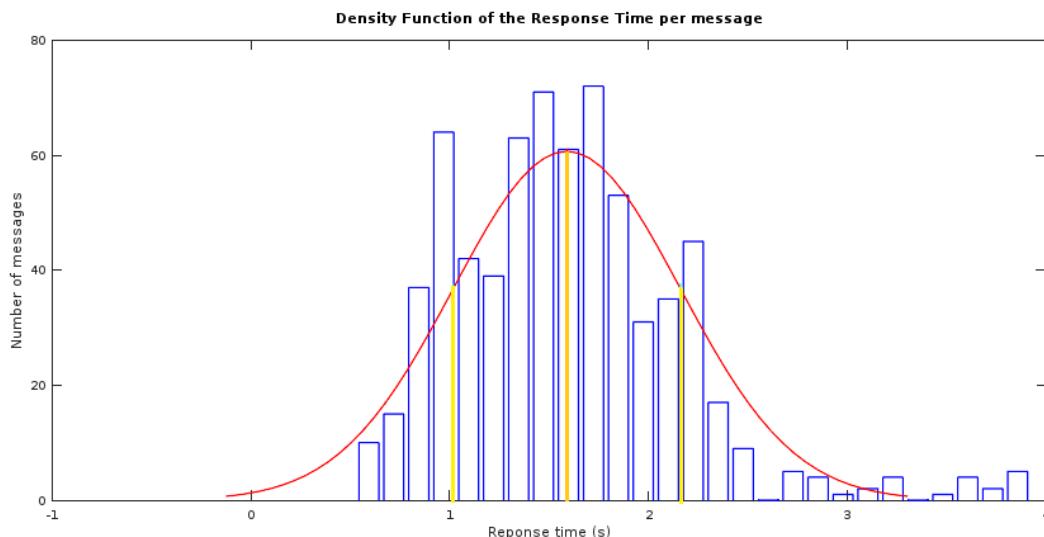


Ilustración 77 Función de densidad del tiempo de respuesta

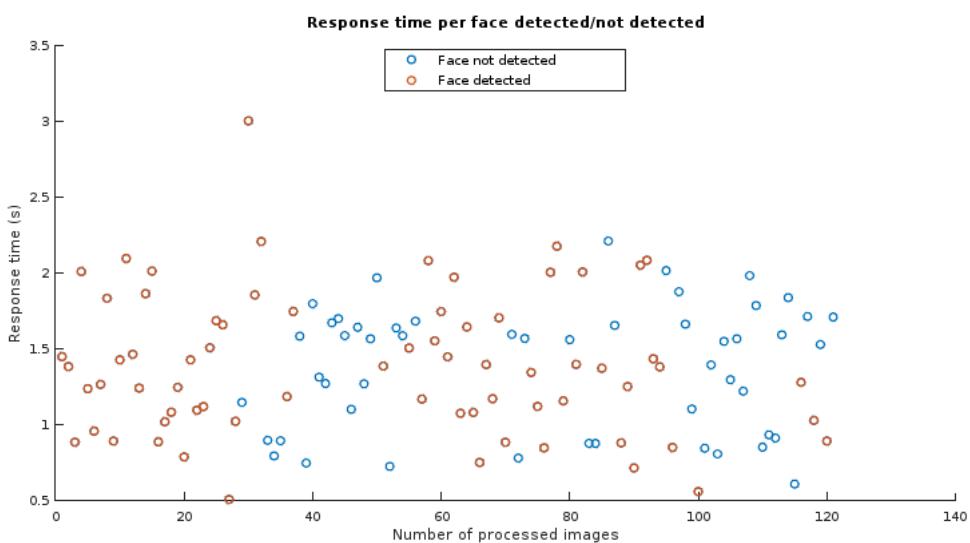


Ilustración 78 Tiempo de respuesta para cada frame con cara o no

Sabiendo lo anterior, queda analizar el tiempo de espera, que, en este caso, representa los datos más precisos en términos de sistemas de colas.

El gráfico de la Ilustración 79 representa los tiempos de espera para un diferente número de *workers*. En él se aprecia la diferencia previamente mencionada de la diferencia de tiempos, dado que el tiempo de espera de un paquete está sujeto al tiempo de procesado. En este gráfico dicha diferencia se plasma desde la perspectiva del mensaje y no de la tarea, siendo el tiempo de espera de un mensaje siempre mayor al del anterior y siendo también aproximadamente igual al de los demás mensajes con una desviación bastante baja. Así podrían obtenerse las pendientes de las rectas y estimar tareas con

mayor número de mensajes. Un estudio intensivo del sistema permitiría escalarlo dinámicamente en función de las necesidades o requerimientos.

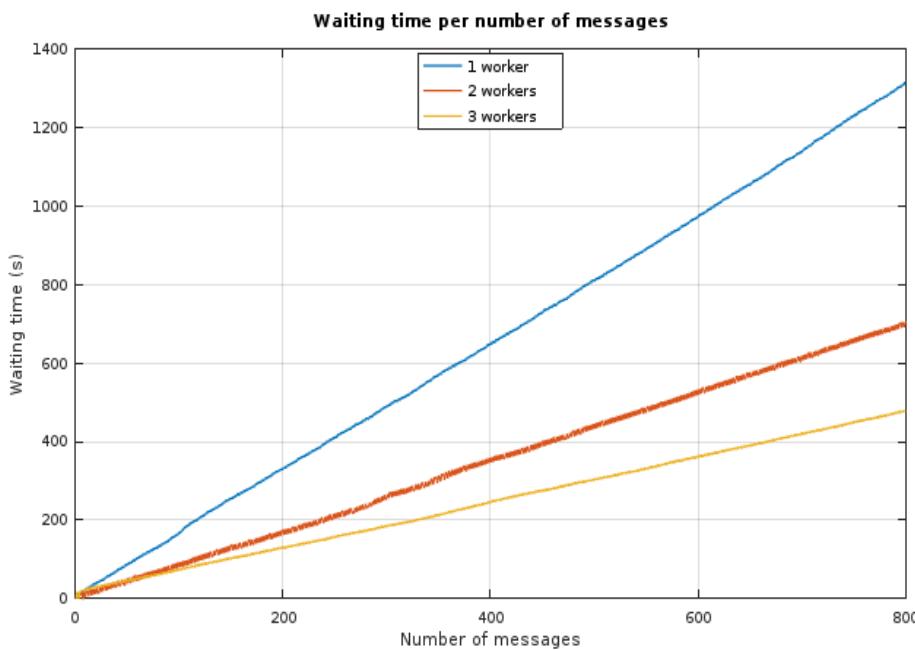


Ilustración 79 Tiempo de espera de un paquete por número de workers

El último experimento, el más sencillo, permite comprobar el correcto funcionamiento de la librería de procesamiento de imágenes. Como explicado previamente, los *frames* de los segundos cincuenta y setenta y cinco aparecen como detectados y el resto no. En la Ilustración 80 aparecen los resultados



Ilustración 80 Comprobación de detección facial

6.4. Evaluación temporal y presupuestaria

En este apartado se contrastarán las estimaciones del Capítulo 3 con los costes reales tras la finalización del proyecto. Dada la falta de experiencia a la hora de planificar, los tiempos reales divergen de los estimados significativamente.

TAREAS		ESTIMADO	REAL
1	GESTIÓN DEL PROYECTO	9,5	11,5
1.1	Inicio	1	1
1.2	Planificación	3,5	4,5
1.3	Ejecución y Control	4	5
1.4	Cierre	1	1
1.5	Proyecto Aprobado		
2	FASE DE ANÁLISIS	6	8
2.1	Obtención de requisitos	2	4
2.2	Plan de Proyecto	4	4
2.3	Entrega de análisis y plan de proyecto		
3	ESTUDIO	14	17
3.1	Estudio de herramientas de virtualización	1	2
3.2	Estudio del paradigma de computación distribuida	2	4
3.3	Estudio de RabbitMQ	4	3
3.4	Estudio de OpenCV	2	2
3.5	Estudio de la topología de red	1	1
3.6	Instalación y configuración de herramientas	2	3
3.7	Evaluación y prueba de herramientas	2	2
3.8	Entrega del estudio		
4	DISEÑO	13	13,25
4.1	Diseño de la aplicación en máquina Web	5	7
4.1.1	Diseño de Servlet de subida de vídeo	1	1

4.1.2	Diseño de Servlet de carga de imágenes	1	1
4.1.3	Diseño de componente de transferencia de datos	1	3
4.1.4	Diseño de componente de recolección de estadísticas	2	2
4.2	Diseño de la aplicación en máquina Storage	2,5	2
4.2.1	Diseño de componente de división de vídeo	0,5	0,25
4.2.2	Diseño de componente de publicación de mensajes	1,5	1,5
4.2.3	Diseño de componente de recolección de estadísticas	0,5	0,25
4.3	Diseño de aplicación en máquinas Worker	4,5	4,25
4.3.1	Diseño de componente de consumición de mensajes	0,5	1
4.3.2	Diseño de componente de detección facial	1,5	1
4.3.3	Diseño de componente de publicación de mensajes	0,5	0,5
4.3.4	Diseño de componente de transferencia de datos	0,5	0,25
4.4	Diseño de script en OCTAVE para presentación de estadísticas	1,5	1,5
4.5	Entrega Diseño		
5	IMPLEMENTACIÓN	28,5	16,75
5.1	Implementación de la aplicación en máquina Web	6,5	6
5.1.1	Implementación de Servlet de subida de vídeo	1,5	1,5
5.1.2	Implementación de Servlet de carga de imágenes	2	2
5.1.3	Implementación de componente de transferencia de datos	1	2
5.1.4	Implementación de componente de recolección de estadísticas	2	0,5

5.2	Implementación de la aplicación en máquina Storage	5,5	2,25
5.2.1	Implementación de componente de división de vídeo	1	0,25
5.2.2	Implementación de componente de publicación de mensajes	2	1
5.2.3	Implementación de componente de recolección de estadísticas	2,5	1
5.3	Implementación de aplicación en máquinas Worker	16,5	8,5
5.3.1	Implementación de componente de consumición de mensajes	5	3
5.3.2	Implementación de componente de detección facial	7	1
5.3.3	Implementación de componente de publicación de mensajes	1	1
5.3.4	Implementación de componente de transferencia de datos	0,5	0,5
5.4	Implementación de script en OCTAVE para presentación de estadísticas	3	3
5.5	Entrega Implementación		
6	PRUEBAS	8,5	17
6.1	Pruebas de funcionamiento de la aplicación	3,5	7
6.2	Pruebas de rendimiento	5	10
6.3	Entrega Pruebas		
7	ELABORACIÓN DE LA MEMORIA	29	25
7.1	Entrega Documento		

Tabla 15 Comparación estimación temporal con real

En la Tabla 8 se observa una estimación correcta para las primeras fases del proyecto, con una ligera desviación, pero asumible. Sin embargo, se sobreestimó el tiempo de desarrollo excesivamente, pensando que sería más complejo de lo que realmente fue, gracias a la buena documentación y facilidad de uso de librerías como *RabbitMQ* o *OpenCV*. En contraposición, el tiempo de ejecución de las pruebas y la

recolección de resultados se subestimó. Esta desviación se debe principalmente a los problemas encontrados con los recursos *hardware* de la máquina y los errores de memoria anunciados en la sección anterior. En las Ilustraciones 81 a 84 se presenta el diagrama de *Gantt* actualizado con los tiempos reales. En él se puede observar que la fecha de fin del proyecto es el 04/06/2019 mientras que la fecha de fin estimada era el 14/03/2018, un año y tres meses más tarde. A parte de la desviación en la estimación, motivos laborales en el extranjero desde principio de año de 2018 hasta principio de año de 2019 han obligado a posponer el desarrollo del proyecto casi un año entero. Destacar que en la cuarta parte del Diagrama de la Ilustración 84 se ha suprimido el periodo de tiempo en el que no se ha trabajado en el proyecto.

Así, el proyecto tiene una duración de 19,5 meses contando el periodo de *hold*. Con un total de 118,5 días laborales, que representa una desviación de 10 días laborales con respecto a la estimación inicial.

Las desviaciones entre las estimaciones y la realidad incurren directamente en un aumento de los costes, tanto de *software* y *hardware* como de recursos humanos. En la Tablas siguientes se plasman dichos aumentos.

Hardware	Unidades	Precio	Amortización	Coste final
Lenovo IdeaPad U330T	1	699,00 €	54%	377,46 €
TOTAL (+ IVA 21%)				456,75 €

Tabla 16 Costes de hardware reales

Evidentemente, dada la baja amortización del producto y habiéndose alargado el proyecto un año más de lo esperado los costes de *hardware* aumentan significativamente.

Software	Descripción	Precio	Amortización	Precio final
Windows 10 Ultimate	Sistema Operativo	135,00 €	27%	36,45 €
Ubuntu 14.04	Sistema Operativo	0,00 €	27%	0,00 €
VirtualBox 5.0.1	Software Virtualización	0,00 €	27%	0,00 €
RabbitMQ	Broker de mensajes	0,00 €	27%	0,00 €
OpenCV	Librería procesado imág.	0,00 €	27%	0,00 €
AvConv	Librería procesado imág.	0,00 €	27%	0,00 €

NetBeans 13	Entorno de desarrollo	0,00 €	27%	0,00 €
Visual Paradigm 14.1	Documentación Ing. Soft.	86,00 €	27%	23,2 €
Microsoft Office 2016	Herramienta ofimática	127,99 €	27%	34,55 €
Microsoft Projects 2016	Administración proyectos	119,00 €	27%	32,1 €
Google Chrome v61	Navegador web	0,00 €	27%	0,00 €
Mozilla Firefox 45	Navegador web	0,00 €	27%	0,00 €
TOTAL (+ IVA 21%)				152,8 €

Tabla 17 Costes de software reales

Con los costes de *software* ocurre algo similar que con los de *hardware*, sólo que el aumento no es tan importante dada su amortización de 6 años.

Personal	Sueldo anual	Tasa S.S.	Sueldo Anual Total	Cantidad de trabajo	Coste total 5,3 meses
Analista de Datos	14 329,54 €	4 055,26 €	18 384,80 €	55%	4 466 €
Programador	13 508,57 €	3 822,92 €	17 331,49 €	33%	2 526,1 €
Jefe Proyecto	15 150,50 €	4 287,59 €	19 438,09 €	12%	1 030,2 €
Total					
Total media jornada					

Tabla 18 Costes de recursos humanos reales

Los costes de recursos humanos aumentan en función de los días de desviación en la planificación.

Por último, los costes totales del proyecto, en la Tabla 12, derivan de los costes estimados (con un valor de 4702,1 €) en 830,78 €. Diferencia marcada principalmente por las amortizaciones de los productos y no por los costes de recursos humanos.

Tipo de coste	Coste Directo	Coste Indirecto (20% coste directo)	Coste total
Hardware	456,75 €	91,3 €	548,1 €
Software	152,8 €	30,5 €	183,4 €
Recursos Humanos	4 001,15 €	800,25 €	4 801,38 €
Total			

Tabla 19 Costes totales reales

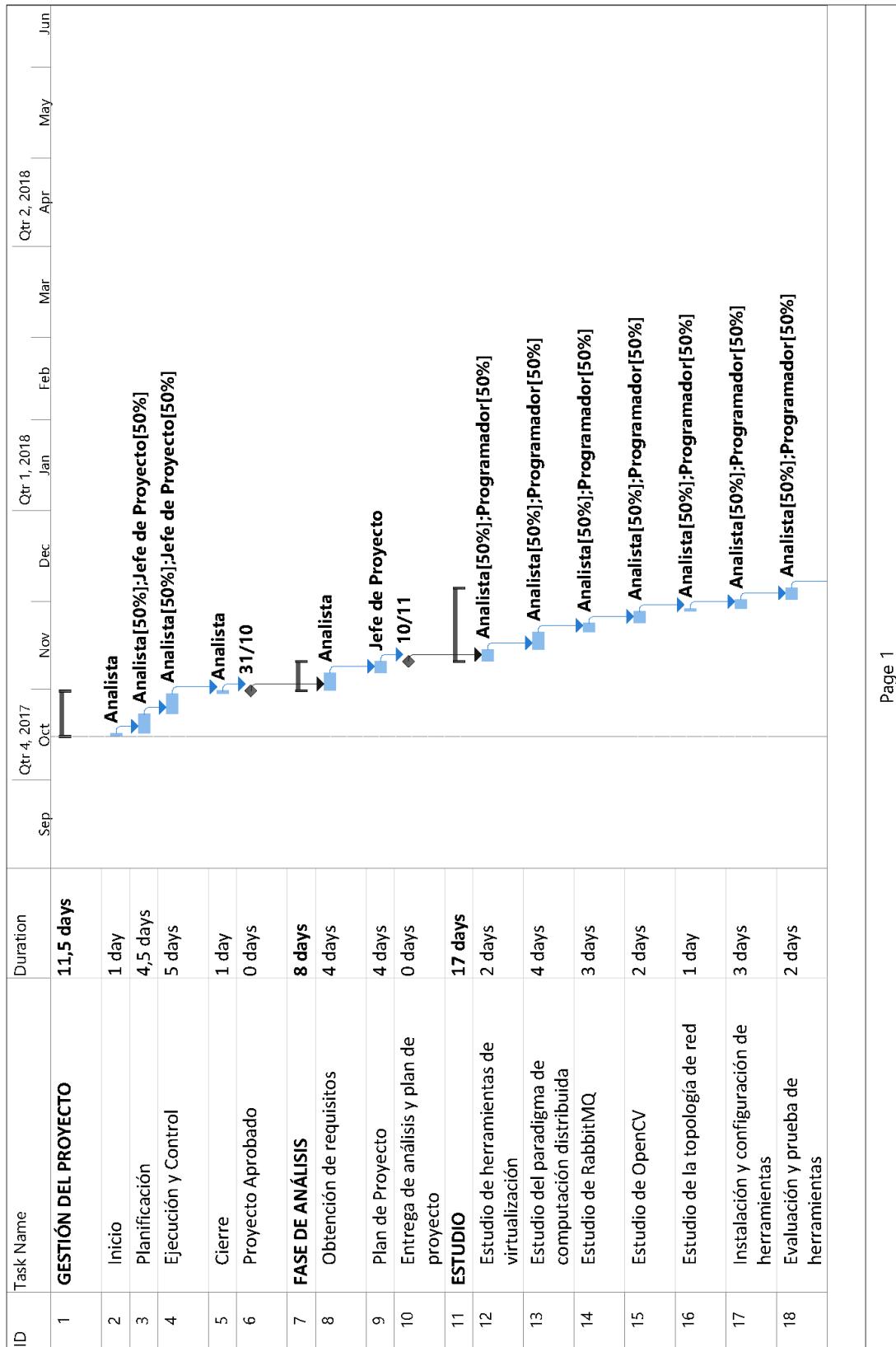
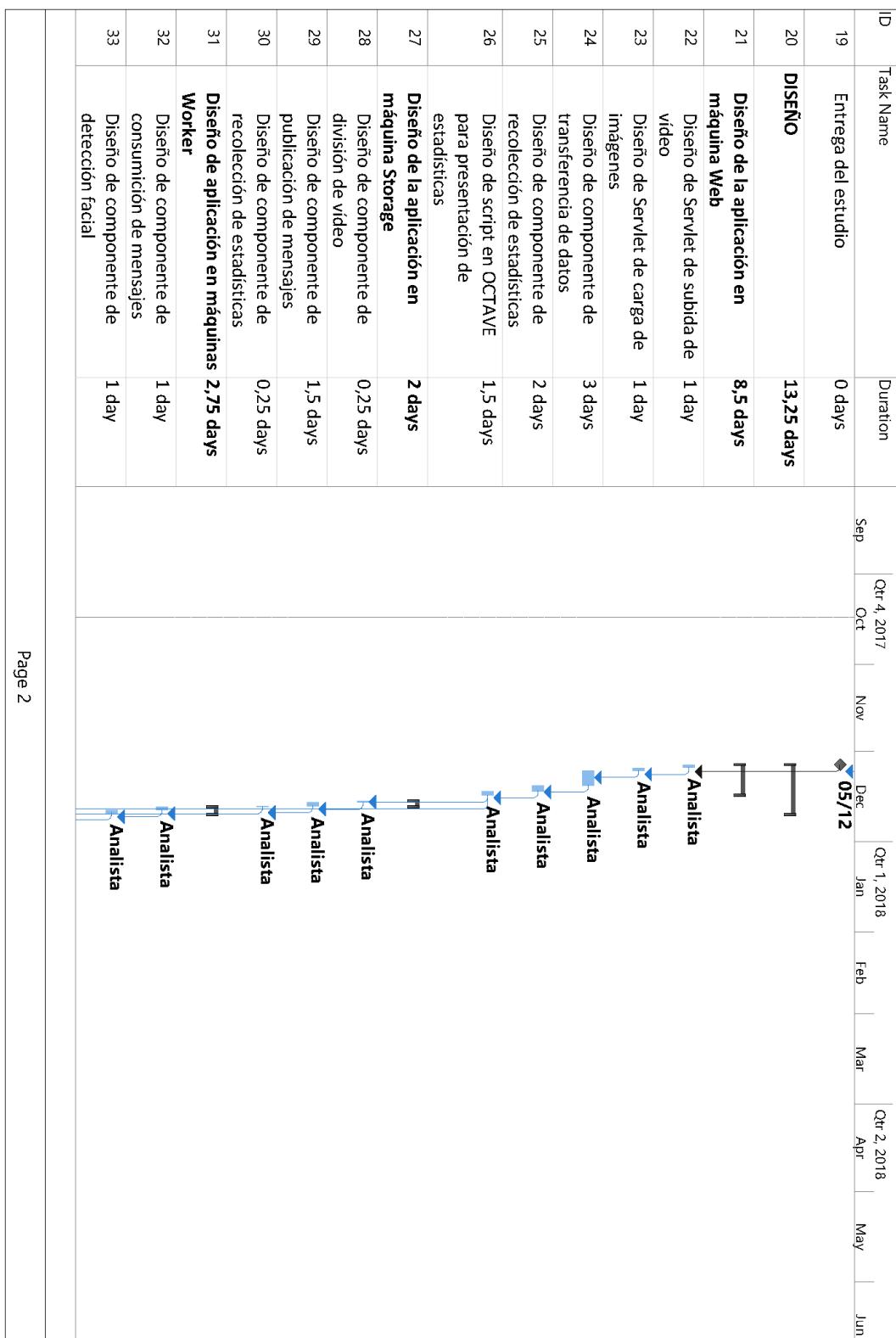
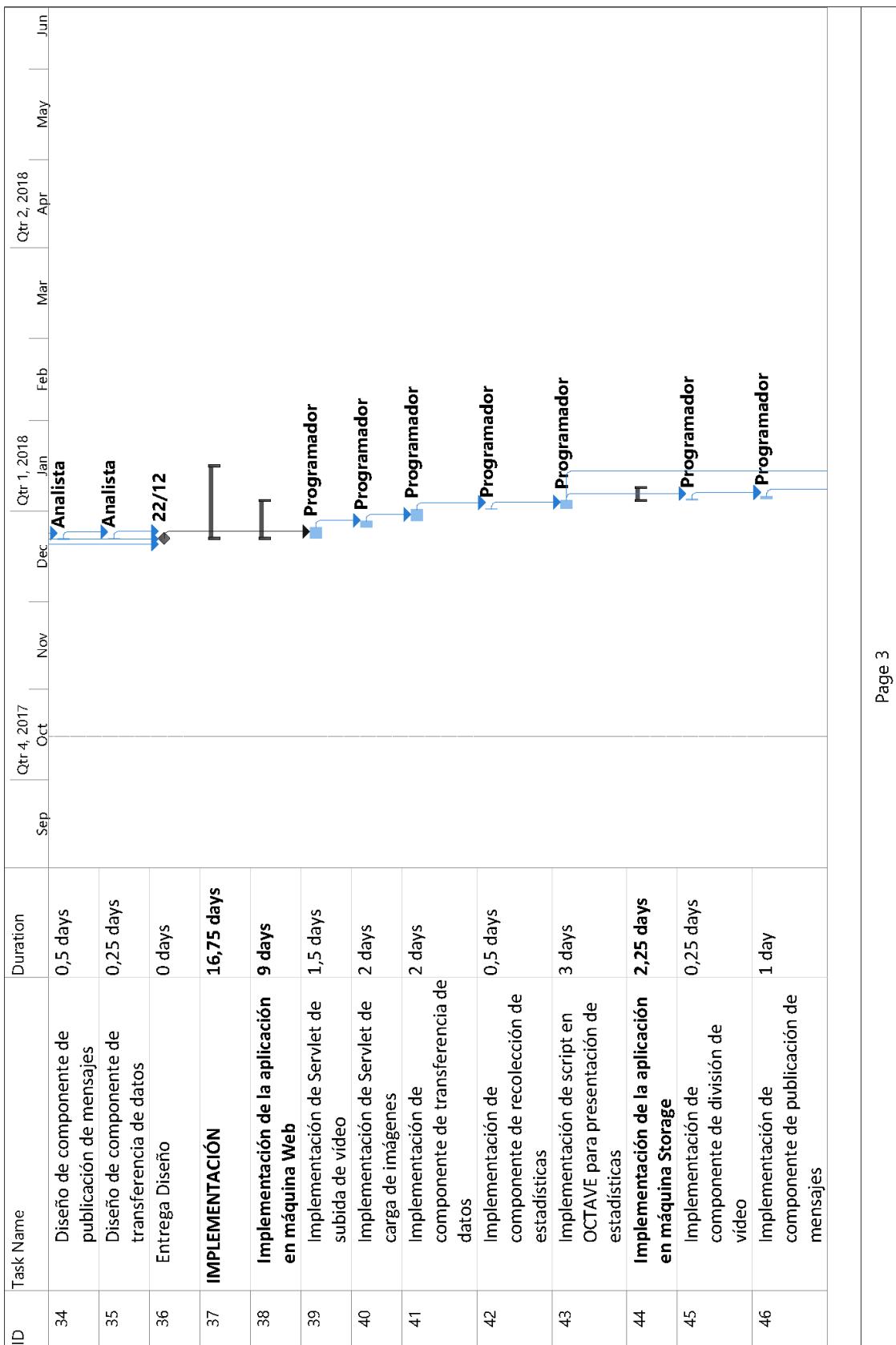


Ilustración 81 Diagrama de Gantt real/p.1

Page 1



Page 2



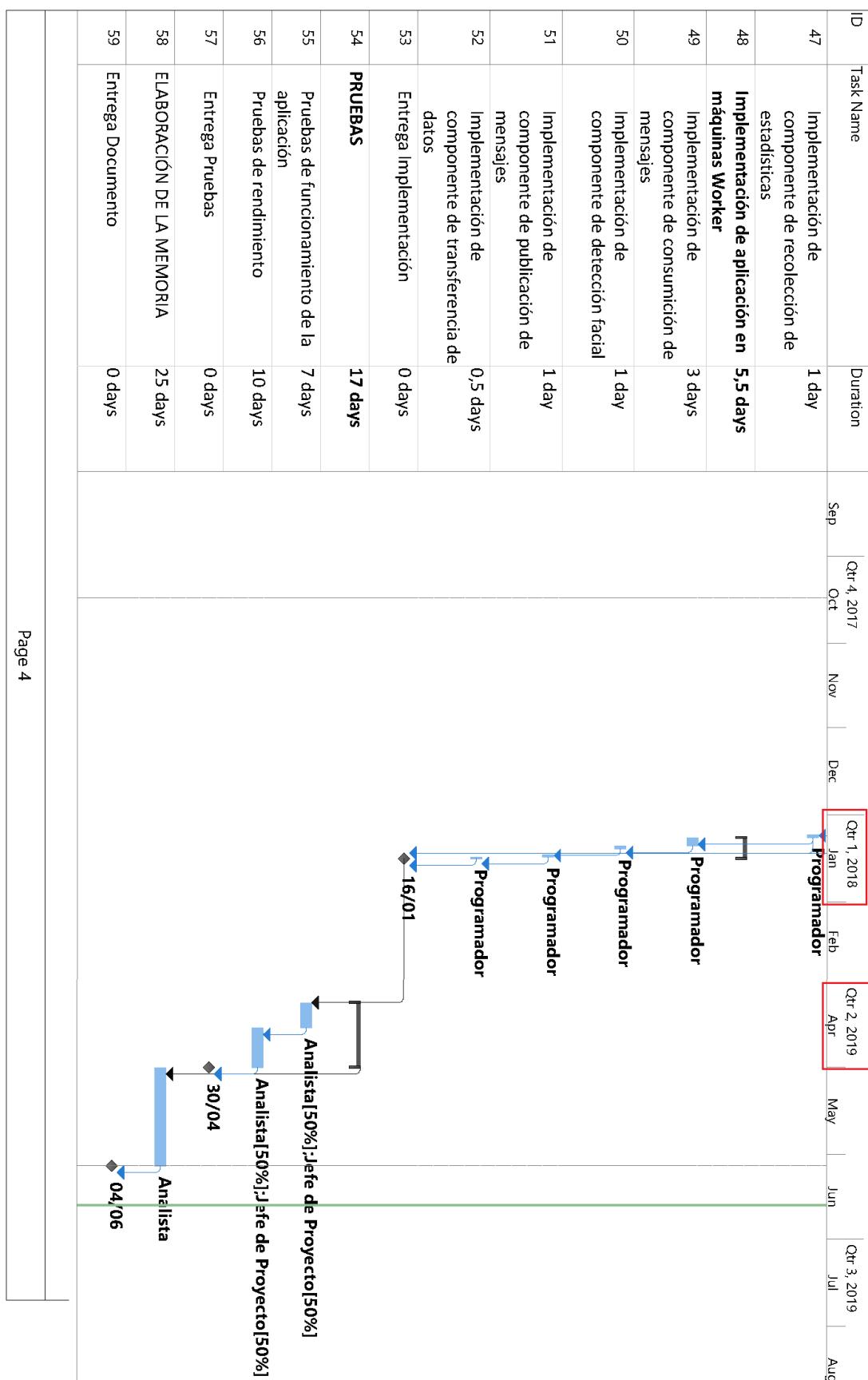


Ilustración 84 Diagrama de Gantt real/p.4

Capítulo 7. Conclusiones y trabajos futuros

7.1. Conclusiones

Tras la finalización del proyecto, pasando por todas sus fases e incluso repitiendo algunas, puede concluirse que los objetivos han sido alcanzados satisfactoriamente. Se han dado diferencias con respecto a la planificación inicial, tanto temporal como económica, debido a una emigración al extranjero por trabajo. Sin embargo, no ha impedido que, aunque fuera de plazo, los requisitos han sido cumplidos.

Los objetivos principales eran desplegar una red de computación distribuida y probar su eficiencia con respecto a la computación clásica en una única máquina. Se puede decir que la red ha sido desplegada correctamente y que la eficiencia queda más que probada; cosa que era de esperar debido al paralelismo del procesamiento, algo ampliamente estudiado durante la carrera.

Con respecto al desarrollo, las aplicaciones han sido implementadas y re-implementadas debido a la refactorización llevada a cabo. Buscando siempre la máxima coherencia, eficiencia y escalabilidad. Produciendo un resultado satisfactorio y en el que se han añadido técnicas aprendidas en el mundo laboral.

A nivel de usuario, las interfaces gráficas, aunque sencillas, proporcionan las funcionalidades especificadas de una manera *user-friendly*. Permitiendo subir un vídeo y recuperar el resultado de su procesamiento.

Con la consecución de estos objetivos, se ha llegado a muchas conclusiones. Entre las más destacables estaría la importancia de la buena especificación y diseño del sistema, que permite definir precisamente y de manera estándar cómo implementar dicho diseño. Habiendo tenido la oportunidad de trabajar con metodologías ágiles como *SCRUM* es fácil darse cuenta de los dolores de cabeza, pérdidas de tiempo y malentendidos que podrían evitarse con un buen diseño de las funcionalidades. No siendo un experto, es fácil cometer errores en la arquitectura de la aplicación y que más tarde a la hora de implementarla sean descubiertos y sea necesario reiterar sobre el diseño y refactorización del código; pero esto permite crear una aplicación robusta desde el inicio y fácil de analizar y entender por otros desarrolladores.

También a nivel de gestión de proyectos se obtiene una conclusión muy clara: infraestimación de los tiempos de desarrollo, subestimación de oportunidades o inconvenientes externos al proyecto o la ignorancia de que volver a repetir fases del proyecto fuera a ser necesaria.

Cabe recalcar que todo el sistema ha sido diseñado de manera escalable para ser capaz de funcionar simplemente añadiendo más máquinas a la red y con una configuración de red mínima de éstas. También se han desarrollado las aplicaciones de manera escalable, permitiendo que si en un futuro se quisieran añadir nuevas

funcionalidades al sistema, como por ejemplo otro tipo de procesado de la imagen, la implementación de los mensajes contiene un campo que indica el tipo de trabajo a realizar.

Así, este sistema podría ser ampliado tanto cualitativa como cuantitativamente para ejecutar mayor número de procesos diferentes en menor tiempo. Siendo el límite la capacidad física de los dispositivos.

Para abordar el proyecto ha sido necesario el aprendizaje y familiarización con nuevos entornos, aplicaciones, librerías y lenguajes de desarrollo. Se ha complementado y aplicado los conocimientos sobre la utilidad de las colas en la informática en este caso de la mano de *RabbitMQ*. También con tecnologías nuevas de procesado de imágenes como *OpenCV* y una asimilación del sistema *Linux* en muchos de sus aspectos, desde configuración hasta *bash scripting*. Por supuesto se han aprendido una ingente cantidad de nuevas técnicas de desarrollo *Java* aumentando las competencias de manera significativa, desde *servlets*, pasando por técnicas de comunicación a través de la red o ejecución de comandos a distancia al estilo *RMI*, hasta una comprensión mayor del funcionamiento de la máquina virtual de *Java*, la compilación y ejecución de aplicaciones y su mantenimiento y gestión.

Subrayar que, en este proyecto, el uso de algún *framework* como *Spring* o un gestor de dependencias como *Maven* habrían facilitado bastantes tareas, pero que, debido al desconocimiento de estas tecnologías al comienzo del proyecto no se pusieron en práctica.

No obstante, gracias a los conocimientos adquiridos en la carrera, este proyecto pluri-competente (por llamarlo de alguna manera) ha podido ser llevado a cabo. Tanto a nivel de desarrollo, como de colas, de aplicaciones web, de administración de sistemas operativos y de gestión de proyectos.

Por último, destacar que este proyecto no busca mejorar los sistemas distribuidos actuales, ni competir con proyectos de grandes empresas como *Amazon* o *Microsoft* que los poseen a escala macroscópica; más bien el objetivo es poner en práctica dicho sistema, hacerlo escalable y utilizable por cualquier usuario y por supuesto evaluar sus beneficios.

7.2. Trabajos futuros

En este proyecto se han llevado a cabo los objetivos iniciales, sin embargo, hay muchas variables que se podrían mejorar. La principal y que más problemas ha dado ha sido el uso de máquinas virtuales, que agotan los recursos del anfitrión. En la misma línea de desarrollo habría sido interesante poder probar el sistema en un entorno mayor, con una veintena de máquinas y estudiar los resultados con tamaños mayores de archivos y diferentes operaciones. Además, una posibilidad contemplada fue subir el proyecto a *GitHub* de manera estructurada y con un manual de usuario para que cualquier persona pudiese beneficiarse del sistema distribuido. También habría sido interesante automatizar al cien por cien la recogida de estadísticas y producción de resultados cosa que hasta el

momento requiere algunas acciones por parte del administrador. Otra mejora importante sería la de poder recibir un flujo de vídeo en tiempo real y no un archivo de vídeo.

Este proyecto abre una antesala de posibilidades, no para continuar con el desarrollo –cosa que también sería posible- sino para utilizar la salida o resultado de este sistema como entrada de datos para otros sistemas. Por ejemplo, podría utilizarse para nutrir de datos a un sistema de entrenamiento para el reconocimiento facial, o incluso a redes neuronales que aprendan de este proceso de reconocimiento. Este sistema podría también servir de receptor de datos para su posterior computación en la nube, administrando hacia qué servidores distribuir la información.

Referencias

- [1] VMWare, "What is virtualization technology virtual machine? | VMware," 2019. [Online]. Available: <https://www.vmware.com/solutions/virtualization.html>. [Accessed: 20-Jun-2019].
- [2] "Oracle VM VirtualBox," [Online]. Available: <https://www.virtualbox.org/>. [Accessed: 20-Jun-2019].
- [3] "IBM Archives: System/370 Model 145." [Online]. Available: https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP3145.html. [Accessed: 20-Jun-2019].
- [4] A. S. Tanenbaum, *Distributed systems : principles and paradigms*. Upper Saddle River: Upper Saddle River : Prentice Hall : Pearson Education International, cop. 2002., 2002.
- [5] "Sun Java System Message Queue 4.3 Technical Overview." [Online]. Available: <https://docs.oracle.com/cd/E19575-01/820-6424/index.html>. [Accessed: 20-Jun-2019].
- [6] "MQTT v5.0-os Specification URIs." 2019. [Online]. Available: <https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html>. [Accessed: 20-Jun-2019].
- [7] R. Jeyaraman, R. J. Com), A. Telfer, and R. Godfrey, "OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0 Specification URIs Chairs: Editors," AMQP TC, 2012.
- [8] "STOMP Protocol Specification, Version 1.2." [Online]. Available: <http://stomp.github.io/stomp-specification-1.2.html>.
- [9] "Hub and Spoke [or] Zen and the Art of Message Broker Maintenance - Enterprise Integration Patterns." [Online]. Available: https://www.enterpriseintegrationpatterns.com/ramblings/03_hubandspoke.html. [Accessed: 20-Jun-2019].
- [10] "Messaging that just works — RabbitMQ." [Online]. Available: <https://www.rabbitmq.com/>. [Accessed: 20-Jun-2019].
- [11] "Apache Kafka." [Online]. Available: <https://kafka.apache.org/uses>. [Accessed: 20-Jun-2019].

- [12] "ActiveMQ." [Online]. Available:
] <https://activemq.apache.org/components/classic/>. [Accessed: 20-Jun-2019].
- [13] J. R. Mejía Vilet, "Apuntes de procesamiento digital de imágenes," Área *Comput. e Infomática la Fac. Ing. Univ. Autónoma San Luis Potosí. Pags*, vol. 95, 2005.
- [14] T. Díaz Miranda and J. L. Viloria Rodríguez, "Algoritmo de Viola-Jones para detección de rostros en procesadores gráficos," *Rev. Estud. Nac. Ing. y Arquit. RNPS 2359. ISSN 2307-471X*, vol. 2, no. 3, pp. 23–33, 2012.
- [15] P. Viola, M. Jones, and others, "Robust real-time object detection," *Int. J. Comput. Vis.*, vol. 4, no. 34–47, p. 4, 2001.
- [16] "OpenCV: OpenCV modules." [Online]. Available:
] <https://docs.opencv.org/4.1.0/>. [Accessed: 20-Jun-2019].

Anexo Código Fuente

Web

UploadServlet.java

```

public class UploadServlet extends HttpServlet {

    private final static int MAX_FILE_SIZE = 1024*1024*1000;
    private final static int MAX_REQUEST_SIZE = 1024*1024*1100;
    private final static String SUCCESS_MESSAGE ="<span class=\"glyphicon glyphicon-ok\" style=\"color: green; margin-right: 4px;\">></span> Upload has been done successfully! <br/><br/> You can retrieve your files on the next url soon: <br/>";
    private final static String RETRIEVE_URL =
"http://localhost:8084/Front/ServeImages?f=";
    private final static String UPLOAD_PATH = "/home/victor/apache-tomcat-8.0.27/webapps/data/";

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException, FileUploadException {

        ServletFileUpload upload = configureUploadFile();
        // Get files from request
        List fileItems = upload.parseRequest(new ServletRequestContext(request));
        String directoryPath = createUploadDirecotory();
        createStatisticsFile(directoryPath);
        File uploadedFile = saveUploadedFile(fileItems, directoryPath);

        processVideo(uploadedFile);

        String urlFolder = RETRIEVE_URL + uploadedFile.getParentFile().getName();
        request.setAttribute("url", urlFolder);
        request.setAttribute("message", SUCCESS_MESSAGE);

        getServletContext().getRequestDispatcher("/uploadDone.jsp").forward(request,
        response);
    }

    private ServletFileUpload configureUploadFile() {
        DiskFileItemFactory factory = new DiskFileItemFactory(MAX_FILE_SIZE, new
File(System.getProperty("java.io.tmpdir")));
        ServletFileUpload upload = new ServletFileUpload(factory);
        upload.setSizeMax(MAX_REQUEST_SIZE);
        upload.setFileSizeMax(MAX_FILE_SIZE);
        return upload;
    }

    private File saveUploadedFile(List files, String directory) {
        Iterator i = files.iterator();
        File storeFile = null;
        while (i.hasNext()) {
            FileItem item = (FileItem)i.next();
            if (!item.isFormField()) {
                String filePath = directory + File.separator + item.getName();
                storeFile = new File(filePath);
                try {
                    // saves files on disk
                    item.write(storeFile);
                } catch (Exception ex) {

Logger.getLogger(UploadServlet.class.getName()).log(Level.SEVERE, null, ex);
}
}

```

```
        }
    }

    return storeFile;
}

private void createStatisticsFile(String directoryPath) {
    try {
        new File(directoryPath + File.separator +
"statistics.txt").createNewFile();
    } catch (IOException ex) {
        Logger.getLogger(UploadServlet.class.getName()).log(Level.SEVERE,
null, ex);
    }
}

private String createUploadDirecotory() {
    File uploadDir = new File(UPLOAD_PATH);
    String UIDDIRECTORY = UUID.randomUUID().toString();
    String dir = UPLOAD_PATH + UIDDIRECTORY;
    new File(dir).mkdirs();
    return dir;
}

private void processVideo(File video) {
    ProcessVideo cp = new ProcessVideo(video.getAbsolutePath());
    Thread thread = new Thread(cp);
    thread.start();
}
```

Storage

Tasker.java

```
public class Tasker {
    private final static String USERNAME = "guest";
    private final static String PASSWORD = "guest";
    private final static String HOST = "192.168.10.30";
    private final static int PORT = 5672;
    private final static String EXCHANGE = "img_process";
    private final static String ROUTING_KEY = "img_process_rk";
    private final static String FORMAT = "jpg";
    private final static String[] fileFormat = {"png"};

    public static void main(String[] argv)
        throws java.io.IOException, TimeoutException {
        Channel channel = QueueUtil.createChannel(HOST, PORT);

        Iterator it = FileUtils.iterateFiles(new File(argv[0]), fileFormat, true);
        while (it.hasNext()) {
            File file = (File) it.next();
            String fileName = file.getName();
            String message = createMessage(argv[0], argv[1], argv[2], argv[3], fileName,
                "", argv[4]);

            channel.basicPublish(EXCHANGE, ROUTING_KEY,
                MessageProperties.PERSISTENT_TEXT_PLAIN, message.getBytes());
            System.out.println(" [x] Sent '" + message + "'");
        }
    }

    public static String createMessage(String sourceDir, String sourceIP, String
destinationIP, String destinationDir, String fileName, String operation, String
frameTime) {
        long time = System.currentTimeMillis();
        Message message = new Message(sourceDir, sourceIP, destinationIP,
destinationDir, fileName, operation, time, Long.valueOf(frameTime));
        Gson gson = new Gson();
        return gson.toJson(message);
    }
}
```

ReceiveLog.java

```

public class ReceiveLog {
    private final static String QUEUE_NAME = "storelog_queue";
    private final static String HOST = "192.168.10.30";
    private final static int PORT = 5672;
    private final static String PUBLIC_RSAKEY_FILE = "~/.ssh/id_rsa";
    private final static String REMOTE_HOST = "victor@192.168.10.50";
    private final static String STRG_DIR = "~/video-storage/";
    private final static String STATS_FILE = "/home/victor/video-
storage/statistics.txt";
    private static Map<String, AtomicInteger> countMap = new
    ConcurrentHashMap<String, AtomicInteger>();
    private static Map<String, Long> timeMap = new ConcurrentHashMap<String,
    Long>();
    private static Gson gson;

    public static void main(String[] argv) throws Exception {
        gson = new Gson();
        Channel channel = QueueUtil.createChannel(HOST, PORT);

        System.out.println(" [*] Waiting for messages");

        Consumer consumer = createConsumer(channel);
        channel.basicConsume(QUEUE_NAME, true, consumer);
    }

    private static Consumer createConsumer(Channel channel) {
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws
        IOException {
                String message = new String(body, "UTF-8");
                System.out.println(" [x] Received '" + message + "'");

                Message imgMsg = convertFromJson(message);
                String folder = imgMsg.getSourceDir();

                mapLogs(folder);
            }
        };
        return consumer;
    }

    private static void mapLogs(String folder) throws IOException {
        String UUID = folder.substring(folder.lastIndexOf(File.separator));
        AtomicInteger count = countMap.get(UUID);
        if (count==null) {
            count = new AtomicInteger(1);
            countMap.put(UUID, count);
            timeMap.put(UUID, System.currentTimeMillis());
        } else {
            count.incrementAndGet();
            countMap.put(UUID, count);
        }
        if (countMap.get(UUID).get() == new File(folder).list().length) {
            long processTime = System.currentTimeMillis() - timeMap.get(UUID);
            generateStatistics(UUID, processTime);
            deleteDir(folder);
        }
    }

    private static void generateStatistics(String UUID, long time) throws
    IOException {
        String stat = UUID + " : " + String.valueOf(time) + "ms" + "\n";
        File file = new File(STATS_FILE);
        Files.write(file.toPath(), stat.getBytes(), StandardOpenOption.APPEND);
    }
}

```

```
}
```

```
private static Message convertFromJson(String json) {
    return gson.fromJson(json, Message.class);
}

private static void deleteDir(String directory) {
    try {
        File dir = new File(directory);
        FileUtils.deleteDirectory(dir);
    } catch (IOException e){}
}
```

Worker

Worker.java

```

public class Worker {

    private final static String QUEUE_NAME = "task_queue";
    private final static String HOST = "192.168.10.30";
    private final static int PORT = 5672;
    private final static int QOS = 1;
    private final static String FORMAT = "png";
    private final static String PUBLIC_RSAKEY_FILE = "/home/victor/.ssh/id_rsa";
    private final static String REMOTE_HOST = "victor@192.168.10.20";
    private final static String TMP_STORAGE_DIR = "/home/victor";
    private final static String EXCHANGE_SEND = "logs";
    private final static String EXCHANGE_RCV = "img_process";
    private final static String ROUTING_KEY_SEND = "log_rk";
    private final static String ROUTING_KEY_RCV = "img_process_rk";
    private final static String WORKER_ID = "worker1";
    private final static String PROCESSED_IMG_DIR = TMP_STORAGE_DIR +
File.separator + "processed_images";

    public static void main(String[] argv) throws Exception, TimeoutException,
IOException, InterruptedException {
        boolean autoAck = false;
        createStoreFile();

        final Channel channel = QueueUtil.createChannel(HOST, PORT);
        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
        final Consumer consumer = createConsumer(channel);
        channel.basicConsume(QUEUE_NAME, autoAck, consumer);
    }

    private static void createStoreFile() {
        new File(PROCESSED_IMG_DIR).mkdirs();
    }

    private static Consumer createConsumer(Channel channel) {
        final Consumer consumer = new DefaultConsumer(channel) {
            public void handleDelivery(
                String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties,
                byte[] body) throws IOException
            {
                String message = new String(body);
                System.out.println(" [x] Received, processing " + message);
                try {
                    doWork(message);
                } finally {
                    System.out.println(" [x] Done");
                    channel.basicAck(envelope.getDeliveryTag(), true);
                }
            }
        };
        return consumer;
    }

    private static void doWork(String task) {
        Message message = convertFromJson(task);
        message.setQueueOutTime(System.currentTimeMillis());

        String destinationDirectory = PROCESSED_IMG_DIR + File.separator +
getFileFolder(message.getDestinationDir());
        String tmpSourceFile = TMP_STORAGE_DIR + File.separator +
message.getFileName();
        String filePath = destinationDirectory + File.separator +
message.getFileName();
    }
}

```

```

copyFileFromStorage(message.getSourceDir(), message.getFileName());

// Process the image
boolean faceDetected;
FaceDetector fd = new FaceDetector();

long inTime = System.currentTimeMillis();
faceDetected = fd.detectFaces(tmpSourceFile);
long outTime = System.currentTimeMillis();

message.setProcessTime(outTime - inTime);
message.setWorkerFile(filePath);
String file;
// Set the File if it was processed OK
if (faceDetected) {
    checkDir(destinationDirectory);
    System.out.println("Face Detected in frame" + message.getFileName());
    copyProcessedFile(tmpSourceFile, filePath);
    file = message.getDestinationDir() + File.separator +
message.getFileName();
    copyFileToWeb(filePath, file, message.getSourceIP());
}
try {
    logWebAndStorage(message);
    deleteDir(new File(message.getSourceDir()).getParent());
} catch (Exception ex) {
    Logger.getLogger(Worker.class.getName()).log(Level.SEVERE, null, ex);
}

private static void deleteDir(String dirName) {
    String path = PROCESSED_IMG_DIR + File.separator + dirName;
    File folder = new File(path);
    if (folder.exists()) {
        folder.delete();
    }
}

private static void copyProcessedFile(String path, String target) {
    try {
        Files.copy(new File(path).toPath(), new File(target).toPath(),
StandardCopyOption.REPLACE_EXISTING);
        Files.deleteIfExists(Paths.get(path));
    } catch (IOException ex) {
        Logger.getLogger(Worker.class.getName()).log(Level.SEVERE, null, ex);
    }
}

private static Message convertFromJson(String json) {
    Gson gson = new Gson();
    return gson.fromJson(json, Message.class);
}

private static String getFileFolder(String path) {
    String[] route = path.split("/");
    return route[route.length - 1];
}

// Check if exists the folder for an image and creates it if not
private static void checkDir(String folder) {
    File dir = new File(folder + "/");
    boolean created = false;
    if (!dir.exists()) {
        dir.setWritable(true, false);
        dir.mkdirs();
    }
}

```

```

    private static void copyFileToWeb(String targetDirectory, String file, String
remoteHost) {
    String[] retrieveCommand = {
        "/bin/bash",
        "-c",
        "scp -i " + PUBLIC_RSAKEY_FILE + " " + targetDirectory + " " +
"victor@" + remoteHost + ":" + file,};
    ProcessBuilder pb = new ProcessBuilder(retrieveCommand);
    try {
        Process copyProcess = pb.start();
    } catch (IOException ex) {
        Logger.getLogger(Worker.class.getName()).log(Level.SEVERE, null, ex);
    }
}

private static void copyFileFromStorage(String sourceDirectory, String
fileName) {
    String[] retrieveCommand = {
        "/bin/bash",
        "-c",
        "scp -i " + PUBLIC_RSAKEY_FILE + " " + REMOTE_HOST + ":" +
sourceDirectory + File.separator + fileName + " " + TMP_STORAGE_DIR,};

    ProcessBuilder pb = new ProcessBuilder(retrieveCommand);
//System.out.println(Arrays.toString(retrieveCommand));
    try {
        Process copyProcess = pb.start();
        copyProcess.waitFor();
    } catch (InterruptedException ie) {
        Logger.getLogger(Worker.class.getName()).log(Level.SEVERE, null, ie);
    } catch (IOException ex) {
        Logger.getLogger(Worker.class.getName()).log(Level.SEVERE, null, ex);
    }
}

private static void logWebAndStorage(Message message) throws Exception {

    Channel channel = QueueUtil.createChannel(HOST, PORT);

    message.setWorkerID(WORKER_ID);
    Gson gson = new Gson();
    String logMessage = gson.toJson(message);

    channel.basicPublish(EXCHANGE_SEND, ROUTING_KEY_SEND, null,
logMessage.getBytes());
    System.out.println(" [*] Log Sent '" + logMessage + "'");
}
}

```

FaceDetector.java

```
public class FaceDetector {  
  
    private final static String HAAR_PATH = "/home/victor/libraries/opencv-  
3.3.1/data/haarcascades/haarcascade_frontalface_alt.xml";  
  
    public FaceDetector() {  
    }  
  
    public boolean detectFaces(String filePath) {  
        System.loadLibrary(Core.NATIVE_LIBRARY_NAME);  
        CascadeClassifier faceDetector = new CascadeClassifier(HAAR_PATH);  
        Mat image = imageToMat(filePath);  
  
        MatOfRect faces = new MatOfRect();  
        faceDetector.detectMultiScale(image, faces);  
  
        return !faces.empty();  
    }  
  
    private Mat imageToMat(String filePath) {  
        BufferedImage image;  
        Mat mat = null;  
        try {  
            image = ImageIO.read(new File(filePath));  
            mat = new Mat(image.getHeight(), image.getWidth(), CvType.CV_8UC3);  
            byte[] data = ((DataBufferByte)  
image.getRaster().getDataBuffer()).getData();  
            mat.put(0, 0, data);  
        } catch (IOException e) {  
        }  
        return mat;  
    }  
}
```

RabbitMQ

QueueInitializer.java

```
public class QueueInitializer {

    private final static String TASK_QUEUE_NAME = "task_queue";
    private final static String WEBLOG_QUEUE_NAME = "weblog_queue";
    private final static String STORELOG_QUEUE_NAME = "storelog_queue";
    private final static String USERNAME = "guest";
    private final static String PASSWORD = "guest";
    private final static String HOST = "192.168.10.30";
    private final static int PORT = 5672;
    private final static int QOS = 1;
    private final static String EXCHANGE_LOG = "logs";
    private final static String EXCHANGE_TASK = "img_process";
    private final static String ROUTING_KEY_LOG = "log_rk";
    private final static String ROUTING_KEY_TASK = "img_process_rk";

    public static void main(String[] argv) throws Exception, TimeoutException,
    IOException, InterruptedException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(HOST);
        factory.setPort(PORT);
        final Connection connection = factory.newConnection();
        final Channel channel = connection.createChannel();

        channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null);
        channel.exchangeDeclare(EXCHANGE_TASK, BuiltinExchangeType.DIRECT, true);
        channel.queueBind(TASK_QUEUE_NAME, EXCHANGE_TASK, ROUTING_KEY_TASK);

        channel.queueDeclare(WEBLOG_QUEUE_NAME, true, false, false, null);
        channel.exchangeDeclare(EXCHANGE_LOG, BuiltinExchangeType.FANOUT, true);
        channel.queueBind(WEBLOG_QUEUE_NAME, EXCHANGE_LOG, ROUTING_KEY_LOG);

        channel.queueDeclare(STORELOG_QUEUE_NAME, true, false, false, null);
        channel.exchangeDeclare(EXCHANGE_LOG, BuiltinExchangeType.FANOUT, true);
        channel.queueBind(STORELOG_QUEUE_NAME, EXCHANGE_LOG, ROUTING_KEY_LOG);

        System.out.println("Queues declared correctly");
        System.exit(0);
    }
}
```