

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA



PROYECTO FIN DE CARRERA

PATRÓN OBJETO ACTIVO: SERVIDOR MULTIMEDIA CON C#



AUTOR: Mario López Mompeán

DIRECTOR: Juan Ángel Pastor Franco

Autor	Mario López Mompeán
E-mail del autor	mario.lopez@regmurcia.com
Director	Juan Ángel Pastor Franco
E-mail del director	Juanangel.pastor@upct.es
Título del PFC	Patrón Objeto Activo: Servidor Multimedia con C#
Descriptores	
Resumen	
<p>Se diseñará una aplicación sobre Visual Studio 2003. Usará el patrón objeto activo y el lenguaje de programación C# de la plataforma .NET. Esta aplicación será un servidor de archivos multimedia, permitiendo la descarga, la subida y la reproducción de videos por parte de los clientes.</p>	
Titulación	Ingeniero Técnico de Telecomunicación, especialidad Telemática
Intensificación	
Departamento	Tecnología de la Información y las Comunicaciones
Fecha de presentación	Diciembre - 2007

Índice

Capítulo 1: Introducción.....	11
1.1. Antecedentes, contexto.....	11
1.2. Trabajo planteado, objetivos, motivaciones.....	12
1.3. Contenido y alcance de este documento.....	12
Capítulo 2: .NET Framework.....	15
2.1. Objetivos y principales características.....	15
2.2. Arquitectura.....	16
2.2.1. CLI.....	17
2.2.2. Ensamblados.....	17
2.2.3. Metadatos.....	18
2.2.4. Librería de clases.....	18
2.2.5. Seguridad.....	18
2.2.6. Manejo de la memoria.....	19
2.3. CLR: Lenguaje común en tiempo de ejecución.....	20
2.4. Biblioteca de clases .NET.....	21
2.5. .NET y JAVA.....	22
2.6. Inconvenientes de .NET.....	23
Capítulo 3: Patrón de diseño objeto activo.....	25
3.1. Ejemplo del gateway.....	25
3.2. Contexto.....	26
3.3. Problema.....	26
3.4. Solución.....	27
3.5. Estructura.....	27
3.6. Dinámica.....	30
3.7. Implementación.....	31
3.7.1. Implementar el servant.....	31
3.7.2. Implementar el proxy y los method requests.....	32
3.7.3. Implementar la activation list.....	34
3.7.4. Implementar el scheduler.....	35
3.7.5. Concretar el future y cómo acceder a él.....	37
Capítulo 4: Paquetes y librerías.....	39
4.1. Paquetes externos.....	39
4.2. Paquetes desarrollados.....	39

Capítulo 5: Implementación del patrón..... 41

5.1. Problemas encontrados.....	41
5.1.1. Referentes al lenguaje.....	41
5.1.2. Referentes al patrón.....	42
5.2. Organización del código.....	43
5.2.1. Proxy y skeleton.....	43
5.2.2. Future y method requests.....	44
5.2.3. Scheduler.....	44
5.2.4. Activation list.....	44
5.2.5. Servant.....	45
5.3. Dinámica.....	45
5.3.1. Arranque de los programas.....	45
5.3.2. Comunicaciones proxy – skeleton.....	46
5.3.3. Objeto activo.....	48
5.3.4. Comunicaciones objeto activo – proxy.....	48
5.3.5. Otros.....	52
5.4. Implementación.....	53
5.4.1. Proxy y skeleton.....	53
5.4.2. Future y method requests.....	56
5.4.3. Scheduler.....	59
5.4.4. Activation list.....	60
5.4.5. Servant.....	61

Capítulo 6: Plan de prácticas..... 63

6.1. Práctica 1: Sistema de paso de mensajes.....	64
6.1.1. Objetivos de la práctica.....	64
6.1.2. Contenidos.....	64
6.1.3. Especificación de los requisitos de la aplicación.....	65
6.1.4. Diseño.....	65
6.1.5. Implementación.....	66
6.2. Práctica 2: Sistema de paso de mensajes con sockets.....	67
6.2.1. Objetivos de la práctica.....	67
6.2.2. Contenidos.....	67
6.2.3. Especificación de los requisitos de la aplicación.....	67
6.2.4. Diseño.....	68
6.2.5. Implementación.....	70
6.3. Práctica 3: Sistema de video. Estructura básica.....	71
6.3.1. Objetivos de la práctica.....	71
6.3.2. Contenidos.....	71
6.3.3. Especificación de los requisitos de la aplicación.....	71
6.3.4. Diseño.....	72
6.3.5. Implementación.....	74

6.4. Práctica 4: Sistema de video. Aplicación final.....	75
6.4.1. Objetivos de la práctica.....	75
6.4.2. Contenidos.....	75
6.4.3. Especificación de los requisitos de la aplicación.....	75
6.4.4. Diseño.....	76
6.4.5. Implementación.....	77
Capítulo 7: Manual de usuario.....	79
7.1. Servidor.....	79
7.2. Cliente.....	80
Capítulo 8: Conclusiones y trabajos futuros.....	83
8.1. Conclusiones.....	83
8.2. Trabajos futuros.....	83
Capítulo 9: Bibliografía y referencias.....	85
Anexo 1: Glosario de términos y siglas.....	87

Índice de figuras

Capítulo 2

Figura 1: Estructura básica de CLI.....	17
Figura 2: Biblioteca de clases de .NET Framework.....	22

Capítulo 3

Figura 3: Ejemplo gateway.....	25
Figura 4: Estructura del patrón objeto activo.....	29
Figura 5: Diagrama de secuencia del patrón objeto activo.....	30

Capítulo 5

Figura 6: Estructura del skeleton.....	43
Figura 7: Estructura del scheduler.....	44
Figura 8: Estructura del servant.....	45
Figura 9: Secuencia de arranque del servidor.....	46
Figura 10: Secuencia de arranque del cliente.....	46
Figura 11: Secuencia de conexión del proxy con el skeleton.....	46
Figura 12: Secuencia de petición.....	47
Figura 13: Secuencia de extracción de petición.....	48
Figura 14: Secuencia de ejecución de la petición de visionado.....	49
Figura 15: Secuencia de ejecución de la petición de descarga.....	50
Figura 16: Secuencia de ejecución de la petición de subida.....	51
Figura 17: Secuencia de configuración.....	52
Figura 18: Interfaz del proxy.....	53
Figura 19: Interfaz del skeleton.....	54
Figura 20: Interfaz de method request.....	56
Figura 21: interfaz de la petición verVideo.....	57
Figura 22: Interfaz de la petición transferirArchivo.....	57
Figura 23: Interfaz de la petición recibirArchivo.....	57
Figura 24: Interfaz del future.....	58
Figura 25: Interfaz del scheduler.....	59
Figura 26: Interfaz de la activation list.....	60
Figura 27: Interfaz del objeto Peticion.....	60
Figura 28: Interfaz del iterador.....	61
Figura 29: Interfaz del servant.....	61

Capítulo 6

Figura 30: Estructura de la práctica 1.....	65
Figura 31: Interfaz gráfica de la práctica 1.....	66
Figura 32: Estructura del programa cliente de la práctica 2.....	68
Figura 33: Interfaz gráfica del programa cliente de la práctica 2.....	68

Figura 34: Estructura del programa servidor de la práctica 2.....	69
Figura 35: Interfaz gráfica del programa servidor de la práctica 2.....	69
Figura 36: Interfaz gráfica del programa servidor de la práctica 3.....	72
Figura 37: Panel de configuración del servidor de la práctica 3.....	72
Figura 38: Interfaz gráfica del programa cliente de la práctica 3.....	73
Figura 39: Panel de configuración del cliente de la práctica 3.....	74
Figura 40: Interfaz gráfica del programa servidor de la práctica 4.....	76
Figura 41: Interfaz gráfica del programa cliente de la práctica 4.....	76

Capítulo 7

Figura 42: Ventana principal del servidor.....	79
Figura 43: Ventana de configuración del servidor.....	80
Figura 44: Ventana principal del cliente.....	81
Figura 45: Ventana de configuración del cliente.....	82

Capítulo 1: Introducción

El objetivo principal de las comunicaciones es el intercambio de información, pero si algo ha cambiado en los últimos años es el tipo de información que solicitan los usuarios de internet. Desde la puesta en marcha de la plataforma conocida como Web 2.0, cuya propuesta es dar un protagonismo a los usuarios que antes no tenían convirtiéndolos en generadores de contenidos y dándoles la completa libertad de poder elegir la información que quieren ver, y gracias a los cada vez más grandes anchos de banda de que se disponen, se han popularizado en Internet multitud de servicios que antes se consideraban inconcebibles. Hace poco más de 10 años que se puso en marcha la internet tal como la conocemos hoy en día, pero ha sido cosa de estos últimos años en que ha habido una explosión de servicios, llevando internet a algo más que un conjunto de páginas web, correos electrónicos, servidores de noticias y demás servicios básicos. No hace demasiado que existen directorios de fotografías suministradas y organizadas por los propios usuarios, o auténticas enciclopedias en multitud de idiomas cuyos contenidos son aportados por usuarios desinteresados.

1.1. Antecedentes, contexto

Si hay un servicio que realmente ha explotado aprovechando los cada vez mayores anchos de banda y las redes sociales características de la Web2.0 ha sido el de difusión de video.

Han aparecido multitud de empresas que ofrecen este servicio y cada una tiene su propio modelo de negocio, lo que deja patente que todavía no está todo inventado pero que a la vez cada nueva empresa debe darle una vuelta más de tuerca al mercado para poder desmarcarse. Probablemente los modelos más extendidos en la actualidad sean los que cobran una cuota de mantenimiento por alojar y difundir los videos o los que hacen esto a cambio de incluir publicidad de una forma más o menos encubierta. Y al final de tantas opciones, el denominador común es el mismo, al final son los propios usuarios quienes generan el contenido y quienes deciden qué merece la pena ser visto.

Una prueba de que este es un mercado con muchas posibilidades es la compra de YouTube por parte de un gigante como Google por 1650 millones de dólares. Cifra elevada para un portal con pocos años de vida que todo lo que ofrece es que los usuarios puedan colgar sus videos y que otros los vean. Probablemente lo que más llama la atención de esta compra sea que Google ya disponía de su propio portal con los mismos servicios que YouTube pero con una diferencia: el número de usuarios. En un servicio donde son los usuarios quienes proveen de contenido, cuantos más usuarios tengas, más garantías tienes que tu negocio vaya a crecer y generar beneficios.

Estos son negocios con fines comerciales, pero la idea de la difusión de video es interesante a muchos niveles: medicina, reuniones virtuales, entretenimiento... En el ámbito universitario se podría utilizar para la educación a distancia o para ampliar los conocimientos del alumno más allá de los libros: conferencias de profesores, clases...

1.2. Trabajo planteado, objetivos, motivaciones

Dado el cada vez mayor número de posibilidades de elección donde albergar contenido multimedia para su difusión, quizás haya llegado el momento de pensar si no sería mejor utilizar una opción no tan masificada, un servicio más bien privado y que sólo llegue a aquellas personas que realmente interese que llegue.

Con esta idea en mente, podría ser interesante construir un sistema propio para este fin. En proyectos anteriores esto ya se ha hecho aunque de una forma más rudimentaria y en lenguaje Java. En este proyecto se propone llevar estos proyectos un paso más allá y además portarlos a C#. Para conseguir esto, este proyecto se va a centrar más en los medios con los que los usuarios recibirán los videos que en el fin mismo de la transmisión de video, y es por ello por lo que todo gira en torno al patrón objeto activo.

Objetivos:

- Proporcionar ejemplos de utilización del lenguaje C# y del entorno de desarrollo .NET para la docencia del área LSI.
- Proporcionar una descripción en castellano del patrón a partir del original en inglés.
- Proporcionar una implementación del patrón Objeto Activo en el lenguaje C# y documentarla de forma que sirva a los objetivos docentes del área.
- Proporcionar una aplicación multimedia en el entorno .NET que utilice el patrón antes mencionado.

1.3. Contenido y alcance del documento

Capítulo 1: Introducción, que se subdivide en tres apartados:

- **Antecedentes, contexto:** Se hace breve introducción del mercado multimedia disponible en internet y cómo ha llegado hasta aquí.
- **Trabajo planteado, objetivos, motivaciones:** Se describen los motivos y las metas que persigue este proyecto.
- **Contenido y alcance del documento:** Se describe la estructura del documento.

Capítulo 2: Tecnología .NET, se realiza un pequeño resumen de las características de la tecnología .NET.

- **Objetivos y principales características.** Detalles de la motivación para crear .NET.
- **Arquitectura.** Resumen de la arquitectura en la que se basa .NET.

Capítulo 3: Patrón objeto activo, explicación detallada de la estructura básica del patrón objeto activo.

- **Ejemplo del Gateway:** Presentación del caso teórico sobre el que se explicará el patrón.
- **Contexto:** Escenario sobre el que actúa el patrón.
- **Problema:** Resumen del problema que intenta solucionar.
- **Solución:** Se detalla cómo el patrón solucionará los problemas planteados.
- **Estructura:** Consta de varios apartados, cada uno dedicado a un componente del patrón. Detalla la estructura de cada uno.
- **Dinámica:** Explica las fases en que se divide el patrón según lo que esté haciendo.
- **Implementación:** Al igual que la estructura, dedica un apartado a la implementación de cada componente del patrón en el caso concreto del gateway.

Capítulo 4: Paquetes y librerías, este capítulo va destinado a identificar los paquetes usados en el proyecto ajenos a los de .NET y a los paquetes en que se va a organizar el código.

- **Paquetes externos:** Librerías y paquetes externos a .NET que se han incorporado al proyecto.
- **Paquetes desarrollados:** Paquetes en que se ha organizado el código.

Capítulo 5: Implementación del patrón, se divide en cuatro apartados y detalla todos los aspectos de la aplicación construida.

- **Problemas encontrados:** Resumen de los problemas encontrados antes, durante y después de la implementación del código. Se divide en los problemas debidos a la estructura del patrón y los problemas debidos al lenguaje C#.
- **Organización del código:** Se resume cómo va a estar organizado el código.
- **Dinámica:** Diagramas de secuencia del funcionamiento del patrón y de arranque de los programas.
- **Implementación:** Explicación del código.

Capítulo 6: Plan de prácticas, completo plan de prácticas para construir la aplicación en varios pasos.

- **Práctica 1: Sistema de paso de mensajes**
- **Práctica 2: Sistema de paso de mensajes con sockets**
- **Práctica 3: Sistema de video. Estructura básica**
- **Práctica 4: Sistema de video. Aplicación final**

Capítulo 7: Manual de usuario completo de los dos programas.

- **Servidor:** Configuración y puesta en marcha del programa servidor.
- **Cliente:** Configuración y puesta en marcha del programa cliente

Capítulo 8: Conclusiones y trabajos futuros, resumen de los objetivos cumplidos y posibles vías de actuación y mejora de la aplicación.

Capítulo 9: Bibliografía y referencias

Anexo 1: Glosario de términos y siglas

Capítulo 2: .NET Framework

La nueva tecnología de Microsoft ofrece soluciones a los problemas de programación actuales, como son la administración de código o la programación para Internet. Para aprovechar al máximo las características de .NET es necesario entender la arquitectura básica en la que esta implementada esta tecnología y así beneficiarse de todas las características que ofrece esta nueva plataforma.

El Framework de .NET es una infraestructura sobre la que se reúne todo un conjunto de lenguajes y servicios que simplifican enormemente el desarrollo de aplicaciones. Mediante esta herramienta se ofrece un entorno de ejecución altamente distribuido, que permite crear aplicaciones robustas y escalables. Los principales componentes de este entorno son:

- Lenguajes de compilación
- Biblioteca de clases de .NET
- CLR (Common Language Runtime)

Actualmente, el Framework de .NET es una plataforma no incluida en los diferentes sistemas operativos distribuidos por Microsoft, por lo que es necesaria su instalación previa a la ejecución de programas creados mediante .NET. El Framework se puede descargar gratuitamente desde la web oficial de Microsoft (ver link de descarga en el capítulo de bibliografía [1]).

.NET Framework soporta múltiples lenguajes de programación y aunque cada lenguaje tiene sus características propias, es posible desarrollar cualquier tipo de aplicación con cualquiera de estos lenguajes. Existen más de 30 lenguajes adaptados a .NET, desde los más conocidos como C# (C Sharp), Visual Basic o C++ hasta otros lenguajes menos conocidos como Perl o Cobol.

2.1. Objetivos y principales características

La tecnología .NET se planteó los siguientes objetivos:

- **Interoperabilidad.** La interacción entre nuevas y viejas aplicaciones eran una exigencia, por lo que .NET debía ofrecer medios para poder interactuar con programas que se ejecutan de forma externa al entorno .NET.
- **Un entorno común de ejecución.** Los lenguajes soportados se compilan en un lenguaje intermedio conocido como Common Intermediate Language (CIL) o Microsoft Intermediate Language (MSIL). Sin embargo, este código no es interpretado durante la ejecución, sino que se recompila durante la fase de ejecución mediante compiladores “just-in-time” en código nativo. Este modo de funcionamiento se conoce como

Common Language Infrastructure (CLI) y en el caso particular de Microsoft ha recibido el nombre de Common Language Runtime (CLR).

- **Independencia del lenguaje.** La independencia se consigue mediante la especificación Common Type System (CTS). En ella se definen todos los tipos de datos posibles, todos los operadores y cómo actúan unos sobre otros en caso de ser posible esta interacción. Por esta razón, .NET soporta desarrollos en varios lenguajes.
- **Librería de clases básicas.** Esta librería de clases forma parte de la plataforma y ofrece todas sus funciones a todos los lenguajes que la usan. Proporciona clases que encapsulan funciones de uso común, como lectura y escritura de ficheros, tratamiento de imágenes, comunicación con bases de datos y manipulación de documentos XML.
- **Instalación simple** de los programas creados desde la plataforma. La instalación de cualquier software debe realizarse cuidadosamente para que asegure que no interfiere con software instalado previamente y que cumple los requisitos de seguridad adecuados. En .NET se incluyen herramientas que ayudan a cumplir estos requisitos.
- **Seguridad.** Evitar vulnerabilidades debe ser una prioridad, por ejemplo las relaciones con desbordamientos de buffer son demasiado peligrosas y fácilmente aprovechar por código malicioso, por lo que se deben evitar a toda costa. Por ello .NET proporciona un modelo de seguridad común para todas las aplicaciones.
- **Portabilidad.** Una de las metas de la plataforma es permanecer independiente de la plataforma desde la que se ejecuta. Esto significa que una aplicación escrita en .NET debería poder ejecutarse en cualquier máquina donde esté implementada la plataforma. En el momento de escribir este documento, solamente los sistemas operativos Windows y Windows CE soportan plenamente todas las funciones de .NET. Microsoft también ha liberado implementaciones que funcionen sobre algunos sistemas basados en Unix como FreeBSD y Mac OSX bajo licencias que sólo permiten su uso para propósitos educativos únicamente. Además Microsoft envió la especificación del CLI, el lenguaje C# y el conversor C++/CLI para convertirlos en estándares abiertos. Esto hace posible que se puedan realizar implementaciones compatibles de la plataforma en cualquier sistema operativo o sistema que se desee.

2.2. Arquitectura

En este apartado se van a detallar los componentes más importantes de la arquitectura de .NET. Se dejarán un par de casos para apartados posteriores por considerarlos lo suficientemente importantes como para entrar en profundidad.

2.2.1. CLI

El núcleo de .NET respeta la estructura tipo CLI. El propósito de CLI es proporcionar un entorno independiente del lenguaje de programación para el desarrollo y ejecución de aplicaciones, incluyendo funciones para el tratamiento de excepciones, recolección de basura, seguridad e interoperabilidad. La implementación que utiliza Microsoft de la estructura CLI recibe el nombre de CLR. La siguiente figura muestra la estructura básica de CLI:

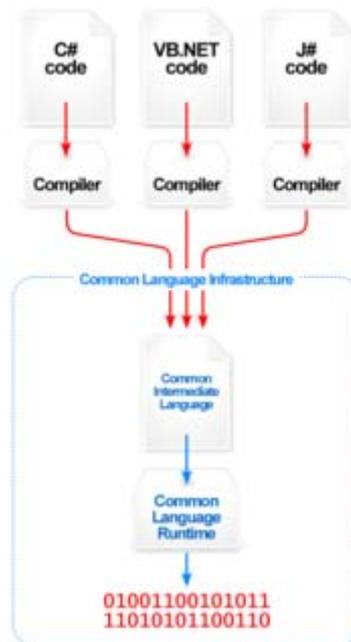


Figura 1: Estructura básica de CLI

2.2.2. Ensamblados

Uno de los mayores problemas de las aplicaciones actuales es que en muchos casos tienen que tratar con diferentes archivos binarios (DLL's), elementos de registro, conectividad abierta a bases de datos (ODBC), etc.

Para solucionarlo el Framework de .NET maneja un nuevo concepto denominado ensamblado. Los ensamblados son ficheros con forma de EXE o DLL que contienen toda la funcionalidad de la aplicación de forma encapsulada. Por tanto la solución al problema puede ser tan fácil como copiar todos los ensamblados en el directorio de la aplicación.

Con los ensamblados ya no es necesario registrar los componentes de la aplicación. Esto se debe a que los ensamblados almacenan dentro de sí mismos toda la información necesaria en lo que se denomina el manifiesto del ensamblado. El manifiesto recoge todos los métodos y

propiedades en forma de meta-datos junto con otra información descriptiva, como permisos, dependencias, etc.

Para gestionar el uso que hacen las aplicaciones de los ensamblados .NET utiliza la llamada caché global de ensamblados (GAC, Global Assembly Cache). Así, .NET Framework puede albergar en el GAC los ensamblados que puedan ser usados por varias aplicaciones e incluso distintas versiones de un mismo ensamblado, algo que no era posible con el anterior modelo COM.

2.2.3. Metadatos

Todo código CIL es autosuficiente gracias a los metadatos. El CLR comprueba en los metadatos que se está llamando al método correcto. Normalmente estos metadatos los genera el compilador, aunque es posible que el programador pueda crear los suyos propios a través de atributos personalizables.

2.2.4. Librería de clases

Existen dos librerías de clases. Por un lado se encuentra la librería básica, que incluye todas las clases de todos los lenguajes soportados. Esta se conoce como Base Class Library (BCL). Es más grande que otras librerías de este tipo, pero también incluye muchas más funciones en un solo paquete. A veces esta librería se confunde con la librería que incluye además de las básicas, todas las clases del espacio de nombres de "Microsoft." [2]. Este conjunto de BCL y el espacio de nombres de Microsoft se llama Framework Class Library (FCL).

En un apartado posterior se entrará en detalle en este punto.

2.2.5. Seguridad

.NET tiene su propio mecanismo de seguridad. Tiene dos características principales:

- Code Access Security (CAS). Se basa en cierta información, *evidence*, asociada a un ensamblado específico. Esta información determinará la fuente desde la que se ejecuta el ensamblado, si está instalado en la propia máquina o si se ha descargado desde una intranet o desde internet. A partir de esta información, CAS determina los permisos permitidos al código. Cuando se hace una llamada que requiere permisos específicos, desde el CLR se realiza una búsqueda en cada ensamblado que tenga un método en la cola de llamadas, si ninguno de estos ensamblados tiene permiso para realizar la llamada se lanza una excepción de seguridad.

- Validación y verificación. Cuando se carga un ensamblado, el CLR realiza varias comprobaciones. Dos de ellas se conocen como validación y verificación. Durante la fase de validación se comprueba que el ensamblado contiene metadatos válidos y que las tablas internas están correctas, es decir, comprueba que no ha sido modificado. La verificación busca si el código puede realizar operaciones no seguras. La comprobación es muy restrictiva y puede dar lugar a que código seguro sea catalogado como inseguro. El código marcado como inseguro sólo se ejecuta si se otorga permiso al ensamblado para que se salte la fase de verificación. El código que se instala en la máquina desde la que se ejecuta normalmente permite la ejecución de código inseguro.

Como punto final, otra característica que incorpora .NET es la posibilidad de otorgar permisos según por dominios. Así, no toda la aplicación tiene los mismos permisos en todas las partes. Estos dominios se conocen como *appdomains*. Esto mejora la tolerancia a fallos de la aplicación y evita que un fallo en un dominio afecte al resto. Estos dominios se han de establecer por el programador.

2.2.6. Manejo de la memoria

En el apartado de manejo de la memoria, .NET funciona de una forma similar de JAVA. Se libera al programador de las tareas de manejo de la memoria, automatizando los procesos de reserva y liberación de ésta. El proceso de reserva de memoria lo realiza el CLR. Este componente guarda un espacio y va asignando memoria a los objetos que van creándose. Los objetos se colocan uno al lado del otro en la memoria, sin dejar espacios entre ellos. Cuando este espacio se llena se reserva más. Mientras exista una referencia a cualquier objeto en el espacio reservado por el CLR, se considera que el objeto está en uso por él. Cuando se eliminan todas las referencias a un objeto y no es posible usarlo ni acceder a él de ninguna manera, se vuelve basura. Sin embargo, sigue ocupando el espacio de memoria que le fue asignado. Para liberar la memoria utilizada por datos basura se utiliza el recolector de basura.

Este recolector se ejecuta periódicamente en un hilo separado al de la aplicación. Se activa sólo cuando una cierta cantidad de memoria ha sido usada o si se está llenando la memoria que reservó el CLR. Como el momento en el que va a entrar en funcionamiento el recolector no está a priori determinado, se denomina que es ejecuta de una forma no determinista.

Toda aplicación .NET tiene una serie de punteros mantenidos por el CLR que apuntan a cada objeto en memoria. Estos incluyen referencias a objetos estáticos y a objetos definidos como variables locales, parámetros de métodos en uso u objetos referenciados por los registros de la CPU. Cuando el recolector entra en funcionamiento, pausa la aplicación y por cada objeto referenciado por un puntero, recorre todos los objetos que se pueden llegar a través de él. También es capaz de recorrer objetos encapsulados dentro de otros objetos gracias a los metadatos y recorrer los que están relacionados con ellos. Mientras recorre todos los objetos a los que puede llegar utilizando los punteros del CLR y todas sus ramificaciones también los marcar como *Reachable* (alcanzable). Una vez hecho esto, recorre todos los objetos que hay en

memoria marcando como basura aquellos que no estén marcados como alcanzables. Esta es la fase de marcado. Los objetos marcados como basura se tratan a partir de aquí como espacios de memoria libres. Pero como a los nuevos objetos se les asigna memoria al lado de los últimos que se crearon, estos espacios de memoria libre no son aprovechables tal cual. Es necesario un proceso de compactación de la memoria para que todos los objetos estén juntos de nuevo.

Para mejorar la eficiencia, el recolector de basura es generacional. A todos los objetos se les asigna una generación y en función de ésta, son recolectados más o menos a menudo. Los objetos nuevos tienen generación 0. Los que sobreviven a un recolector de basura, se les asigna la generación 1 y los que sobreviven a un segundo recolector son los de generación 2, no hay más niveles. Los archivos de mayor generación son recolectados menos a menudo, ya que son objetos que tienen un tiempo de vida más largo. Así se acelera el funcionamiento del recolector al eliminar los objetos más viejos de una ejecución, pues se tienen que comprobar, marcar y compactar menos objetos.

2.3. CLR: Lenguaje común en tiempo de ejecución

El CLR es el verdadero núcleo del Framework de .NET, ya que es el entorno de ejecución en el que se cargan las aplicaciones desarrolladas en los distintos lenguajes, ampliando el conjunto de servicios que ofrece el sistema operativo.

Se compone principalmente de cuatro partes:

- Common Type System (CTS). En términos generales, describe el tratamiento de todos los tipos de datos que deben soportar todos los lenguajes .NET. Su función es establecer una integración multilenguaje segura y de alto rendimiento, proporcionar un modelo orientado a objetos y definir reglas para la interacción entre distintos lenguajes de programación.
- Common Language Specification (CLS). Conjunto de tipos y operaciones que deben soportar todos los lenguajes compatibles con .NET.
- Just-In-Time Compiler (JIT). Compilador en tiempo de ejecución, transforma el código en MSIL a código nativo de la máquina.
- Virtual Execution System (VES). Proporciona lo necesario para manejar código en MSIL.

La herramienta de desarrollo compila el código fuente de cualquiera de los lenguajes soportados por .NET en un mismo código, denominado código intermedio (MSIL, Microsoft Intermediate Language). Para generar dicho código el compilador se basa en el Common Language Specification (CLS) que determina las reglas necesarias para crear código MSIL compatible con el CLR.

De esta forma, indistintamente de la herramienta de desarrollo utilizada y del lenguaje elegido, el código generado es siempre el mismo, ya que el MSIL es el único lenguaje que entiende directamente el CLR. Este código es transparente al desarrollo de la aplicación ya que lo genera automáticamente el compilador.

Sin embargo, el código generado en MSIL no es código máquina y por tanto no puede ejecutarse directamente. Se necesita un segundo paso en el que una herramienta denominada compilador JIT (Just-In-Time) genera el código máquina real que se ejecuta en la plataforma que tenga la computadora.

Así se consigue con .NET cierta independencia de la plataforma, ya que cada plataforma puede tener su compilador JIT y crear su propio código máquina a partir del código MSIL.

La compilación JIT la realiza el CLR a medida que se invocan los métodos en el programa y, el código ejecutable obtenido, se almacena en la memoria caché de la computadora, siendo recompilado sólo cuando se produce algún cambio en el código fuente.

2.4. Biblioteca de clases de .NET

Cuando se está programando una aplicación muchas veces se necesitan realizar acciones como manipulación de archivos, acceso a datos, conocer el estado del sistema, implementar seguridad, etc. El Framework organiza toda la funcionalidad del sistema operativo en un espacio de nombres jerárquico de forma que a la hora de programar resulta bastante sencillo encontrar lo que se necesita.

Para ello, el Framework posee un sistema de tipos universal, denominado Common Type System (CTS). Este sistema permite que el programador pueda interactuar los tipos que se incluyen en el propio Framework (biblioteca de clases de .Net) con los creados por él mismo (clases). De esta forma se aprovechan las ventajas propias de la programación orientada a objetos, como la herencia de clases predefinidas para crear nuevas clases, o el polimorfismo de clases para modificar o ampliar funcionalidades de clases ya existentes.

La biblioteca de clases de .NET incluye, entre otros, tres componentes clave:

- ASP.NET para construir aplicaciones y servicios Web.
- Windows Forms para desarrollar interfaces de usuario.
- ADO.NET para conectar las aplicaciones a bases de datos.

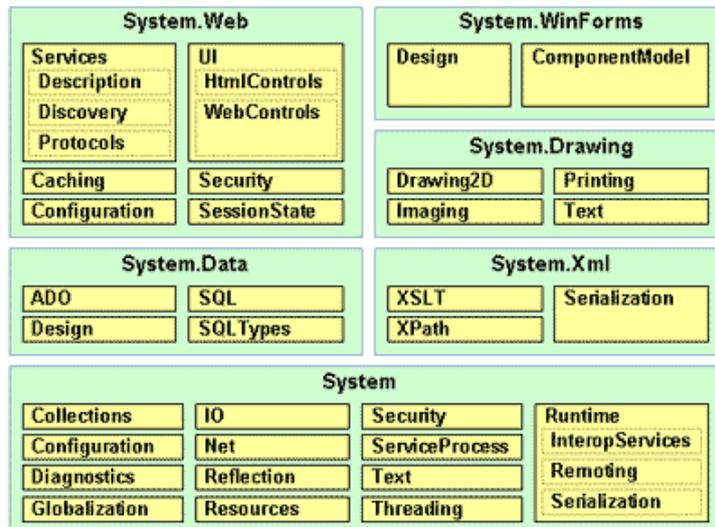


Figura 2: Biblioteca de clases de .NET Framework

La forma de organizar la biblioteca de clases de .NET dentro del código es a través de los espacios de nombres (namespaces), donde cada clase está organizada en espacios de nombres según su funcionalidad. Por ejemplo, para manejar ficheros se utiliza el espacio de nombres System.IO y si lo que se quiere es obtener información de una fuente de datos se utilizará el espacio de nombres System.Data.

La principal ventaja de los espacios de nombres de .NET es que de esta forma se tiene toda la biblioteca de clases de .NET centralizada bajo el mismo espacio de nombres (System). Además, desde cualquier lenguaje se usa la misma sintaxis de invocación, ya que a todos los lenguajes se aplica la misma biblioteca de clases.

2.5. .NET y Java

La tecnología .NET y Java tienen muchos puntos en común[3]:

- Utilizan máquinas virtuales que ocultan los detalles del hardware sobre el que funcionan.
- Usan código intermedio. Byte-codes en el caso de Java, MSIL en el caso de .NET.
- El código MSIL es siempre compilado antes de la ejecución. Los byte-codes además de compilado, puede ser interpretado.
- Proporcionan grandes librerías de clases con todas las necesidades habituales ya cubiertas, incluida la seguridad.
- El espacio de nombres de .NET es bastante similar al sistema de paquetes de la especificación de la API Java EE tanto en estilo como en invocación.

Hasta aquí las similitudes. Las diferencias se pueden resumir en varios puntos:

- .NET sólo está completo para Windows, existe una implementación parcial en Linux y MAC OSX. Java está completamente soportado en estas tres plataformas, además de muchas más otras.
- .NET tiene soporte nativo para múltiples lenguajes, mientras que en JAVA esto fue añadido a posteriori y su uso es mínimo.
- Java está bajo licencias de libre distribución y es de código abierto, mientras que .NET está bajo licencias que restringen su uso.

2.6. Inconvenientes de .NET

Procesos como la recolección de basura de .NET o la administración de código introducen factores de sobrecarga que repercuten en la demanda de más requisitos del sistema.

El código administrado proporciona una mayor velocidad de desarrollo y mayor seguridad de que el código sea bueno. En contrapartida el consumo de recursos durante la ejecución es mucho mayor, aunque con los procesadores actuales esto cada vez es menos inconveniente.

El nivel de administración del código dependerá en gran medida del lenguaje que utilicemos para programar. Por ejemplo, mientras que Visual Basic .Net es un lenguaje totalmente administrado, C# permite la administración de código de forma manual, siendo por defecto también un lenguaje administrado. Mientras que C++ es un lenguaje no administrado en el que se tiene un control mucho mayor del uso de la memoria que hace la aplicación.

Capítulo 3: Patrón objeto activo

El patrón objeto activo[4][5] se engloba en los patrones de concurrencia. Tiene por objetivo dar orden y mejorar la concurrencia en el acceso a un objeto desacoplando la invocación de un método de este objeto de su ejecución. De esta forma se mejora la concurrencia, pues varios hilos pueden acceder al objeto intercaladamente, y la sincronización, pues sólo uno de ellos actúa sobre el objeto cada vez. A este patrón también se le conoce como objeto concurrente o en inglés active object o concurrent object.

Tratar de explicar un patrón de diseño sin dar un ejemplo de uso para comprender el problema que intenta dar solución sería una labor compleja, por lo que se utilizará un ejemplo a lo largo de la explicación del patrón. Este ejemplo consistirá de un gateway en un sistema de paso de mensajes. Mediante este ejemplo será bastante más sencillo comprender el patrón y el por qué tiene la estructura que tiene.

Antes de empezar, hay que aclarar que un gateway cumple la función de interconexión entre diferentes dispositivos o redes. Por lo tanto, el objeto activo será este gateway, mientras que los dispositivos o máquinas conectadas a él tendrán la función de clientes.

3.1. Ejemplo: gateway

Este gateway de comunicaciones permite la cooperación entre todos los componentes sin que haya dependencias directas entre ellos. Es un sistema distribuido, por lo que el gateway debe ser capaz de llevar los mensajes desde los proveedores hasta tantas máquinas destino como sean necesarias. En la figura inferior se muestra una posible estructura:

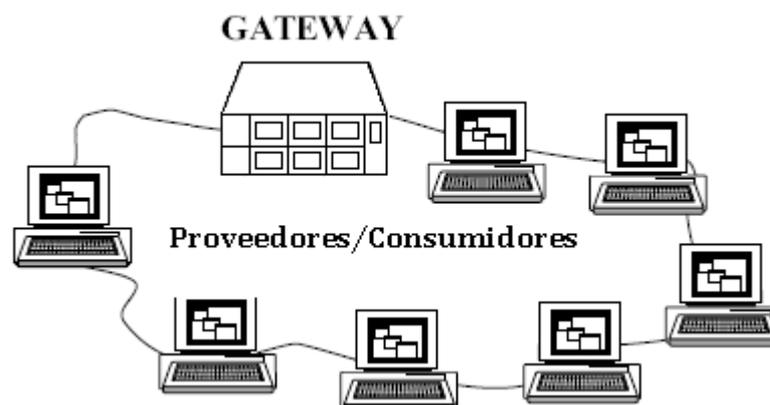


Figura 3: Ejemplo gateway

Se va a suponer que todas las partes se comunican entre sí mediante el protocolo TCP aprovechando que es un protocolo orientado a la conexión. Por tanto, el gateway se encontrará con los problemas de control de flujo provenientes de la capa de transporte TCP a

la hora de enviar datos y deberá actuar en consecuencia. Para más información, TCP usa el control de flujo para asegurar que fuentes de información muy rápidas o el propio gateway no saturan consumidores lentos o congestionan redes incapaces de almacenar y procesar los paquetes. Por ello, para mejorar la calidad de servicio para todos los componentes, el gateway no deberá bloquearse si el control de flujo del protocolo TCP entra en funcionamiento. Además el gateway debe poder escalarse eficientemente con el incremento de proveedores y consumidores.

Uno de los mejores métodos para evitar el bloqueo del gateway y mejorar su rendimiento es añadir concurrencia al diseño del gateway, por ejemplo asociar cada conexión TCP a un hilo de control diferente. Esto consigue que, aunque los hilos de las conexiones TCP afectadas por el control de flujo se encuentren bloqueados, el resto de hilos siguen funcionando con normalidad. No obstante es necesario programar los hilos de ejecución del gateway y cómo estos van a interactuar con los de los proveedores y los de los consumidores.

3.2. Contexto

Clientes que acceden a objetos ejecutándose en hilos de control diferentes.

Esto es, máquinas que generan mensajes que han de llegar a otras máquinas pasando por el gateway.

3.3. Problema

Muchas aplicaciones intentan dar mejor calidad de servicio permitiendo que se conecten varios clientes simultáneamente. En vez de usar un único objeto pasivo que ejecuta sus métodos en el hilo de ejecución del cliente que lo invocó, un objeto concurrente tiene su propio hilo. Sin embargo, aunque los clientes se atienden concurrentemente, es necesario sincronizar los accesos a este objeto en caso de que pudiera ser modificado por varios de estos clientes a la vez.

Por tanto, el patrón intenta solucionar tres problemas:

- El acceso a este objeto por parte de los clientes no debería bloquear ni a los clientes ni al servidor de ningún modo para no degradar la calidad de servicio de todos los participantes.
- Simplificar el acceso a los objetos compartidos, de forma que toda la sincronización necesaria sea totalmente transparente para el cliente.
- Hacer un diseño donde el software equilibre los paralelismos entre hardware y software.

3.4. Solución

Por cada objeto afectado por los tres problemas anteriores, habrá que desacoplar la invocación de los métodos que afecten al objeto de su ejecución. La invocación tiene lugar en el hilo de ejecución del cliente, mientras que la ejecución lo hará en un hilo distinto. Además, el cliente invocará estos métodos como si fueran cualquier otro.

Entrarán en juego otros dos patrones de diseño: Proxy y Servant. El primero representará la interfaz del objeto activo y el segundo proporcionará su implementación. El proxy y el servant estarán en hilos de control diferentes para que la invocación y la ejecución puedan ocurrir concurrentemente. Concretamente, el proxy correrá en el hilo del cliente y el servant lo hará en otro distinto.

El proxy se encargará de transformar la invocación del cliente en una petición de métodos (method request), que será almacenada en una lista de activación (activation list) por un organizador (scheduler). El scheduler tendrá un hilo propio encargado de desencolar peticiones y de provocar su ejecución, por lo que el servant correrá en el hilo del scheduler. Los clientes obtienen el resultado de la petición a través del future que devolvió el proxy durante la invocación.

3.5. Estructura

El diseño básico del patrón cuenta con seis componentes: proxy, method request, scheduler, activation list, servant y future.

Un proxy proporciona la interfaz que permite a los clientes invocar los métodos públicos del objeto activo, dicho de otra forma, se encarga de ocultar toda la implementación del objeto activo al cliente. Reside en el hilo del cliente.

Cuando un cliente invoca un método definido en el proxy, se dispara la construcción de objeto tipo method request. Este objeto contiene toda la información necesaria para la ejecución del método y devolver el resultado al cliente. El method request define un interfaz para la ejecución de los métodos de un objeto activo. También contiene la sincronización requerida para determinar cuándo puede ejecutarse una petición y cuándo no. Se implementa un method request distinto (concrete method request) por cada método disponible al público por parte del proxy.

Clase Proxy	Responsabilidad Define el interfaz del objeto a los clientes.
Colaboradores Method Request Scheduler Future	Crea el Method Request correspondiente. Corre en el hilo del cliente

Clase Method Request	Responsabilidad Representa la llamada al método en el objeto activo.	Clase Concrete Method Request	Responsabilidad Implementa la representación de la llamada de un método.
Colaboradores Servant Future	Proporciona métodos para la sincronización cuando una petición pueda ser ejecutada.	Colaboradores Servant Future	Implementa métodos de guarda.

El proxy crea un concrete method request durante la invocación de uno de sus métodos por parte de los clientes y lo inserta en la activation list. Esta lista acumula todas las peticiones realizadas y pendientes de ejecución, además de decidir qué peticiones pueden ejecutarse. La activation list es la encargada de desacoplar el hilo del cliente donde reside el proxy del hilo donde el servant ejecuta la petición. El estado interno de la lista debe estar programado de tal forma que esté protegido contra accesos concurrentes que pudieran modificarla simultáneamente.

El scheduler corre en un hilo diferente del de los clientes, normalmente en el del propio objeto activo. Decide qué petición se ejecutará a continuación. Esta decisión se puede tomar según el criterio que se programe, por ejemplo en el orden en que fueron encoladas las peticiones o según algunas propiedades de las propias peticiones (tiempo necesario para su ejecución por ejemplo). El scheduler puede hacer esto gracias a los métodos de guarda del method request y a la información que contiene para su ejecución. El scheduler usa la activation list para organizar los method requests pendientes de ejecución. Los method requests los inserta el proxy cuando el cliente invoca un método.

Clase Activation List	Responsabilidad Almacena method requests pendientes de ejecución. El scheduler añade los method requests a petición del proxy y los extrae cuando sea posible ejecutarlos.	Clase Scheduler	Responsabilidad Extrae method requests de la activación list. Añade method requests cuando el proxy lo solicite. Se ejecuta en el hilo del objeto activo.
Colaboradores		Colaboradores Activation List Method Request	

Un servant define el comportamiento y el estado que se modela como objeto activo. Los métodos que el servant implementa se corresponden con aquellos a los que da interfaz el proxy y los method request que el proxy crea. También puede contener mecanismos que pueden usar los method request para sus métodos de guarda. Se invocan los métodos del servant cuando el scheduler lo solicita a través de los method requests, por lo que se ejecuta en el hilo del scheduler.

Cuando un cliente invoca un método en el proxy, éste le devuelve un future. Con este future el cliente es capaz de encontrar el resultado de su petición cuando esté disponible. Cada future reserva el espacio necesario para almacenar el resultado. El cliente puede comprobar si está disponible el resultado bien bloqueándose hasta que lo recibe o realizando sondeos.

Clase Servant	Responsabilidad Implementa el objeto activo.	Clase Future	Responsabilidad Almacena el resultado de una petición al objeto activo.
Colaboradores	Se ejecuta en el hilo del scheduler, el hilo del objeto activo.	Colaboradores	Da un punto de encuentro entre el cliente y el resultado.

El diagrama de clases al que responde el patrón es el siguiente:

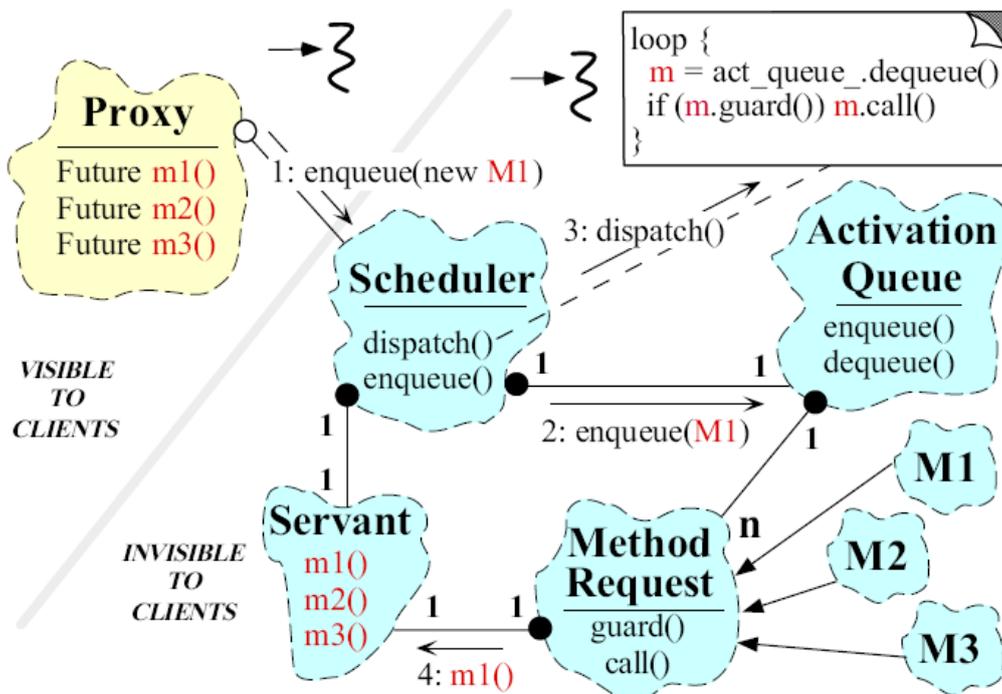


Figura 4: Estructura del patrón objeto activo

3.6. Dinámica

El patrón se divide en tres fases:

1. Creación de un Method Request y su encolamiento. El cliente invoca el método en el proxy. Esto provoca la creación de un method request, que guarda lo necesario para la ejecución de la petición. El proxy envía al scheduler el method request creado para que lo encole en la activation list. Si la petición espera un resultado, el proxy le devuelve al cliente un future donde podrá encontrarlo cuando esté disponible. Si el método no devuelve ningún resultado, no se le devuelve al cliente ningún future.
2. Ejecución del Method Request. El scheduler monitoriza su activation list y determina qué method request puede ser ejecutado a continuación a través de sus métodos de guarda. Para ello tiene un hilo propio corriendo continuamente. Cuando un method request va a ser ejecutado, el scheduler lo saca de la activation list y se lo pasa al servant para que ejecute el método correspondiente. Mientras se ejecuta este método se puede acceder al estado del servant y crear el resultado si se requiere.
3. Finalización. En esta fase se almacena el resultado, si lo hay, en el future y el scheduler vuelve a monitorizar la activation list buscando method requests que puedan ser ejecutados. Si se esperaba un resultado, el cliente puede encontrarlo a través del future. Normalmente cualquier cliente que pueda encontrarse con el future podrá encontrarse con el resultado. Una vez el cliente se ha encontrado con el resultado, tanto el method request como el future se pueden borrar explícitamente o mediante el recolector de basura cuando ya no tengan referencias que les referencien.

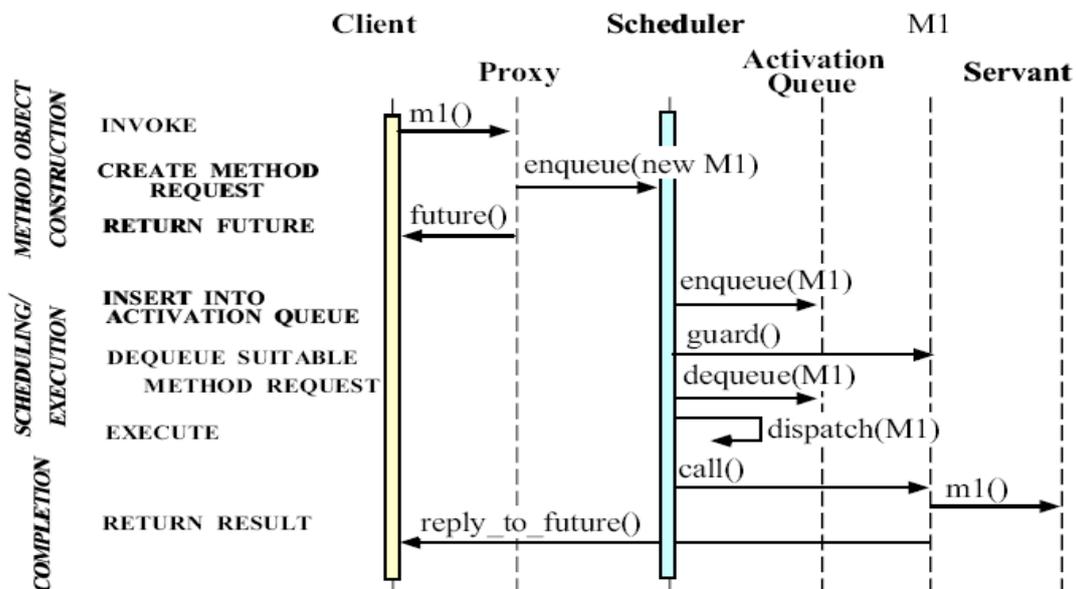


Figura 5: Diagrama de secuencia del patrón objeto activo

3.7. Implementación

Se recuperará el ejemplo del gateway durante este apartado para dar una mejor explicación de cómo implementar este patrón. La implementación del patrón se puede realizar en los cinco pasos descritos a continuación.

3.7.1. Implementar el servant

El servant define el comportamiento y el estado que va a ser modelado como un objeto activo. Además el servant puede contener métodos que podrán ser utilizados por los method requests para sus propios métodos de guarda.

En el ejemplo del gateway, el servant es una cola de mensajes pendientes de ser transmitidos a los consumidores. Por cada consumidor remoto, existe un “consumer handler” que contiene una conexión TCP que apunta al proceso de un consumidor. Además este handler contiene una cola mensajes modelada como un objeto activo e implementado con un MQ_Servant. Cada handler almacena mensajes procedentes de los proveedores al gateway mientras esperan ser transmitidos al consumidor remoto. La interfaz más básica del MQ_Servant se muestra a continuación en lenguaje C#:

```
class MQ_Servant{
    //Constructor.
    public MQ_Servant (int mq_size);
    //Métodos para encolar y desencolar mensajes.
    void put_i (string msg);
    string get_i ();
    //Métodos en los que los method request basarán los métodos de guarda
    bool empty_i ();
    bool full_i ();
}
```

Los métodos put() y get() implementan la inserción y la extracción de mensajes de la cola respectivamente. El servant define los métodos empty() y full() que distinguen tres estados internos del servant: vacío, lleno o ninguno de los dos. Con ellos puede determinarse cuándo se puede llamar a los métodos put() y get(), y por tanto ejecutarse los method request.

Como en este caso, la sincronización que protege secciones compartidas servant no debe estar implementada dentro del servant. Éste sólo debe proveer de los medios necesarios para realizarla mostrando su estado interno a los method request, que serán los encargados de implementar los mecanismos de sincronización necesarios. De esta forma se evitan problemas

si se necesitase crear sub-clases del servant que requirieran políticas diferentes de sincronización.

3.7.2. Implementar el proxy y los method requests

El proxy proporciona a los clientes un interfaz con el que poder hacer llamadas a los métodos del servant. Por cada invocación del cliente, el proxy crea un method request con toda la información necesaria para su ejecución, que suele componerse con varios parámetros, una referencia al servant en el que se ejecutará, un future donde almacenar el resultado y el código que ejecuta el method request.

En el gateway, la clase MQ_Proxy proporciona un interfaz a la clase MQ_Servant definida en el paso 1.

```
class MQ_Proxy{
    //Parámetros
    private int MAX_SIZE = 100; //Número máximo de mensajes en cola
    private MQ_Servant servant;
    private MQ_Scheduler scheduler;
    //Constructor.
    public MQ_Proxy (int size){
        servant = new MQ_Servant(size);
        scheduler = new MQ_Scheduler(size);
    }
    //Métodos para encolar y desencolar mensajes.
    void put (string msg){
        Method_Request mr = new Put(servant, msg);
        scheduler.enqueue(mr);
    }
    future get (){
        future resultado;
        Method_Request mr = new Get(servant, resultado);
        scheduler.enqueue(mr);
        return resultado;
    }
}
```

Cada método del MQ_Proxy transforma una invocación en un method request y pasa la petición al scheduler que se encargará de encolarla. El acceso al proxy no está limitado a un único cliente, además pueden acceder concurrentemente.

La clase Method_Request es el interfaz que usarán todas las implementaciones de los method requests, hay una por método que se ofrece en el proxy. Proporciona al scheduler métodos con los que realizar la sincronización y provocar su ejecución. Normalmente esto es mediante un método can_run() que devuelve un valor booleano si es posible ejecutarlo o no, y un método run() que contiene el código para realizar la petición.

```
abstract class Method_Request{
    public bool can_run();
    public void run();
}
```

Los métodos son implementados en sus subclases, por lo que necesitamos dos: Get y Put.

```
class Put : Method_Request{
    private MQ_Servant servant;
    private string mensaje;

    //Constructor
    public Put(MQ_Servant servant, string mensaje){
        this.servant = servant;
        this.mensaje = mensaje;
    }
    //Método de guarda
    bool can_run(){
        return !servant.full();
    }
    //Método con el código para la ejecución
    void run(){
        servant.Put(mensaje);
    }
}
```

Se puede observar cómo se usa el método full() del servant para realizar la sincronización. Cuando la petición puede ser ejecutada, el scheduler llama al método run(), que hace que se introduzca un mensaje en la cola de mensajes del servant. Éste método no requiere de más sincronización ya que es el scheduler el que se encarga de todo con los métodos de guarda de los method requests.

```

class Get : Method_Request{
    private MQ_Servant servant;
    private future resultado;

    //Constructor
    public Get(MQ_Servant servant, future resultado){
        this.servant = servant;
        this.resultado = resultado;
    }
    //Método de guarda
    bool can_run(){
        return !servant.full();
    }
    //Método con el código para la ejecución
    void run(){
        resultado = servant.Get();
    }
}

```

Lo único que cambia con respecto a la petición Put es que se devuelve un resultado y se guarda en un future. Como el cliente tiene la misma referencia, al actualizar el future con el resultado, el cliente tiene acceso a él inmediatamente.

3.7.3. Implementar la activation list

Cada method request se encola en la activation list. Esta lista se puede implementar como con venga, siendo lo más típico hacerlo como una lista enlazada y que proporcione iteradores con los que el scheduler puede extraer e introducir elementos. Se suele diseñar usando patrones de control de la concurrencia, como el del objeto Monitor. Si se combina con temporizadores, es posible delimitar el tiempo que se puede estar esperando a que una operación se realice.

```

class ActivationList{
    //Constructor
    public ActivationList();

    //Método para añadir method requests
    public void enqueue(Method_Request CMR);

    //Método para extraer method requests
    public void dequeue(Method_Request CMR);
}

```

Es posible llamar a los métodos enqueue y dequeue simultáneamente ya que estos incorporan los mecanismos necesarios para que no se produzcan inconsistencias y no se corrompa la lista.

En el gateway los clientes actúan como proveedores añadiendo method requests mientras que el scheduler hace de consumidor extrayéndolos cuando sea posible y ejecutando su método run().

3.7.4. Implementar el scheduler

El scheduler mantiene la activation list y ejecuta los method request cuando sea posible. La interfaz del scheduler suele proporcionar un método al proxy para añadir method requests a la lista y otro método para extraerlos y ejecutarlos.

```
class MQ_Scheduler{
    private ActivationList actList;

    //Constructor
    public MQ_Scheduler(int size);

    //Método para añadir method requests
    public void enqueue(Method_Request CMR){
        actList.enqueue(CMR);
    }

    //Método para extraer method requests
    public Dispatch();
}
```

El scheduler crea un hilo sólo para ejecutar el método Dispatch. A petición de los clientes, el proxy encola method requets en la activation list mientras que el scheduler monitoriza la lista desde otro hilo, el del método Dispatch. Desde éste hilo, se comprueban los métodos de guarda de los method requests y se ejecuta la petición cuando sea posible.

Se lanzaría el nuevo desde el mismo constructor del scheduler:

```
class MQ_Scheduler{
    private ActivationList actList;

    //Constructor
    public MQ_Scheduler(int size){
        actList = new ActivationList(size);
        Thread bucle = new Thread(new ThreadStart(Dispatch));
        bucle.Start();
    }
}
```

La implementación del método Dispatch determina en qué orden pueden ejecutarse los métodos Put y Get del servant mediante los métodos full() y empty() de éste.

```
public Dispatch(){
    while(true){
        if(!actList.numpet = 0){
            IEnumerator enume = actList.tope.GetEnumerator();
            Method_Request peticion;

            for(enume.Reset(); enume.MoveNext());{
                peticion = ((ActivationList.Peticion)enume.Current).metodo;
                if(peticion.can_run()){
                    actList.dequeue(peticion);
                    peticion.run();
                }
            }
        }
    }
}
```

En el gateway, la implementación del método Dispatch ejecuta continuamente los method request que estén disponibles. No obstante se puede complicar más dando por ejemplo prioridad a unos method requests sobre otros o añadiendo variables para tener en cuenta el estado del servant.

3.7.5. Concretar el future y cómo acceder a él

El último paso es determinar cómo los clientes podrán acceder al resultado de sus peticiones. Es necesario establecer un procedimiento porque el servant, el encargado de realizar la petición, se ejecuta en un hilo distinto al del cliente y por lo tanto se tendrán que poner en contacto de alguna manera.

Hay multitud de opciones para realizar este intercambio de información, pero existen unas pocas políticas que son las usadas más habitualmente:

- Synchronous waiting (parada síncrona). El cliente se bloquea en el proxy hasta que el scheduler manda ejecutar el method request y el resultado se ha almacenado en el future. Esto significa que el cliente no podrá hacer nada hasta que tenga el resultado de su petición.
- Synchronous timed wait (parada síncrona limitada). El cliente se bloquea un cierto tiempo mientras su petición se realiza. Si vence el temporizador, el cliente retoma el control pero no se encola ningún method request porque no se puede ejecutar inmediatamente.
- Asynchronous (asíncrono). El cliente realiza su petición al proxy y éste le devuelve un future donde encontrará el resultado si es que espera uno, devolviendo así el control del hilo al cliente. Aquí hay muchos métodos para que el cliente sepa que la petición está ya disponible. El mismo servant podría avisarle de que esto ha ocurrido, también lo podría hacer el proxy o quizás el propio cliente se encargue de realizar sondeos para comprobar si el resultado está disponible.

En este ejemplo se ha optado por la tercera opción porque da un poco más de complejidad y permite establecer reglas para establecer el orden en que se van a ejecutar las peticiones.

El future permite realizar llamadas asíncronas que devuelvan un resultado. Cuando el servant completa la ejecución, almacena el resultado en el future. Cualquier cliente con acceso al future, pueden recoger el resultado concurrentemente. El future se puede eliminar una vez el servant y todos los consumidores lo han usado.

En el gateway el método get() del proxy devuelve un future y su implementación podría ser la siguiente:

```
Class Future{
    //Parámetros
    private bool listo;
    private string mensaje;

    public Future (){
        listo = false;
    }

    public Future(string mensaje){
        this.mensaje = mensaje;
        listo = true;
    }

    public setResultado(string mensaje){
        this.mensaje = mensaje;
        listo = true;
    }

    public bool listo(){
        return listo;
    }

    public string getResultado(){
        return mensaje;
    }
}
```

Capítulo 4: Paquetes y librerías

La motivación de este capítulo es la presentación de las diferentes librerías y paquetes que forman parte de la aplicación desarrollada. Es necesario realizar un control de la estructura para poder adoptar el patrón de diseño como base.

4.1. Paquetes externos

Se ha utilizado el paquete DirectShowLib[6] en la versión liberada en abril de 2006 para las labores multimedia. Es un paquete de libre distribución que trata de portar la biblioteca DirectShow que existe en C++ al lenguaje C#. Está siendo desarrollado por un grupo de voluntarios que decidieron tomar la iniciativa cuando Microsoft anunció su intención de abandonar las librerías DirectShow para centrarse en el proyecto Windows Media.

Por el hecho de estar siendo desarrollado por programadores independientes y de forma voluntaria, no está terminado y algunas funciones que podrían ser interesantes para este proyecto han tenido que ser sustituidas por otras menos eficientes. Concretamente la posibilidad de enviar ficheros multimedia y manipularlos al vuelo como lo que son, es decir, tener flujos de datos identificados como multimedia y sobre los que se puede actuar como tales. Esto que se puede hacer en Java, en C# no está soportado nativamente y son necesarias librerías externas. En caso de tener este soporte nativo para flujos de datos multimedia, sería posible unificar la descarga y el visionado de imagen y video en directo. En el capítulo de implementación de la aplicación se detallará en qué afecta esto.

Concretamente en el servidor se ha utilizado el componente DxPlay de la librería Capture del paquete DirectShowLib. Permite reproducir video y hacer capturas del mismo. Con la transmisión de estas capturas se realizará un sistema de visionado en directo, quizás no del todo eficiente, pero sí posible con las herramientas disponibles en este momento.



4.2. Paquetes propios

La aplicación se ha dividido en dos espacios de nombres: cliente y servidor.

El paquete cliente es el más simple de los dos, pues sólo contiene la interfaz gráfica del usuario. Esta interfaz contiene un componente del espacio de nombres del servidor, el proxy, al que se realizarán las peticiones del cliente. El proxy se ha incluido dentro del espacio de nombres del servidor porque tiene la función de ser un representante de éste para el cliente pero no forma parte de éste: se ejecuta en el hilo de control del cliente.

Además en el patrón original, el proxy completo es una parte del servidor, por lo que es lógico que aunque se separen las funciones relacionadas con el cliente y las relacionadas con el servidor en dos partes, estas dos partes sigan siendo parte del servidor. El cliente también utiliza la clase future del servidor para poder acceder al resultado de su petición.

El paquete servidor contiene todas las clases relacionadas con el patrón objeto activo, incluidos proxy y future. Además desde el paquete servidor se utiliza el componente DxPlay del paquete DirectShowLib para las labores multimedia.



Capítulo 5: Implementación del patrón

Este proyecto buscaba construir una aplicación multimedia de tipo cliente/servidor basándose en el patrón objeto activo. Se ha intentado respetar al máximo posible la estructura original, y así se ha conseguido, excepto en algunos aspectos no contemplados en la descripción original. Se podría decir que se ha implementado una variante del patrón y como más adelante se va a haber, los cambios no sólo han sido necesarios, sino que además inevitables.

5.1. Problemas encontrados

Durante el desarrollo de la aplicación se han encontrado dificultades relacionadas tanto por el lenguaje utilizado como por el patrón.

5.1.1. Referentes al lenguaje

- Librerías multimedia inacabadas

Aunque en teoría el lenguaje C# es multiplataforma y de hecho hay proyectos más o menos maduros que dan la posibilidad de utilizarlo en sistemas Linux, en la práctica sólo se utiliza en sistemas Windows. Además Microsoft es su gran valedor, se podría decir que es suyo, y se demuestra en que son los encargados de escribir las librerías que utiliza. Esto, que puede parecer bueno a priori ya que asegura su desarrollo, resulta dañino a largo plazo por las aspiraciones de Microsoft a promocionar sus productos.

Un ejemplo de esto son las librerías multimedia que la aplicación debería utilizar. Las librerías que deberían ser equivalentes a las utilizadas a C++, no sólo no están acabadas, sino que se encuentran abandonadas. La razón es que Microsoft las olvidó cuando lanzó la versión 9 de su Windows Media Player y vio que la forma más fácil para tratar audio y video con C# es simplemente incrustar su reproductor allá donde sea necesario.

El problema de esto es que oficialmente la única manera de ver, transmitir o tratar video o audio requiere un sistema Windows y utilizar programas de Microsoft. Esto puede no ser del todo deseable, por ejemplo porque se esté acostumbrado a C++ y se prefiera utilizar las librerías equivalentes a ese lenguaje o que se esté pensando en una posible compilación en Linux donde no estarán disponibles estos programas.

Por ello algunos desarrolladores independientes han ideado sistemas alternativos con los que tratar audio y video. La mayoría son demasiado simples y simplemente permiten ver video y aplicar filtros a la imagen aunque existe un proyecto de software libre bastante avanzado que es el que se ha usado en la aplicación. Es el caso de la librería DirectShowNet. No obstante esta librería no está acabada y tiene una función imprescindible para la aplicación todavía sin implementar, y es el soporte de archivos multimedia para sockets y flujos de datos. Por ahora

esta librería es incapaz de diferenciar un flujo de datos binarios de uno de datos multimedia, por lo que el visionado de videos en tiempo se tiene que realizar dando un pequeño rodeo.

El impacto de este problema sobre la aplicación es tener que elegir entre ver el video o descargarlo y verlo después. Cuando la librería DirectShowNet esté acabada será posible descargar un video y verlo al mismo tiempo.

- Lanzamiento de hilos

El lenguaje C# tiene una forma muy particular de crear hilos. Si en Java se modela un objeto tipo Thread que se ejecutaba en un hilo independiente, en C# lo que se modela es un método que tiene una signatura particular. Concretamente ha de ser public y no devolver nada. Surgen dos dificultades principalmente. Por un lado no es posible crear un hilo que dependa de uno parámetros, al menos no de forma directa. Por otro lado, este hilo no puede devolver nada, por lo que no se podrá obtener el resultado cualquier proceso que lleve a cabo, también de forma directa en este caso.

Lo que se ha optado es la solución más utilizada, y es utilizar objetos a través de los cuales poder adecuar el contexto del hilo que va a funcionar. Esto significa que un objeto contiene, además del método que se lanzará, los campos necesarios para poder adecuar el proceso a la situación que está transcurriendo. Además, mediante estos campos es posible acceder a cualquier resultado que deba devolver el hilo. Una vez se obtiene este resultado o finaliza la ejecución del hilo, basta con eliminar este objeto.

5.1.2. Referentes al patrón

- Programa cliente y programa servidor distintos

La versión original no contemplaba que la aplicación pueda estar partida en dos partes y por tanto, no haya forma posible que el future sea un objeto compartido por cliente y servidor. Si el cliente es un programa y el servidor es otro, no es posible tener referencias cruzadas y que el servidor tuviera una referencia a un objeto del programa del cliente (el future).

Por la misma razón, el proxy no puede ser implementado tal cual está en la teoría. Por un lado el cliente tendría que acceder a este proxy como si del método del servidor se tratase para poder obtener su future, no debería utilizar sockets en ningún momento. Por otro lado el proxy debería tener referencias a los objetos del scheduler y el servant que se encuentran en el servidor para construir los method requests y poder encolarlos.

- Ejemplo en C++

El documento original que describía el patrón objeto activo utilizaba el lenguaje C++ como base. A pesar de que tanto C++ como C# son lenguajes orientados a objetos, han sido necesarias ciertas adaptaciones de funciones de C++ que no existían en C#. Por ejemplo

5.2. Organización del código

La aplicación se dividirá en dos programas al tener estructura tipo cliente/servidor. Por un lado estará el servidor y por otro el cliente, y las funciones del patrón estarán repartidas entre ambos.

El componente más afectado por la estructura cliente/servidor es el proxy. La solución escogida ha sido partir el proxy en dos partes. La parte que estará en el programa del cliente hará de interfaz para que éste pueda realizar peticiones al servidor y se encargará de hacer llegar el resultado de la petición al cliente. La otra parte estará en el programa del servidor y será el encargado de crear los method requests y pedirle al scheduler que los encole en la activation list.

Al partir el proxy en dos partes, surge la necesidad de un medio de comunicación entre ellas. Aquí, sin embargo, sí se puede utilizar sockets para este fin porque ésta comunicación será completamente transparente al cliente. Para aclarar conceptos, la parte del proxy que está en el lado del cliente se denominará a partir de ahora Proxy mientras que la otra será Skeleton.

5.2.1. Proxy y Skeleton

La clase Proxy se encuentra en el espacio de nombres del servidor y es la puerta de acceso de los clientes al objeto activo. No tiene hilo propio, se ejecuta en el hilo de control del cliente y atiende sus peticiones. Sólo se encargará de las funciones relacionadas con el cliente: atender sus peticiones y encauzarlas hacia quien deberá introducirlas en la lista, devolver un future al cliente y poner los medios para que el resultado llegue a él.

Proxy mantendrá una lista de los archivos tanto propios de la máquina del cliente como los de la máquina del servidor.

El Skeleton tampoco tiene hilo propio, se ejecuta en el hilo del cliente. Es la parte visible del objeto activo para el resto de proxies. El skeleton y los proxies se comunican por sockets mediante mensajes debidamente formateados.

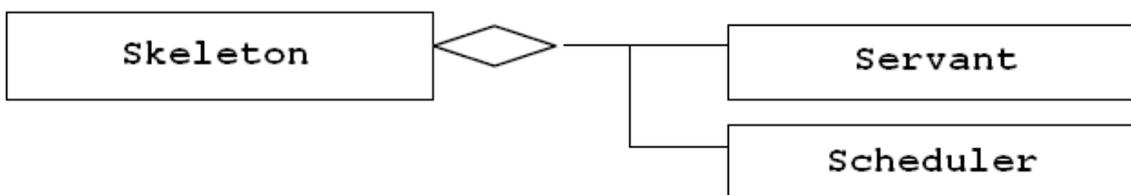


Figura 6: Estructura del skeleton

5.2.2. Future y Method Requests

El **Future** es común a todas las peticiones, aunque no todas usan todos los campos, de ahí que haya dos constructores distintos. Si la petición es de visionado de video, ni siquiera el nombre del fichero sería necesario para que el cliente lo vea. Para las peticiones de descarga o subida de archivos, tanto el nombre como el tamaño de los ficheros es necesario.

El interfaz de Method Request al que se deberán ajustar todas las peticiones es el mostrado en la figura siguiente. El campo "tipo" de la interfaz Method Request se usa únicamente para labores de depuración. El método *can_run()* es el método de guarda que devolverá true cuando sea posible realizar la petición. El método *ejecutar()* realiza una llamada al servant para que realice la petición.

La implementación que se realiza del Method Request para cada uno de las peticiones posibles es prácticamente la misma ya que dentro de estas clases no está el código para ejecutarla, sino la llamada al servant para que la realice. La única diferencia es la presencia o no del campo size, que indica el tamaño del fichero en el caso de descarga o subida de ficheros.

5.2.3. Scheduler

El Scheduler es similar al del ejemplo del gateway. Contiene una referencia a una activacion list donde irá almacenando los method requests que le lleguen desde el skeleton. Sólo contiene dos métodos, uno para incluir method requests y otro para extraerlos.

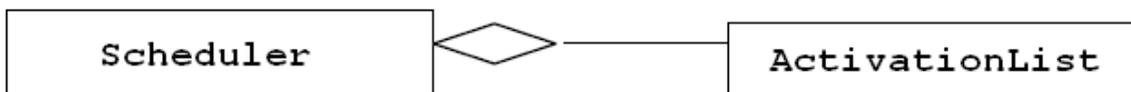


Figura 7: Estructura del scheduler

Además cuenta con el método Dispatch que correrá en un hilo independiente. Este hilo se encarga de recorrer los Method Requests almacenados, comprobar sus métodos de guarda, extraerlo de la lista si es posible realizar su ejecución y avisar al servant para que la realice.

5.2.4. Activation List

Está construida mediante un iterador utilizando los interfaces que proporciona Microsoft para este fin: IEnumerable y IEnumerator. De esta forma se consigue compatibilidad con la instrucción *foreach()* de C#.

El interfaz IEnumerable debe ser implementada por el objeto que vaya a ser iterado. En este caso se usa una clase interna llamada petición que incluye el Method Request y una referencia a la siguiente petición de la lista.

El interfaz IEnumerator lo implementa otra clase interna que será la encargada de recorrer la lista de objetos tipo petición enlazados.

5.2.5. Servant

El servant es el componente último de la estructura, el encargado de realizar la ejecución de la petición y devolver al cliente el resultado. Utiliza el componente DxPlay de la librería DirectShowLib para enviar video en directo.

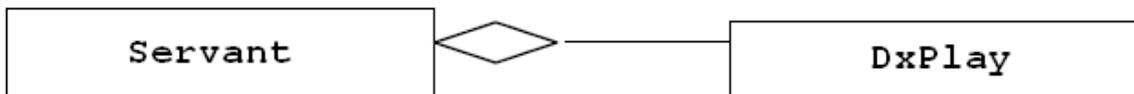


Figura 8: Estructura del servant

Para realizar esta transmisión de video en directo, el servant necesita reproducir el video y hacer capturas de la imagen para enviarlas. Para poder reproducir el video, ha de tener una referencia a un objeto tipo panel que se le provee en el constructor.

5.3. Dinámica

Se realizan comunicaciones simultáneas a varios niveles ya que existen varios hilos de ejecución. Están las comunicaciones entre el proxy y el skeleton provocadas por las peticiones de los clientes. Además existe intercambio de mensajes entre las clases del objeto activo para decidir si se puede ejecutar una petición. Por último están las comunicaciones entre el servant y el cliente para hacerle llegar el resultado de su petición.

5.3.1. Arranque de los programas

En este apartado se van a detallar los procesos de arranque e inicio de los programas cliente y servidor.

Empezando por el programa servidor, hay que diferenciar entre el arranque del programa y el inicio del servidor. Al arrancar el programa todo lo que se obtiene es una ventana con varias opciones, pero todavía no se escucha ninguna petición de los clientes. Para comenzar a escuchar las peticiones de los clientes, hay que pulsar sobre el botón *Iniciar*. Esto dispara la secuencia de arranque del objeto activo que se reduce a crear un objeto de tipo skeleton

desde la ventana. En el constructor de éste se lanzarán los diferentes componentes del objeto activo, que a su vez iniciarán los componentes que necesiten para funcionar:

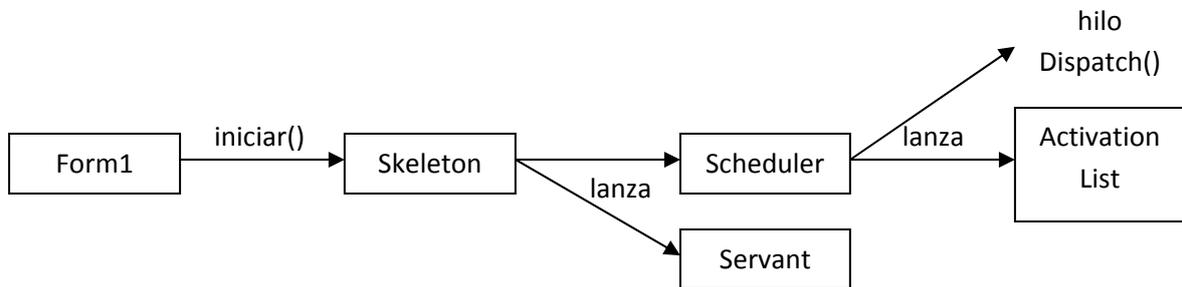


Figura 9: Secuencia de arranque del servidor

El caso del cliente es aun más simple. Partiendo de la ventana inicial, el proxy se crea cuando se pulsa sobre la opción *Conectar* del menú principal.



Figura 10: Secuencia de arranque del cliente

5.3.2. Comunicaciones proxy – skeleton

Las primeras comunicaciones entre el proxy y el skeleton se producen durante la secuencia de arranque del proxy. En este periodo, el proxy se conecta al servidor y descarga la lista de videos remotos. Después de esto, permanece a la espera de que se produzcan peticiones que deba encaminar hacia el skeleton.

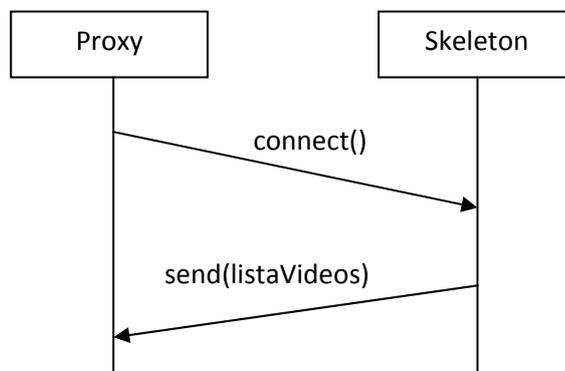


Figura 11: Secuencia conexión del proxy con el skeleton

El segundo y último contacto que se produce entre proxy y skeleton es en los primeros pasos de una petición:

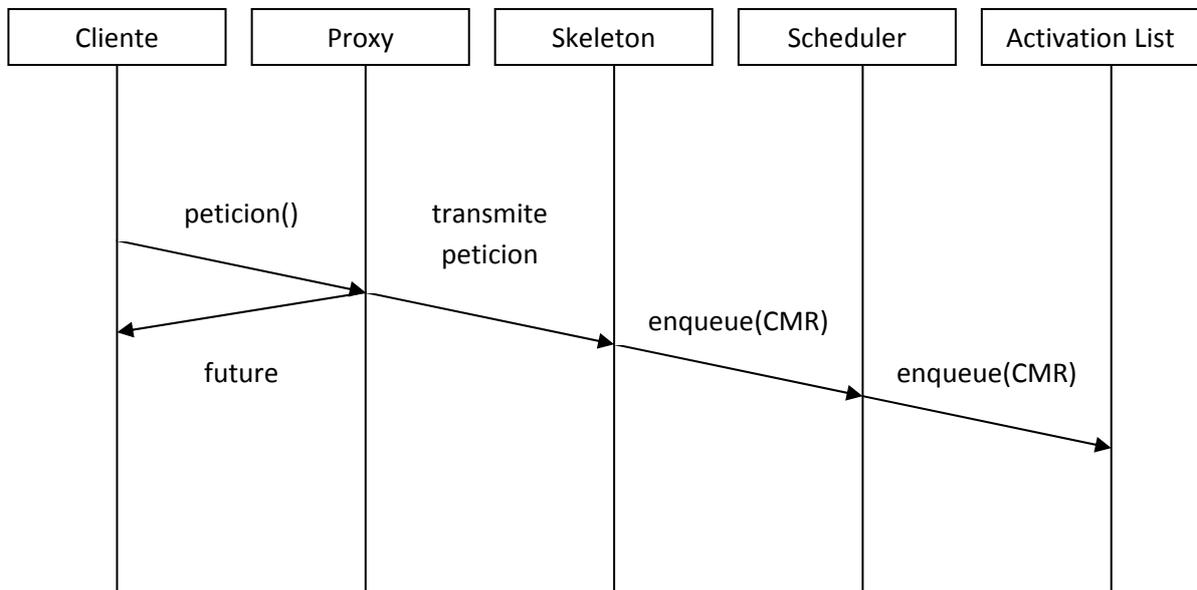


Figura 12: Secuencia de petición

El cliente realiza la petición como si de un método normal del proxy se tratara. Desconoce completamente que esa llamada a un método del proxy supone conectar con un servidor externo. Obviamente el usuario sí sabe lo que esa petición conlleva, pero desde el punto de vista del programa cliente, esto es totalmente transparente.

Una vez el proxy recibe la petición del cliente, construye un vector tipo string de dos o tres posiciones según el tipo de petición. En este vector incluye la información debidamente formateada que necesita el skeleton para saber la petición que realizó el cliente y los datos necesarios de los archivos que haya implicados en esta petición. Una vez estos datos han sido transmitidos al skeleton, el proxy le devuelve un future al cliente a través del cual deberá esperar recibir el resultado. Más adelante se detallará el proceso de petición según el tipo que sea, ya que no todas requieren los mismos pasos una vez ha sido encolada en la activation list.

El skeleton se limita a recibir la petición y construir un objeto tipo Concrete Method Request (CMR) que incluye tanto el tipo de petición y las características de los videos implicados, como un socket al que enviar el resultado. Este objeto se lo envía al scheduler para que lo encole en la activation list. Cuando el scheduler recibe el objeto, se limita a enviarlo a la activation list y ésta a colocarlo en la última posición de su lista.

5.3.3. Objeto activo

El objeto activo tiene un hilo propio encargado de extraer peticiones de la lista y de ejecutarlas posteriormente. El proceso es el siguiente:

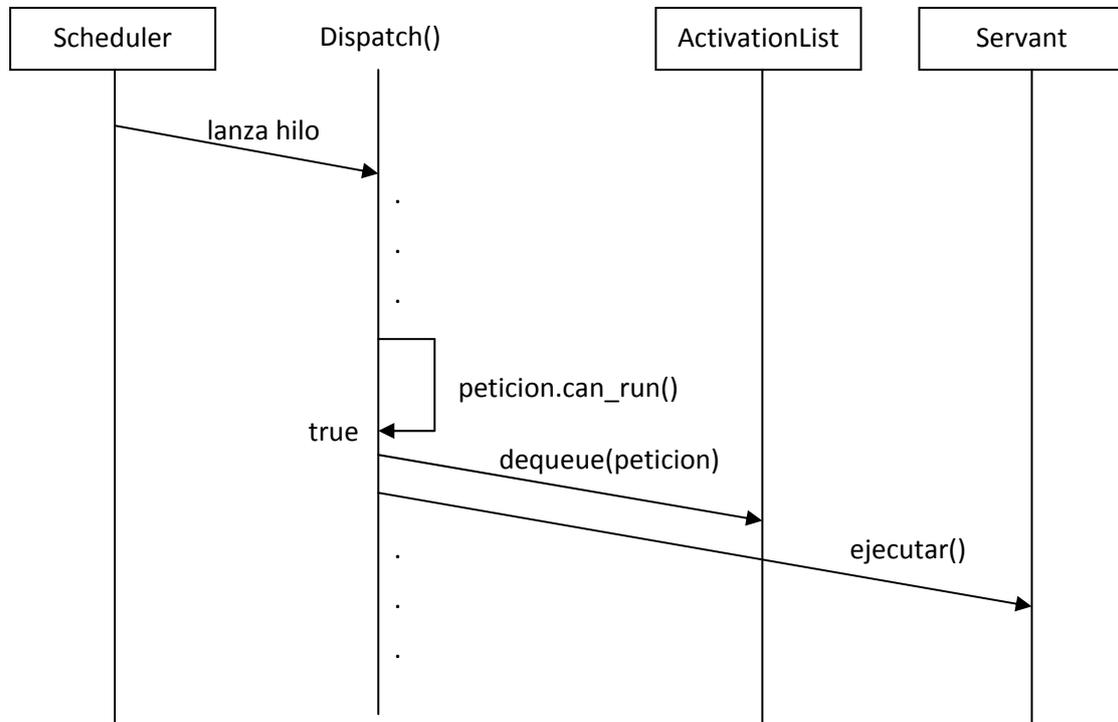


Figura 13: Secuencia de extracción de petición

El método Dispatch tiene un hilo propio donde se ejecuta continuamente. Este método recorre la lista de peticiones pendientes de ejecución mediante un iterador. Comprueba con el método de guarda de cada petición si es posible ejecutarla y cuando encuentra una que se puede ejecutar, la extrae de la lista y avisa al servant para que realice la ejecución.

5.3.4. Comunicaciones objeto activo - proxy

Una vez se ha extraído una petición y se ha llamado al servant para que la ejecute, se produce un intercambio de mensajes entre el objeto activo, concretamente desde el servant, y el proxy alojado en el hilo del cliente. Este intercambio de mensajes varía con el tipo de petición que se va a realizar, por lo que se van a detallar a continuación los tres casos implementados. En todos los casos se parte desde el momento en que el Scheduler avisa al servant para que ejecute la petición.

Primer caso: Petición tipo descarga de video.

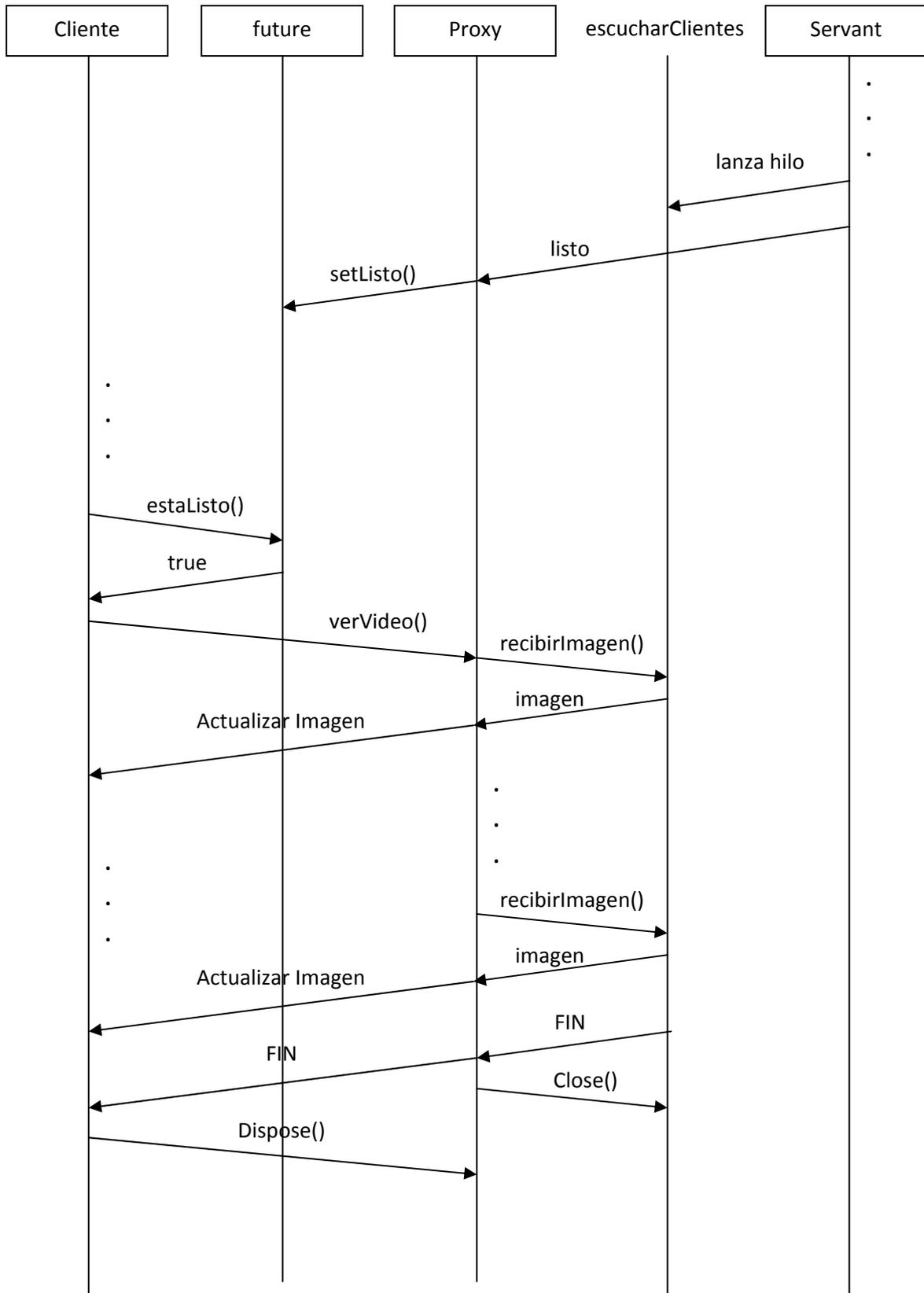


Figura 14: Secuencia de ejecución de la petición de visionado

Segundo caso: Petición tipo descarga de video.

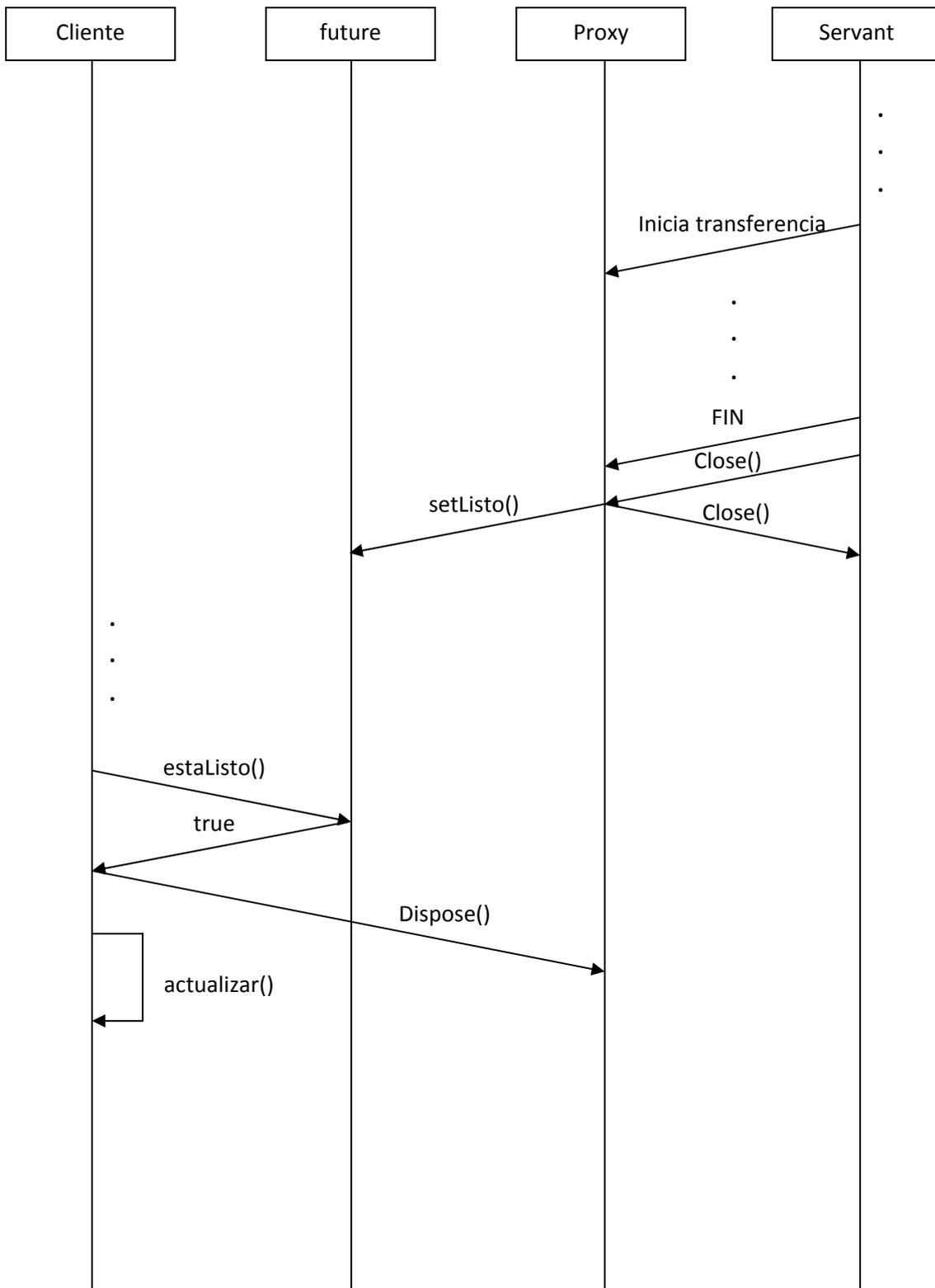


Figura 15: Secuencia de ejecución de la petición de descarga

Tercer caso: Subida de video.

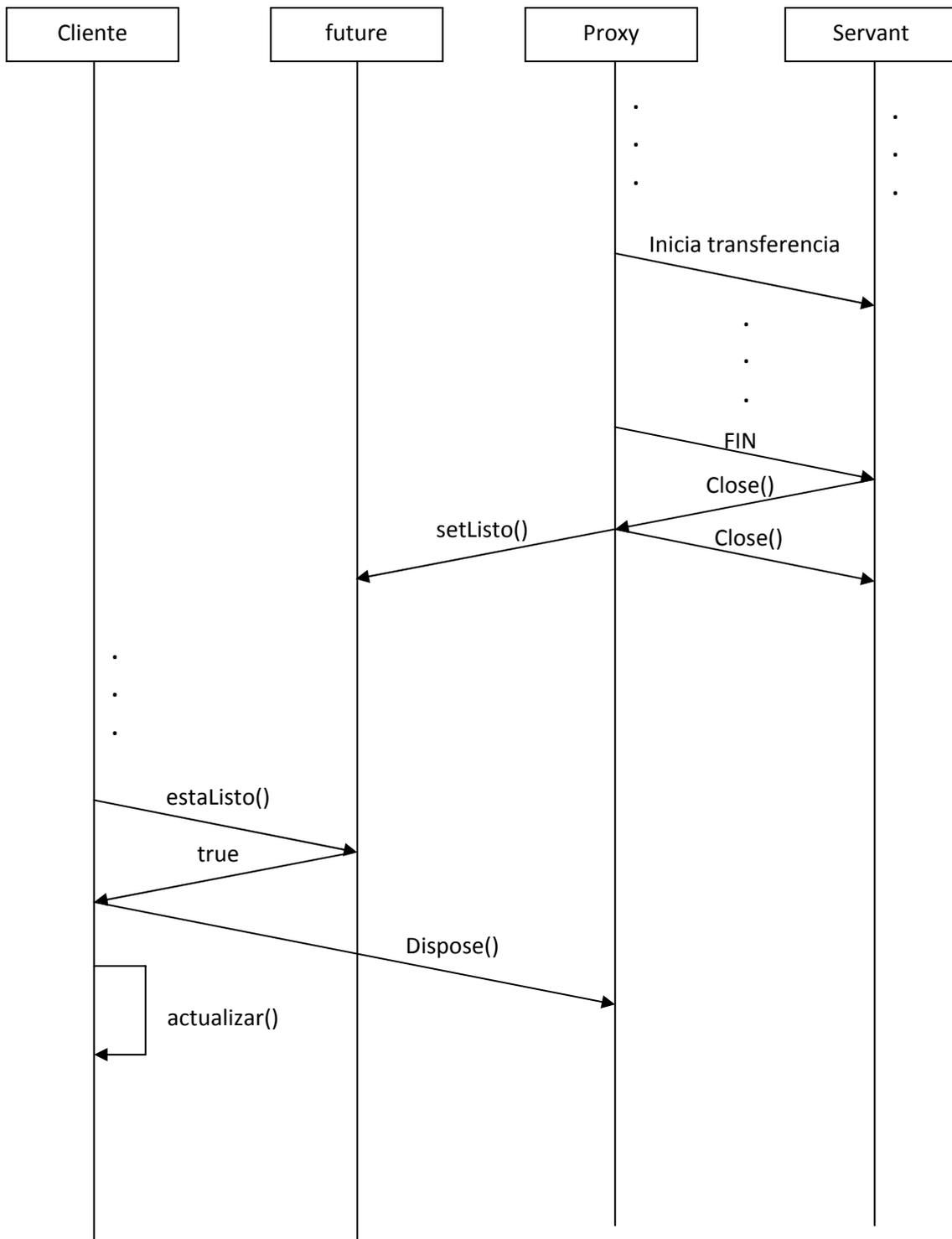


Figura 16: Secuencia de ejecución de la petición de subida

En todos los casos, una vez ha recibido el resultado el cliente, se elimina la referencia al proxy y se crea una nueva. Para poder volver a realizar peticiones es necesario reconectar con el servidor.

Las peticiones tipo descarga y subida funcionan de forma exactamente igual tal como se verá a continuación, con la excepción del sentido de la transferencia del fichero principalmente.

La petición tipo visionado tiene un funcionamiento algo más complicado. Esta petición

5.3.5. Otros

La secuencia que se produce cuando se cambian los parámetros de configuración es la misma tanto para el servidor como para el cliente. Lo único que varía es el nombre de las variables.

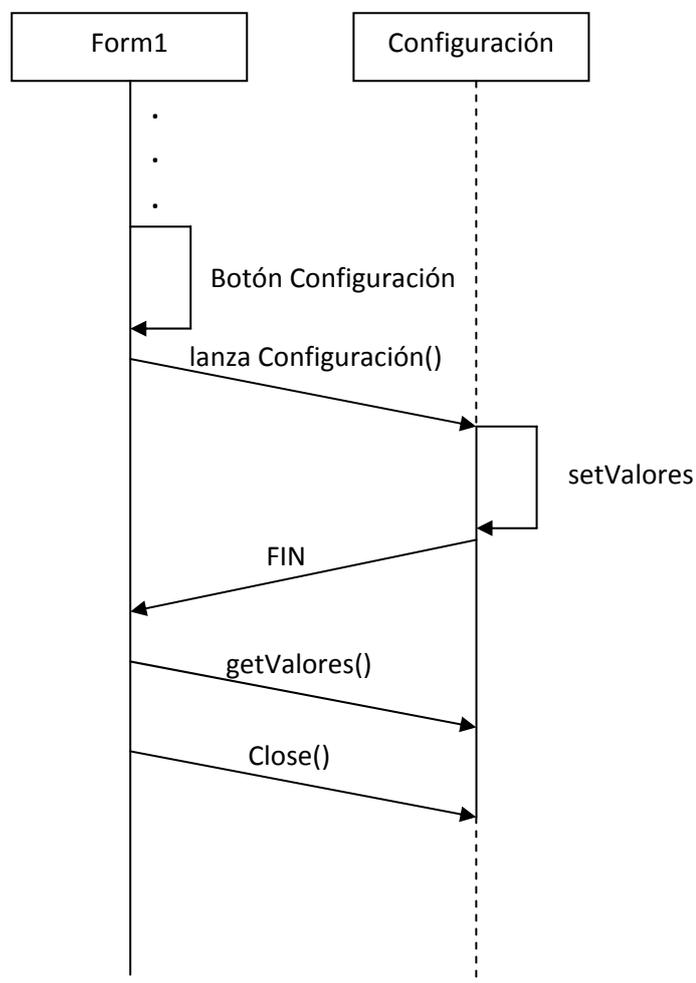


Figura 17: Secuencia de configuración

Cuando se pulsa sobre el botón de configuración, en cualquiera de los dos casos, se abre una ventana que asume el control de la aplicación. Una vez se cierra, se actualizan los valores.

5.4. Implementación

En este capítulo se va a entrar en detalle en el código de los dos programas desarrollados, cliente y servidor. Puesto que ya ha quedado claro en capítulos anteriores qué componentes pertenecen a cada programa, en este capítulo se van a documentar directamente el código siguiendo la estructura del patrón y no la de los programas. Se hará así para seguir el modelo que se ha utilizado en la descripción del patrón en su versión original.

El método consistirá en mostrar primero el interfaz de cada clase y una pequeña explicación de ésta. Después se entrará en detalle con los métodos, explicando la función de cada uno.

5.4.1. Proxy y skeleton

Puesto que en el patrón original estas dos clases eran una misma, en este apartado se han incluido ambas. Como ya se ha explicado en apartados anteriores, el proxy se encargará de proporcionar al cliente un interfaz con el que acceder al servidor. El skeleton hará de enlace entre los proxies que se ejecutan en cada máquina cliente con el servidor.

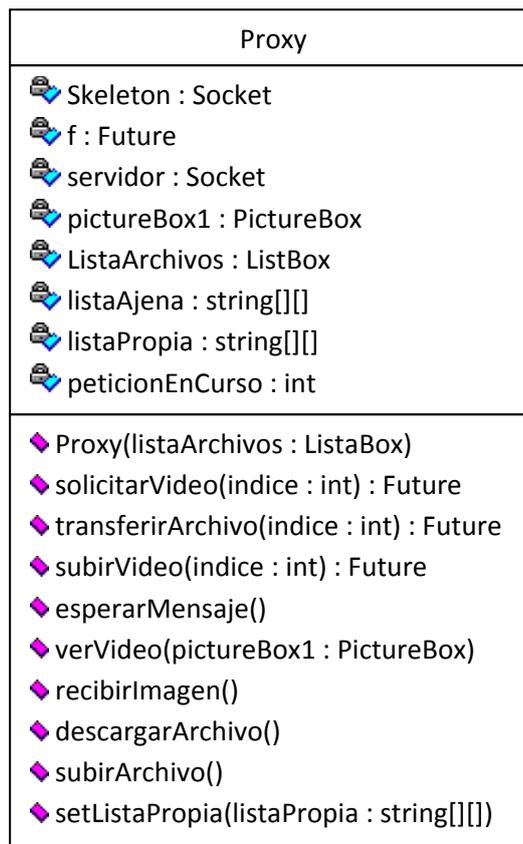


Figura 18: Interfaz del proxy

Método	Descripción
Proxy (listaArchivos ListBox)	Es el constructor de la clase. La lista que se le pasa como argumento es la lista de archivos de la máquina cliente disponibles en ese momento. Durante la construcción, se descarga la lista de archivos del servidor.
solicitarVideo (int índice)	Método que invoca el cliente para solicitar el visionado de un video. El proxy se encarga aquí de encaminar la petición al skeleton. El argumento es el índice del archivo en la lista de archivos del servidor.
transferirArchivo (int índice)	Método que invoca el cliente para solicitar la descarga de un video. El proxy se encarga aquí de encaminar la petición al skeleton. El argumento es el índice del archivo en la lista de archivos del servidor.
subirVideo (int índice)	Método que invoca el cliente para solicitar la subida de un video. El proxy se encarga aquí de encaminar la petición al skeleton. El argumento es el índice del archivo en la lista de archivos del cliente.
esperarMensaje ()	Con este método se realiza una espera si el servidor debe avisar cuándo se debe actualizar el future.
verVideo (PictureBox pictureBox1)	Cuando la petición de visionado está lista y el future así lo indica, el cliente invoca este método para comenzar a ver las imágenes del video. El argumento es el cuadro donde el cliente quiere ver las imágenes.
recibirImagen ()	Mediante este método el proxy recoge las imágenes del servidor y se las hace llegar al cliente.
descargarArchivo ()	Con este método el proxy realiza la descarga el archivo. Al finalizar actualiza el future para que el cliente pueda encontrar el resultado de su petición.
subirArchivo ()	Con este método el proxy realiza la subida el archivo. Al finalizar actualiza el future para que el cliente sepa que ya se ha subido.
setListaPropia (string[] listaPropia)	En caso de que cambie la lista de archivos propia, aquí se actualiza.

Skeleton	
 TCPListener : TCPListener  lista : string[][]  organizador : Scheduler  servant : Servant  panel : Panel  IEventos : ListBox	
 Skeleton(panel : Panel, IEventos : ListBox)  iniciar()  actualizarLista()  esperarClientes()  esperarMensaje()  recibirVideo(client : Socket, fichero : string, size : long)  verVideo(client : Socket, fichero : string)  transferirArchivo(client : Socket, fichero : string, size : long)	

Figura 19: Interfaz del skeleton

Método	Descripción
Skeleton(Panel panel, ListBox IEventos)	Constructor de la clase. Sobre el panel se reproducirá el video del que se tomarán capturas y IEventos es una lista que se actualizará con los eventos que ocurran.
Iniciar ()	Inicia el objeto activo.
actualizarLista ()	Actualiza la lista de archivos del servidor.
esperarClientes ()	Método de espera de conexión de clientes.
recibirVideo (Socket client, string fichero, long size)	Método que avisa al scheduler de que debe introducir una petición de tipo recibirVideo en la activation list.
verVideo (Socket client, string fichero)	Método que avisa al scheduler de que debe introducir una petición de tipo verVideo en la activation list.
transferirArchivo (Socket client, string fichero, long size)	Método que avisa al scheduler de que debe introducir una petición de tipo transferirArchivo en la activation list.

5.4.2. Future y method requests

Los interfaces de las peticiones, aunque similares, no son del todo iguales entre ellos. Además de diferir en el constructor, unos requieren distintos datos para la ejecución, por lo que tienen campos distintos.

El interfaz base para todos los Method Request y que deberán implementar es el siguiente:

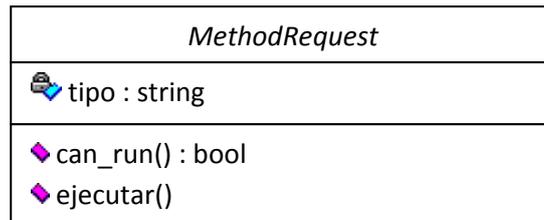


Figura 20: Interfaz de Method Request

Método	Descripción
<code>can_run()</code>	Es el método de guarda. Devuelve un valor booleano: true si es posible la ejecución y false si no es así.
<code>ejecutar()</code>	Cuando sea posible la ejecución, se llamará a este método. Este método hace una llamada al servant.

La clase *MethodRequest* se ha declarado como una clase abstracta, así como sus métodos. Por esta razón, no aporta ningún código, sólo el interfaz que deben implementar las peticiones de cada tipo. El método *ejecutar()* resulta obvio que cada petición necesitará uno propio que lance una llamada diferente al servant, pero se podría pensar que el método *can_run()* sí es común a todos. Este planteamiento es incorrecto por varias razones. Por un lado los distintos tipos de peticiones no tienen por qué depender de las mismas variables para su ejecución, aunque en la aplicación esto sí es así. Por otro lado, así no se depende de una implementación específica del servant, y así, si este cambiara en algún sentido, sólo variarían las peticiones que se vieran afectadas por este cambio.

El campo `tipo` se ha utilizado en labores de depuración de errores para poder identificar qué peticiones hay en la cola y saber qué estaba ocurriendo durante la ejecución. Para el funcionamiento normal de la aplicación, este campo es totalmente irrelevante.

En la aplicación se han usado tres tipos de peticiones: Ver video en directo, descargar video y subir video. Los interfaces de cada una son los siguientes.

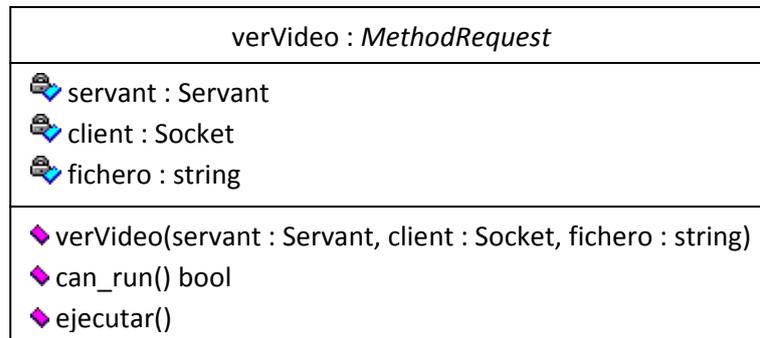


Figura 21: Interfaz de la petición verVideo

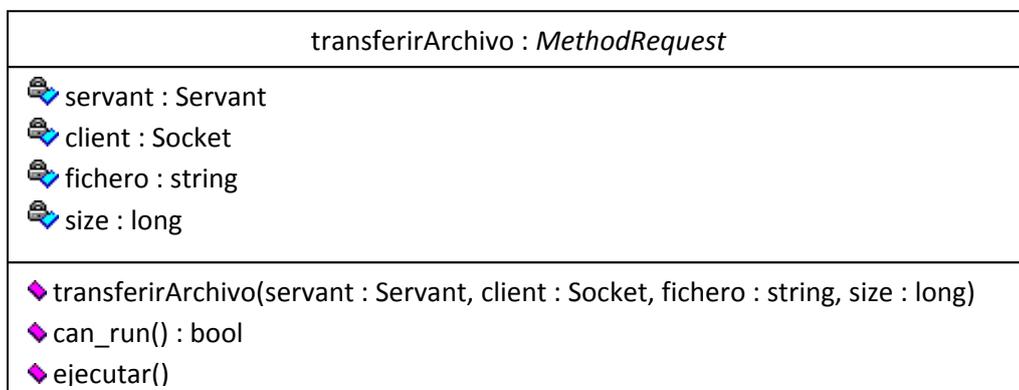


Figura 22: Interfaz de la petición transferirArchivo

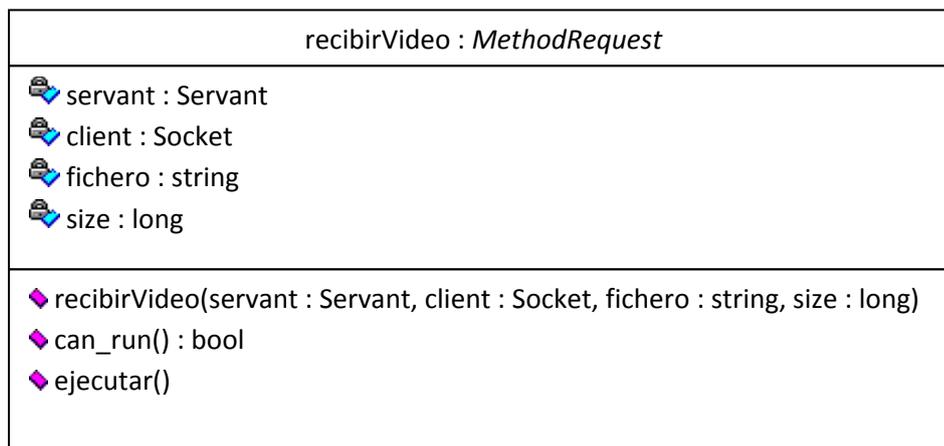


Figura 23: Interfaz de la petición recibirArchivo

En los tres casos realizan una implementación de los métodos abstractos. El método de guarda es el mismo en la aplicación, aunque no tendría por qué serlo como ya se ha explicado.

Como se puede ver cada petición necesita de una serie de datos para su ejecución, y estos datos se almacenan dentro del Method Request.

Por último, antes de seguir con la estructura del patrón, queda detallar el future. Se ha optado por dar una implementación única del future para todas las peticiones, aunque se podría haber dado diferentes versiones de future para cada una. Este objeto se encontrará en el cliente, por lo que ha de llevar aquella información que se requiera en la máquina cliente para la recepción del resultado. Por ello, se ha implementado el constructor sobrecargado, para atender a las exigencias de cada solicitud. El interfaz es el siguiente:

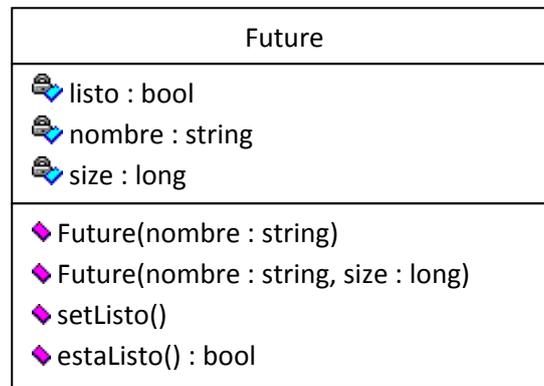


Figura 24: Interfaz del future

El único campo necesario en todos los casos es el campo *listo*, que indica si el resultado está ya disponible. Las peticiones de descarga y subida de videos requieren de los otros dos campos, *nombre* y *size*, para saber el nombre del fichero a crear o abrir y su tamaño, para saber cuándo se ha completado la transferencia. En caso de ver el video en directo, no se necesitaría de ninguno de estos dos campos, aunque se deja el del nombre por si se quisiera mostrar el nombre del video en la ventana por ejemplo.

Método	Descripción
Future(nombre : string)	Primer constructor. Sólo requiere del nombre del fichero de video.
Future(nombre : string, size : long)	Este es el segundo constructor. Requiere además del nombre del fichero, el tamaño de éste.
setListo()	Cuando el resultado de la petición esté disponible, con este método se actualiza el estado.
estaListo() : bool	Con este método el cliente puede sondear si el resultado de su petición está ya disponible.

5.4.3. Scheduler

A primera vista, el scheduler puede parecer la clase más simple del patrón. Si observamos el interfaz, se ve que además del constructor, sólo hay dos métodos y el componente de la Activation List.

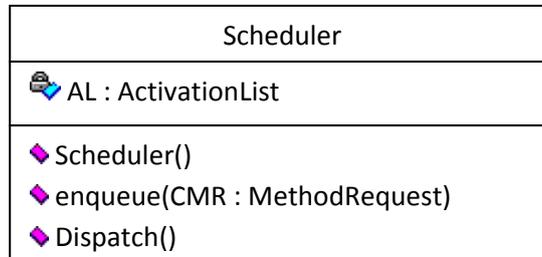


Figura 25: Interfaz del scheduler

Método	Descripción
Scheduler()	Constructor.
enqueue(CMR : MethodRequest)	Así se encolan peticiones en la activation list.
Dispatch()	Este método se ejecuta continuamente en un hilo independiente. Se encarga de extraer Method Request de la activation list cuando sea posible ejecutarlos comprobando los métodos de guarda.

Sin embargo es en este método donde se podrían definir unas reglas y un orden a la hora de decidir qué MethodRequest se extraerá a continuación. Esto queda para futuras versiones, pero bastaría con modificar el método *Dispatch()* para que, además de comprobar los métodos de guarda, tuviera lógica adicional que tuviera en cuenta información sobre las peticiones. Por ejemplo, sería posible establecer prioridades entre tipos de peticiones y dentro del tipo de peticiones, prioridades por tamaños de archivos o duración de los videos.

Además, se podría añadir un nivel más de complejidad permitiendo que las opciones se pudieran cambiar desde un menú de opciones, sin necesidad de programarlas manualmente.

5.4.4. Activation list

El interfaz de la activation list es el siguiente:

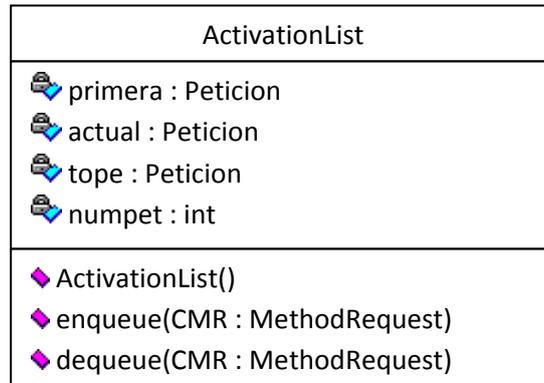


Figura 26: Interfaz de la activation list

Método	Descripción
ActivationList()	Constructor. No recibe ningún parámetro.
enqueue(CMR : MethodRequest)	Añade peticiones a la lista.
dequeue(CMR : MethodRequest)	Extrae la petición que se le pasa como argumento de la lista.

Los objetos tipo Peticion se modelan mediante una clase interna de la activation list.

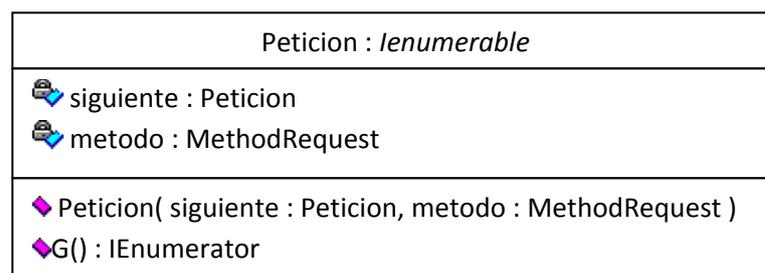


Figura 27: Interfaz del objeto Peticion

El objeto Peticion encapsula, además del Method Request que ocupa esa posición en la lista, una referencia al siguiente objeto Peticion en la lista enlazada.

El método *GetEnumerator()* devuelve un iterador con el que recorrer la lista enlazada a partir del objeto Peticion desde el que se hace la llamada.

El iterador ha de implementar un interfaz concreto, *IEnumerator*, para que el operador *foreach()* sea compatible con él.

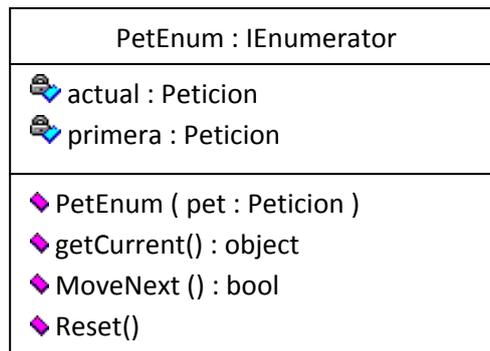


Figura 28: Interfaz del iterador

Método	Descripción
PetEnum(pet : Peticion)	Constructor. El argumento es el objeto que se tomará como el primero de la lista.
getCurrent() : object	Devuelve una referencia del objeto de la lista por el que va la iteración.
MoveNext() : bool	Adelante una posición en la lista enlazada la iteración
Reset()	Reinicia la iteración al primer objeto.

5.4.5. Servant

Por último queda de tallar la implementación del servant. Este es el objeto activo propiamente dicho, aunque ello no significa que tenga hilo propio de ejecución. Depende del scheduler para ejecutar las peticiones.

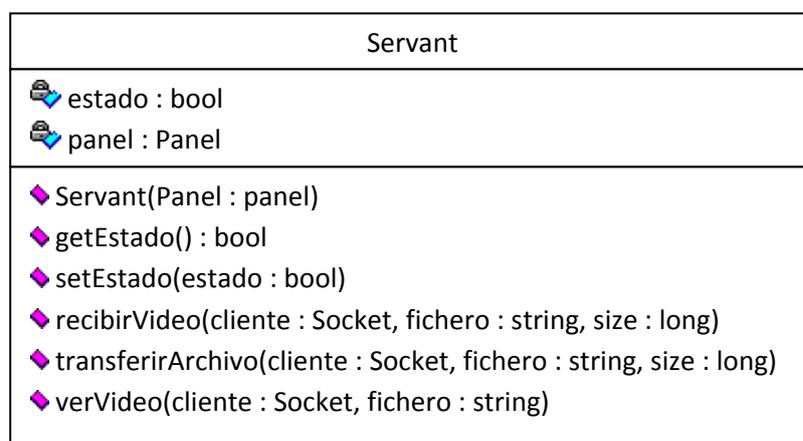


Figura 29: Interfaz del servant

Método	Descripción
Servant (Panel : panel)	Constructor. El argumento es el panel sobre el que se reproduce el video para la petición de visionado.
getEstado () : bool	Indica si el servant está ocupado realizando una petición. Devuelve True cuando está ocupado, y False cuando está libre.
setEstado (estado : bool)	Actualiza el estado del servant al valor que se le pasa por argumento.
recibirVideo (cliente : Socket, fichero : string, size : long)	Método que ejecuta una petición de recepción de video. Con los argumentos es capaz de conectar al cliente.
transferirArchivo (cliente : Socket, fichero : string, size : long)	Método que ejecuta una petición de transferencia de video.
verVideo(cliente : Socket, fichero : string)	Método para ejecutar una petición de visionado a distancia de video.

Capítulo 6: Plan de prácticas

Uno de los objetivos de este proyecto era proporcionar al área LSI de la UPCT ejemplos de uso tanto de la tecnología .NET como del patrón objeto activo. Tras el trabajo realizado, parece una buena idea aprovechar la evolución que tuvo la aplicación para crear un plan de prácticas que sirva para el fin de proporcionar ejemplos de uso. Por ello, se ha resumido y adaptado el desarrollo de la aplicación para que se pueda realizar en unos pocos pasos que vayan incrementando gradualmente la dificultad.

El objetivo será un plan de prácticas para la docencia tanto de la tecnología .NET como del patrón objeto activo. Por esta razón se presuponen aprendidos conceptos básicos tanto de programación orientada a objetos como un conocimiento básico del lenguaje C#. Aun así, se ha intentado simplificar lo máximo posible aquellos aspectos únicos en C#, por lo que probablemente un programador Java debería ser capaz de seguir y comprender el código C# casi en su totalidad.

El plan de prácticas seguirá, más o menos, el mismo orden que siguió el proyectista original en la realización de su aplicación. Se empezará por el sistema más básico, un sistema de paso de mensajes parecido al ejemplo del gateway que se encuentra en la documentación del patrón en el capítulo 3, aunque con la limitación de que todos los objetos se encuentran en el mismo programa y no en máquinas separadas. El siguiente paso en la creación de la aplicación será separar en dos programas diferentes los clientes y el servidor. Para conseguir esto habrá que hacer una separación del proxy en funciones relacionadas con los clientes y funciones relacionadas con el servidor. Además, las comunicaciones entre clientes y servidor se harán a partir de este momento mediante el protocolo de transporte TCP.

Una vez se han separado los programas cliente y servidor con éxito, se supone que ya se conoce y comprende la estructura del patrón en su totalidad. A partir de este momento el plan de prácticas se centra más en ampliar los conocimientos de .NET y C#. Se sustituirá el sistema de paso de mensajes por uno de paso de ficheros, con esto se consigue un conocimiento básico del manejo de archivos desde C#. El paso final será una pequeña introducción al tratamiento de ficheros multimedia.

6.1. Práctica 1: Sistema de paso de mensajes

En esta sesión se realizará una implementación muy básica del patrón mediante el lenguaje C#. Se construirá un sistema de paso de mensajes simple. Se proporcionarán varias clases con tan sólo los interfaces declarados, a excepción de la lista de activación y el organizador, que estarán ya implementados y no será necesario modificar.

El sistema de mensajes es el más simple que se puede pensar. En el sistema de mensajes que se propone, se dejan mensajes y se extraen sin ningún tipo de identificación sobre a quién iban dirigidos o si los ha extraído la persona correcta. Funcionaría de forma similar a un servicio técnico, los clientes van dejando incidencias y los técnicos las van resolviendo por orden de llegada, una cada vez.

La aplicación, en definitiva, es muy sencilla. Todo lo que se pide es que sea posible dejar mensajes en el objeto activo y se puedan extraer posteriormente. Pero el objetivo es un primer contacto con el patrón objeto activo más que dar un sistema realista de paso de mensajes.

6.1.1. Objetivos de la práctica

Los objetivos principales de esta práctica son:

- Establecer la estructura básica del patrón objeto activo.
- Ofrecer un caso práctico de uso del patrón.
- Un primer contacto con el lenguaje C# y el entorno de desarrollo Visual Studio.

6.1.2. Contenidos

Además del apartado de objetivo y éste de contenidos, este guión de la primera práctica contiene los siguientes apartados:

- *Apartado 3: Especificación de los requisitos de la aplicación.* Se describe la aplicación que se pide construir y sus requisitos funcionales.
- *Apartado 4: Diseño.* Se describe la solución adoptada.
- *Apartado 5: Implementación.* Se describe el código que se proporciona.

6.1.3. Especificación de los requisitos de la aplicación

En esta primera práctica se trata de construir una aplicación que sea capaz de almacenar mensajes y enviarlos a quien los solicite con las siguientes características:

- Se almacenarán y enviarán en el orden cronológico en que llegaron.
- Para simplificar lo máximo posible, los mensajes no contienen ningún tipo de identificación sobre quien lo ha enviado o quien lo debe recibir.
- La aplicación debe incorporar un interfaz gráfico desde el que dejar y recibir mensajes.
- Se extraen los mensajes de uno en uno.

El objetivo de esta práctica es el de establecer la estructura básica del patrón y no un sistema complejo de paso de mensajes, por lo que bastará con la posibilidad de dejar mensajes y poder extraerlos después.

6.1.4. Diseño

La figura XX muestra la estructura de la aplicación a desarrollar en esta práctica. El punto de partido es un proyecto de Visual Studio para una aplicación de ventana. El entorno gráfico contiene un objeto Proxy. Este objeto Proxy es la puerta de entrada al servidor, y es al objeto que se le harán peticiones.

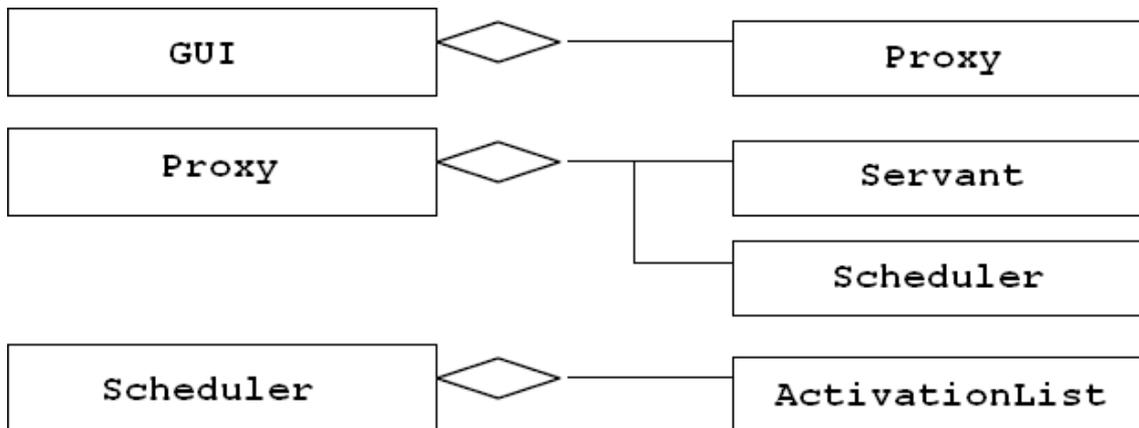


Figura 30: Estructura de la práctica 1

El interfaz gráfico que se usará es el mostrado en la figura XX. Sólo son dos necesarios dos botones y dos cajas de texto. Cuando se pulse el botón *enviar mensaje* se realizará una

petición al Proxy para que almacene un mensaje en su lista. Cuando se realiza una petición con el botón *recibir mensaje* aparecerá en la caja de texto en gris el primer mensaje almacenado de la lista.

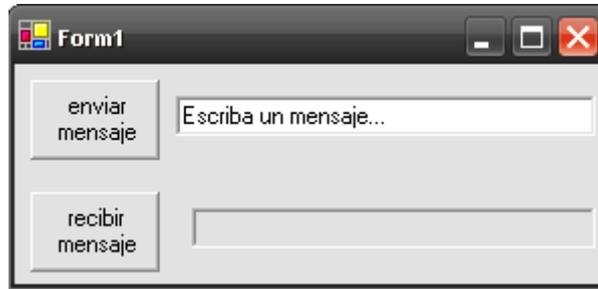


Figura 31: Interfaz gráfica de la práctica 1

6.1.5. Implementación

La práctica consiste en la implementación del proxy, el servant, los method requests correspondientes con cada petición y completar la interfaz gráfica.

En la interfaz gráfica hay que añadir la gestión de los eventos relacionados con la pulsación de los botones.

Respecto al Proxy, hay que completar tres métodos: el constructor *Proxy(int size)* y los métodos relaciones con las peticiones *leerMensaje()* y *dejarMensaje(string mensaje)*.

En el servant son cinco los métodos que hay que completar: el constructor *Servant(int size)*, los métodos de guarda *puedoDejarMensaje()* y *puedoLeerMensaje()* y los dos métodos que ejecutan las peticiones de los clientes *LDejarMensaje(string msg)* y *leerMensaje()*.

Por último, los method requests relacionados con ambas peticiones deberán ser completados con todo lo necesario para que el servant sea capaz de su ejecución. Además deberán implementar la interfaz del method request.

6.2. Práctica 2: Sistema de paso de mensajes con sockets.

Una vez se tiene la estructura básica del patrón establecida en la práctica 1, es el momento de realizar la primera modificación para que se acomode a las necesidades que tiene la aplicación final. La primera de estas necesidades es que clientes y servidor se encuentran en máquinas distintas, por lo que requieren de comunicaciones mediante sockets y sobre todo, separación de las funciones del proxy.

6.2.1. Objetivos de la práctica

Los principales objetivos de esta sesión son:

- Separar clientes de servidor, convirtiéndolos en aplicaciones independientes mediante el uso de sockets.
- Realizar modificaciones al patrón para adaptarlo a las necesidades de la aplicación.

6.2.2. Contenidos

Además del apartado de objetivo y éste de contenidos, este guión de la segunda práctica contiene los siguientes apartados:

- Apartado 3: Especificación de los requisitos de la aplicación. Se describe la aplicación que se pide construir y sus requisitos funcionales.
- Apartado 4: Diseño. Se describe la solución adoptada.
- Apartado 5: Implementación. Se describe el código que se proporciona.

6.2.3. Especificación de los requisitos de la aplicación

Los requisitos para dar por complete esta práctica son:

- Los clientes no deben bloquearse mientras esperar respuesta.
- El servidor deberá escuchar a los clientes concurrentemente.
- Debe estar disponible una GUI para cada uno de los programas desde la que poder acceder a las opciones que ofrezcan cada uno.

6.2.4. Diseño

La separación de la aplicación en dos programas independientes supone tener que separar las funciones del proxy y del comportamiento de los Method Request y el Servant. En la práctica anterior el proxy hacía de intermediario entre el cliente y el objeto activo, al separar el servidor del cliente, esta labor no se puede realizar directamente.

El proxy se encuentra en la máquina cliente y por tanto no tiene acceso directo al objeto activo, que se encuentra en una máquina diferente. Por ello, hay dos consecuencias: las funciones para encolar peticiones ya no son necesarias y necesita poder acceder al objeto activo de alguna manera. Para esto se crea una nueva clase, que correrá en el lado del servidor, que será con la que el proxy se pondrá en contacto y se encargará de redirigir las peticiones del proxy hacia el objeto activo. En definitiva, las funciones de encolamiento del proxy se transfieren a esta nueva clase, Skeleton, y a su vez se comunicará con ella mediante sockets para comunicarle las peticiones de los clientes.

La estructura del programa cliente es la más simple y se muestra en la siguiente figura:

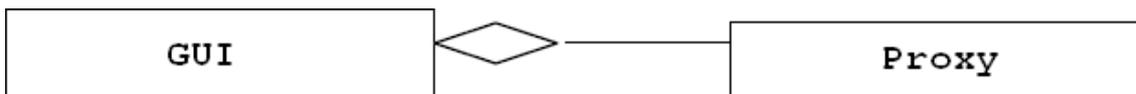


Figura 32: Estructura del programa cliente de la práctica 2

Consta de tan sólo dos componentes. La interfaz gráfica y el proxy que será creado desde la GUI durante su arranque. El aspecto de la GUI es el siguiente:



Figura 33: Interfaz gráfica del programa cliente de la práctica 2

El funcionamiento es exactamente igual que en la práctica anterior. Con el botón *Dejar Mensaje* se pide al proxy que deje un mensaje en la cola y con el botón *Leer Mensaje* se extrae. Además no debería bloquearse la ventana mientras se realiza la petición.

La estructura del servidor no ha cambiado demasiado ya que sólo se han separado funciones del proxy y no se ha añadido ninguna nueva, excepto funcionar por sockets.

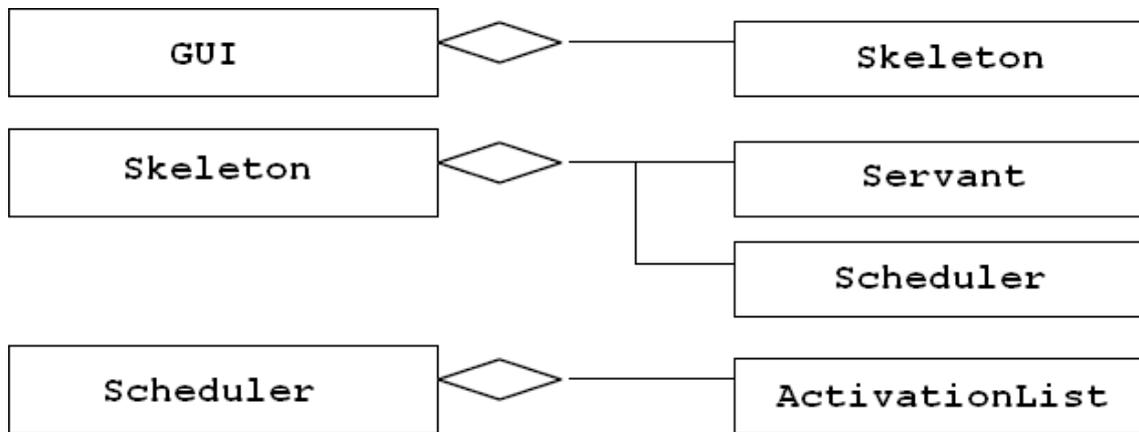


Figura 34: Estructura del programa servidor de la práctica 2

A primera vista, la única diferencia es que el Proxy se ha sustituido por una nueva clase llamada Skeleton. Internamente debe modificarse el comportamiento del Servant y los Method Request para que tengan en cuenta que deben hacer llegar el resultado mediante sockets a una máquina externa.

El aspecto de la interfaz gráfica es el siguiente:



Figura 35: Interfaz gráfica del programa servidor de la práctica 2

Al pulsar sobre el botón *Iniciar* el servidor empieza a escuchar peticiones. En el cuadro en blanco de abajo se pueden ir registrando los eventos que suceden.

6.2.5. Implementación

La práctica consiste en la implementación del proxy, el skeleton, el servant y los method requests de cada petición, además de completar la interfaz gráfica.

En la interfaz gráfica hay que añadir la gestión de los eventos relacionados con la pulsación de los botones.

Respecto al proxy, hay que completar tres métodos: el constructor *Proxy()* y los métodos relaciones con las peticiones *leerMensaje()* y *dejarMensaje(string mensaje)*.

El skeleton ahora contendrá toda la funcionalidad del proxy de la práctica 1 en lo que al servidor se refiere. Por tanto, hay que completar el constructor *Skeleton(int size)*, *leerMensaje(Socket client)* y *dejarMensaje(string mensaje)*. Además habrá que añadir cualquier método o clase necesaria para el tratamiento de los sockets de forma que se puedan conectar varios clientes concurrentemente.

En el servant son cinco los métodos que hay que completar, los mismos que en la práctica 1, que debido al cambio de contexto, hay que volverlos a implementar: el constructor *Servant(int size)*, los métodos de guarda *puedoDejarMensaje()* y *puedoLeerMensaje()* y los dos métodos que ejecutan las peticiones de los clientes *LDejarMensaje(string msg)* y *leerMensaje()*.

Por último, los method requests relacionados con ambas peticiones deberán ser completados con todo lo necesario para que el servant sea capaz de su ejecución. Además deberán implementar la interfaz del method request.

6.3. Práctica 3: Sistema de video. Transferencia de ficheros.

Esta práctica requiere de la aplicación de la práctica 2. Partiendo de esta aplicación, se sustituirá el sistema de paso de mensajes para que en vez de enviar y recibir mensajes, sea capaz de enviar y recibir archivos.

6.3.1. Objetivos de la práctica

Los principales objetivos de esta práctica son:

- Adaptar la aplicación de la práctica 3 para que, en vez de enviar y recibir mensajes, permita, enviar y recibir video.
- Toma de contacto con la librería IO de .Net.

6.3.2. Contenidos

Además del apartado de objetivo y éste de contenidos, este guión de la tercera práctica contiene los siguientes apartados:

- Apartado 3: Especificación de los requisitos de la aplicación. Se describe la aplicación que se pide construir y sus requisitos funcionales.
- Apartado 4: Diseño. Se describe la solución adoptada.
- Apartado 5: Implementación. Se describe el código que se proporciona.

6.3.3. Especificación de los requisitos de la aplicación

Los requisitos que debe cumplir la aplicación para considerarla correcta son:

- Sólo se deberán modificar las clases Servant, Proxy y Skeleton.
- Crear los nuevos Method Requests respetando el interfaz base.
- Disponer de todas las funciones de los programas Servidor y Cliente desde una GUI.

6.3.4. Diseño

El diseño de la aplicación es prácticamente el mismo que el de la práctica anterior. En el lado del servidor la interfaz gráfica es la misma, excepto por el botón añadido de opciones:



Figura 36: Interfaz gráfica del programa servidor de la práctica 3

Se ha añadido una ventana de opciones de configuración que permite cambiar los parámetros del tamaño de la cola, el puerto TCP que usará y el directorio donde se encuentran los videos a servir.



Figura 37: Panel de configuración del servidor de la práctica 3

En el lado del cliente sí ha habido más cambios. Ya no se pasan mensajes, sino archivos, por lo que habrá que tener una lista con los archivos disponibles, tanto remotos como locales, con los que se puede trabajar.



Figura 38: Interfaz del programa cliente de la práctica 3

También se han cambiado los botones, actualizándolos para que se correspondan con las peticiones que se pueden realizar ahora. El botón *Actualizar* es completamente nuevo y es el encargado de actualizar la lista de videos locales.

Además se ha añadido un menú principal desde el que se pueden seleccionar los botones *Conectar* y *Configuración*.

El botón de *Configuración* muestra la ventana de la siguiente figura. En ella se pueden cambiar algunos valores que serán tenidos en cuenta durante la conexión con el servidor. El parámetro de directorio de videos se puede cambiar en cualquier momento, incluso cuando ya se haya conectado con el servidor, y será plenamente funcional.



Figura 39: Panel de configuración del cliente de la práctica 3

Bastará con cerrar las ventanas de configuración de ambos programas para que se actualicen los parámetros.

6.3.5. Implementación

En esta práctica sólo se pide actualizar las interfaces gráficas de ambos programas para que se adecúen a los nuevos tipos de peticiones.

Por otro lado, habrá que añadir el menú principal y las ventanas de configuración donde corresponda. Estableciendo los correspondientes eventos para que se actualicen los parámetros de conexión.

Los métodos del proxy que han cambiado se proporcionan vacíos y será necesario completarlos para que permitan la transferencia de archivos en ambos sentidos de la comunicación.

En el servidor de nuevo hay que rehacer la clase servant por el cambio en las peticiones. También el skeleton requiere de modificaciones. Los method requests de las nuevas peticiones también cambian y hay que implementarlos.

6.4. Práctica 4: Sistema de video. Aplicación final.

La aplicación final se basará en la creada en la práctica 3, a la que habrá que añadir la opción de ver video sin necesidad de descargarlo.

6.4.1. Objetivos de la práctica

Los principales objetivos de esta práctica son:

- Creación de la petición tipo visionado en directo.
- Utilizar la librería DirectShowLib.

6.4.2. Contenidos

Además del apartado de objetivo y éste de contenidos, este guión de la primera práctica contiene los siguientes apartados:

- Apartado 3: Especificación de los requisitos de la aplicación. Se describe la aplicación que se pide construir y sus requisitos funcionales.
- Apartado 4: Diseño. Se describe la solución adoptada.
- Apartado 5: Implementación. Se describe el código que se proporciona.

6.4.3. Especificación de los requisitos de la aplicación

Los requisitos para dar por concluido el desarrollo de la aplicación son:

- Debido a las limitaciones de DirectShowLib, utilizar la clase Capture.cs y su objeto DxDPlay para hacer capturar de la imagen de un video.
- Enviar estas imágenes por el sockets del proxy del cliente.
- Una vez recibe estas imágenes el proxy, se encargará de mostrárselas por pantalla al usuario.

6.4.4. Diseño

En esta última práctica el diseño de los interfaces de usuario no ha variado prácticamente en nada. Sólo se ha añadido el botón correspondiente al nuevo tipo de petición soportado en el programa del cliente.



Figura 40: Interfaz gráfica del programa servidor de la práctica 4

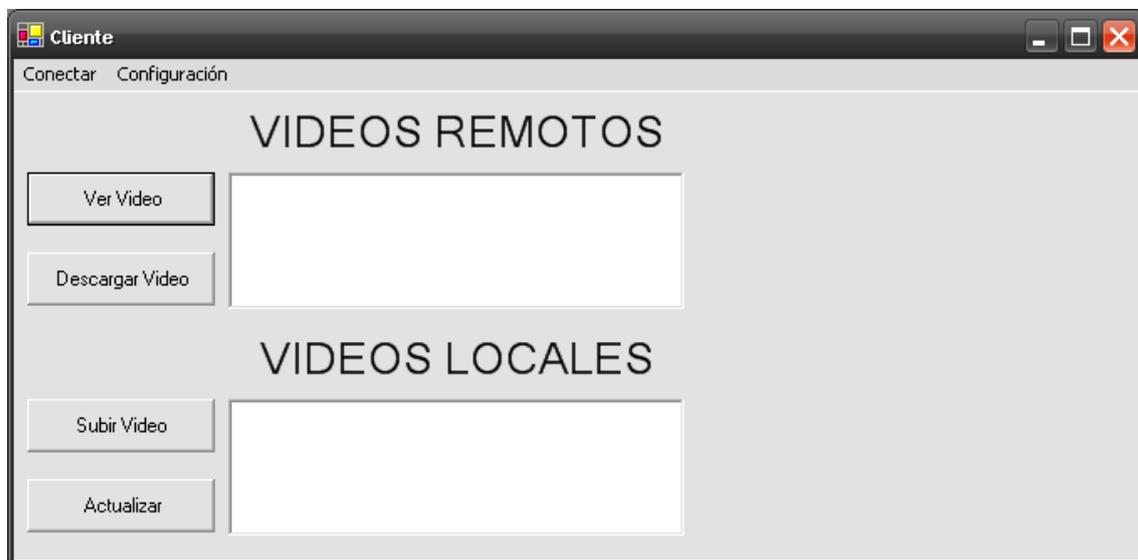


Figura 41: Interfaz gráfica del programa cliente de la práctica 4

6.4.5. Implementación

Lo último que queda por implementar desde cero es el Method Request correspondiente a la nueva petición.

El servant, skeleton y proxy requieren también de modificaciones, como en ocasiones anteriores, para soportar la nueva petición y sean capaces de agregarla a la lista de activación.

Capítulo 7: Manual de usuario

A lo largo de este manual se va a instruir al lector para que pueda utilizar la aplicación desarrollada de forma correcta. Esta explicación tratará de explicar cómo funcionan los programas Cliente y Servidor además de poner en conocimiento del usuario algunas librerías externas que son necesarias.

7.1. Programa Servidor

Como su nombre indica, el programa servidor recibirá las peticiones de los clientes y las ejecutará cuando sea posible hacerlo. El programa requiere de la librería DirectShowLib que debería venir incluida junto al ejecutable. Además es necesario tener instalados los codecs de video para poder ejecutar las peticiones de visionado en directo.

A la hora de arrancar el programa basta con lanzar el ejecutable. Una vez lanzado, aparecerá la ventana principal del servidor:



Figura 42: Ventana principal del servidor

La ventana se divide en dos zonas: la que contiene los botones *Iniciar* y *Opciones* y la que contiene la ventana de registro de eventos. En este registro de eventos se irá informando de conexiones que se produzcan y en versiones posteriores se podrá añadir información adicional como el tipo de peticiones que se realizan o mostrar un informe del número de peticiones almacenadas.

El botón de *Iniciar* es el que pone en marcha el servidor. No obstante, antes de iniciar conviene comprobar las opciones de configuración para ver si son correctas. Para ello basta con pulsa sobre el botón *Opciones*.

La ventana de opciones del servidor es la siguiente:



Figura 43: Ventana de configuración del servidor

Los parámetros que son posibles configurar se limitan al tamaño de peticiones que es posible encolar en la lista de activación, al puerto desde el que se escucharán peticiones y el directorio donde se encuentran los videos a servir.

Estas opciones deben estar determinadas antes de iniciar el servidor, pues una vez que se inicie no es posible cambiar ninguna opción de estas.

El programa utiliza el protocolo TCP, por lo que en caso de utilizar un router o un firewall, es necesario dar permisos de acceso a la red sobre este protocolo y al puerto elegido para que funcione.

Los videos se encuentran por defecto en la carpeta `.\Videos\` que se encuentra en la misma carpeta donde se encuentre el ejecutable del programa. Esta carpeta se puede cambiar por otra cualquiera. Es posible utilizar tanto rutas relativas como la que viene por defecto, como rutas absolutas si se necesita.

7.2. Programa Cliente

El programa cliente es absolutamente autónomo, por lo que no necesita de ninguna librería para funcionar. No obstante, en caso de descargar un video, es necesario el códec

correspondiente que permita su reproducción, aunque este proceso es ajeno al programa cliente.

Al igual que con el programa servidor, para arrancar el cliente basta con lanzar el ejecutable correspondiente. Lo primero que se mostrará será la siguiente ventana:

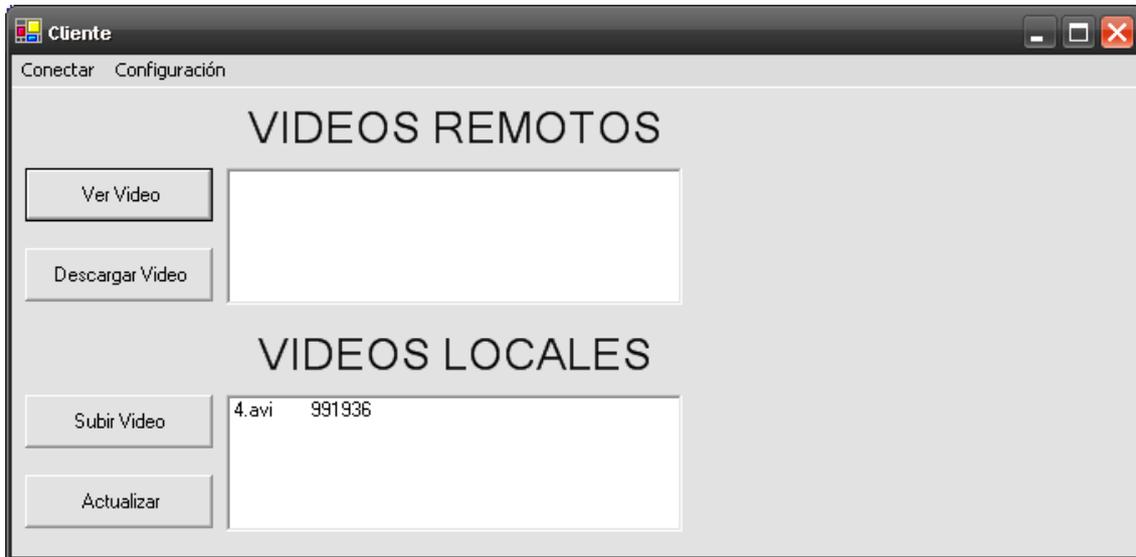


Figura 44: Ventana principal del cliente

Esta es la ventana principal del programa cliente. En ella se puede distinguir tres zonas claramente, además de un menú principal.

1.- El menú principal tiene dos botones: *Conectar* y *Configuración*. El primero realiza la conexión con el servidor según las opciones de configuración introducidas. El programa viene con unas opciones por defecto, que es posible cambiar con el botón correspondiente del menú principal, que suponen que el servidor está en la misma máquina y que atiende peticiones en el puerto 8888, más adelante se detallará el proceso para cambiar estas opciones.

2.- Peticiones disponibles. A la izquierda tenemos los tres botones con los que es posible realizar peticiones al servidor. Antes de poder realizar cualquier petición es necesario seleccionar un video de la lista de archivos correspondiente, además de haber conectado con el servidor. En caso de querer ver o descargar un video es necesario seleccionarlo de la lista de videos remotos. En caso de querer subir un video, habrá que seleccionarlo de la lista de videos locales.

3.- Listas de videos remotos y locales. Se muestran todos los videos disponibles tanto en la propia máquina, videos locales, como en la máquina del servidor, videos remotos. La lista de videos remotos se actualiza una vez se realice la conexión con el servidor. En caso de añadir videos a la carpeta local mientras se ejecuta el cliente, es necesario renovar la lista manualmente mediante el botón *Actualizar*.

4.- Zona de visualización de videos. En este espacio en blanco es donde se muestra el video en el caso de realizar la petición de visionado.

Antes de conectar con el servidor, es recomendable repasar las opciones de configuración para comprobar que son las correctas. Para ello se pulsa sobre el botón *Configuración*. Se mostrará la siguiente ventana:



Figura 45: Ventana de configuración del cliente

La función de las opciones es directa. En la **IP del Servidor** se especifica la dirección en la que se encuentra la máquina servidor, el **Puerto** determina el puerto TCP por el que se accederá a esta máquina y **Directorio Videos** especifica el directorio local donde se encuentran los videos del cliente además del directorio donde se guardarán los videos que se descarguen.

Una vez establecidas las opciones que se deseen, basta con pulsar sobre el botón *Conectar* del menú principal para establecer la conexión con el servidor. Automáticamente se mostrarán los videos disponibles para ver y descargar en el listado correspondiente.

Capítulo 8: Conclusiones y trabajos futuros

El proyecto está relacionado con uno de los servicios de más proyección actualmente en internet, la transmisión y la reproducción de contenido multimedia por la red. Aunque el eje central del proyecto giraba alrededor del patrón de diseño objeto activo, se ha intentado crear una aplicación que diera este servicio a una escala mucho menor de las disponibles ahora mismo en la red.

8.1. Conclusiones

Al inicio del proyecto se marcaron ciertos objetivos que debían cumplirse para dar por finalizado el proyecto.

El primer objetivo consistía en proporcionar al área de LSI ejemplos de utilización del lenguaje C# y del entorno .NET. En las primeras fases del proyecto se cumplió este objetivo al portar al lenguaje C# las prácticas de la asignatura de Fundamentos de la Programación de primero de Ingeniería Técnica Superior de Telecomunicaciones. Además, para cubrir temas relacionados con concurrencia, se portaron prácticas de otras asignaturas sobre este tema.

En este primer objetivo se puede incluir el plan de prácticas realizado, que proporciona al área un programa para la docencia del patrón y del lenguaje C#.

La documentación en castellano del patrón se encuentra en el capítulo 3 de esta misma memoria. Además se han proporcionado dos implementaciones diferentes del mismo. Una de ellas es la aplicación final en la que se utilizan objetos multimedia. La otra se encuentra en el plan de prácticas y se limita a realizar un sistema de paso de mensajes.

8.2. Trabajos futuros

Aunque acabada, la aplicación está lejos de estar completa o ser perfecta. Se le pueden realizar numerosas modificaciones para mejorarla.

Una de ellas consistiría en eliminar los cuellos de botella de los que adolece y ralentiza su funcionamiento. En principio existen dos claramente identificados: Visionado en directo y servant.

El primero de ellos se encuentra en las peticiones de visionado en directo de video. El sistema actual supone mantener al servidor ocupado el tiempo que dura el video. Lo ideal sería implementar este servicio de forma que se integrase con el de descarga. Así, mientras se estuviera descargando el video, sería posible su visualizado simultáneamente.

El segundo cuello de botella viene provocado por la estructura propia del patrón. Un único servant sirviendo peticiones puede resultar insuficiente ante altas cantidades de clientes y

peticiones. Mejorar este punto supondría dejar de usar el patrón objeto activo y otro más complejo, el patrón aceptor-conector. Este patrón tiene muchos puntos en común con el de objeto activo, pero lo importante es lo que los diferencia: en el patrón objeto activo hay un único servant, en el otro habrá tantos servant como sean necesarios y el hardware permita. Además con esta medida se proporcionaría una nueva posibilidad: repartir las peticiones según la carga de los servidores.

Otras líneas de trabajo más allá de solucionar cuellos de botella sería añadir más opciones a la hora de elegir qué petición se ejecutará en siguiente lugar. Añadiendo más variables a los objetos petición que se encolan en la activation list se podría hacer aun más complejo el sistema.

Capítulo 9: Bibliografía y referencias

[1] <http://msdn.microsoft.com/netframework/>

[2] <http://msdn.microsoft.com>

[3] “C# for Java Programmers”; Syngress

[4] “Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects” Volume 2; Douglas C. Schmidt; Wiley & Sons

[5] “Active Object: An Object Behavioral Pattern for Concurrent Programming”; R. Greg Lavender, Douglas C. Schmidt

[6] <http://directshownet.sourceforge.net/>

Anexo 1: Glosario de términos y siglas

En este anexo se proporciona un glosario con muchos de los términos y siglas utilizados a lo largo de esta memoria.

.NET: Tecnología de Microsoft que busca dar mayor facilidad y recursos al programador de aplicaciones.

Activation List: Componente del patrón objeto activo que mantiene la lista de peticiones pendientes de ejecución.

Active Object: Objeto activo en español. Patrón de diseño.

Aplicación: Programa informático que se ejecuta sin necesidad de ningún otro programa.

Appdomains: Dominios que se establecen internamente en una aplicación para aplicarles diferentes políticas de seguridad.

BCL: Biblioteca de clases básica de .NET.

C#: Lenguaje de programación orientado a objetos.

CAS: Code Access Security. Mecanismo de seguridad de .NET mediante ficheros adjuntos a los ensamblados para controlar los permisos de ejecución que estos tienen.

CIL: Common Intermediate Language. Es un lenguaje intermedio entre el de alto nivel como puede ser C# o Java y el lenguaje nativo que entiende la máquina.

Clase: Unidad fundamental en la programación orientada a objetos, que sirve como plantilla para la creación de objetos. Una clase define datos y métodos y es la unidad de organización básica de un programa.

CLI: Arquitectura software que se basa en la compilación de los programas en un lenguaje intermedio para dotar de interoperabilidad entre distintos lenguajes además de otras características.

CLR: Common Language Runtime. Es el nombre de la implementación de la arquitectura CLI que ha realizado Microsoft en .NET.

CLS: Common Language Specification. Es la especificación donde se recogen todos los tipos de datos, estructuras y reglas que debe seguir un lenguaje para que se pueda usar en la plataforma .NET.

Concurrent Object: Objeto concurrente en español. Es un nombre alternativo para el patrón de diseño objeto activo.

Componente: Parte de una estructura o un objeto.

Concrete Method Request: Implementación específica de un Method Request.

Constructor: Método que tiene el mismo nombre que la clase que inicia. Toma cero o más parámetros y proporciona unos datos u operaciones iniciales dentro de una clase, que no se pueden expresar como una simple asignación.

CTS: Common Type System. Documento con el que Microsoft describe el tratamiento que se le da a los diferentes tipos de datos soportados por .NET.

DirectShow: Librería de objetos, clases y funciones multimedia existente en varios lenguajes de programación, C y C# entre ellos.

DirectShowNet: Nombre del proyecto encargado del desarrollo de la librería DirectShowLib.

DirectShowLib: Nombre de la librería de libre distribución que intenta sustituir la librería DirectShow oficial de Microsoft.

Dispatch: Método en el scheduler que se encarga de comprobar qué peticiones cumplen sus métodos de guarda, extraerlas de la activation list y mandarlas ejecutarlas por el servant.

DxPlay: Componente de la librería Capture del paquete DirectShowLib que permite tomar capturas de la imagen de un video.

FCL: Biblioteca de clases que incorpora el espacio de nombres de Microsoft a la librería básica BCL de .NET.

Framework: Marco de trabajo. Conjunto de librerías, especificaciones, etc, que permite trabajar siguiendo unos estándares.

Future: Componente del patrón objeto activo que sirve de punto de unión entre el resultado de una petición y el cliente que la solicitó.

GAC: Global Assembly Cache. Caché en la que .NET va almacenando todos los ensamblados que estén siendo utilizados.

Gateway: Sistema encargado de redirigir mensajes u otra información desde un ordenador a otro.

GUI: Graphical User Interface. Interfaz gráfica de usuario, es la ventana desde la que se usa y configura un programa.

Interfaz: Un interfaz ofrece la funcionalidad de un objeto a través de sus métodos.

Java: Lenguaje de programación orientado a objetos.

JIT: Just-In-Time compiler. Compilador en tiempo real, se encarga de convertir el código intermedio en código nativo de la máquina durante la ejecución.

Librería: Conjunto de funciones y clases que ofrecen una funcionalidad.

Method Request: Componente del patrón objeto activo que ofrece el interfaz que deben cumplir todas las peticiones para su incorporación al patrón.

Método: Conjunto de sentencias que operan sobre los datos de la clase para manipular su estado.

MSIL: Microsoft Intermediate Language. Nombre que recibe la implementación de Microsoft del lenguaje intermedio usado en .NET.

Paquete: Nombre para una librería de clases.

Patrón de diseño: Solución a un problema concreto.

Proceso: Instancia de un programa ejecutable.

Proxy: Componente del patrón objeto activo encargado de ofrecer la funcionalidad del servidor a los clientes enmascarando la implementación de éste.

Reachable: Etiqueta que se asigna a los objetos que están siendo utilizados y no se deben borrar en el proceso de recolección de basura.

Scheduler: Componente del patrón objeto activo encargado de introducir y extraer peticiones en la activation list.

Servant: Componente del patrón objeto activo encargado de ejecutar las peticiones que le mande el scheduler.

Skeleton: Componente del patrón objeto activo encargado de hacer de enlace entre el proxy y el servidor.

TCP: Protocolo del nivel de transporte.

VES: Virtual System Execution. Entorno de ejecución que permite la ejecución de código intermedio.

Visual Studio: Entorno de trabajo de Microsoft para la plataforma .NET.

Web2.0: Concepto de la nueva generación de servicios ofrecidos por internet en el que el usuario toma el control de los contenidos.