

# **Programación Cocoa con Foundation Framework**

## Acerca de este documento

---

Una vez que el programador Cocoa aprende el lenguaje Objective-C, su siguiente paso es empezar a conocer la extensa librería de clases y funciones que proporciona Cocoa. Este documento estudia Foundation Framework un conjunto de librerías para tareas comunes compartidas por Mac OS X y iPhone OS. Foundation Framework proporciona funcionalidad para el manejo de ficheros, los procesos e hilos, el runtime de configuración del sistema, la programación multihilo y sus técnicas de sincronización, la programación en red y los objetos distribuidos. Conocer estas librerías es una inestimable ayuda antes de afrontar el aprendizaje del extenso grupo de clases que proporciona Cocoa para crear interfaces gráficas de usuario. Application Kit Framework es el kit de desarrollo de aplicaciones gráficas de Mac OS X. UI Kit Framework es el correspondiente grupo de librerías para desarrollar aplicaciones para iPhone OS. En este documento, Foundation Framework se explica en el contexto de Mac OS X, pero casi todas las técnicas estudiadas aquí se pueden transportar sin cambios o con pocos cambios a iPhone OS.

Si no conoce el lenguaje Objective-C le recomendamos empezar leyendo "El Lenguaje Objective-C para programadores C++ y Java", también publicado en MacProgramadores. Si no conoce las librerías de programación Cocoa, le recomendamos estudiar primero Foundation Framework leyendo este tutorial. Al acabar este tutorial el lector estará mejor preparado para elegir entre iniciar el aprendizaje del Application Kit Framework o del UI Kit Framework.

## Nota legal

---

Este tutorial ha sido escrito por Fernando López Hernández para MacProgramadores, y de acuerdo a los derechos que le concede la legislación española e internacional el autor prohíbe la publicación de este documento en cualquier otro servidor web, así como su venta, o difusión en cualquier otro medio sin autorización previa.

Sin embargo el autor anima a todos los servidores web a colocar enlaces a este documento. El autor también anima a cualquier persona interesada en conocer el Foundation Framework a bajarse o imprimirse este tutorial.

Madrid, Enero del 2009

Para cualquier aclaración contacte con:

[fernando@DELITmacprogramadores.org](mailto:fernando@DELITmacprogramadores.org)

# Tabla de contenido

---

## TEMA 1: Manejo de ficheros y directorios

---

|   |    |
|---|----|
| 1. Gestión del sistema de ficheros .....                  | 8  |
| 1.1. Crear, copiar, y borrar ficheros y directorios ..... | 8  |
| 1.2. Enlaces y enlaces simbólicos .....                   | 9  |
| 1.3. Permisos .....                                       | 10 |
| 1.4. Atributos .....                                      | 10 |
| 1.5. Listar directorios .....                             | 13 |
| 1.6. Acceso al contenido de los ficheros .....            | 13 |
| 1.7. Current directory .....                              | 14 |
| 2. Handles de ficheros .....                              | 14 |
| 2.1. Crear un handle de fichero .....                     | 14 |
| 2.2. Leer y escribir .....                                | 15 |
| 2.3. Cerrar el handle de fichero .....                    | 15 |
| 3. Path utilities .....                                   | 16 |
| 3.1. Localizar ficheros del sistema .....                 | 17 |
| 4. Resolver alias .....                                   | 18 |
| 5. Servicios del workspace .....                          | 19 |
| 5.1. Abrir ficheros .....                                 | 19 |
| 5.2. Lanzar aplicaciones .....                            | 20 |
| 5.3. Obtener información de ficheros .....                | 21 |
| 5.4. Montar y desmontar unidades .....                    | 23 |

## TEMA 2: Objetos dato

---

|   |    |
|---|----|
| 1. Qué son los objetos dato .....               | 25 |
| 2. Objetos dato inmutables .....                | 26 |
| 2.1. Creación a partir de un buffer .....       | 26 |
| 2.2. Creación a partir de un fichero .....      | 26 |
| 2.3. Acceder al buffer .....                    | 27 |
| 2.4. Guardar el contenido del objeto dato ..... | 28 |
| 3. Objetos dato mutables .....                  | 28 |
| 3.1. Creación .....                             | 28 |
| 3.2. Leer y modificar el contenido .....        | 28 |
| 3.3. Copiar objetos dato .....                  | 29 |

## TEMA 3: Archivado y serialización

---

|   |    |
|---|----|
| 1. Introducción .....                                   | 31 |
| 2. Archivado .....                                      | 32 |
| 2.1. Objetos codificadores y objetos codificables ..... | 32 |
| 2.2. Crear un objeto archivador .....                   | 35 |
| 2.3. Crear un objeto desarchivador .....                | 35 |
| 2.4. Archivar un grafo de objetos .....                 | 36 |

|   |    |
|---|----|
| 2.5. Implementar un objeto codificable .....                | 37 |
| 2.6. Codificar y decodificar tipos fundamentales .....      | 40 |
| 2.7. Codificación condicional .....                         | 40 |
| 2.8. Restringir el soporte para objetos codificadores ..... | 41 |
| 3. Serialización .....                                      | 43 |
| 3.1. Listas de propiedades .....                            | 43 |
| 3.2. API de serialización.....                              | 44 |

---

TEMA 4: Configuración del runtime

|  |    |
|--|----|
| 1. Introducción .....                                | 47 |
| 2. Bundles.....                                      | 47 |
| 2.1. Anatomía de un bundle moderno .....             | 48 |
| 2.2. Información de configuración de un bundle ..... | 53 |
| 2.3. Paquetes y Finder.....                          | 60 |
| 2.4. API para gestión de bundles.....                | 61 |
| 2.5. Variables de entorno.....                       | 67 |
| 3. El sistema de preferencias .....                  | 68 |
| 3.1. Los dominios .....                              | 68 |
| 3.2. Acceso programático a las preferencias .....    | 70 |
| 3.3. Fijar las preferencias por defecto .....        | 71 |

---

TEMA 5: Gestión de procesos

|   |    |
|---|----|
| 1. Información de nuestro proceso.....                    | 73 |
| 1.1. Obtener información del proceso .....                | 73 |
| 1.2. Obtener información del host .....                   | 73 |
| 2. Crear nuevos procesos .....                            | 74 |
| 2.1. Obtener información sobre el proceso .....           | 74 |
| 2.2. Modificar el entorno de ejecución de un proceso..... | 75 |

---

TEMA 6: Programación multihilo

|   |    |
|---|----|
| 1. Conceptos básicos.....                           | 77 |
| 1.1. Tipos de hilos.....                            | 77 |
| 1.2. Técnicas de sincronización .....               | 78 |
| 1.3. Las señales .....                              | 79 |
| 1.4. Los bucles de sondeo.....                      | 79 |
| 1.5. Consideraciones de diseño .....                | 80 |
| 1.6. Alternativas a la programación multihilo ..... | 81 |
| 1.7. Los objetos operación .....                    | 82 |
| 1.8. Protección multihilo .....                     | 82 |
| 2. Crear hilos.....                                 | 83 |
| 2.1. Crear hilos Cocoa.....                         | 83 |
| 2.2. Crear hilos POSIX .....                        | 85 |
| 2.3. Hilos POSIX en Cocoa .....                     | 87 |
| 2.4. Configurar los hilos .....                     | 87 |

|   |     |
|---|-----|
| 2.5. Terminar los hilos .....                           | 89  |
| 3. Programación con objetos operación .....             | 91  |
| 3.1. Ejecución directa frente a colas de operación..... | 91  |
| 3.2. Operaciones no concurrentes y concurrentes.....    | 91  |
| 3.3. Operaciones no concurrentes .....                  | 92  |
| 3.4. Dependencias entre operaciones .....               | 94  |
| 3.5. Colas de operación .....                           | 95  |
| 3.6. Multiplicador de matrices.....                     | 96  |
| 3.7. Operaciones concurrentes .....                     | 103 |
| 3.8. Métodos sincronizados .....                        | 104 |
| 3.9. Responder a errores .....                          | 104 |

---

TEMA 7: Técnicas de comunicación y sincronización

---

|  |     |
|--|-----|
| 1. Comunicación y sincronización .....             | 107 |
| 2. Los cerrojos.....                               | 107 |
| 2.1. Cerrojos POSIX.....                           | 108 |
| 2.2. Cerrojos Cocoa .....                          | 108 |
| 2.3. Los bloques sincronizados .....               | 110 |
| 2.4. Secciones críticas en colecciones .....       | 111 |
| 3. Barreras de memoria y variables volátiles ..... | 112 |
| 4. Operaciones atómicas .....                      | 113 |
| 5. Cerrojos de sondeo .....                        | 114 |
| 6. Condiciones .....                               | 115 |
| 6.1. Condiciones POSIX .....                       | 115 |
| 6.2. Condiciones Cocoa.....                        | 118 |
| 7. Pipes.....                                      | 120 |
| 7.1. Pipes BSD .....                               | 120 |
| 7.2. Pipes Cocoa .....                             | 121 |
| 8. Memoria compartida .....                        | 123 |

---

TEMA 8: Gestión de eventos

---

|  |     |
|--|-----|
| 1. El gestor de ventanas de Cocoa .....          | 127 |
| 2. Gestión de eventos .....                      | 128 |
| 3. Los bucles de sondeo .....                    | 130 |
| 3.1. Anatomía de un bucle de sondeo .....        | 131 |
| 3.2. Los modos del bucle de sondeo .....         | 132 |
| 3.3. Obtener y ejecutar el bucle de sondeo ..... | 133 |
| 3.4. Terminar el bucle de sondeo .....           | 134 |
| 3.5. Tipos de fuentes.....                       | 136 |
| 3.6. Los temporizadores.....                     | 137 |
| 3.7. Ejecutar selectores en otros hilos.....     | 139 |
| 3.8. Bucle de sondeo Core Foundation .....       | 140 |
| 4. Las notificaciones.....                       | 142 |
| 4.1. Para qué sirven las notificaciones.....     | 142 |
| 4.2. Cuándo usar notificaciones.....             | 143 |

|      |  |     |
|------|--|-----|
| 4.3. | Los objetos notificación .....                 | 144 |
| 4.4. | Los centros de notificación .....              | 144 |
| 4.5. | Los centros de notificación distribuidos ..... | 145 |
| 4.6. | Las colas de notificación.....                 | 150 |

---

**TEMA 9: Programación en red**

---

|      |  |     |
|------|--|-----|
| 1.   | Introducción a la programación en red .....  | 153 |
| 2.   | Sockets BSD .....                            | 154 |
| 2.1. | Implementar un cliente.....                  | 154 |
| 2.2. | Implementar un servidor.....                 | 158 |
| 3.   | Resolución de nombres DNS .....              | 162 |
| 4.   | Objetos stream .....                         | 163 |
| 4.1. | Políticas de bloqueo .....                   | 163 |
| 4.2. | Procesar el stream.....                      | 165 |
| 5.   | Sockets con objetos Cocoa .....              | 172 |
| 5.1. | Representar descriptores de fichero .....    | 172 |
| 5.2. | Operaciones asíncronas.....                  | 172 |
| 6.   | Programación en red con Core Foundation..... | 178 |
| 6.1. | CFSocket.....                                | 178 |
| 6.2. | CFStream.....                                | 187 |
| 6.3. | CFHTTP y CFFTP .....                         | 187 |
| 6.4. | Servidor de HTTP .....                       | 192 |
| 7.   | Sistema de carga de URLs .....               | 194 |
| 7.1. | Conexión síncrona .....                      | 195 |
| 7.2. | Conexión asíncrona.....                      | 196 |
| 7.3. | Bajar ficheros .....                         | 198 |

---

**TEMA 10: Objetos distribuidos**

---

|      |  |     |
|------|--|-----|
| 1.   | Introducción .....                       | 203 |
| 2.   | Crear y acceder a objetos remotos .....  | 203 |
| 2.1. | Definir el objeto remoto .....           | 203 |
| 2.2. | Exportar un objeto remoto .....          | 204 |
| 2.3. | Acceder al objeto remoto .....           | 206 |
| 3.   | Arquitectura, conexiones y proxies ..... | 207 |
| 3.1. | Las conexiones.....                      | 208 |
| 3.2. | Los proxies .....                        | 209 |
| 4.   | Los puertos .....                        | 212 |
| 4.1. | Envío y recepción de mensajes .....      | 212 |
| 4.2. | Registrar los puertos.....               | 213 |
| 5.   | Autorizar conexiones .....               | 215 |
| 6.   | Manejo de errores en la conexión.....    | 216 |

# Tema 1

## Manejo de ficheros y directorio

---

### **Sinopsis:**

*Casi cualquier aplicación que queramos implementar acabará teniendo que usar ficheros. En este tema empezaremos viendo qué clases proporciona Cocoa para manejar ficheros y directorios.*

# 1. Gestión del sistema de ficheros

Para gestionar el sistema de ficheros, Foundation Framework proporciona la clase `NSFileManager`, la cual implementa operaciones como la creación, copia y borrado de ficheros y directorios, creación y evaluación de enlaces simbólicos, determinar o modificar sus atributos, extraer el contenido de los ficheros, listar el contenido de un directorio, o cambiar el current directory (como explicaremos en el apartado 1.7).

En este apartado vamos a empezar resumiendo qué métodos de la clase `NSFileManager` implementan esta funcionalidad. Esta clase proporciona métodos de instancia, con lo que no se puede usar directamente, sino que debemos obtener su instancia por defecto con el método `defaultManager`. Realmente este método nos proporciona una instancia de clase derivada de `NSFileManager` donde se encuentra implementada su funcionalidad:

```
NSFileManager* fm = [NSFileManager defaultManager];
```

## 1.1. Crear, copiar, y borrar ficheros y directorios

Para crear un directorio disponemos del método:

- `(BOOL)createDirectoryAtPath: (NSString*)path  
 attributes: (NSDictionary*)attributes`

Este método recibe el path absoluto o relativo del directorio a crear y un diccionario con los atributos del directorio (permisos, dueño, grupo, fecha de modificación, etc). Los posibles valores del parámetro `attributes` se resumen en la Tabla 1.1. Si en `attributes` pasamos `nil` se crea el directorio con los atributos por defecto. La operación retorna `YES` si tiene éxito.

Para crear un fichero se usa:

- `(BOOL)createFileAtPath: (NSString*)path  
 contents: (NSData*)contents  
 attributes: (NSDictionary*)attributes`

El parámetro `contents` recibirá un objeto `NSData` que es un buffer con los datos a escribir en el fichero. En caso de que este parámetro sea `nil`, se creará un fichero con 0 bytes. Por ejemplo, si tenemos una instancia de `NSFileManager` llamada `fm` podemos crear un fichero con el nombre `lock` vacío en el current directory de la forma:

```
if ([fm createFileAtPath:@"lock" contents:nil attributes:nil]
    == YES)
    printf("lock creado\n");
```

Para copiar un fichero o directorio desde un `srcPath` origen a un `toPath` destino podemos usar:

- `(BOOL)copyItemAtPath: (NSString*)srcPath  
 toPath: (NSString*)dstPath  
 error: (NSError**)error`

Si el parámetro `error` no es `nil` y se produce un error (es decir, el retorno es `NO`), se depositará en el objeto `error` una descripción del error que ha ocurrido.

También podemos mover o borrar un fichero o directorio con las operaciones:

- `(BOOL)moveItemAtPath: (NSString*)srcPath  
 toPath: (NSString*)dstPath  
 error: (NSError**)error`
- `(BOOL)removeItemAtPath: (NSString*)path  
 error: (NSError**)error`

## 1.2. Enlaces y enlaces simbólicos

Podemos crear un enlace (hard link) con el siguiente método, donde `srcPath` debe existir y `dstPath` es el enlace que queremos crear:

- `(BOOL)linkItemAtPath: (NSString*)srcPath  
 toPath: (NSString*)dstPath  
 error: (NSError**)error`

En UNIX los enlaces (hard links) nos permite asociar varios nombres al mismo fichero. Existe otro tipo de enlaces llamados enlaces simbólicos (soft link) que nos permiten crear un fichero especial que lo único que contiene es la ruta de otro fichero. Para crear un enlace simbólico usamos:

- `(BOOL)createSymbolicLinkAtPath: (NSString*)path  
 withDestinationPath: (NSString*)destPath  
 error: (NSError**)error`

Donde `path` es el fichero a crear apuntando a `destPath`.

Una característica de los enlaces simbólicos es que si el origen se borra el enlace simbólico seguirá existiendo aunque estará roto (broken link). Podemos preguntar por el fichero apuntado por un enlace simbólico y saber si este enlace simbólico está roto con:

- `(NSString*) destinationOfSymbolicLinkAtPath: (NSString*)path  
error: (NSError**)error`

El método devuelve el nombre del fichero apuntado por el enlace simbólico, o `nil` si se produce un error (p.e. `path` no es un enlace simbólico).

## 1.3. Permisos

Podemos comprobar si un fichero existe con los métodos:

- `(BOOL) fileExistsAtPath: (NSString*)path`
- `(BOOL) fileExistsAtPath: (NSString*)path  
isDirectory: (BOOL*)isDirectory`

El segundo método nos sirve además para comprobar si el `path` es un directorio. En caso de que `path` sea un enlace simbólico, lo que indican es si existe el fichero apuntado por el enlace simbólico.

También podemos preguntar si un `path` tiene permisos de lectura, escritura, ejecución y borrado con los métodos:

- `(BOOL) isReadableFileAtPath: (NSString*)path`
- `(BOOL) isWritableFileAtPath: (NSString*)path`
- `(BOOL) isExecutableFileAtPath: (NSString*)path`
- `(BOOL) isDeletableFileAtPath: (NSString*)path`

## 1.4. Atributos

Podemos obtener los atributos de un fichero con el método:

- `(NSDictionary*) attributesOfItemAtPath: (NSString*)path  
error: (NSError**)error`

El método devuelve un puntero a `NSDictionary` donde las posibles claves se resumen en la Tabla 1.1. Cada clave es un puntero a un objeto de tipo `NSString`.

| Atributo                            | Descripción  |
|-------------------------------------|--|
| <code>NSFileType</code>             | Indica si se trata de un fichero, directorio, enlace simbólico, socket, fichero especial de carácter o fichero especial de bloque. |
| <code>NSFileSize</code>             | Tamaño en bytes del fichero  |
| <code>NSFileCreationDate</code>     | Fecha de creación del fichero  |
| <code>NSFileModificationDate</code> | Fecha de modificación del fichero  |
| <code>NSFileReferenceCount</code>   | Número de enlaces (hard link) al fichero   |

|                             |  |
|-----------------------------|--|
| NSFileSystemNumber          | Número único que identifica al fichero en el sistema de ficheros                                 |
| NSFileSystemFileNumber      | Número único del fichero dentro del sistema de ficheros tal como lo devuelve <code>stat()</code> |
| NSFileDeviceIdentifier      | Identificador del dispositivo en el que reside el fichero  |
| NSFilePosixPermissions      | Permisos POSIX del fichero   |
| NSFileOwnerAccountName      | Nombre del owner del fichero   |
| NSFileGroupOwnerAccountName | Nombre del grupo del fichero   |
| NSFileOwnerAccountID        | ID del owner del fichero   |
| NSFileGroupOwnerAccountID   | ID del grupo del fichero   |
| NSFileExtensionHidden       | Indica que es un fichero oculto  |
| NSFileHFSCreatorCode        | HFS creator code del fichero   |
| NSFileHFSTypeCode           | HFS type del fichero   |
| NSFileImmutable             | Indica que el fichero es inmutable   |
| NSFileAppendOnly            | Indica que al fichero sólo se puede añadir datos (útil para ficheros de log)                     |
| NSFileBusy                  | Booleano que indica si el fichero está siendo usado  |

**Tabla 1.1:** Atributos de fichero

El Listado 1.1 contiene un programa que enumera todos los atributos de un fichero pasado como argumento.

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
    if (argc<2) {
        fprintf(stderr,"Indique fichero como argumento\n");
        return 1;
    }
    NSFileManager* fm = [NSFileManager defaultManager];
    NSString* fichero =
        [fm stringWithFileSystemRepresentation:argv[1]
            length:strlen(argv[1])];
    NSDictionary* dict = [fm attributesOfItemAtPath:fichero
                           error:nil];
    for (NSString* clave in dict) {
        printf("%s=%s\n", [clave UTF8String],
               [[dict valueForKey:clave] UTF8String]);
    }
    [pool drain];
    return 0;
}
```

**Listado 1.1:** Programa que muestra los atributos de un fichero

El método `stringWithFileSystemRepresentation:length:` permite convertir el parámetro `argv` de la función `main()` en un `NSString`. Un ejemplo de ejecución sería:

```
$ atributos /Users/fernando
NSFileReferenceCount=28
NSFileModificationDate=2008-03-11 19:20:40 +0100
NSFileSystemFileNumber=297512
NSFileExtensionHidden=0
NSFileOwnerAccountName=fernando
NSFileGroupOwnerAccountName=staff
NSFileSize=1360
NSFileType=NSFileTypeDirectory
NSFileOwnerAccountId=501
NSFileCreationDate=2007-11-11 10:35:33 +0100
NSFileGroupOwnerId=20
NSFilePosixPermissions=493
NSFileSystemNumber=234881027
```

También podemos modificar los atributos de un fichero con el método:

- `(BOOL) setAttributes: (NSDictionary*) attributes  
ofItemAtPath: (NSString*) path  
error: (NSError**) error`

Además de los atributos de fichero, la clase `NSFileManager` también permite obtener los atributos de un sistema de ficheros que tengamos montado con el método:

- `(NSDictionary*) attributesOfFilesystemForPath: (NSString*) path  
error: (NSError**) error`

La Tabla 1.2 resume estos atributos.

| <b>Atributo</b>                    | <b>Descripción</b>  |
|------------------------------------|---|
| <code>NSFileSystemSize</code>      | Tamaño en bytes del sistema de ficheros tal como devuelve <code>statfs()</code> |
| <code>NSFileSystemFreeSize</code>  | Bytes libres en el sistema de ficheros tal como devuelve <code>statfs()</code>  |
| <code>NSFileSystemNodes</code>     | Número de nodos en el sistema de ficheros                                       |
| <code>NSFileSystemFreeNodes</code> | Número de nodos libres en el sistema de ficheros                                |
| <code>NSFileSystemNumber</code>    | Número del sistema de ficheros tal como devuelve <code>stat()</code>            |

**Tabla 1.2:** Atributos de sistema de ficheros

## 1.5. Listar directorios

Podemos obtener el contenido de un directorio con el método:

- `(NSArray*) contentsOfDirectoryAtPath: (NSString*) path  
error: (NSError**) error`

El cual devuelve un array de punteros a objetos de tipo `NSString` con los ficheros y subdirectorios del directorio dado en `path`. Este listado no es recursivo y no contiene los subdirectorios especiales `"."` y `".."`, pero sí que contiene ficheros cuyo nombre empieza por punto (p.e. `.bash_profile`).

Si queremos hacer un listado recursivo podemos usar:

- `(NSArray*) subpathsOfDirectoryAtPath: (NSString*) path  
error: (NSError**) error`

## 1.6. Acceso al contenido de los ficheros

Para acceder al contenido de un fichero podemos usar el método:

- `(NSData*) contentsAtPath: (NSString*) path`

Que dado un fichero devuelve un `NSData` que representa un buffer con el contenido del fichero. La forma de trabajar con los objetos de tipo `NSData` la veremos en Tema 2.

Podemos crear un fichero a partir de un `NSData` con el método:

- `(BOOL) createFileAtPath: (NSString*) path  
contents: (NSData*) contents  
attributes: (NSDictionary*) attributes`

También podemos comprobar si dos ficheros tienen el mismo contenido con el método:

- `(BOOL) contentsEqualAtPath: (NSString*) path1  
andPath: (NSString*) path2`

El cual devuelve `YES` cuando los ficheros tengan el mismo contenido.

## 1.7. Current directory

El **current directory** es el directorio desde donde se ejecuta la aplicación y desde donde se evalúan los paths relativos. Podemos preguntar con el current directory de una aplicación con el método:

- `(NSString*)currentDirectoryPath`

Y cambiar el current directory con el método:

- `(BOOL)changeCurrentDirectoryPath:(NSString*)path`

## 2. Handles de ficheros

---

Las instancias de la clase `NSFileHandle` representan ficheros abiertos, así como otro tipos de recursos como los sockets (que veremos en el apartado 2 del Tema 9).

### 2.1. Crear un handle de fichero

Para crear un handle de fichero usamos métodos factory de `NSFileHandle` como:

- + `(id)fileHandleForReadingAtPath:(NSString*)path`
- + `(id)fileHandleForWritingAtPath:(NSString*)path`
- + `(id)fileHandleForUpdatingAtPath:(NSString*)path`

El primer método permite sólo leer de fichero, el segundo permite sólo escribir, y el tercero permite leer y escribir.

También podemos obtener un handle a `stdin`, `stdout` y `stderr` con los métodos factory singleton:

- + `(id)fileHandleWithStandardInput`
- + `(id)fileHandleWithStandardOutput`
- + `(id)fileHandleWithStandardError`

Y podemos crear un handle de fichero a partir de un **descriptor de fichero** POSIX, es decir, un descriptor obtenido con `open()` usando el método:

- `(id)initWithFileDescriptor:(int)fileDescriptor`

En el apartado 5.1 del Tema 9 veremos que también podemos pasar a este método un descriptor de socket POSIX obtenido de una llamada a `socket()`.

## 2.2. Leer y escribir

Podemos usar los siguientes métodos de `NSFileHandle` para leer o escribir de un handle de fichero:

- `(NSData*) readDataOfLength: (NSUInteger) length`
- `(NSData*) readDataToEndOfFile`
- `(void) writeData: (NSData*) data`

Para garantizar que los datos se escriben a disco disponemos de la operación:

- `(void) synchronizeFile`

También podemos movernos por el contenido de un fichero con:

- `(void) seekToFileOffset: (unsigned long long) offset`
- `(unsigned long long) seekToEndOfFile`

O preguntar por la posición actual con:

- `(unsigned long long) offsetInFile`

Para truncar el fichero a un determinado offset podemos usar la operación:

- `(void) truncateFileAtOffset: (unsigned long long) offset`

## 2.3. Cerrar el handle de fichero

Para cerrar un handle de fichero, `NSFileHandle` proporciona la operación:

- `(void) closeFile`

En caso de que el fichero lo hayamos creado a partir de un descriptor de fichero POSIX, no basta con llamar a este método, sino que además debemos ejecutar `close()` sobre el handle de fichero.

Una alternativa es crear un handle de fichero a partir de un descriptor de fichero POSIX con la operación:

- `(id) initWithFileDescriptor: (int) fileDescriptor  
closeOnDealloc: (BOOL) flag`

Y pasar YES en `flag`. En este caso cuando llamado a `closeFile`, también se cierra el descriptor de fichero POSIX.

## 3. Path utilities

---

El fichero `NSPathUtilities.h` incluye varias categorías y funciones relacionadas con la gestión de ficheros.

En concreto, el Listado 1.2 muestra los métodos para gestión de ficheros que añade a la clase `NSString` la categoría `NSStringPathExtensions`. El Listado 1.3 muestra el método `pathsMatchingExtensions` que se añade a `NSArray`.

```
@interface NSString (NSStringPathExtensions)

+ (NSString*)pathWithComponents:(NSArray*)components;
- (NSArray*)pathComponents;

- (BOOL)isAbsolutePath;

- (NSString*)lastPathComponent;
- (NSString*)stringByDeletingLastPathComponent;
- (NSString*)stringByAppendingPathComponent:(NSString*)str;

- (NSString*)pathExtension;
- (NSString*)stringByDeletingPathExtension;
- (NSString*)stringByAppendingPathExtension:(NSString*)str;

- (NSString*)stringByAbbreviatingWithTildeInPath;
- (NSString*)stringByExpandingTildeInPath;

- (NSString*)stringByStandardizingPath;

- (NSString*)stringByResolvingSymlinksInPath;

- (NSArray*)stringsByAppendingPaths:(NSArray*)paths;

- (NSUInteger)completePathToString:(NSString**)outputName
                           caseSensitive:(BOOL)flag
                           matches:(NSArray**)matches;

- (const char*)fileSystemRepresentation;
- (BOOL)getFileSystemRepresentation:(char*)cname
                           maxLength:(NSUInteger)max;
@end
```

**Listado 1.2:** Categoría de utilidades de fichero para `NSString`

```
@interface NSArray (NSArrayPathExtensions)

- (NSArray*)pathsMatchingExtensions:(NSArray*)filterTypes;
@end
```

**Listado 1.3:** Categoría de utilidades de fichero para `NSArray`

El fichero `NSPathUtilities.h` también declara las funciones:

```
NSString* NSHomeDirectory(void);
NSString* NSHomeDirectoryForUser(NSString*userName);
```

Que nos permiten conocer el directorio home de un usuario, y la función:

```
NSString* NSTemporaryDirectory(void);
```

Que nos permite conocer el directorio temporal.

Podemos combinar estas funciones con el método de `NSString`:

- `(NSString*)stringByAppendingPathComponent:(NSString*)aString`

El cual añade un nombre fichero a un path de directorio. Si el path de directorio no acaba en barra (/), la añade. Por ejemplo:

```
NSString* path = NSTemporaryDirectory();
path = [path stringByAppendingPathComponent:@"tmp_file.txt"];
```

En este ejemplo, tras ejecutar la primera línea `path` vale @"`/tmp`" y tras ejecutar la segunda línea acaba valiendo @"`/tmp/tmp_file.txt`".

### 3.1. Localizar ficheros del sistema

Cuando accedemos a los ficheros y directorios del sistema se recomienda evitar escribir su path hardcode en el código fuente, ya que si en el futuro cambiase la localización de estos ficheros nuestro programa dejaría de funcionar.

Para localizar ficheros del sistema se recomienda usar la función:

```
NSArray* NSSearchPathForDirectoriesInDomains(
    NSSearchPathDirectory directory,
    NSSearchPathDomainMask domainMask,
    BOOL expandTilde);
```

Que devuelve un array de objetos `NSString` con los paths que cumplen el criterio de búsqueda. El parámetro `directory` es un enumerado cuyos valores indican el directorio de interés (p.e. `/Applications`, `/Library`, `/System`, etc). El parámetros `domainMask` es otro enumerado que indica en qué dominios estamos interesados (`user`, `system`, `local`, `network`). A diferencia del parámetro anterior, el parámetro `domainMask` es una máscara en la que se pueden pasar varios parámetros con un or binario. Por último `expandTilde` indica si expandir la tilde (~) en los directorios retornados.

Tanto la función `NSSearchPathForDirectoriesInDomains` como los valores de sus parámetros se encuentran en el fichero de cabecera `NSPathUtilities.h`.

Como la función `NSSearchPathForDirectoriesInDomains` puede devolver varios paths, podemos recorrerlos todos (p.e. cuando buscamos un fichero) o bien usar el primero de la lista (p.e. cuando buscamos el directorio donde depositar un fichero) ya que el primero de la lista es el que más preferencia tiene para ese fin.

## 4. Resolver alias

---

Los enlaces y enlaces simbólicos son un concepto introducido en el sistema operativo de Apple con la llegada de Mac OS X. Cocoa maneja estos enlaces nativamente. Por el contrario, los alias son un concepto similar heredado de Mac OS Classic. Las aplicaciones Cocoa pueden encontrarse con alias, que para ellas son indistinguibles de los ficheros regulares, ya que Cocoa no proporciona ninguna operación para su gestión. Para gestionar alias debemos recurrir a funciones de Core Services.

Las funciones de Core Services son funciones de bajo nivel compartidas por Cocoa y Carbon. Estas funciones se encuentran declaradas en el fichero de cabecera `<CoreServices/CoreServices.h>` y para usarlas debemos de enlazar con `CoreServices.framework`. El Listado 1.4 muestra cómo implementar una función `ResuelveAlias()` para que dado un path a un alias devuelva el path al que apunta este alias.

```
#import <Foundation/Foundation.h>
#import <CoreServices/CoreServices.h>

NSString* ResuelveAlias(NSString* path_alias) {
    // Esta variable contendrá el path resuelto
    NSString* resolved_path = nil;
    // Crea una URL al path
    CFURLRef url = CFURLCreateWithFileSystemPath(kCFAllocatorDefault,
                                                    (CFStringRef)path_alias, kCFURLPOSIXPathStyle, NO);
    if (url==NULL)
        return NULL;
    // Resuelve el alias
    FSRef fs_ref;
    Boolean target_is_folder, was_aliased;
    if (CFURLGetFSRef(url, &fs_ref)) {
        OSerr err = FSResolveAliasFile(&fs_ref, true,
                                        &target_is_folder, &was_aliased);
        if (err==noErr && was_aliased) {
            CFURLRef resolved_url =
                CFURLCreateFromFSRef(kCFAllocatorDefault,
                                     &fs_ref);
```

```

        if (resolved_url!=NULL) {
            resolved_path = (NSString*) CFURLCopyFileSystemPath(
                resolved_url, kCFURLPOSIXPathStyle);
            CFRelease(resolved_url);
        }
    }
}
CFRelease(url);
// Si no se ha resuelto el alias devolvemos el path recibido
if (resolved_path == nil)
    resolved_path = [NSString stringWithFormat:path_alias];
return resolved_path;
}

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    if (argc<2) {
        fprintf(stderr,"Indique alias como argumento\n");
        return 1;
    }
    NSString* path_alias = [NSString stringWithUTF8String:argv[1]];
    NSString* path_resuelto = ResuelveAlias(path_alias);
    printf("%s apunta a %s\n", [path_alias UTF8String],
           [path_resuelto UTF8String]);
    [pool drain];
    return 0;
}

```

**Listado 1.4:** Programa que resuelve un alias

## 5. Servicios del workspace

---

La clase `NSWorkspace` encapsula servicios proporcionados por Finder. Para acceder a estos servicios la clase proporciona el método factory singleton:

+ (`NSWorkspace*`) sharedWorkspace

En los siguientes apartados vamos a ver qué funcionalidad proporciona este objeto.

### 5.1. Abrir ficheros

Cuando hacemos doble-click en un fichero, Finder abre el fichero. Podemos acceder a esta funcionalidad con el método:

- (`BOOL`) openFile: (`NSString*`) fullPath

Que abre el fichero usando su aplicación por defecto y devuelve YES si la operación ha tenido éxito.

En caso de querer abrir el fichero con una aplicación distinta a la por defecto podemos usar:

- (BOOL)openFile: (NSString\*) fullPath  
withApplication: (NSString\*) appName

En este caso nuestra aplicación es desactivada para dar el foco a la aplicación lanzada. Si no queremos que nuestra aplicación pierda el foco podemos pasar NO en el parámetro `flag` del siguiente método:

- (BOOL)openFile: (NSString\*) fullPath  
withApplication: (NSString\*) appName  
andDeactivate: (BOOL) flag

Si lo que queremos es mostrar un fichero en Finder, podemos usar el método:

- (BOOL)selectFile: (NSString\*) fullPath  
inFileViewerRootedAtPath: (NSString\*) rootFullPath

Si `rootFullPath` es @"" se muestra el fichero en la ventana de Finder actualmente abierta, sino se crea otra ventana de Finder.

En caso de que lo que tengamos sea una URL, podemos pedir abrir la URL en el navegador web por defecto usando el método:

- (BOOL)openURL: (NSURL\*) url

## 5.2. Lanzar aplicaciones

Para ejecutar una aplicación disponemos del método:

- (BOOL)launchApplication: (NSString\*) appName

En `appName` debemos de proporcionar sólo en nombre (no el path) de la aplicación. La extensión .app es opcional. Por ejemplo:

```
NSWorkspace* w = [NSWorkspace sharedWorkspace];
[w launchApplication:@"Chess"];
```

Dado el nombre de una aplicación, podemos conocer el path de ésta usando el método:

- (NSString\*)fullPathForApplication: (NSString\*) appName

Podemos conocer las aplicaciones que actualmente están en el Dock con el método:

- (NSArray\*)launchedApplications

El método nos devuelve un array con un diccionario por cada aplicación. Por ejemplo, si tenemos este programa:

```
NSWorkspace* w = [NSWorkspace sharedWorkspace];
NSArray* apps = [w launchedApplications];
printf("%s\n", [[apps description] UTF8String]);
```

Al ejecutarlo obtendremos:

```
(  
{  
    NSApplicationBundleIdentifier = "com.apple.finder";  
    NSApplicationName = Finder;  
    NSApplicationPath =  
        "/System/Library/CoreServices/Finder.app";  
    NSApplicationProcessIdentifier = 468;  
    NSApplicationProcessSerialNumberHigh = 0;  
    NSApplicationProcessSerialNumberLow = 61455;  
},  
{  
    NSApplicationBundleIdentifier = "com.apple.Safari";  
    NSApplicationName = Safari;  
    NSApplicationPath = "/Applications/Safari.app";  
    NSApplicationProcessIdentifier = 536;  
    NSApplicationProcessSerialNumberHigh = 0;  
    NSApplicationProcessSerialNumberLow = 118813;  
},  
{  
    NSApplicationBundleIdentifier = "com.apple.Xcode";  
    NSApplicationName = Xcode;  
    NSApplicationPath = "/Developer/Applications/Xcode.app";  
    NSApplicationProcessIdentifier = 638;  
    NSApplicationProcessSerialNumberHigh = 0;  
    NSApplicationProcessSerialNumberLow = 159783;  
}  
)
```

También podemos saber cuál es la aplicación activa con el método:

- (NSDictionary\*)activeApplication

que nos devuelve el diccionario de la aplicación activa.

### 5.3. Obtener información de ficheros

Podemos solicitar al objeto `NSWorkspace` información sobre la aplicación que abre un fichero y el tipo del fichero con el método:

- (BOOL) getInfoForFile: (NSString\*) fullPath  
application: (NSString\*\*) appName  
type: (NSString\*\*) type

Los parámetros `appName` y `type` son parámetros de salida con el nombre de la aplicación y su tipo (extensión).

También podemos obtener el UTI (Uniform Type Identifier) de un fichero con el método:

- `(NSString*) typeOfFile:(NSString*) absoluteFilePath  
error:(NSError**) outError`

Por ejemplo, el siguiente programa:

```
NSWorkspace* w = [NSWorkspace sharedWorkspace];
NSString* f = @"/Users/fernando/Alumnos.xls";
NSString* app,* tipo;
[w getInfoForFile:f application:&app type:&tipo];
[app retain];
[tipo retain];
NSString* uti = [w typeOfFile:f error:nil];
printf("App:%s\ntipo:%s\nUTI:%s\n"
      ,[app UTF8String],[tipo UTF8String],[uti UTF8String]);
```

produce la salida:

```
App:/Applications/Microsoft Office 2004/Microsoft Excel
tipo:xls
UTI:com.microsoft.excel.xls
```

Dado un tipo UTI podemos preguntar por su extensión preferida con el método:

- `(NSString*)  
preferredFilenameExtensionForType:(NSString*) typeName`

También podemos preguntar por el ícono que corresponde a un fichero o a un tipo de fichero usando respectivamente:

- `(NSImage*) iconForFile:(NSString*) fullPath`
- `(NSImage*) iconForFileType:(NSString*) fileType`

Cuando un fichero no tiene asociado un ícono explícitamente, utiliza el ícono de su tipo.

Podemos cambiar el ícono de un determinado fichero con el método:

- `(BOOL)setIcon:(NSImage*) image  
forFile:(NSString*) fullPath  
options:(NSWorkspaceIconCreationOptions) options`

En `options` se suele especificar 0 a no ser que queramos asignar sólo ícono para determinados tamaños.

## 5.4. Montar y desmontar unidades

Podemos preguntar por las unidades de disco que están montadas con el método:

- `(NSArray*)mountedRemovableMedia`

Que devuelve un array de objetos cadena con el path de cada unidad montada.

También podemos pedir desmontar una unidad con el método:

- `(BOOL)unmountAndEjectDeviceAtPath: (NSString*)path`

Podemos pedir al objeto `NSWorkspace` que nos envíe una notificación cuando se monte o desmonte una unidad. Las notificaciones se estudiarán en el apartado 4 del Tema 8.

Si este documento le está resultando útil puede plantearse el ayudarnos a mejorarlo:

- Anotando los errores editoriales y problemas que encuentre y enviándlos al sistema de Bug Report de Mac Programadores.
- Realizando una donación a través de la web de Mac Programadores.

# Tema 2

## Objetos dato

---

### **Sinopsis:**

*Este tema está dedicado al estudio de los buffers de datos binarios y a su representación mediante objetos del Foundation Framework y del Core Foundation Framework.*

## 1. Qué son los objetos dato

---

Los **objetos dato** (data objects) son objetos que representan un buffer de bytes binario. El objeto dato no contiene información sobre el tipo de los datos contenidos en el buffer, sino que es responsabilidad del programa que crea el objeto dato decidir cómo interpretar los datos. Entre otros aspectos, el programa deberá tener en cuenta si los datos del buffer están guardados en big-endian o little-endian y realizar las conversiones oportunas. Además, el buffer debe de ser lineal, es decir, no puede contener punteros a otros datos.

Los objetos dato son objetos puente, es decir, se pueden usar tanto desde Cocoa como desde Core Foundation: Cocoa dispone de las clases `NSData` y `NSMutableData` para manejarlos. Por su parte Core Foundation usa los tipos `CFDataRef` y `CFMutableDataRef` para representarlos. Al ser objetos puente, los punteros a sus tipos son intercambiables, es decir, podemos hacer casting entre `NSData*` y `CFDataRef`, así como entre `NSMutableData*` y `CFMutableDataRef`.

El término *objeto dato* se utiliza para referirse tanto a buffers binarios mutables como inmutables. Cuando queramos indicar que el buffer puede ser modificado después de la creación del objeto dato se utiliza el término **objeto dato mutable**. Cuando sea importante la distinción se usa el término **objeto dato inmutable** para indicar que su contenido no puede ser modificado una vez creado.

Un objeto dato queda definido por los bytes que almacena y por su longitud. Después podemos extraer un rango de bytes, comparar dos objetos dato o escribir los datos del objeto dato en un fichero o conexión de red. En caso de que el objeto dato sea mutable podemos modificar los datos que contenga después de crearlo, así como añadir nuevos datos o modificar su longitud.

## 2. Objetos dato inmutables

---

En este apartado empezaremos comentando los objetos dato inmutables representados por la clase `NSData` y en el siguiente apartado comentaremos los objetos dato mutables representados la clase `NSMutableData`. La clase `NSMutableData` deriva de `NSData` para añadir método que permiten modificar su contenido.

### 2.1. Creación a partir de un buffer

Podemos crear un objeto `NSData` a partir de los siguientes métodos factory:

```
+ (id)dataWithBytes:(const void*)bytes
              length:(NSUInteger)length
```

El cual, a partir de un array de bytes y su longitud crea el objeto buffer. Este método hace una copia de los bytes que le pasamos en el parámetro `bytes` y libera este buffer cuando se destruye el objeto. Es nuestra responsabilidad el liberar el buffer original.

También podemos usar el método factory:

```
+ (id)dataWithBytesNoCopy:(void*)bytes
                     length:(NSUInteger)length
```

para que no se haga una copia de los `bytes` pasados, sino que el objeto dato se hace responsable de este buffer y lo libera cuando acaba de trabajar con él<sup>1</sup>. Para que el objeto dato pueda liberar el buffer, éste deberá de haber sido creado con `malloc()`.

Si preferimos que no se copie ni se libere el buffer al destruir el objeto podemos pasar `NO` en el tercer parámetro del método factory:

```
+ (id)dataWithBytesNoCopy:(void*)bytes
                     length:(NSUInteger)length
                   freeWhenDone:(BOOL)freeWhenDone
```

### 2.2. Creación a partir de un fichero

Podemos crear un objeto dato a partir del contenido de un fichero con los métodos factory:

---

<sup>1</sup> La clase `NSMutableData` también puede usar este método, pero en este caso se hace una copia del buffer y se libera cuando se destruye el objeto.

```
+ (id) dataWithContentsOfFile:(NSString*)path
+ (id) dataWithContentsOfMappedFile:(NSString*)path
```

Estos métodos devuelven un objeto `NSData` con el contenido leído. El segundo de ellos no carga el contenido del fichero sino que lo mapea en memoria y su contenido sólo es cargado cuando es leído en memoria.

También podemos leer el contenido de una URL con:

```
+ (id) dataWithContentsOfURL:(NSURL*)aURL
```

## 2.3. Acceder al buffer

Para acceder al buffer de un objeto dato disponemos de los métodos:

- `(const void*) bytes`
- `(NSUInteger) length`

El primero devuelve un puntero al buffer almacenado en el objeto dato, el segundo devuelve la longitud del buffer.

Si queremos hacer una copia del buffer – en vez de acceder directamente al buffer almacenado en el objeto dato – podemos usar:

- `(void)getBytes:(void*)buffer`
- `(void)getBytes:(void*)buffer length:(NSUInteger)length`
- `(void)getBytes:(void*)buffer range:(NSRange)range`

Antes de usar estos métodos deberemos haber reservado memoria suficiente en `buffer`. El primero de ellos copia todos los datos en el buffer del objeto dato, con lo que `buffer` deberá medir al menos lo que devuelva el método `length` del objeto dato. El segundo de ellos permite indicar la longitud de los datos a copiar, y el tercero permite indicar un rango de datos a copiar.

Podemos crear un objeto dato a partir de parte del contenido de otro objeto dato con:

- `(NSData*) subdataWithRange:(NSRange)range`

Y también podemos preguntar si el contenido de los buffers de dos objetos dato es el mismo con el método:

- `(BOOL) isEqualToData:(NSData*)otherData`

## 2.4. Guardar el contenido del objeto dato

Podemos guardar el contenido de un objeto dato en fichero usando:

- `(BOOL)writeToFile:(NSString*)path atomically:(BOOL)flag`

También podemos escribir el contenido del objeto dato en una URL con:

- `(BOOL)writeToURL:(NSURL*)aURL atomically:(BOOL)atomically`

Aunque actualmente sólo se soporta el protocolo `file://`, con lo que este método es funcionalmente similar al anterior.

## 3. Objetos dato mutables

---

En este apartado estudiaremos los objetos dato mutables que están representados por la clase `NSMutableData`, que es una derivada de `NSData`.

### 3.1. Creación

Podemos crear un objeto `NSMutableData` usando las familias de métodos `data...` y `init...` que `NSMutableData` hereda de `NSData`.

Además podemos crear un objeto `NSMutableData` con un buffer de un determinado tamaño con el método `dactory`:

- + `(id)dataWithCapacity:(NSUInteger)aNumItems`

### 3.2. Leer y modificar el contenido

Podemos usar el siguiente método de `NSMutableData` para acceder al buffer mutable:

- `(void*)mutableBytes`

Si sólo queremos leer el buffer podemos usar el método heredado de `NSData`:

- `(const void*)bytes`

El modificador `const` está ahí para recordarnos que no debemos modificar el contenido de este buffer.

También podemos extender o truncar el buffer con:

- `(void)setLength:(NSUInteger)length`
- `(void)increaseLengthBy:(NSUInteger)extraLength`

El segundo de ellos sólo permite extender el buffer y los nuevos bytes se rellenan con ceros.

También podemos poner un rango de bytes a cero con:

- `(void)resetBytesInRange:(NSRange)range`

Y reemplazar un rango de bytes con el método:

- `(void)replaceBytesInRange:(NSRange)range withBytes:(const void*)bytes`

O bien cambiar el contenido del buffer con:

- `(void)setData:(NSData*)aData`

Por último podemos añadir datos al final del buffer con los métodos:

- `(void)appendData:(NSData*)otherData`
- `(void)appendBytes:(const void*)bytes length:(NSUInteger)length`

### 3.3. Copiar objetos dato

Tanto `NSData` como `NSMutableData` implementan las interfaces `NSCopying` y `NSMutableCopying` con lo que los objetos de su tipo se pueden copiar tanto de forma mutable como inmutable. En concreto, si enviamos al objeto el mensaje `copy` crearemos una copia inmutable (es decir, de tipo `NSData`) y si le enviamos el mensaje `mutableCopy` crearemos una copia mutable (es decir, de tipo `NSMutableData`).

# Tema 3

## Archivado y serialización

---

### **Sinopsis:**

*En el tema anterior aprendimos a trabajar con objetos dato que nos permiten leer y guardar en ficheros buffers binarios. En este tema se profundiza en dos técnicas que implementa Cocoa para una cómoda lectura y almacenamiento en ficheros de objetos y estructuras de datos: el archivado y la serialización.*

## 1. Introducción

---

El archivado y la serialización son dos formas de crear jerarquías de datos que se pueden leer y almacenar en buffers binarios (a través de objetos dato) de forma independiente de la arquitectura (big endian, little endian). El **archivado** permite almacenar de forma detallada los valores de un objeto, así como las relaciones entre estos objetos. Por contra, la **serialización** permite almacenar simples jerarquías de propiedades de forma más sencilla, aunque muy útil para determinadas aplicaciones.

Los ficheros .nib (NeXTSTEP Interface Builder) son un buen ejemplo de aplicación del archivado: Durante el desarrollo de una aplicación el programador crea un **grafo de objetos** formado por instancias de objetos y conexiones entre ellos (p.e. ventanas y jerarquías de vistas). Despues el programador (usando Interface Builder) guarda este grafo de objeto mediante archivado en disco. Cuando la aplicación se va a ejecutar este grafo de objetos se reconstruye en memoria. Se llama **conexiones** a las asociaciones entre los objetos del grafo de objetos que se almacenan y recuperan del fichero. En Xcode 3.0 se introdujeron los ficheros .xib (Xcode Interface Builder) los cuales almacenan la misma información que los .nib, pero en formato XML. La razón por la que se cambió el formato de estos ficheros fue para poder almacenarlos en gestores de versiones (como CVS o Subversion) de forma más eficiente. En concreto, al estar los ficheros .xib en formato texto, se puede enviar al gestor de versiones sólo los cambios, en vez de enviar todo el fichero binario. Cuando compilamos la aplicación se compila el fichero .xib en otro con formato .nib que es el que se distribuirá junto con la aplicación.

Los ficheros de propiedades (que tienen las extensión .plist) son un buen ejemplo de serialización, los cuales almacenan **listas de propiedades** (property lists) con una jerarquía simple muy usada para almacenar información de configuración de las aplicaciones.

En Mac OS X no se recomienda que las aplicaciones creen (y mantengan) sus propios formatos de datos, sino que se recomienda utilizar el archivado o la serialización para escribir y leer rápidamente los objetos que conforman la aplicación. De esta forma se pueden implementar rápidamente las operaciones de abrir y guardar ficheros.

No todos los objetos Cocoa soportan archivado, sólo los que implementan el protocolo `NSCoding` pueden ser archivados. El protocolo `NSCoding` dispone de dos métodos que guardan y restauran el objeto. Todos los objetos del Foundation Framework, y muchos objetos (pero no todos) del Application Kit Framework soportan el archivado (es decir, implementan `NSCoding`). En el Modelo-Vista-Controlador (MVC), muy usado por las aplicaciones Cocoa, los objetos del modelo siempre implementan el protocolo `NSCoding`. Los objetos

de la vista también suelen implementar el protocolo `NSCoding` para poder guardarse en ficheros `.nib`. Aunque no es estrictamente necesario, se recomienda que los objetos del modelo implementen también el protocolo `NSCopying`. De esta forma resulta fácil hacer copias de los objetos del modelo.

Por su parte, la serialización está restringida a un subconjunto de objetos del Foundation Framework que permiten crear diccionarios, arrays, cadenas y datos binarios: `NSArray`, `NSDictionary`, `NSString`, `NSDate`, `NSNumber` y `NSData`. A diferencia del archivado, la serialización no mantiene el grafo de objetos serializado, sólo mantiene los valores y su posición en la jerarquía. Múltiples referencias al mismo valor se expanden en objetos distintos durante la deserialización. Además la mutabilidad de los objetos serializados no se mantiene cuando se deserializan.

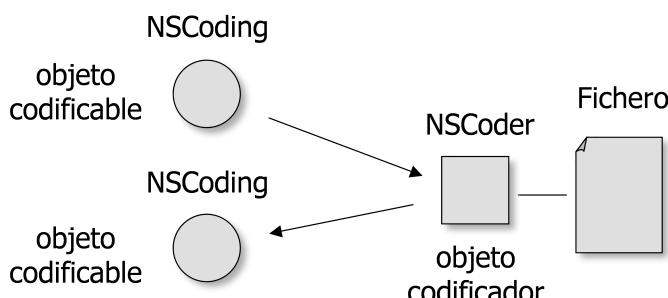
## 2. Archivado

---

En este apartado vamos a empezar estudiando las técnicas de archivado y en el apartado 3 profundizaremos en las técnicas de serialización.

### 2.1. Objetos codificadores y objetos codificables

Sabemos que el archivado nos permite convertir objetos Cocoa en un stream de bytes que preserva el contenido y las relaciones entre los objetos. Se llama **objeto codificador** al objeto destinado a realizar la acción de leer y escribir objetos. El objeto codificador está representado por la clase  `NSCoder` y derivadas. Se llama **objeto codificable** a cualquier objeto que se puede archivar. Para que un objeto sea codificable basta con que adopte el protocolo `NSCoding`. La Figura 3.1 resume gráficamente esta idea.

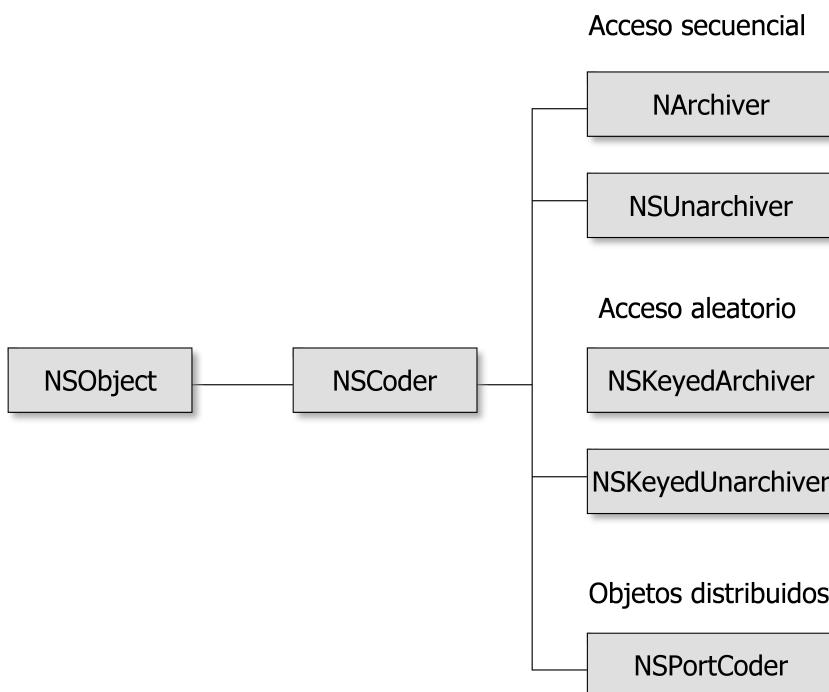


**Figura 3.1:** Objeto codificador y objetos codificables

La clase  `NSCoder` es una clase abstracta que declara las operaciones comunes a todos los objetos codificadores. Dado que es abstracta, la clase  `NSCoder` no dispone de métodos que permitan instanciarla, sino que tan sólo declara

operaciones `encode...` que permiten enviar datos a fichero y operaciones `decode...` que permiten recuperar datos a memoria.

La Figura 3.2 muestra las clases derivadas de  `NSCoder`. Estas clases derivadas representan los distintos tipos de objetos codificadores. Como muestra la figura existen tres tipos de objetos codificadores: (1) **Objetos archivadores de acceso secuencial** que permiten leer y escribir objetos de forma secuencial, es decir, el orden en que escribimos los objetos deberá ser el mismo que el orden en que los recuperamos. (2) **Objetos archivadores de acceso aleatorio** que permiten almacenar y recuperar los objetos en cualquier orden. Para ello a cada objeto que se archiva se le asigna una clave, que luego se usa para recuperar el objeto. (3) **Objetos codificadores distribuidos** los cuales se usan sólo cuando se implementa una arquitectura de objetos distribuidos.



**Figura 3.2:** Jerarquía de clases de los objetos codificables

Los objetos archivadores de acceso secuencial están representados por las clases  `NSArchiver` y  `NSUnarchiver`, y Apple desaconseja su uso a partir de Mac OS X 10.2 en favor de los objetos archivadores de acceso aleatorio que están representados por las clases  `NSKeyedArchiver` y  `NSKeyedUnarchiver`. La razón por la que Apple recomienda no usar objetos archivadores de acceso secuencial es que es muy difícil mantener la compatibilidad entre versiones de aplicaciones cuando el grafo de objetos se archiva de forma secuencial. Es decir, la aparición y desaparición de variables de instancia en los objetos, así como la aparición y desaparición de nuevos objetos dificulta el archivar los valores de forma que versiones anteriores de la aplicación puedan recuperar esta secuencia. En adelante nos centraremos en estudiar sólo los objetos

archivadores de acceso aleatorio. Los objetos codificadores distribuidos los veremos en el Tema 10.

La clase  `NSCoder` define dos conjuntos separados de métodos: Los métodos sin clave como:

- `(void)encodeObject:(id) object`
- `(id)decodeObject`

están destinados a ser usados por los objetos archivadores de acceso secuencial, y los métodos con clave como:

- `(void)encodeObject:(id) objv forKey:(NSString*) key`
- `(id)decodeObjectForKey:(NSString*) key`

están destinados a ser usados por los objetos archivadores de acceso aleatorio. De hecho, por razones de compatibilidad los objetos archivadores de acceso aleatorio también permiten usar los métodos de archivado sin clave, pero ya hemos dicho que Apple no recomienda su uso, sino que recomienda usar sólo los métodos de acceso con clave.

En  `NSCoder`, además de  `encodeObject:forKey:`, encontramos métodos para cada tipo de dato fundamental como  `encodeBool:forKey:` o  `encodeDouble:forKey:`. Análogamente existen métodos  `decode...` para cada tipo fundamental.

Respecto a los objetos codificables, su protocolo  `NSCoding` sólo dispone de dos métodos:

- `(void)encodeWithCoder:(NSCoder*) encoder`
- `(id)initWithCoder:(NSCoder*) decoder`

El primero se usa para pedir al objeto codificable que guarde su contenido en el objeto codificador que se pasa como argumento, y el segundo se usa para indicar al objeto codificable que restaure su contenido a partir del objeto codificador que se le pasa como argumento. Normalmente el método  `initWithCoder:` se ejecuta sobre un objeto inmediatamente después de haber ejecutado sobre él la operación  `alloc`, y el método es una alternativa a  `init`. Sólo los objetos que implementan  `NSCoding` pueden leerse o escribirse en un objeto codificador.

Téngase en cuenta que durante la codificación sólo se guarda el nombre de clase de los objetos archivados y sus variables de instancia, pero no sus métodos. En consecuencia, la aplicación que decodifica debe de tener los métodos que corresponden a estos objetos.

## 2.2. Crear un objeto archivador

Para crear un objeto archivador de acceso aleatorio podemos empezar creando un buffer mutable (de la clase `NSMutableData`) y luego crear un `NSKeyedArchiver` usando el método de inicialización:

- `(id) initForWritingWithMutableData: (NSMutableData*) data`

Una vez que tengamos el objeto archivador de acceso aleatorio podemos escribir en él usando los métodos `encode....`. En el ejemplo del Listado 3.1 a `encodeObject:forKey:` le pasamos un `NSArray` que a su vez contiene objetos `NSString`. En este caso el array se encarga de archivarse a si mismo y a los objetos que agrega. Cuando acabemos de escribir en el objeto archivador es importante acordarse de llamar a:

- `(void) finishEncoding`

Por último, el contenido del objeto dato mutable lo podemos guardar en un fichero tal como hacemos en el Listado 3.1.

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // Crea un array de nombres
    NSArray* array = [NSArray arrayWithObjects:@"Pedro", @"Juan",
                      @"Marta", nil];
    // Crea un archivador
    NSMutableData* buffer = [NSMutableData dataWithCapacity:1024];
    NSKeyedArchiver* archivador = [[NSKeyedArchiver alloc]
                                    initForWritingWithMutableData:buffer];
    // Guarda el array de nombres en el archivador
    [archivador encodeObject:array forKey:@"nombres"];
    [archivador finishEncoding];
    [archivador release];
    // Guarda el archivador en disco
    NSFileManager* fm = [NSFileManager defaultManager];
    [fm createFileAtPath:@"nombres.data"
                  contents:buffer attributes:nil];
    [pool drain];
    return 0;
}
```

**Listado 3.1:** Programa que archiva objetos

## 2.3. Crear un objeto desarchivador

La creación de un objeto desarchivador de acceso aleatorio es similar a la de un objeto archivador de acceso aleatorio. Primero debemos de disponer de un

objeto dato con los datos archivados a recuperar. El Listado 3.2 muestra cómo obtenerlo.

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // Lee el contenido del fichero
    NSFileManager* fm = [NSFileManager defaultManager];
    NSData* buffer_recuperado = [fm contentsAtPath:@"nombres.data"];
    NSKeyedUnarchiver* desarchivador = [[NSKeyedUnarchiver alloc]
                                         initForReadingWithData:buffer_recuperado];
    NSArray* array = [desarchivador decodeObjectForKey:@"nombres"];
    [desarchivador finishDecoding];
    for (NSString* str in array) {
        printf("%s\n", [str UTF8String]);
    }
    [pool drain];
    return 0;
}
```

**Listado 3.2:** Programa que desarchiva objetos

Después podemos crear un objeto de tipo `NSKeyedUnarchiver` usando el método de inicialización:

- `(id)initForReadingWithData:(NSData*)data`

Y después podemos usar los métodos `decode...` para recuperar los objetos archivados. Por último debemos llamar al método:

- `(void)finishDecoding`

Que indica al objeto desarchivador que no vamos a pedir más datos y que puede liberar recursos.

## 2.4. Archivar un grafo de objetos

Un grafo de objetos es una estructura más compleja que un árbol. Dos objetos pueden contener referencias entre sí. Si el objeto codificador se limitase a seguir las referencias en un grafo se metería en un bucle infinito. Además un mismo objeto puede ser referenciado desde distintos objetos. El objeto codificador debe conocer esta circunstancia y codificar cada objeto sólo una vez, así como regenerar todas las referencias a los objetos cuando realice la decodificación.

Para tener en cuenta estas redundancias durante el archivado y recuperación de un grafo de objetos, los objetos archivadores introducen el concepto de **objeto raíz** (root object) que es el objeto a partir del cual se empieza a

codificar el grafo de objetos. Todos los objetos referenciados, directa o indirectamente, por el objeto raíz serán archivados.

Además, cuando el objeto archivador inicia el archivado de un grafo de objetos a partir del objeto raíz, comprueba que los objetos no se archiven más de una vez. Para ello, cada vez que se llama a `encodeObject:` (en los objetos archivadores de acceso secuencial) o a `encodeObject:forKey:` (en los objetos archivadores de acceso aleatorio), el objeto archivador anota que el objeto ha sido archivado, y si se vuelve a pedir archivarlo no lo vuelve a archivar, sino que guarda una referencia al objeto previamente archivado.

La clase `NSKeyedArchiver` (y la antigua clase `NSArchiver`) implementa los métodos de clase:

```
+ (NSData*) archivedDataWithRootObject: (id)rootObject
+ (BOOL) archiveRootObject: (id)rootObject
                      toFile: (NSString*)path
```

Los cuales reciben el objeto raíz y archivan todo el grafo de objetos empezando por el objeto raíz. El primero archiva el grafo de objetos en un objeto dato que devuelve en su retorno. El segundo método guarda el grafo de objetos en el fichero pasado en el parámetro `path`.

Para recuperar el grafo de objetos la clase `NSKeyedUnarchiver` (y la antigua clase `NSUnarchiver`) disponen de los métodos de clase:

```
+ (id) unarchiveObjectWithData: (NSData*) data
+ (id) unarchiveObjectWithFile: (NSString*)path
```

Ambos devuelven el objeto raíz que se usó en el archivado.

## 2.5. Implementar un objeto codificable

Hay que tener en cuenta que para que los objetos se puedan archivar y desarchivar deben de ser objetos codificables, es decir, deben de implementar el protocolo `NSCoding`.

En el apartado 2.1 indicamos que el protocolo `NSCoding` tenía dos métodos:

- `(void)encodeWithCoder: (NSCoder*)encoder`
- `(id)initWithCoder: (NSCoder*)decoder`

Estos métodos están implementados por muchas clases Cocoa. Si nosotros queremos crear un objeto codificable también podemos hacer que nuestro objeto adopte el protocolo `NSCoding`. El Listado 3.3 y Listado 3.4 muestran la interfaz e implementación de una clase `Fecha` que implementa el protocolo `NSCoding`.

```
/* Fecha.h */
#import <Foundation/Foundation.h>

@interface Fecha : NSObject <NSCoding> {
    @private
    NSNumber* dia;
    NSNumber* mes;
    NSNumber* anno;
    BOOL bisiesto;
}
@property(retain) NSNumber* dia;
@property(retain) NSNumber* mes;
@property(retain) NSNumber* anno;
@property(nonatomic,readonly) BOOL bisiesto;
- init;
- initWithDia:(NSNumber*)d mes:(NSNumber*)m anno:(NSNumber*)a;
- initWithCoder:(NSCoder*)coder;
- (void)encodeWithCoder:(NSCoder*) coder;
- (BOOL)bisiesto;
- (NSString*) description;
- (void)dealloc;
@end
```

**Listado 3.3:** Interfaz de objeto codificable

```
/* Fecha.m */
#import "Fecha.h"

@implementation Fecha
@synthesize dia;
@synthesize mes;
@synthesize anno;
- init {
    return [self initWithDia:[NSNumber numberWithInt:1]
                  mes:[NSNumber numberWithInt:1]
                  anno:[NSNumber numberWithInt:1900]];
}
- initWithDia:(NSNumber*)d mes:(NSNumber*)m anno:(NSNumber*)a {
    if (self = [super init]) {
        [self setDia: d];
        [self setMes: m];
        [self setAnno: a];
    }
    return self;
}
- initWithCoder:(NSCoder*)coder {
    if (self = [super init]) {
        dia = [[coder decodeObjectForKey:@"dia"] retain];
        mes = [[coder decodeObjectForKey:@"mes"] retain];
        anno = [[coder decodeObjectForKey:@"anno"] retain];
        bisiesto = [coder decodeBoolForKey:@"bisiesto"];
    }
    return self;
}
- (void)encodeWithCoder:(NSCoder*)coder {
    [coder encodeObject:dia forKey:@"dia"];
    [coder encodeObject:mes forKey:@"mes"];
    [coder encodeObject:anno forKey:@"anno"];
    [coder encodeBool:bisiesto forKey:@"bisiesto"];
}
```

```

}
- (BOOL)bisiesto {
    if( ([anno intValue]%4==0 && [anno intValue]%100!=0)
        || [anno intValue]%400==0)
        return YES;
    return NO;
}
- (NSString*)description {
    return [NSString stringWithFormat:@"%@:%@:%@",dia,mes,anno];
}
- (void)dealloc {
    [dia release];
    [mes release];
    [anno release];
    [super dealloc];
}
@end

```

**Listado 3.4:** Implementación de objeto codificable

Es importante tener en cuenta que si en `initWithCoder:forKey:` se crean nuevos objetos con `decodeObject:forKey:`, es responsabilidad del método `initWithCoder:forKey:` el llamar a `retain` sobre los objetos obtenidos, ya que en caso contrario estos objetos pertenecerían al objeto desarchivador, y éste podría liberarlos.

El Listado 3.5 muestra un programa que archiva y desarchiva dos instancias de la clase `Fecha`. Obsérvese que aunque en el método `encodeWithCoder:` las dos instancias de `Fecha` usan las mismas claves (`@"dia"`, `@"mes"` y `@"anno"`), no se produce conflicto entre ellas ya que cada llamada a `encodeObject:forKey:` inicia un nuevo namespace.

```

#import <Foundation/Foundation.h>
#import "Fecha.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // Crea dos fechas
    Fecha* f1= [[Fecha alloc] initWithDia:[NSNumber numberWithInt:22]
                                         mes:[NSNumber numberWithInt:2]
                                         anno:[NSNumber numberWithInt:2007]];
    Fecha* f2= [[Fecha alloc] initWithDia:[NSNumber numberWithInt:23]
                                         mes:[NSNumber numberWithInt:3]
                                         anno:[NSNumber numberWithInt:2008]];
    // Archiva las fechas
    NSFileManager* fm = [NSFileManager defaultManager];
    NSMutableData* buffer = [NSMutableData dataWithCapacity:1024];
    NSKeyedArchiver* archivador = [[NSKeyedArchiver alloc]
                                    initForWritingWithData:buffer];
    [archivador encodeObject:f1 forKey:@"f1"];
    [archivador encodeObject:f2 forKey:@"f2"];
    [archivador finishEncoding];
    // Libera las fechas y el archivador
    [f1 release];
    [f2 release];
    [archivador release];
    // Guarda el buffer en fichero
    [fm createFileAtPath:@"fechas.data" contents:buffer]

```

```

        attributes:nil];
    // Recupera el buffer de fichero
    NSData* buffer_recuperado = [fm contentsAtPath:@"fechas.data"];
    // Desarchiva las fechas
    NSKeyedUnarchiver* desarchivador = [[NSKeyedUnarchiver alloc]
                                         initForReadingWithData:buffer_recuperado];
    f1 = [desarchivador decodeObjectForKey:@"f1"];
    f2 = [desarchivador decodeObjectForKey:@"f2"];
    printf("%s\n", [[f1 description] UTF8String]);
    printf("%s\n", [[f2 description] UTF8String]);
    // Libera las fechas y el desarchivador
    [desarchivador release];
    [pool drain];
    return 0;
}

```

**Listado 3.5:** Programa que archiva y desarchiva objetos

## 2.6. Codificar y decodificar tipos fundamentales

Las clases `NSKeyedArchiver` y `NSKeyedUnarchiver`, además de métodos como `encodeObject:forKey:` para codificar objetos, proporcionan métodos como `encodeBool:forKey:`, `encodeDouble:forKey:` para codificar tipos de datos fundamentales.

En el caso de las estructuras o campos de bits, Apple recomienda descomponer sus miembros en partes y almacenar cada parte como un tipo fundamental distinto.

En principio no existen métodos para codificar punteros, ya que al decodificar el puntero no obtendríamos nada útil, pero sí que podemos almacenar arrays de bytes (como `char*`) usando el método:

- `(void)encodeBytes:(const uint8_t*)bytesp  
length:(NSUInteger)len  
forKey:(NSString*)key`

En caso de usar arrays de bytes, es nuestra responsabilidad el tratar con la representación big/little endian.

## 2.7. Codificación condicional

Un problema que se produce cuando se intenta guardar un grafo de objetos es que a veces el grafo se extiende demasiado, llegando a cubrir toda la aplicación. Para marcar objetos donde queremos que termine el grafo se usa la **codificación condicional**, que consiste en indicar que un objeto sólo lo queremos codificar cuando en el archivo se haya almacenado este objeto por otra vía. En este caso, al recuperar el objeto queremos obtener un puntero a él. Si no se ha almacenado este objeto por ningún otro camino, queremos

que al recuperarlo se nos devuelva `nil`. Muchas veces la codificación condicional se aplica sobre las referencias débiles a objetos. La codificación condicional se hace usando el método:

```
- (void)encodeConditionalObject:(id)objv forKey:(NSString*)key
```

No existe el correspondiente método `decodeConditionalObject:forKey:` porque la decodificación se haría con `decodeObject:forKey:`, y el método devolvería `nil` o el objeto encontrado.

Por ejemplo, si modificamos el método `encodeWithCoder:` del Listado 3.4 para que incluya de forma condicional otro atributo `ayer` que apunte a otro objeto `NSDate` que se decodifique de forma condicional así:

```
- (void)encodeWithCoder:(NSCoder*)coder {
    [coder encodeObject:dia forKey:@"dia"];
    [coder encodeObject:mes forKey:@"mes"];
    [coder encodeObject:anno forKey:@"anno"];
    [coder encodeBool:bisiesto forKey:@"bisiesto"];
    [coder encodeConditionalObject:ayer forKey:@"ayer"];
}
```

El método `encodeConditionalObject:forKey:` aparece en la clase abstracta  `NSCoder`, pero este método se limita a llamar a `encodeObject:forKey:`. La clase  `NSKeyedArchiver` lo redefine para implementar la codificación condicional cuando guarda un grafo de objetos.

## 2.8. Restringir el soporte para objetos codificadores

Hemos visto que los objetos codificables son los que implementan  `NSCoder`. En ocasiones un objeto codificable puede querer no soportar la codificación para todos los tipos de objetos codificadores, sino limitarse a un subconjunto de ellos. Por ejemplo, las clases  `NSDistantObject`,  `NSInvocation` y  `NSPort` adoptan el protocolo  `NSCoder`, pero pueden ser usados sólo por un objeto codificable de tipo  `NSPortCoder` (no pueden ser usados por objetos archivadores). En este caso, una clase puede comprobar el tipo del codificador y lanzar una excepción si no soporta ese codificador. Por ejemplo, podemos modificar el método  `encodeWithCoder:` del Listado 3.4 para que sólo soporte objetos codificadores de tipo  `NSKeyedArchiver` de la forma:

```
- (void)encodeWithCoder:(NSCoder*)coder {
    if (![coder isKindOfClass:[NSKeyedArchiver class]]) {
        [NSEception raise:NSInvalidArgumentException
        format:
        @"Solo objetos codificadores NSKeyedArchiver"];
    }
    [coder encodeObject:dia forKey:@"dia"];
    [coder encodeObject:mes forKey:@"mes"];
```

```
[coder encodeObject:anno forKey:@"anno"];
[coder encodeBool:bisiesto forKey:@"bisiesto"];
}
```

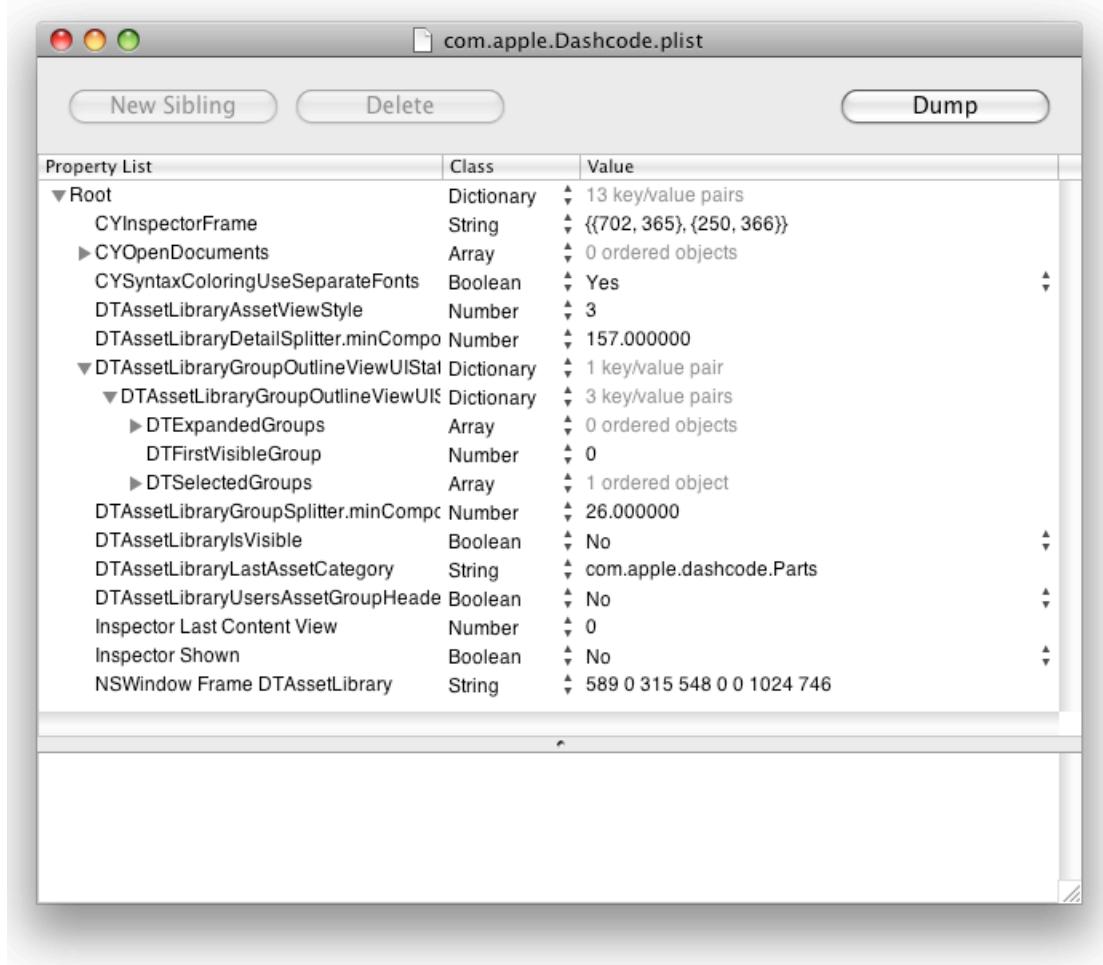
En otras ocasiones una clase podría heredar de una clase base que adopta `NSCoding`, pero la nueva clase derivada podría no querer seguir siendo codificable. Por ejemplo `NSWindow` hereda de `NSResponder`, pero no soporta la codificación. En este caso la clase derivada redefine `initWithCoder:` y `encodeWithCoder:` para que lancen una excepción si se ejecutan.

## 3. Serialización

La serialización permite almacenar simples jerarquías de propiedades de forma más sencilla que el archivado. En esta apartado vamos a detallar cómo se realiza la serialización.

### 3.1. Listas de propiedades

En el apartado 1 introdujimos las listas de propiedades (property lists), una forma usada por Cocoa y Core Foundation para almacenar jerarquías de propiedades. A diferencia del archivado, la serialización se limita a poder almacenar jerarquías (y no grafos como en el caso del archivado) de objetos. Además el conjunto de objetos que puede almacenar la serialización es más restringido que el del archivado. En concreto, el tipo de los objetos debe ser uno de los siguientes: `NSDictionary`, `NSArray`, `NSData`, `NSString`, `NSNumber`, `NSDate`.



**Figura 3.3:** Property List Editor

Las listas de propiedades se suelen almacenar en ficheros de propiedades con la extensión `.plist`.

Dentro de las herramientas de desarrollo de Apple encontramos la aplicación Property List Editor (ver Figura 3.3), la cual permite editar estos ficheros de propiedades de forma cómoda.

En el Tema 4 profundizaremos en el estudio de ejemplos prácticos donde se usan ficheros de propiedades tanto para almacenar preferencias como para almacenar información de configuración de las aplicaciones y del sistema operativo.

Existen tres formatos para representar las listas de propiedades en ficheros de propiedades: (1) Formato ASCII heredado de OpenStep, (2) Formato XML y (3) formato binario.

## 3.2. API de serialización

Existen dos formas de serializar y deserializar una lista de propiedades. La primera se puede aplicar cuando tenemos la lista de propiedades almacenada en un objeto `NSArray` o `NSDictionary` como objeto raíz. Estas clases tienen el método de instancia:

- `(BOOL)writeToFile:(NSString*)path atomically:(BOOL)flag`

El cual permite volcar el contenido del objeto directamente a un fichero dado en `path`.

Para recuperar la lista de propiedades en memoria a partir del contenido del fichero las clases `NSArray` y `NSDictionary` disponen respectivamente de los métodos factory:

```
+ (id)arrayWithContentsOfFile:(NSString*)aPath
+ (id)dictionaryWithContentsOfFile:(NSString*)path
```

Así como del método de instancia:

- `(id)initWithContentsOfFile:(NSString*)aPath`

La otra forma de almacenar y recuperar listas de propiedades es usando la clase `NSPropertyListSerialization`. Para almacenar una lista de propiedades en memoria usamos su método de clase:

```
+ (NSData*)dataFromPropertyList:(id)plist
                           format:(NSPropertyListFormat)format
                           errorDescription:(NSString**)errorString
```

El parámetro `format` nos permite indicar el formato de serialización deseado de acuerdo a los valores de la Tabla 3.1. El método `writeToFile atomically:` de las clases `NSArray` y `NSDictionary` siempre serializar en formato XML, pero son capaces de deserializar cualquier formato de los presentes en la Tabla 3.1.

Para deserializar usamos el método de clase:

```
+ (id)propertyListWithData:(NSData*)data
    mutabilityOption:(NSPropertyListMutabilityOptions)opt
        format:(NSPropertyListFormat*)format
    errorDescription:(NSString**)errorString
```

| Formato                                      | Descripción                         |
|--|-------------------------------------|
| <code>NSPropertyListOpenStepFormat</code>    | Formato ASCII heredado de OpenStep. |
| <code>NSPropertyListXMLFormat_v1_0</code>    | Formato XML.                        |
| <code>NSPropertyListBinaryFormat_v1_0</code> | Formato binario                     |

**Tabla 3.1:** Formatos de serialización

La clase `NSPropertyListSerialization`, además de poder serializar listas de propiedades cuyo objeto raíz sea un `NSArray` o `NSDictionary`, también permite serializar objetos individuales de tipo `NSData`, `NSString`, `NSNumber`, `NSDate`.

Si deserializamos una lista de propiedades con `NSArray` de la forma:

```
NSArray* array = [NSArray
    arrayWithContentsOfFile:@"array.plist"];
```

Obtenemos un contenedor inmutable. Si queremos que sea mutable podemos hacer:

```
NSMutableArray* array = [NSMutableArray
    arrayWithContentsOfFile:@"array.plist"];
```

O bien hacer:

```
NSMutableArray* array = [NSPropertyListSerialization
    propertyListWithData:buffer
    mutabilityOption:NSPropertyListMutableContainersAndLeaves
    format: nil errorDescription:nil];
```

# Tema 4

## Configuración del runtime

---

### **Sinopsis:**

*En el tema anterior aprendimos a serializar jerarquías de propiedades. En este tema vamos a ver cómo Mac OS X hace uso de propiedades serializadas para almacenar información de configuración de las aplicaciones y del propio sistema operativo.*

## 1. Introducción

---

La **configuración del runtime** es una forma de poder modificar el comportamiento de un ejecutable sin recompilar su código fuente. En sistemas operativos como Microsoft Windows o Linux, la configuración del runtime se hace de forma poco homogénea, es decir, cada parte del sistema operativo y cada aplicación disponen de su propio mecanismo de almacenamiento de información de configuración (p.e. ficheros .ini, registro de Windows, directorio /etc). En Mac OS X se recomienda almacenar esta información de configuración siempre usando ficheros de propiedades. Tanto el sistema operativo como las aplicaciones suelen seguir esta recomendación, aunque siempre existen aplicaciones (muchas veces aplicaciones portadas desde otros sistemas operativos) que no atienden a este consejo.

## 2. Bundles

---

El problema de gestionar un código ejecutable y sus recursos asociados no es un problema nuevo, sino que es un problema que se ha tratado en los sistemas operativos de forma diferente. En el caso de Microsoft Windows los ficheros ejecutables incorporan un conjunto de recursos estándar (iconos, cuadros de diálogo), y el resto de los recursos se depositan en diferentes partes del disco duro tal como en cada caso decide el programador. En el caso de Linux también se han depositado los recursos en carpetas más o menos estándar como puedan ser /usr/share o /usr/local/share. En el caso de Mac OS X la forma recomendada de almacenar estos elementos es que el ejecutable y sus recursos asociados se almacenen en un directorio al que se llama **bundle**. Las aplicaciones, los frameworks y los plug-ins son ejemplos de bundles. Ejemplos de plug-ins son los paneles de preferencias y los widgets de Dashboard. Algunas aplicaciones (p.e. Mail) también definen plug-ins para extender su funcionalidad.

No todos los directorios son bundles, para que un directorio sea un bundle tiene que seguir las recomendaciones de Apple a la hora de almacenar, nombrar y organizar jerárquicamente su contenido.

Algunos bundles son también **paquetes**, es decir, Finder se los muestra al usuario como un sólo fichero. Las aplicaciones o los plug-ins son ejemplos de paquetes. Para que un bundle sea considerado un paquete debe cumplir unas condiciones que veremos en el apartado 2.3.

En el apartado 2.2 veremos cómo los bundles usan ficheros de propiedades para almacenar su información de configuración. Esta información de configuración es usada por Finder y Launch Services para decidir qué tipo de bundle es y cómo actuar sobre el bundle. Por ejemplo, Launch Services usa la infor-

mación de configuración almacenada en un fichero de propiedades de cada aplicación para asociar tipos de documentos a la aplicación.

El tipo de un bundle determina su estructura, su contenido y el tipo de recursos que almacena. Mac OS X soporta dos tipos de bundles: **bundles modernos** que son el tipo usado por las aplicaciones y los plug-ins, y **bundles versionados** que son los usados por los frameworks. En los siguientes apartados vamos a estudiar los bundles modernos. Los bundles versionados se explican en el documento "Compilar y depurar aplicaciones con las herramientas de programación de GNU", que también encontrará publicado en MacProgramadores.

## 2.1. Anatomía de un bundle moderno

Los bundles son aplicaciones autocontenidoas. Esto permite que el usuario pueda instalar una aplicación con sólo arrastrar el bundle a su disco, o desinstalar una aplicación con sólo llevarla a la papelera. La Figura 4.1 muestra la estructura de un bundle básico. Dentro del bundle sólo debe haber un único directorio llamado `Content`. Dentro de este directorio estará el resto de los ficheros y directorios del bundle<sup>1</sup>. El único fichero que es estrictamente necesario encontrar en un bundle es el fichero `Info.plist`, que es un fichero de propiedades con información sobre el bundle. Entre la información que encontramos en este fichero está el nombre del bundle, un ID único para el bundle, información sobre cómo Launch Services debe de manejar el bundle o información sobre los tipos de ficheros que reconoce el bundle.

El contenido del fichero `Info.plist` lo iremos introduciendo a lo largo de los siguientes apartados. En el apartado 2.2 estudiaremos el contenido de este fichero con más detalle.

Cuando a Xcode le pedimos que cree una aplicación Cocoa, Xcode genera el bundle y sus ficheros automáticamente. Nosotros también podemos crear el bundle manualmente, aunque nos arriesgamos a cometer errores. En los siguientes apartados nos vamos a dedicar a explicar los distintos ficheros y directorios que componen el bundle de una aplicación. Creemos que es importante conocer la estructura del bundle antes de usar Xcode para crearlo.

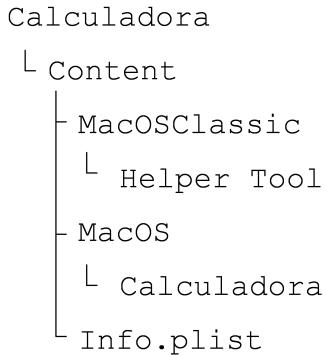
### 2.1.1. Añadir ejecutables

Los bundles suelen contener uno o más binarios ejecutables. En el caso de que estos binarios estén compilados para Mac OS X deberán de ir en el directorio `MacOS`. Si el bundle incluyese aplicaciones Mac OS Classic, estas irían en

---

<sup>1</sup> Aunque pueda parecer superfluo el requisito de la existencia del directorio `Content`, su razón de ser es diferenciar al bundle moderno de otros tipos de bundles.

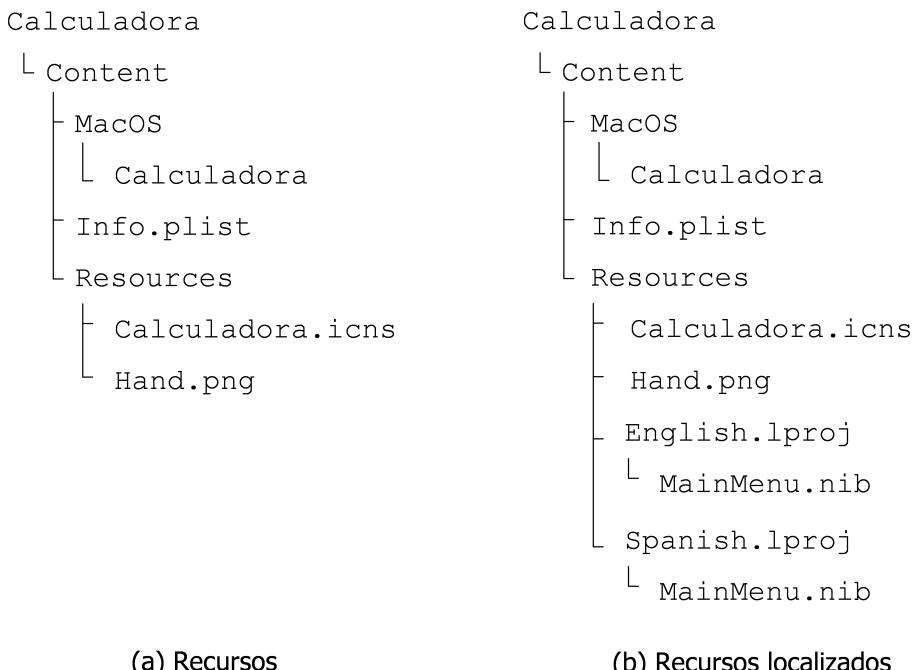
el directorio `MacOSClassic`. Estas últimas aplicaciones se ejecutarían en el Classic compatibility environment. La Figura 4.1 muestra un bundle con un binario Mac OS X llamado `Calculadora` y un binario Mac OS Classic llamado `Helper Tool`.



**Figura 4.1:** Bundle moderno con ejecutables

## 2.1.2. Añadir recursos

Los recursos son ficheros adicionales usados por la aplicación como puedan ser imágenes, iconos o ficheros de audio. Como muestra la Figura 4.2 (a), los recursos se almacenan en la carpeta `Resources`.



**Figura 4.2:** Bundle moderno con recursos

Un recurso que normalmente debemos tener en la carpeta `Resources` (aunque Xcode no lo crea por defecto) es el ícono del bundle, que normalmente

tiene el nombre del bundle y la extensión `.icns`. Para crear el ícono puede usar la herramienta Icon Composer, que se distribuye con las herramientas de desarrollo de Apple. Además, en el fichero `Info.plist` deberá existir la entrada `CFBundleIconFile` con el nombre (no el path) del fichero de ícono contenido en la carpeta `Resources`. Además, si queremos que Xcode nos copie el ícono a la carpeta `Resources` de la aplicación que genera cada vez que compilamos el proyecto, deberemos añadir el fichero de íconos al grupo `Resources` de Xcode tal como muestra la Figura 4.3. Los ficheros contenidos en este grupo son copiados a la carpeta `Resources` del bundle cada vez que se construye la aplicación tal como muestra la sección `Copy Bundle Resources` del target del proyecto.

### 2.1.3. Adaptar al lugar los recursos

La **adaptación al lugar**<sup>1</sup> permite crear recursos diferentes para diferentes idiomas y regiones. Los recursos más comunes de adaptar al lugar son los que contienen texto, pero podemos adaptar al lugar cualquier recurso (p.e. un ícono).

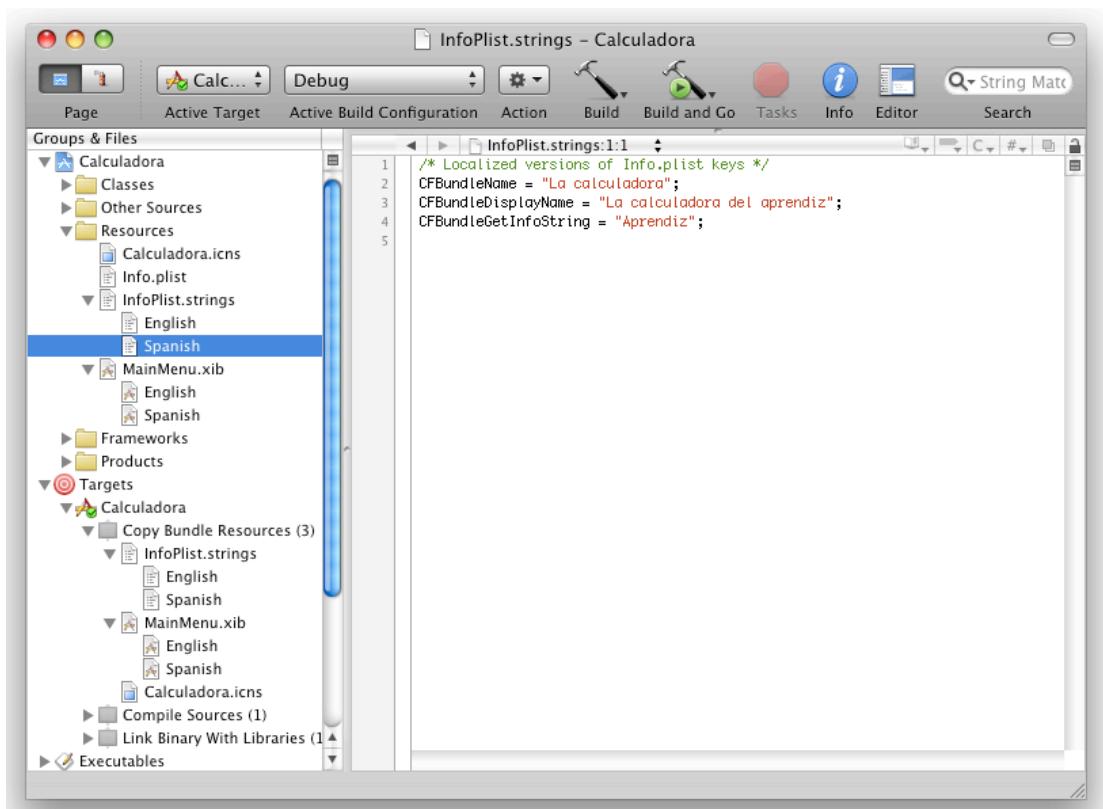


Figura 4.3: Proyecto con ícono

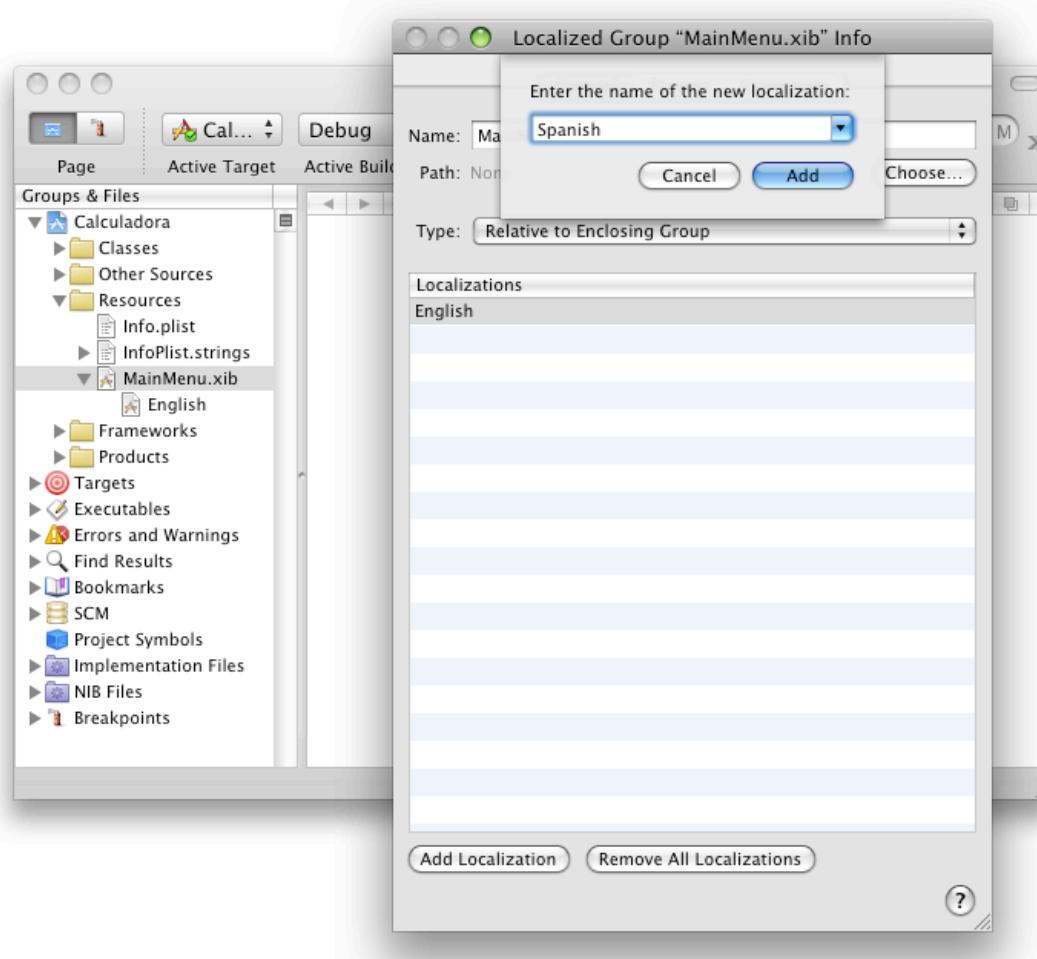
---

<sup>1</sup> En inglés se usa el término *localization* para indicar que un recurso se adapta al lugar del usuario (p.e. cuadros de diálogo en distintos idiomas) y el término *internationalization* para indicar que un recurso se puede usar en distintos idiomas y regiones (p.e. ficheros Unicode). Nosotros traduciremos *localization* por *adaptación al lugar* y *internationalization* por *internacionalización*.

Los recursos adaptados al lugar se almacenan en el bundle en **subdirectorios de recursos adaptados al lugar** dentro del directorio `Resources`. Cada subdirectorio de recursos adaptados al lugar debe de tener el nombre del idioma o región donde se va a usar, y la extensión `.lproj`. Estos nombres están formados por las letras de la norma ISO 639 para el idioma (p.e. `en.lproj` para inglés o `es.lproj` para Español), o bien dos letras para el idioma y dos letras para la región de acuerdo a la norma ISO 3166. Por ejemplo `en_US.lproj` para el inglés de USA. Además se pueden usar nombres para lenguajes comunes como `English.lproj` o `Spanish.lproj`.

La Figura 4.2 (b) muestra un bundle donde los ficheros `Calculadora.icns` y `Hand.png` están sin adaptar al lugar, y el fichero `MainMenu.nib` está adaptado para inglés y español.

En el apartado 2.4.3 comentaremos el orden en que se buscan los recursos en el directorio de recursos globales `Resources` y en los subdirectorios de recursos adaptados al lugar.



**Figura 4.4:** Crear subdirectorios de recursos adaptados al lugar adicionales

Cuando pedimos a Xcode crear un proyecto Cocoa, por defecto crea el subdirectorio `English.lproj`. Si queremos adaptar al lugar nuestra aplicación deberemos crear subdirectorios de recursos adaptados al lugar adicionales. Podemos crear subdirectorios de recursos adaptados al lugar de dos formas:

1. Creamos una carpeta adicional en el directorio del proyecto (p.e. `Spanish.lproj`) y añadimos sus ficheros a Xcode. Para añadir la carpeta con los recursos adaptados al lugar al proyecto Xcode, seleccionamos el elemento `MainMenu.xib` de la Figura 4.3 y usamos la opción de menú `Project|Add to project...`
2. Seleccionamos el elemento `MainMenu.xib` de la Figura 4.3 y sacamos el panel de información (`Command+I`). Después, como muestra la Figura 4.4, pedimos crear el subdirectorio de recursos adaptados al lugar.

Calculadora

```

└── Calculadora.xcodeproj
    ├── main.m
    ├── Info.plist
    ├── Calculadora.icns
    ├── Hand.png
    └── English.lproj
        ├── InfoPList.strings
        └── MainMenu.xib
    └── Spanish.lproj
        ├── InfoPList.strings
        └── MainMenu.xib

```

(a) Proyecto Xcode

Calculadora

```

└── Content
    └── MacOS
        └── Calculadora
    └── Info.plist
└── Resources
    ├── Calculadora.icns
    ├── Hand.png
    ├── English.lproj
        ├── InfoPList.strings
        └── MainMenu.nib
    └── Spanish.lproj
        ├── InfoPList.strings
        └── MainMenu.nib

```

(b) Bundle correspondiente

**Figura 4.5:** Estructura de un proyecto Xcode y del bundle que genera

Tenga en cuenta que el mantenimiento de una aplicación implica cambiar todos los ficheros de recursos adaptados al lugar, con lo que se recomienda no iniciar el proceso de adaptación al lugar de una aplicación hasta que el desarrollo de la aplicación se de por terminado.

Es importante no confundir los recursos en el proyecto Xcode con los recursos en el bundle obtenido al compilar el proyecto. La Figura 4.5 muestra los ficheros que podría contener un proyecto Xcode y el bundle que genera. En el proyecto Xcode no existe el directorio `Resources`, pero sí que existen los

subdirectorios de recursos adaptados al lugar como `English.lproj` o `Spanish.lproj`. Cuando en el directorio del proyecto Xcode metemos un recurso en un directorio adaptado al lugar, Xcode lo reconoce como tal y lo muestra anidado en el proyecto tal como ocurre en el fichero `MainMenu.xib` de la Figura 4.3. Cuando se crea el bundle, los recursos se copian al directorio `Resources` del bundle, y en caso de que estén adaptados al lugar, cada recurso se copiará a su correspondiente directorio adaptado al lugar. Además, el fichero `MainMenu.xib` se compila para generar `MainMenu.nib`.

### 2.1.4. Añadir otros ficheros

La mayoría de los bundles cuentan con el subdirectorio `MacOS` para ficheros binarios ejecutables de Mac OS X y con el subdirectorio `Resources` para los ficheros de recursos. Pero también existen otros nombres de subdirectorio estándar que Apple ha definido para los bundles.

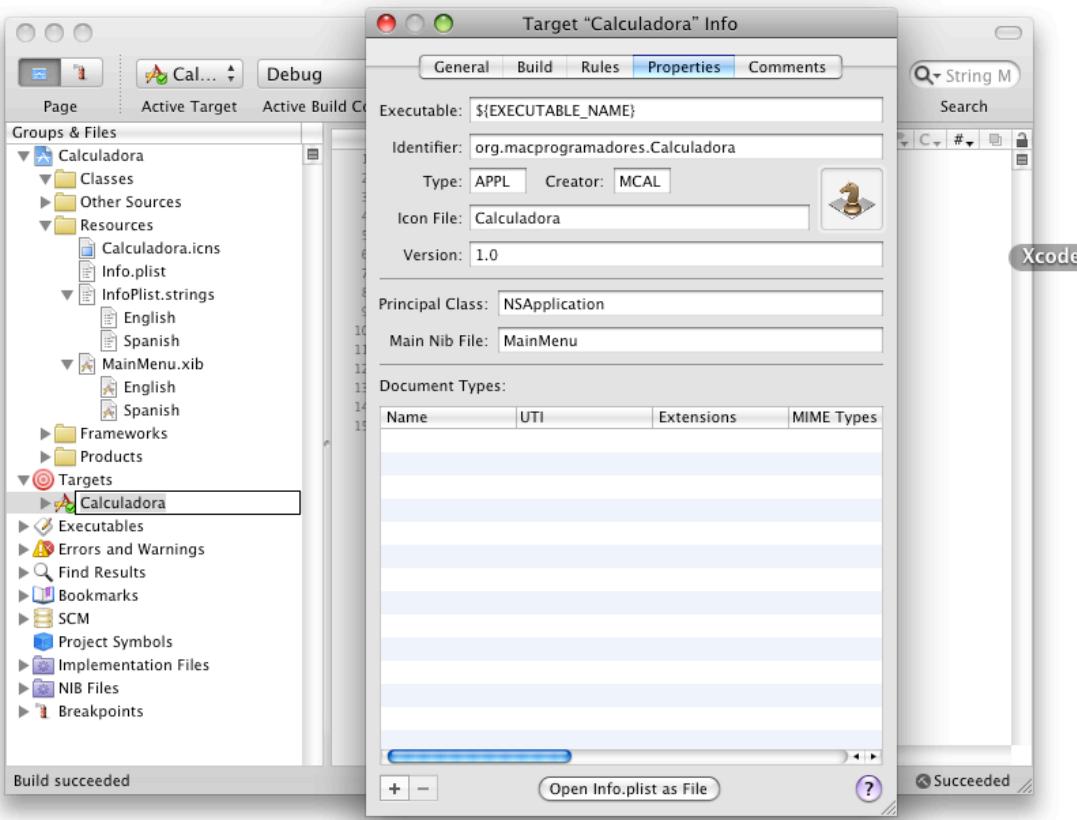
Vamos a comentar tres de ellos: El subdirectorio `Frameworks` se usa para contener librerías y frameworks privados, es decir, que sólo son usados por el bundle. El subdirectorio `PlugIns` se usa para contener bundles cargables dinámicamente que extienden la funcionalidad del ejecutable. El subdirectorio `SharedSupport` se usa para recursos no críticos que no afectan a la aplicación en caso de no estar disponibles. Por ejemplo, este último subdirectorio puede contener documentos, videos o tutoriales.

## 2.2. Información de configuración de un bundle

Cada bundle debe contener un fichero `Info.plist` al que llamaremos **fichero de información de configuración** (information property list file). Este fichero debe encontrarse dentro del directorio `Contents` tal como muestra la Figura 4.1. Finder y Launch Services usan esta información para representar y ejecutar los bundles encontrados. La Figura 4.6 muestra cómo podemos modificar algunas de estas propiedades desde Xcode. Para ello, dentro del grupo Targets seleccione el target de la aplicación y use `Command+I` para mostrar el panel de propiedades de la Figura 4.6. En los siguientes apartados vamos a comentar las principales propiedades que debe de contener este fichero. En general, las propiedades cuyo nombre empieza por `CF` son para Core Foundation, las que su nombre empieza por `LS` son para Launch Services, y las que empieza por `NS` son propiedades heredadas de NeXTSTEP.

En caso de que desee probar el funcionamiento de estas propiedades debe saber que si actualiza el fichero `Info.plist`, posiblemente Finder no detecte el cambio, ya que Finder sólo lee el contenido del Bundle una vez y lo guarda en una base de datos interna. Para actualizar la información de configuración

de un bundle ejecute `touch` sobre el directorio del bundle. Cuando Finder detecta que la fecha del bundle ha cambiado actualiza su base de datos.



**Figura 4.6:** Información de configuración con Xcode

### 2.2.1. Propiedades requeridas

Toda aplicación debe de incluir al menos las siguientes propiedades:

`CFBundleName`. Nombre corto del bundle. Debe tener menos de 16 caracteres y es el nombre que se muestra en la barra de menú y en la ventana de información de propiedades de Finder. Esta propiedad se puede adaptar al lugar tal como se explica en el apartado 2.2.4.

`CFBundleIdentifier`. Un identificador UTI (Universal Type Identifier) para la aplicación (p.e. `org.macprogramadores.Calculadora`). Véase Figura 4.6. Esta clave no identifica de forma única un bundle ya que pueden existir en el sistema varias aplicaciones con el mismo identificador y diferentes versiones (`CFBundleVersion`). O incluso varias copias de la aplicación con el mismo identificador y versión. Launch Services usa este identificador para localizar la aplicación capaz de abrir un tipo de fichero. Para ello lanza la primera aplicación que encuentra con este identificador. Como veremos en el apartado 3.1, el sistema de

preferencias también usa este identificador para almacenar las preferencias de la aplicación.

`CFBundleVersion`. Número de versión. Representa una iteración en el proceso de desarrollo (de release o no). Esta versión estará formada por uno o más números separados por punto (p.e. 1.0).

`CFBundleShortVersionString`. Número de versión de release. Identifica una versión que distribuimos. Debe constar de tres números separados por punto. El primero indica la versión mayor, el segundo la versión menor, y el tercero es la versión de mantenimiento.

`CFBundlePackageType`. Código de 4 letras que identifica el tipo de bundle (es análogo al type code de Mac OS Classic): `APPL` para aplicaciones, `FMWK` para frameworks, `BNDL` para bundles cargables dinámicamente (plug-ins). Este valor se puede proporcionar desde Xcode (ver Figura 4.6).

`CFBundleSignature`. Código de 4 letras que identifica al creador del bundle (es análogo al creator code de Mac OS Classic). Este valor se puede proporcionar desde Xcode (ver Figura 4.6).

Las siguientes propiedades sólo son necesarias en el caso de que el bundle sea una aplicación.

`NSMainNibFile`. Nombre del fichero `.nib` (sin extensión) donde está archivada la interfaz gráfica de la aplicación, así como las conexiones entre el programa y su interfaz gráfica. Este fichero se carga cuando la aplicación se lanza.

`NSPrincipalClass`. Nombre de la **clase principal del bundle**. El sistema de introspección de Objective-C usa este nombre para cargar la clase principal del bundle, la cual controla el resto de la aplicación. En el caso de las aplicaciones esta clase por defecto es `NSApplication`.

## 2.2.2. Propiedades recomendables

Las siguientes propiedades son de uso común, aunque no son estrictamente necesarias.

`CFBundleDisplayName`. Esta clave se usa para personalizar el nombre de la aplicación dependiendo del idioma de usuario. No se suele incluir en `Info.plist` sino en ficheros de cadenas adaptadas al lugar tal como se explica en el apartado 2.2.4.

`LSHasLocalizedDisplayName`. Indica si el nombre de la aplicación varía dependiendo del idioma tal como se explica en el apartado 2.2.4. Para ello debemos de poner esta propiedad con el valor "1". Esta propiedad debe activarse sólo si hemos proporcionado una versión adaptada al lugar de la propiedad `CFBundleDisplayName`. Al activar esta propiedad Finder puede localizar más rápido el nombre de aplicación que debe mostrar.

`CFBundleGetInfoString`. Versión del bundle para Get Info (`Command+I`) de Finder. Su contenido se puede adaptar al lugar como veremos en el apartado 2.2.4.

`CFBundleIconFile`. Fichero de icono del bundle. El fichero debe encontrarse en el directorio `Resources`, tal como se explicó en el apartado 2.1.2. No es necesario indicar la extensión (`.icns`) aunque se puede hacer.

`LSExecutableArchitectures`. Array de cadenas con las arquitecturas soportadas en orden de preferencia. Los posibles valores son: `i386`, `x86_64`, `ppc`, `ppc64`.

`LSRequiresNativeExecution`. Booleano que si está puesto a `YES` indica que no se puede emular la aplicación `ppc` con Rosetta.

`LSMinimumSystemVersion`. Versión mínima de Mac OS X necesaria para ejecutar la aplicación. Puede contener dos números (p.e. 10.2) o tres números (p.e. 10.2.4).

`LSMultipleInstancesProhibited`. Indica a Launch Services si el usuario puede lanzar varias instancias de la misma aplicación.

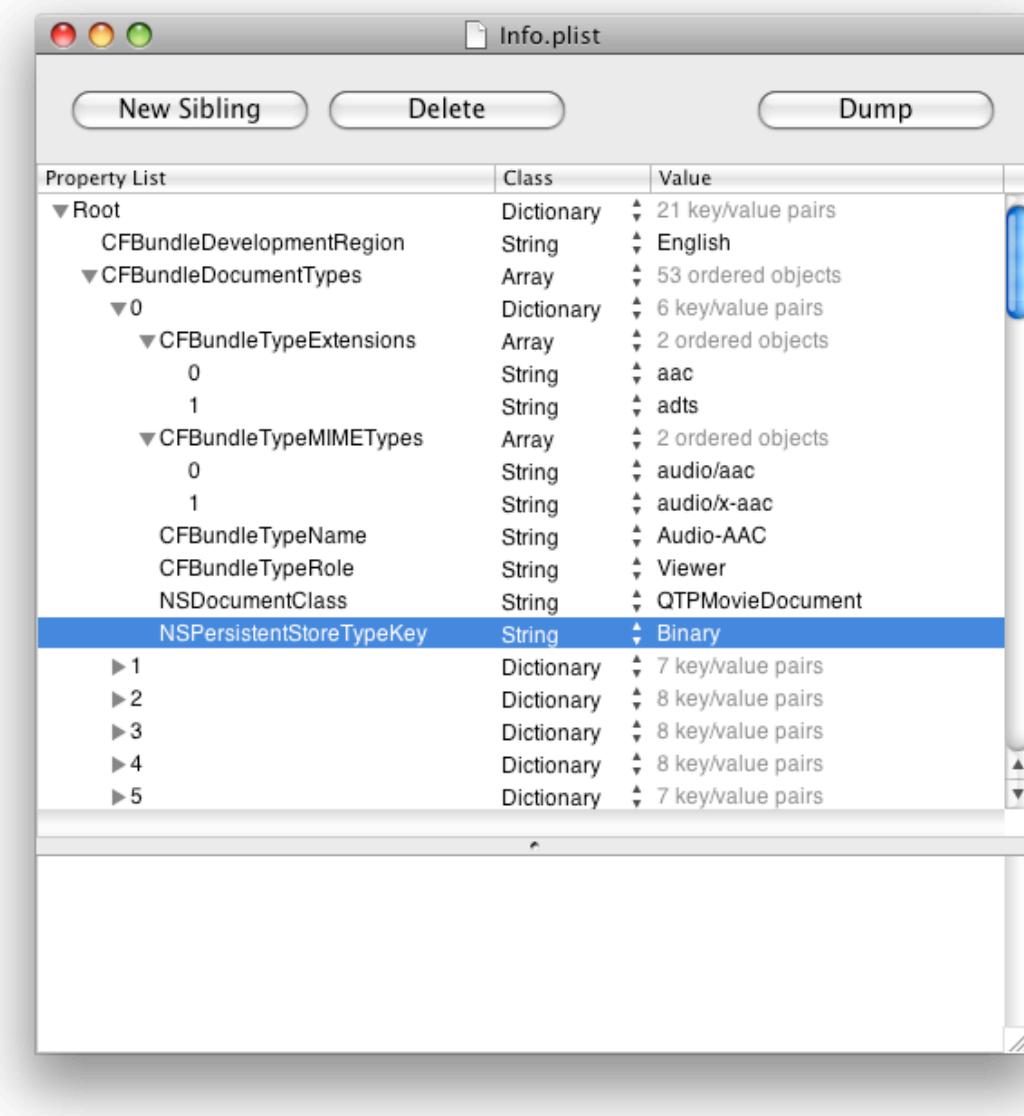
`LSBackgroundOnly`. Indica que la aplicación se ejecuta sólo en background.

`LSUIElement`. Indica que es un **agente de aplicación**. Este tipo de aplicaciones no aparecen en el Dock ni en la ventana de Force Quit y pueden no tener interfaz gráfica y aparecer sólo ante determinados eventos. El Dock y el diálogo de login son dos ejemplos de agentes de aplicación.

`NSAppleScriptEnabled`. Booleano que indica si la aplicación es scriptable. Por defecto las aplicaciones no son scriptables.

### 2.2.3. Asociar documentos

Si queremos asociar nuestra aplicación con uno o más tipos de documentos debemos usar la propiedad `CFBundleTypeExtensions`.



The screenshot shows the 'Info.plist' editor window with the title bar 'Info.plist'. Below the title bar are three buttons: 'New Sibling', 'Delete', and 'Dump'. The main area is a table titled 'Property List' with columns 'Property List', 'Class', and 'Value'. The table displays the following data:

| Property List             | Class      | Value              |
|---------------------------|------------|--------------------|
| Root                      | Dictionary | 21 key/value pairs |
| CFBundleDevelopmentRegion | String     | English            |
| CFBundleDocumentTypes     | Array      | 53 ordered objects |
| 0                         | Dictionary | 6 key/value pairs  |
| CFBundleTypeExtensions    | Array      | 2 ordered objects  |
| 0                         | String     | aac                |
| 1                         | String     | adts               |
| CFBundleTypeMIMETypes     | Array      | 2 ordered objects  |
| 0                         | String     | audio/aac          |
| 1                         | String     | audio/x-aac        |
| CFBundleTypeName          | String     | Audio-AAC          |
| CFBundleTypeRole          | String     | Viewer             |
| NSDocumentClass           | String     | QTMovieDocument    |
| NSPersistentStoreTypeKey  | String     | Binary             |
| ► 1                       | Dictionary | 7 key/value pairs  |
| ► 2                       | Dictionary | 8 key/value pairs  |
| ► 3                       | Dictionary | 8 key/value pairs  |
| ► 4                       | Dictionary | 8 key/value pairs  |
| ► 5                       | Dictionary | 7 key/value pairs  |

**Figura 4.7:** Documentos asociados a QuickTime

QuickTime es una aplicación capaz de abrir gran cantidad de documentos. La Figura 4.7 muestra el contenido de su propiedad `CFBundleDocumentTypes`. Esta propiedad es un array de diccionarios, donde cada diccionario describe un tipo de documento. Cada diccionario debe contener al menos las siguientes propiedades:

`CFBundleTypeName`. Nombre del tipo de documento. Aquí también se pueden usar tipos UTI.

`CFBundleTypeRole`. El role de la aplicación respecto a este tipo de documento: `Viewer`, `Editor`, `Shell` (la aplicación proporciona servicios a otros procesos. P.e. un player de applets Java), `None` (la aplicación no entiende los datos del documento, pero declara información sobre el tipo. P.e. Finder declara un ícono para las fuentes de letra).

`CFBundleTypeIconFile`. Fichero con el icono (`.icns`) a asociar a este tipo de documentos.

Además cada diccionario debe indicar el tipo de ficheros asociados conteniendo al menos una de estas propiedades:

`CFBundleTypeExtension`. Array de cadenas, cada una de las cuales contiene la extensión (sin el punto) del tipo del fichero.

`CFBundleTypeMIMETypes`. Array de cadenas con los tipos MIME del documento que abre la aplicación.

`CFBundleTypeOSTypes`. Contiene un array de cadenas de 4 letras con el type code de Mac OS Classic asociado al documento.

Las tres propiedades anteriores son ignoradas si existe la siguiente propiedad, la cual se introdujo en Mac OS X 10.4 para homogeneizar la representación de tipos de documentos.

`LSItemContentTypes`. Array de cadenas donde cada cadena representa un tipo UTI.

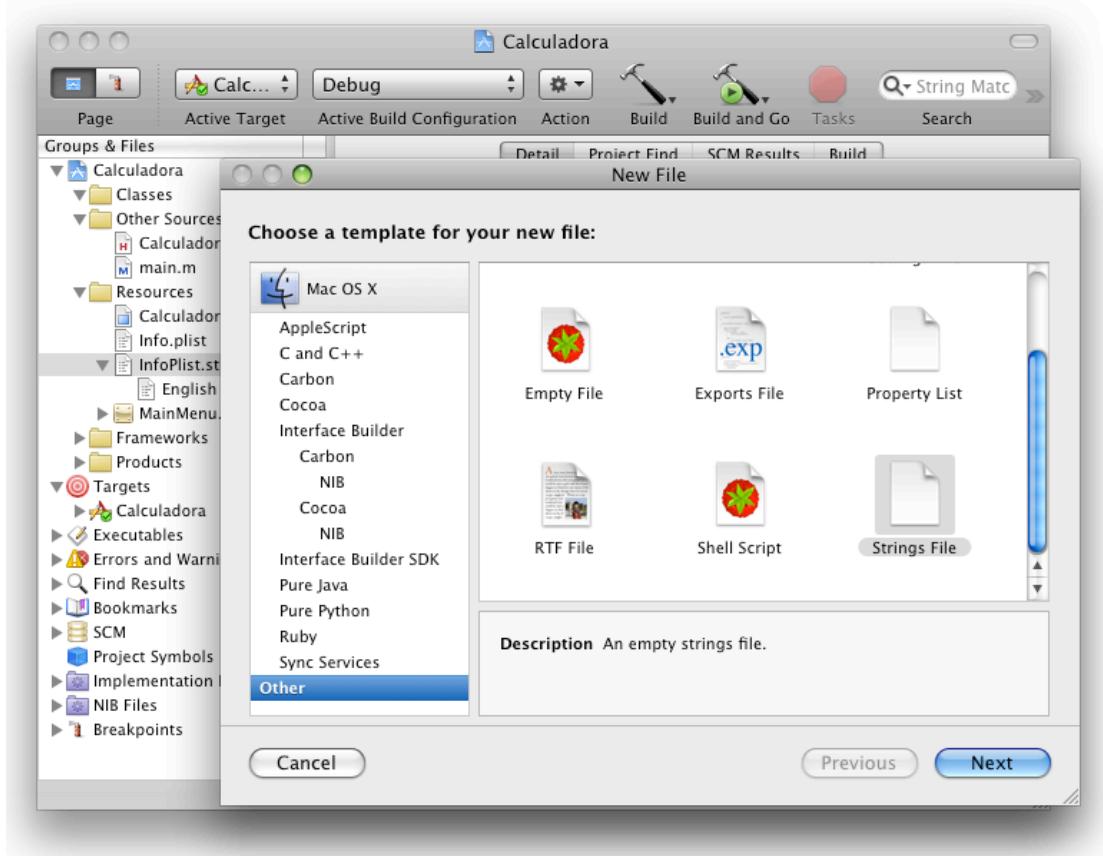
## 2.2.4. Adaptar al lugar las propiedades

Si la información de configuración tiene propiedades que deben ser mostradas directamente al usuario (p.e. `CFBundleName`, `CFBundleDisplayName` o `CFBundleGetInfoString`), es aconsejable adaptar al lugar estas propiedades. En este apartado vamos a ver cómo se adapta al lugar las propiedades de la información de configuración.

Para crear una propiedad adaptada al lugar debemos crear un fichero de texto con el nombre `InfoPList.string` por cada subdirectorio de recursos adaptados al lugar. A este fichero le llamaremos **fichero de cadenas adaptadas al lugar**, y contiene cadenas acabadas en punto y coma de la forma:

```
CFBundleName = "La calculadora del aprendiz";
CFBundleDisplayName = "La calculadora del aprendiz";
CFBundleGetInfoString = "Aprendiz";
```

Por defecto Xcode sólo crea el fichero de propiedades adaptado al lugar `English.lproj`. La Figura 4.8 muestra que para crear otros ficheros de propiedades adaptados al lugar podemos usar la opción `File|New file|Others|String file`.



**Figura 4.8:** Añadir un fichero de cadenas adaptadas al lugar

El sistema de búsqueda de propiedades busca primero las propiedades adaptadas al lugar (las del fichero `InfoPlist.strings`), y sólo si no las encuentra devuelve las propiedades generales (las del fichero `Info.plist`).

## 2.2.5. El fichero PkgInfo

El fichero `PkgInfo` es una forma alternativa y compatible con Mac OS Classic de indicar el type code y creator code de un fichero. Este fichero siempre ocupa 8 bytes, los 4 primeros bytes indican el type code (p.e. `APPL`) y los 4 últimos bytes el creator code (p.e. `ttxt`).

Xcode crea el fichero `PkgInfo` por defecto al crear un bundle (ver Figura 4.6). Aunque la nueva forma recomendada de indicar esta información es usar las propiedades `CFBundlePackageType` y `CFBundleSignature` del fichero `Info.plist`, la existencia de este fichero es habitual en la mayoría de las aplicaciones, aunque no es estrictamente necesaria su presencia.

## 2.2.6. Paso de argumentos durante la ejecución

Existen una serie de argumentos de línea de comandos estándar que todas las aplicaciones Cocoa entienden. Estos argumentos se resumen en la Tabla 4.1.

| Argumento                          | Descripción   |
|------------------------------------|---|
| <code>-NSOpen file</code>          | Abre el fichero <i>file</i> nada más acabar de cargar la aplicación. Para ello usa el método <code>application: openFile:</code> del delegado de la aplicación.   |
| <code>-NSOpenTemp file</code>      | Abre el fichero <i>file</i> como un fichero temporal nada más acabar de cargar la aplicación. Para ello usa el método <code>application: openTempFile:</code> del delegado de la aplicación.  |
| <code>-NSPrint file</code>         | Imprime el fichero <i>file</i> nada más acabar de cargar la aplicación. Para ello usa el método <code>application: printFile:</code> del delegado de la aplicación.   |
| <code>-NSShowAllDrawing YES</code> | Lanza la aplicación y muestra el color amarillo todas las zonas que se repintan de la pantalla. De esta forma podemos ver qué zonas de la vista están siendo actualizadas. Esta opción es equivalente a la que proporciona Quartz Debug, pero opera sólo en la aplicación que ejecutamos. |
| <code>-NSTraceEvents YES</code>    | Escribe en la salida estándar un log con los eventos que recibe el bucle de sondeo de la aplicación ejecutada.  |

**Tabla 4.1:** Argumentos de línea de comandos para aplicaciones Cocoa

## 2.3. Paquetes y Finder

Al comienzo del apartado 2 indicamos que algunos bundles son también paquetes. En concreto, los bundles modernos suelen aparecer como paquetes, mientras que los bundles versionados (los de los frameworks) no suelen aparecer como paquetes. Cuando un bundle aparece como un paquete Finder lo muestra como si fuera un sólo fichero.

Finder identifica los paquetes siguiendo las siguientes reglas:

1. Se considera paquete a los directorios con las extensiones `.app`, `.bundle`, `.plugin`, `.kext`, `.wdgt`, `.prefPane`.
2. Se considera paquete a los directorios que tengan el bit de bundle activado (independientemente de que tengan o no extensión).

3. Se considera paquete a los directorios en los que Finder detecta que su contenido sigue la estructura de un bundle moderno.

Cuando Finder detecta un paquete en el disco duro, modifica su nombre de acuerdo a las siguientes reglas:

1. Si el paquete es una aplicación, Finder oculta su extensión `.app`, siempre que no tengamos activada la opción de Finder `Preferences | Show all file extensions`.
2. Si el paquete implementa la propiedad `CFBundleDisplayName` en los ficheros de cadenas adaptadas al lugar y el usuario no ha cambiado el nombre de la aplicación (es decir, el nombre del directorio del bundle es el mismo que el de `CFBundleName`), Finder muestra el valor de `CFBundleName` adaptado al lugar en vez de el nombre del directorio del paquete.

Para evitar confusiones Finder no oculta la extensión `.app` a los ficheros renombrados con otra extensión. Por ejemplo, si renombramos el programa `Chess` a `Chess.mov`, Finder pasaría a mostrarlo como `Chess.mov.app`.

## 2.4. API para gestión de bundles

Los programas que hacen referencia a bundles, o que son bundles, pueden usar la clase `NSBundle` para interactuar programáticamente con el bundle. Entre otras operaciones, esta clase permite buscar recursos, acceder a información de configuración o cargar dinámicamente código ejecutable.

Las aplicaciones no Cocoa pueden usar el tipo `CFBundleRef` de Core Foundation para acceder a los bundles. A diferencia de otros tipos de datos, los tipos `NSBundle*` y `CFBundleRef` no son objetos puente, y en consecuencia no se pueden usar de forma intercambiable.

### 2.4.1. Localizar y abrir bundles

Se llama **bundle principal** (main bundle) al bundle donde la aplicación está almacenada. Cuando una aplicación quiere conocer su bundle principal, usa el método factory singleton:

```
+ (NSBundle*) mainBundle
```

Este método devuelve `nil` cuando se ejecuta desde una aplicación que no es un bundle.

Cuando queramos acceder a un bundle distinto al bundle principal debemos pasar su path completo al método factory:

```
+ (NSBundle*)bundleWithPath:(NSString*)fullPath
```

Una forma alternativa de localizar un bundle es a partir de una clase Objective-C, usando el método factory:

```
+ (NSBundle*)bundleForClass:(Class)aClass
```

También podemos acceder a un bundle a través de su identificador, es decir, de la propiedad `CFBundleIdentifier` usando el método factory:

```
+ (NSBundle*)bundleWithIdentifier:(NSString*)identifier
```

Pero en este caso el bundle ya debería de haber sido cargado previamente, es decir, ya debería de haberse creado un `NSBundle` previamente, con lo que la utilidad de este método es limitada. Principalmente se usa por los plug-ins y frameworks para localizar a su bundle principal.

Dado un `CFBundleIdentifier` podemos conocer el path absoluto de la aplicación con este identificador usando el método de `NSWorkspace`:

- `(NSString*)absolutePathForAppBundleWithIdentifier:(NSString*)bundleIdentifier`

Por último conviene comentar que una aplicación Cocoa puede conocer todos los bundles asociados a la aplicación usando los métodos:

```
+ (NSArray*)allBundles
+ (NSArray*)allFrameworks
```

Que devuelven todos los bundles asociados a la aplicación. El primer método devuelve los bundles modernos (el bundle de la aplicación y sus plug-ins) y el segundo método los bundles versionados (frameworks de los que depende la aplicación).

## 2.4.2. Obtener información de un bundle

Podemos usar métodos de la clase `NSBundle` para obtener información sobre el directorio donde se encuentra el bundle y para obtener su información de configuración.

Para obtener el directorio de un bundle usamos:

- `(NSString*)bundlePath`

La información de configuración se almacena en el fichero `Info.plist` y para obtener información de configuración del bundle usamos los métodos:

- `(NSDictionary*) infoDictionary`
- `(id) objectForInfoDictionaryKey: (NSString*) key`

El primero de los métodos nos devuelve un diccionario con el contenido del fichero `Info.plist` sin adaptar al lugar. En consecuencia, el mecanismo recomendado para acceder a la información de configuración que se puede adaptar al lugar es usar el segundo método, el cual busca la clave `key`, y si tiene información de adaptación al lugar nos devuelve su valor adaptado al lugar.

Si queremos guardar información de configuración para una aplicación tenemos dos opciones: La primera es crear un fichero de propiedades (`.plist`) y guardarlo en el directorio de recursos (`Content/Resources`). En tiempo de ejecución podemos cargar estas propiedades, por ejemplo con el método `dictionaryWithContentsOfFile:` de `NSDictionary`. La segunda opción es guardar estas propiedades en el fichero `Info.plist`, el cual acepta propiedades no estándar, y el método `infoDictionary` nos devolvería también estas propiedades.

### **2.4.3. Acceso a los recursos de un bundle**

`NSBundle` permite buscar ficheros en el directorio `Resources` del bundle en base a su extensión. Para ello tenemos el método:

- `(NSString*) pathForResource: (NSString*) name  
ofType: (NSString*) extension`

El cual busca el fichero con el nombre y extensión dados. En caso de que el parámetro `extension` valga `nil`, se busca un fichero cuyo nombre y extensión coincida exactamente con el parámetro `name`.

El algoritmo de búsqueda de `pathForResource:type:` busca primero los recursos globales (en el directorio `Resources`), y sólo si no los encuentra busca los recursos adaptados al lugar (en los subdirectorios de `Resources`). Este orden de búsqueda se decidió por eficiencia. En consecuencia los bundles nunca deberían de contener recursos globales y adaptados al lugar con el mismo nombre. Es decir, si queremos adaptar al lugar los recursos, no debemos de crearlos a nivel global.

Obsérvese que este mecanismo es el inverso al mecanismo de búsqueda de propiedades de información de configuración que vimos en el apartado 2.2.4, donde primero se buscaban las propiedades adaptadas al lugar, y luego las generales.

También podemos buscar propiedades en un bundle distinto al bundle principal usando el siguiente método de clase, el cual busca propiedades en el bundle cuyo directorio proporcionamos como parámetro:

```
+ (NSString*)pathForResource:(NSString*)name
                      ofType:(NSString*)extension
                 inDirectory:(NSString*)bundlePath
```

#### **2.4.4. Acceso a las cadenas adaptadas al lugar**

En el apartado 2.2.4 introdujimos los ficheros de cadenas adaptadas al lugar, vimos cómo se podía crear un fichero de cadenas adaptadas al lugar por cada subdirectorio de recursos adaptados al lugar y explicamos cómo se usaban para adaptar al lugar la información de configuración de un bundle.

Los ficheros de cadenas adaptadas al lugar no sólo se usan para adaptar al lugar las propiedades de la información de configuración, sino que pueden contener cadenas clave-valor para mensajes que queramos mostrar al usuario.

En el apartado 2.2.4 creamos un fichero de cadenas adaptadas al lugar con el nombre `InfoPlist.strings`, que es el nombre de fichero que se usa para almacenar las cadenas adaptadas al lugar del fichero `Info.plist`. A estas propiedades adaptadas al lugar podemos acceder con el siguiente método de instancia de `NSBundle`:

- `(NSDictionary*)localizedInfoDictionary`

Es posible crear otros ficheros de cadenas adaptadas al lugar y acceder a ellos con el método:

- `(NSString*)localizedStringForKey:(NSString*)key
 value:(NSString*)value
 tableName:(NSString*)tableName`

Donde `value` es el valor que se retorna si no se encuentra la cadena con clave `key`, y `tableName` es el nombre del fichero de cadenas adaptadas al lugar sin la extensión `.strings`. Por ejemplo, si el fichero de cadenas adaptadas al lugar se llama `Find.strings` el parámetro `tableName` deberá valer `@"Find"`. Se acostumbra a usar el fichero `Localizable.strings` para las cadenas adaptadas al lugar y, de hecho por defecto, cuando el parámetro `tableView` vale `nil`, Cocoa busca el fichero `Localizable.strings`.

Por ejemplo, si en la carpeta `English.lproj` creamos el fichero `Localizable.strings` con este contenido:

```
DELETE = "Delete";
SURE_DELETE = "Do you really want to delete this file?";
CANCEL = "Cancel";
```

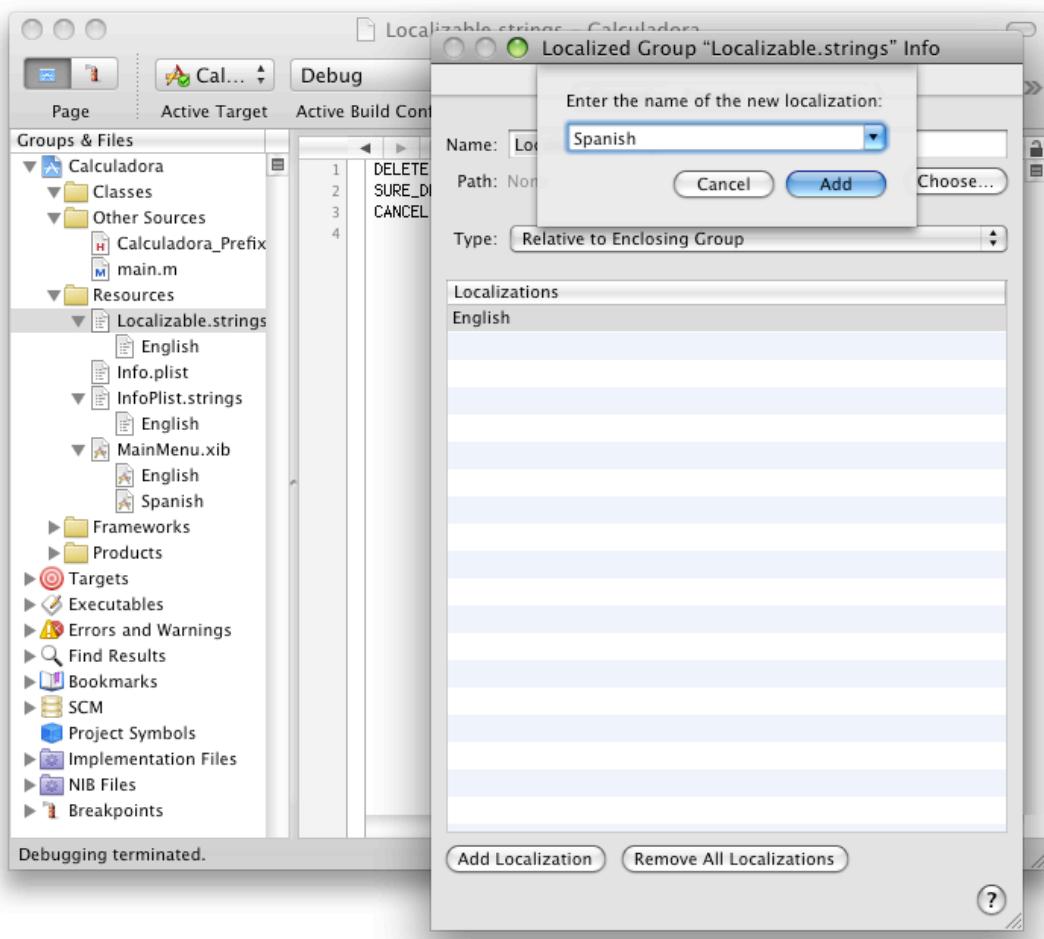
Y en la carpeta Spanish.lproj creamos el fichero Localizable.strings con este otro contenido:

```
DELETE = "Borrar";
SURE_DELETE = "¿Está seguro de querer borrar el fichero?";
CANCEL = "Cancelar";
```

Podemos obtener una cadena adaptada al lugar de la forma:

```
NSBundle* b = [NSBundle mainBundle];
NSString* cancelar = [b localizedStringForKey:@"CANCEL"
    value:@"Default cancel" table:@"Localizable"];
```

Igual que adaptamos al lugar el fichero MainMenu.xib seleccionándolo en Xcode y usando su panel de información (Command+I), podemos adaptar al lugar el fichero Localizable.strings de la misma manera, tal como muestra la Figura 4.9.



**Figura 4.9:** Adaptar al lugar el fichero Localizable.strings

Debido a que el acceso a cadenas adaptadas al lugar es una operación muy común, el fichero `NSBundle.h` proporciona el macro `NSLocalizedString()`. Este macro permite acceder a propiedades adaptadas al lugar de forma más cómoda. En concreto, podemos hacer:

```
NSString* cancelar = NSLocalizedString(@"CANCEL"  
, @"Un cancel de ejemplo");
```

El segundo parámetro es un comentario que ignora el macro, pero que el comando `genstrings` usa.

El comando `genstrings` nos permite generar un fichero `.strings` a partir de uno o más ficheros de código fuente. Es decir, el siguiente comando:

```
$ genstrings *.m
```

leería todos los ficheros `.m` del directorio actual y generaría en el directorio actual el fichero `Localizable.strings`. También podemos buscar cadenas en todos los subdirectorios de la forma:

```
$ find . -name "*.m" | xargs genstrings
```

En este fichero estarían todas las llamadas al macro `NSLocalizedString()` encontradas. Una vez tengamos el fichero `Localizable.strings`, es fácil generar versiones adaptadas a distintos idiomas.

## 2.4.5. Cargar código ejecutable

La mayoría de las aplicaciones usan sólo código de su propio ejecutable. Sin embargo, si nuestra aplicación soporta plug-ins, vamos a necesitar cargar su código dinámicamente.

En general, la forma de cargar código de ejecutables externos en el ejecutable de una aplicación es encontrar un punto de entrada a ese código externo. Normalmente, esto implica una cierta coordinación entre el desarrollador de la aplicación y el desarrollador del plug-in. El desarrollador de la aplicación suele publicar un API que el desarrollador del plug-in implementa.

En el caso de las aplicaciones Core Foundation el desarrollador del plug-in exporta una función o variable global con el modificador `extern "C"`. El desarrollador de la aplicación localiza esa función con la función `CFBundleGetFunctionPointerForName()`. Para localizar una variable global existe la función `CFBundleGetDataPointerForName()`.

En el caso de las aplicaciones Cocoa, podemos cargar una clase del bundle externo con el método:

```
- (Class)classNamed:(NSString*)className
```

A partir de este punto podemos crear instancias de esta clase.

Para evitar que el desarrollador del plug-in tenga que asignar un determinado nombre a la clase, muchas veces se usa el método alternativo:

```
- (Class)principalClass
```

El cual devuelve la clase cuyo nombre se proporciona en la propiedad `NSPrincipalClass` del fichero `Info.plist` del bundle del plug-in.

## 2.5. Variables de entorno

Algunas aplicaciones requieren variables de entorno para funcionar correctamente. Por ejemplo, supongamos que queremos pasar a NetBeans las variables de entorno `PATH` y `CLASSPATH`.

Una primera opción es fijar las variables de entorno en el terminal y ejecutar la aplicación desde el terminal:

```
$ export CLASSPATH=$CLASSPATH:./Library/Java/Home/lib/xjparse  
-1.0.jar  
$ export PATH=$PATH:/usr/local/sw/bin  
$ open /Applications/Development/NetBeans.app
```

Sin embargo, en ocasiones preferimos poder ejecutar la aplicación haciendo doble click en su ícono con Finder. En este caso tenemos dos alternativas: (1) Fijar variables de entorno para todas las aplicaciones que ejecute Finder, o (2) fijar variables de entorno para una determinada aplicación que ejecute Finder.

En el primer caso podemos crear un fichero de propiedades en la ruta `$HOME/.MacOSX/environment.plist`.

La segunda opción es fijar esas variables para una única aplicación. En este caso, usando la herramienta Property List Editor podemos editar el fichero `Info.plist` de la aplicación en cuestión (p.e. NetBeans) y añadir la propiedad `LSEnvironment`. Esta propiedad es un diccionario donde cada entrada representa una variable de entorno.

## 3. El sistema de preferencias

---

El sistema de preferencias de Mac OS X (defaults system) permite a las aplicaciones configurar su runtime mediante valores por defecto que se almacenan de forma persistente como listas de propiedades. Los frameworks y librerías también pueden hacer uso del sistema de preferencias.

Una vez que una aplicación está en ejecución, puede hacer uso del sistema de preferencias para modificar estos valores por defecto para las siguientes ejecuciones.

Existen dos formas principales de acceder a las preferencias. Una es mediante el comando `defaults`. La otra es mediante la clase `NSUserDefaults`. En los siguientes apartados detallaremos su uso. Ambas formas son heredadas de NeXTSTEP.

### 3.1. Los dominios

Las preferencias se agrupan en dominios. Cada dominio tiene un nombre (representado como una cadena) y sus valores se representan mediante un diccionario. Algunos dominios son volátiles, es decir, sus valores se pierden al terminar la ejecución de la aplicación, y otros son persistentes, es decir, sus valores se guardan en disco para usarse la próxima vez que se ejecute la aplicación. Además los dominios forman una lista de prioridad, es decir, cuando se busca una preferencia, se empieza buscando en el dominio de más prioridad, y sólo si este dominio no tiene la propiedad, se busca en el siguiente dominio. La Tabla 4.2 muestra cuáles son los dominios estándar en Mac OS X ordenados por prioridad.

| Nombre                 | Tipo        | Descripción   |
|------------------------|-------------|---|
| "NSArgumentDomain"     | Volátil     | Los argumentos con los que ejecutamos la aplicación |
| Dominio de aplicación  | Persistente | Las preferencias de una aplicación de un usuario    |
| "NSGlobalDomain"       | Persistente | Preferencias compartidas por todas las aplicaciones |
| "NSRegistrationDomain" | Volátil     | Preferencias por defecto de la aplicación           |

**Tabla 4.2:** Dominios ordenados por prioridad

El dominio con mayor prioridad es el dominio "NSArgumentDomain" que engloba los argumentos de línea de comando recibidos por la aplicación durante su ejecución. La clave se precede por guión, y a continuación se pone su

valor. Por ejemplo, si queremos que Safari muestre el menú de desarrolladores podemos modificar su propiedad con el comando:

```
$ /Applications/Safari.app/Contents/MacOS/Safari
-IncludeDevelopMenu 1
```

Como el dominio "NSArgumentDomain" es el que más prioridad tiene, Safari mostrará este menú independientemente de su valor en los demás dominios.

El dominio de aplicación es distinto para cada usuario y para cada aplicación, y se representa mediante el ID de aplicación que proporcionamos en la propiedad de configuración `CFBundleIdentifier` (p.e. "com.apple.Safari").

Las preferencias de aplicación se guardan en el directorio `~/Library/Preferences` mediante un fichero de lista de propiedades (`.plist`) con el ID de la aplicación correspondiente.

Aunque la clase `NSUserDefaults` de Cocoa no usa ni proporciona esta funcionalidad, podemos usar la función `CFPreferencesSetValue()` de Core Foundation para guardar preferencias de aplicación en el directorio `/Library/Preferences`. Este directorio también contiene ficheros cuyo nombre coincide con el ID de la aplicación, y representan propiedades para la aplicación independientes del usuario que ejecuta la aplicación.

Podemos usar el siguiente comando para obtener todos los nombres de dominios registrados en el sistema:

```
$ defaults domains
AddressBookMe, CorelReg, MATLAB, MacSOUP, NewsHunter, OSXnews,
.....
widget-com.apple.widget.worldclock
```

Podemos obtener el diccionario de un determinado dominio con el comando:

```
$ defaults read com.apple.Safari
{
    AddressBarIncludesGoogle = 1;
    AddressBarPreferencesWereConverted = 1;
    .....
}
```

O bien modificar las preferencias de un dominio con el comando:

```
$ defaults write com.apple.Safari AutoOpenSafeDownloads 0
```

Las preferencias globales se guardan en el dominio "NSGlobalDomain" y representan propiedades compartidas por todas las aplicaciones. Podemos usar el siguiente comando para conocer estas propiedades:

```
$ defaults read NSGlobalDomain
{
    AppleAntiAliasingThreshold = 4;
    AppleCollationOrder = en;
    .....
}
```

Por último, el dominio "NSRegistrationDomain" se usa para registrar preferencias por defecto, es decir, actúa como un mecanismo de fallback para cuando se ejecuta una aplicación y determinadas propiedades todavía no han sido fijadas por el usuario. En el apartado 3.3 veremos cómo se registran estas propiedades.

## 3.2. Acceso programático a las preferencias

Los programas Cocoa usan una instancia de la clase `NSUserDefaults` para acceder al sistema de preferencias. Para obtener esta instancia se suele usar el método `factory`:

```
+ (NSUserDefaults*) standardUserDefaults
```

Alternativamente se puede usar el método de instancia:

```
- (id) init
```

Ambos métodos devuelven un objeto `NSUserDefaults` inicializado con las preferencias del usuario. Si quisiéramos obtener las preferencias de otros usuario podemos usar:

```
- (id) initWithUser:(NSString*)username
```

Pero en este caso deberemos tener permiso de lectura para el directorio `home` del usuario dado en el parámetro `username`.

Una vez tengamos el objeto `NSUserDefaults`, podemos obtener estas propiedades con métodos getter como `stringForKey:` y métodos setter como `setInteger:forKey::`

También podemos modificar las propiedades del objeto `NSUserDefaults` mediante método setter. Para enviar estas preferencias a disco podemos usar el método `synchronize`. Este método lo podemos ejecutar antes de salir de la aplicación para garantizar que los cambios que hemos hecho en las preferencias se envíen a disco.

### 3.3. Fijar las preferencias por defecto

Cuando una aplicación llega al ordenador de un nuevo usuario, el usuario no tendrá un fichero de preferencias que `NSUserDefaults` pueda leer. En este caso el objeto devuelto por el método `factory` `standardUserDefaults` no tendrá preferencias a no ser que la aplicación disponga de preferencias por defecto.

Las **preferencias por defecto** son un mecanismo de fallback que permite a la aplicación fijar valores para determinadas preferencias cuando su valor no se encuentra en ningún otro dominio de la Tabla 4.2.

Para fijar las preferencias por defecto la clase `NSUserDefaults` proporciona el método:

- `(void)registerDefaults:(NSDictionary*)dictionary`

Este método se puede ejecutar al arrancar el programa (p.e. en el método `initialize` de una clase). De esta forma cuando el programa pida el objeto `NSUserDefaults`, éste ya estará inicializado con variables por defecto en el dominio "NSRegistrationDomain".

Guardar las preferencias en el código fuente de la aplicación no es muy recomendable de cara al mantenimiento de la aplicación. Como el método `registerDefaults:` recibe un diccionario, también es posible guardar las preferencias por defecto en un fichero de lista de propiedades situado en el subdirectorio `Resources` del bundle de la aplicación.

Tenga en cuenta que el valor por defecto de una propiedad se mantiene mientras que no modifiquemos el valor de esta propiedad mediante algún método setter del objeto `NSUserDefaults`. En consecuencia, mientras que se mantenga el valor por defecto de una propiedad, su valor no será almacenando en el fichero de preferencias de la aplicación (situado en `~/Library/Preferences`).

# Tema 5

## Gestión de procesos

---

### **Sinopsis:**

*En este tema se estudian una serie de clases que permiten acceder a la funcionalidad para gestión de procesos del sistema operativo. Estas clases nos van a permitir conocer información sobre nuestro proceso. Por ejemplo, podemos consultar las variables de entorno o preguntar cuál es el directorio home del usuario asociado al proceso. Además, en este tema también veremos clases que nos permiten crear nuevos procesos.*

# 1. Información de nuestro proceso

La clase `NSProcessInfo` es una clase singleton cuya instancia representa la información asociada al proceso donde se instancia. Para acceder a la información sobre nuestro proceso podemos usar el método factory singleton `processInfo` de la forma:

```
NSProcessInfo* pi = [NSProcessInfo processInfo];
```

## 1.1. Obtener información del proceso

Una vez que tenemos una instancia de `NSProcessInfo` podemos obtener las variables de entorno del proceso con el siguiente método que devuelve un diccionario donde tanto las claves como los valores del diccionario son objeto `NSString`:

- `(NSDictionary*)environment`

También podemos acceder a los argumentos de la línea de comando de `main()` con el siguiente método que devuelve un array de objetos `NSString`:

- `(NSArray*)arguments`

Y podemos obtener el nombre del proceso o su ID con los métodos:

- `(NSString*)processName`
- `(int)processIdentifier`

En ocasiones queremos obtener un ID globalmente único, en este caso podemos usar el método:

- `(NSString*)globallyUniqueString`

Este método devuelve un ID de la forma AF0B5145-5177-4E1F-AB56-0DCC080D4960-561-0000000C98CC5063. Este número se construye a partir de informaciones como el nombre del host, el nombre del proceso o el instante de tiempo. Por su naturaleza aleatoria se considera altamente improbable que este ID se repita en otra ejecución del método.

## 1.2. Obtener información del host

Podemos obtener la versión mayor del sistema operativo con el método de instancia `operatingSystem` (p.e. en Mac OS X 10.5 devolvería 5). O bien una descripción completa de la versión con `operatingSystemVersionString` En

mi actual versión de Mac OS X 10.5 devuelve la cadena: Version 10.5.5 (Build 9F33)

También podemos preguntar por el número de procesadores que tiene la máquina, o el número de ellos que están activos con:

- (NSUInteger)processorCount
- (NSUInteger)activeProcessorCount

O bien por el número de bytes de memoria física con:

- (unsigned long long)physicalMemory

## 2. Crear nuevos procesos

---

Podemos usar la clase `NSTask` para lanzar y monitorizar la ejecución de otro proceso.

La forma más sencilla de lanzar un proceso es usando el método `factory`:

```
+ (NSTask*)launchedTaskWithLaunchPath:(NSString*)path
                                arguments:(NSArray*)arguments
```

En `path` debemos especificar el nombre del binario<sup>1</sup> a ejecutar. En `arguments` podemos poner objetos `NSString` con los argumentos a pasar al nuevo proceso. Por ejemplo, para lanzar Safari podemos hacer:

```
NSTask* t = [NSTask launchedTaskWithLaunchPath:
              @"/Applications/Safari.app/Contents/MacOS/Safari"
              arguments:[NSArray array]];
```

### 2.1. Obtener información sobre el proceso

Existen métodos como `arguments`, `currentDirectoryPath`, `environment`, `launchPath`, `processIdentifier`, `standardError`, `standardInput` o `standardOutput` que nos permiten obtener información sobre el proceso que hemos lanzado.

También podemos obtener información sobre el estado de ejecución del proceso con el método `isRunning` que nos devuelve un booleano indicando si el proceso ha terminado su ejecución. También podemos bloquear nuestro hilo en espera de que el proceso lanzado acabe con el método `waitForExit`. Una vez que el proceso acaba podemos usar el método:

---

<sup>1</sup> Ojo, no funciona si proporcionamos path del bundle de una aplicación.

```
- (int)terminationStatus
```

que nos devuelve el código de terminación del proceso (o si ha terminado bien). El método `terminationStatus` no debe ejecutarse hasta que el proceso haya acabado o se producirá una excepción.

Por último, también existen métodos que nos permiten parar, reanudar o terminar un proceso como `resume`, `suspend`, `interrupt` o `terminate`. La diferencia entre `interrupt` y `terminate` es que el primer método envía la señal `SIGINT` al proceso mientras que el segundo envía la señal `SIGTERM`.

## 2.2. Modificar el entorno de ejecución de un proceso

Por defecto, cuando lanzamos un nuevo proceso con `NSTask`, este proceso hijo hereda las variables de entorno, directorio actual, entrada estándar, salida estándar y salida de errores estándar del proceso padre que lo lanza. Si queremos modificar estos valores, debemos hacerlo antes de lanzar el proceso. Para ello podemos instanciar un objeto de tipo `NSTask`, fijar sus propiedades y ejecutarlo luego llamando al método `launch`. El Listado 5.1 muestra un programa que lanza el comando `grep` en el directorio `home` del usuario (independientemente del directorio desde el que ejecutemos el programa del Listado 5.1). Para ello cambiamos el directorio del proceso hijo con el método `setCurrentDirectoryPath:`. El método `waitForExit` bloquea el padre hasta que el proceso hijo termine. El método `terminationStatus` nos informa sobre el código de terminación del proceso hijo.

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSTask* t = [[NSTask alloc] init];
    NSArray* args = [NSArray
                      arrayWithObjects:@"hola",@"carta.txt",nil];
    [t setArguments:args];
    [t setLaunchPath:@"/usr/bin/grep"];
    [t setCurrentDirectoryPath:NSUTFHomeDirectory()];
    [t launch];
    [t waitForExit];
    int status = [t terminationStatus];
    [t release];
    printf("Código de terminación: %i\n",status);
    [pool drain];
    return 0;
}
```

**Listado 5.1:** Programa que lanza un proceso

Es importante tener en cuenta que cada instancia de `NSTask` se puede usar sólo una vez para ejecutar un programa. Si volvemos a usar el mismo objeto para lanzar otro proceso se produce un error.

# Tema 6

# Programación multihilo

---

## **Sinopsis:**

*En este tema introduciremos los conceptos, ventajas e inconvenientes de la programación multihilo. Después veremos cómo crear y gestionar nuevos hilos. Para acabar el tema estudiaremos cómo encapsular operaciones en objetos operación, y qué ventajas introducen.*

# 1. Conceptos básicos

En los sistemas operativos modernos un proceso puede ejecutar uno o más hilos. Normalmente el proceso empieza ejecutando la función `main()` con el **hilo principal**. A partir de ese momento el hilo principal puede lanzar otros hilos.

En Mac OS Classic y en el núcleo Mach, en el cual Mac OS X está basado, se usó el término "tarea"<sup>1</sup> para referirse a cada aplicación en ejecución, es decir, a cada proceso. A partir de Mac OS X se adoptó la siguiente nomenclatura:

- El término **hilo** (thread) se utiliza para referirse a cada hilo de ejecución.
- El término **proceso** (process) se utiliza para referirse a un programa en ejecución, el cual consta de un espacio de memoria de usuario y uno o más hilos.
- El término **tarea** (task) se usa para referirse a una operación concreta a realizar por el hilo.

En los siguientes subapartados vamos a resumir la terminología y conceptos en que se va a basar nuestra explicación.

## 1.1. Tipos de hilos

El núcleo de Mac OS X implementa los hilos mediante **Mach threads**, que son los únicos hilos que existen para el núcleo de Mac OS X. Sin embargo los Mach threads están destinados al núcleo de Mac OS X y nunca deberemos crear directamente Mach threads en el espacio de memoria de un proceso.

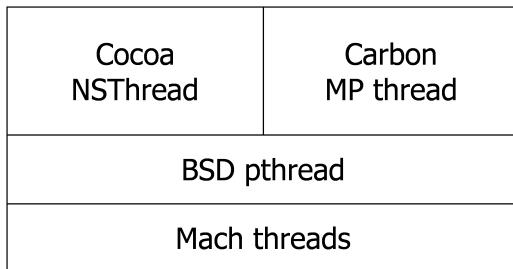
A su nivel más básico el núcleo BSD de Mac OS X proporciona los **pthreads** (POSIX threads) que son hilos que cumplen con el estándar POSIX para programación multihilo. Los pthreads envuelven un Mach thread mediante una interfaz de programación POSIX para el lenguaje C (véase Figura 6.1). La principal ventaja de los pthreads es que los sistemas UNIX (p.e. Linux, Solaris o BSD) implementan esta interfaz de programación, con lo que podemos escribir aplicaciones o librerías fácilmente portables a otras plataformas que hagan uso de pthreads.

Cocoa simplifica la programación multihilo mediante **hilos Cocoa**, representados por la clase `NSThread`, la cual envuelve un pthread y nos da acceso a la interfaz del hilo mediante métodos Objective-C.

---

<sup>1</sup> La clase `NSTask` todavía mantiene este nombre por haber sido heredado de los tiempos en que se estaba pasando de Mac OS Classic a Mac OS X.

Carbon también envuelve los pthreads en **MP threads** (Multi Processing Threads). Pero esta interfaz de programación está desaconsejada por Apple y no la vamos a estudiar aquí.



**Figura 6.1:** Tipos de hilos en Mac OS X

BSD distingue entre dos tipos de threads:

- **Joinable threads** (también llamados **nondetached threads**). Son hilos que se crean con las primitivas `pthread_create()` y `pthread_join()`. Los recursos de este tipo de hilos no los reclama automáticamente el sistema al acabar su ejecución, sino que otro hilo tiene que hacer `pthread_join()` con él para liberar sus recursos. Alternativamente se puede hacer `detach` de este hilo para no tener que esperarle.
- **Detached threads**. Son hilos cuyos recursos son liberados automáticamente al acabar su ejecución, con lo que no hay que hacer `pthread_join()` con el hilo para liberar sus recursos.

BSD permite trabajar tanto con joinable threads como con detached threads, pero Cocoa y Carbon sólo permiten trabajar con detached threads.

Un proceso ejecuta hasta que todos sus joinable threads han terminado. El hilo principal es un joinable thread. Si el hilo principal crea detached threads, estos terminarán tan pronto como termine el hilo principal. Por el contrario, si el hilo principal crea joinable threads y ejecuta la operación `pthread_join()` sobre ellos, todos los joinable threads deberán terminar para que termine el proceso. Si el hilo principal crea joinable threads, pero se olvida de ejecutar `pthread_join()` sobre ellos, el proceso termina cuando termina el hilo principal. Esto se debe a que al acabar el hilo principal la función `main()`, este hilo ejecuta la operación `exit()`, la cual termina con todos los hilos del proceso.

## 1.2. Técnicas de sincronización

El principal problema que tiene la existencia de varios hilos de ejecución dentro de un mismo proceso es que se puede producir un conflicto si varios de ellos intentan acceder a la vez al mismo recurso. Una forma de evitar estos conflictos es que cada hilo posea sus propios recursos no compartidos. Sin embargo, en la práctica es necesario compartir algunos recursos con el fin de

permitir la comunicación entre hilos y de evitar un aumento excesivo del consumo de memoria. En este caso es necesario sincronizar el acceso a los recursos.

Las técnicas de sincronización se pueden dividir en técnicas de sincronización entre hilo y técnicas de sincronización entre procesos. Las **técnicas de sincronización entre hilos** permiten la sincronización entre hilos de un mismo proceso mientras que las **técnicas de sincronización entre procesos** permiten la comunicación entre hilos situados en distintos procesos. Debido a que comparten el mismo espacio de memoria, en general las técnicas de sincronización entre hilos son menos costosas que las técnicas de sincronización entre procesos. Normalmente, las técnicas de comunicación entre procesos recurren al sistema operativo para transferir información entre los espacios de memoria de los distintos procesos. En el Tema 7 veremos que existen muchas técnicas de sincronización.

### 1.3. Las señales

Las señales son un mecanismo introducido por los sistemas BSD para avisar a un proceso de la ocurrencia de un evento. Normalmente los procesos tienen un **manejador de señales** por defecto que responde convenientemente a cada señal, pero en ocasiones queremos personalizar este comportamiento instalando un manejador de señales personalizado. Los manejadores de señales no son más que funciones callback asignadas a una determinada señal.

En una aplicación monohilo, la señal siempre se ejecuta sobre el hilo principal. La complicación surge cuando la señal se envía a una aplicación multihilo. En este caso la señal se ejecuta sobre el hilo que estuviese en ese momento en ejecución. Como no podemos saber cuál va a ser ese hilo, el manejador de señales nunca debe de hacer suposiciones sobre qué hilo va a ejecutar el manejador de señales cuando se produzca la señal.

### 1.4. Los bucles de sondeo

Un **bucle de sondeo** (run loop) es una estructura de programación que permite procesar eventos que llegan al programa de forma asíncrona.

El uso de los bucles de sondeo se ha generalizado en las aplicaciones orientadas a eventos, que normalmente son las aplicaciones que tienen una interfaz gráfica. Sistemas operativos como Microsoft Windows, Linux o Mac OS X usan esta estructura de programación porque permite mantener a un hilo ocupado cuando llegan eventos y poner al hilo a dormir cuando no hay eventos.

En Mac OS X tanto las aplicaciones Cocoa como Carbon usan bucles de sondeo de eventos para implementar la interfaz gráfica. En concreto, una vez

inicializada la aplicación, el hilo principal entra en un bucle de sondeo el cual va procesando los eventos que llegan por las llamadas **fuentes de eventos** (event sources). Un bucle de sondeo siempre tendrá asociada una o más fuentes de eventos. Ejemplos de eventos son los movimientos del ratón, las pulsaciones del teclado, los temporizadores o la gestión de mensajes procedentes de objetos distribuidos.

Aunque el bucle de sondeo de la aplicación normalmente lo gestiona el hilo principal, nosotros podemos crear nuevos bucles de sondeo y asociarlos a otros hilos. En el Tema 8 veremos cómo crear y gestionar bucles de sondeo.

## 1.5. Consideraciones de diseño

Es importante que el programador entienda que el hecho de que un sistema operativo proporcione programación multihilo, no significa que sus aplicaciones deban usar varios hilos. La decisión de usar varios hilos también implica varios efectos perjudiciales que el programador debe de considerar antes de convertir su aplicación en una aplicación multihilo.

La primera consideración es que los hilos son estructuras de programación cuya creación y uso consume memoria y CPU. La memoria consumida por un hilo no es trivial, e implica asignar varias estructuras de programación tanto en el espacio de memoria del kernel como en el espacio de memoria del proceso. En consecuencia, si la tarea a realizar es relativamente corta, seguramente el coste de crear un nuevo hilo para que la ejecute sea mayor que el beneficio que aporte el ejecutar esa tarea en otro hilo. En este caso también conviene considerar el uso de objetos operación (véase apartado 1.7), ya que estos permiten reutilizar hilos en vez de crear un hilo por cada nueva operación.

Una segunda consideración es que la programación multihilo suele complicar la estructura general del programa dificultando su programación, pero sobre todo su mantenimiento. Aunque los datos compartidos entre hilos se pueden reducir, siempre existirá un conjunto de datos que deberán ser compartidos, lo cual implica el uso de técnicas de sincronización no triviales como las que veremos en el Tema 7.

Una tercera consideración es estudiar cómo terminan los hilos. En el apartado 1.1 indicamos que un proceso termina cuando todos sus joinable thread terminan. En principio el único joinable thread es el hilo principal, todos los demás hilos se crean como detached thread. Si queremos asegurarnos de que las operaciones que están realizando otros hilos terminan correctamente, deberemos usar técnicas de sincronización para esperar a que éstos terminen, o bien crear los hilos secundarios como joinable threads.

La cuarta consideración que vamos a resaltar es que cada hilo tiene su propia pila de memoria y su propia pila de llamadas a funciones, con lo que cada hilo es responsable de gestionar sus propias excepciones. Además un hilo no puede lanzar una excepción a otro hilo. Si un hilo no captura una excepción (sea éste el hilo principal o un hilo secundario), el proceso entero se cerrará.

## 1.6. Alternativas a la programación multihilo

Dados los inconvenientes que presentan las aplicaciones multihilo, en ocasiones puede resultar más conveniente para el programador no crear nuevos hilos. En este apartado vamos a comentar algunas alternativas a la programación multihilo que el programador debería de considerar.

Cuando la operación a ejecutar no es urgente, una primera alternativa a la programación multihilo es usar las **notificaciones de tiempo libre** (idle-time notifications). Cocoa proporciona un mecanismo para notificar a la aplicación que no tiene trabajo pendiente de procesar. Para ello debemos de registrar un objeto como observador de la **cola de notificaciones** de la aplicación (representada por la clase `NSNotificationQueue`) y pedirla que le informe cuando la aplicación tenga tiempo libre usando la opción `NSPostWhenIdle`. En el apartado 4.6 del Tema 8 veremos con más detalle cómo funcionan las colas de notificación.

Una segunda alternativa es usar **funciones asíncronas**, que son funciones cuya tarea se sigue ejecutando una vez que retorna la llamada a la función. Cuando una operación sea relativamente costosa y existan tanto funciones síncronas como asíncronas que realicen esa tarea, el uso de funciones asíncronas puede evitarnos el tener que crear un nuevo hilo. Internamente, la función asíncrona puede lanzar un hilo, lanzar un nuevo proceso, o usar un demonio ya existente para realizar la operación. La forma en que esté implementada la operación asíncrona es transparente al programador.

Una tercera alternativa especialmente útil cuando una aplicación deba de ejecutar una tarea periódicamente es usar **temporizadores**. En el apartado 3.6 del Tema 8 veremos cómo responder a temporizadores en el bucle de sondeo del hilo principal. Este mecanismo evita el tener que crear un hilo adicional.

La cuarta alternativa que conviene considerar es **lanzar otro proceso**. Aunque lanzar un proceso es una operación más costosa que lanzar un hilo, tiene la ventaja de que no introduce problemas de sincronización, con lo que es una alternativa que puede resultar bastante interesante cuando la operación a realizar es lo suficientemente costosa como para poder desestimar el coste de crear el nuevo proceso.

## 1.7. Los objetos operación

Los procesadores actuales cada vez cuentan con más cores que permiten ejecutar instrucciones en paralelo. Sin embargo, como hemos visto en los apartados anteriores, la programación concurrente tiene la reputación de ser difícil y propensa a errores. Esto hace que muchos programadores prefieran no usarla. En Mac OS X 10.5 se han introducido los **objetos operación** que permiten evitar los efectos laterales de la programación multihilo, haciendo que ésta sea más sencilla de gestionar. En el apartado 3 veremos cómo se pueden encapsular operaciones concurrentes en objetos operación.

## 1.8. Protección multihilo

En una aplicación multihilo un objeto puede recibir mensajes desde distintos hilos. Se dice que un objeto tiene **protección multihilo** (thread safety) si sus métodos pueden ser ejecutados desde distintos hilos. A grosso modo, las clases del Foundation Framework tienen protección multihilo, mientras que las clases del Application Framework no tienen protección multihilo. Si una clase no tiene protección multihilo, sus instancias sólo deben ser accedidas desde el hilo que las creó (que normalmente es el hilo principal). La razón más común de que una clase no tenga protección multihilo es que acceda a bucles de sondeo para recoger eventos. Lógicamente, esta clasificación es muy genérica: Las clases de QuickTime son clases de interfaz gráfica que sí que tienen protección multihilo, y esto permite que podamos ejecutar sobre ellas operaciones de apertura y renderizado de vídeo desde hilos secundarios. Si queremos saber cuando una determinada clase tiene protección multihilo, debemos consultar la documentación de esa clase.

## 2. Crear hilos

En el apartado 1.1 adelantamos que los Mach threads son los únicos tipos de hilos que realmente existen en Mac OS X. Sin embargo, el programador de aplicaciones crea objetos de alto nivel que envuelven a estos hilos. En este apartado vamos a ver cómo se crear dos de estos objetos de alto nivel: Los hilos Cocoa y los hilos POSIX.

### 2.1. Crear hilos Cocoa

Los hilos Cocoa están representados por la clase `NSThread`. En este apartado vamos a comentar las tres formas que existen para crear un hilo Cocoa:

1. Usar el método `detachNewThreadSelector:toTarget:withObject:` para crear el hilo.
2. Crear una instancia de `NSThread` y lanzarla con el método `start`.
3. Usar `NSObject` para crear un hilo.

Las tres técnicas crear un detached thread. Como indicamos en el apartado 1.1, la única forma de crear un joinable thread es usando POSIX.

La primera forma de crear un hilo Cocoa es usando el método `factory`:

```
+ (void)detachNewThreadSelector: (SEL)aSelector  
                      toTarget: (id)aTarget  
                     withObject: (id)anArgument
```

En este caso debemos de indicar en el parámetro `aSelector` el selector del método donde está el algoritmo a ejecutar por el hilo (el punto de entrada al hilo) y en `aTarget` el objeto donde está implementado ese método. En `anArgument` podemos pasar un puntero a objeto dinámico que se pasará al método con el algoritmo del hilo. Este parámetro puede ser `nil` si no queremos pasar información al hilo.

Por ejemplo, para ejecutar en un hilo distinto el algoritmo del método `sumaVectores:` de la clase donde estemos situados podríamos hacer:

```
[NSThread detachNewThreadSelector:@selector(sumaVectores:  
                                      toTarget:self  
                                     withObject:nil];
```

Obsérvese que esta forma de lanzar el hilo no nos permite obtener una referencia al objeto `NSThread` que hemos creado. Esto es una limitación cuando, por ejemplo, queremos esperar a que el hilo termine.

La segunda forma que vamos a ver de crear un hilo Cocoa es crear una instancia de la clase `NSThread` y después poner el hilo en ejecución con su método `start`. En este caso debemos de tener en cuenta los siguientes métodos:

- `(id) init`
- `(id) initWithTarget: (id) target  
selector: (SEL) selector  
object: (id) argument`
- `(void) start`
- `(void) main`

El método `init` inicializa el objeto con `target` el propio hilo, con `selector @selector(main)` y con argumento `nil`. Si queremos usar otro `target+selector` u otro argumento debemos de inicializar el objeto hilo con el método `initWithTarget:selector:object:..`. Obsérvese que la información que recibe `initWithTarget:selector:object:..` es la misma información que recibe `detachNewThreadSelector:toTarget:withObject:..`.

El método `start` pone en ejecución el hilo, para lo cual usa como punto de entrada al hilo el `target+selector` dados durante la inicialización.

El método `main` por defecto ejecuta el selector pasado como parámetro al método de inicialización. Si no se especificó `target+selector` (el objeto hilo se creó con `init`) el método `main` de la clase `NSThread` por defecto se limita a retornar. En este caso podemos redefinir `main` para que contenga el algoritmo del hilo.

Obsérvese que hay dos formas de indicar el algoritmo del hilo dependiendo de cómo hemos inicializado el objeto:

1. Si hemos inicializados el objeto con `init` podemos redefinir directamente el método `main` para que contenga el algoritmo a ejecutar.
2. Si hemos inicializado el objeto con `initWithTarget:selector:object:..`, `main` ejecutará este `target+selector`, y si hemos proporcionado un argumento, se lo pasará a este método. En este segundo caso no debemos de redefinir `main`.

A la hora de decidir si usar la primera o segunda forma debemos de tener en cuenta que la primera forma exige crear una clase que derive de `NSThread`, mientras que la segunda forma nos permite situar el algoritmo del hilo en otra clase, con lo que no es necesario crear una derivada de `NSThread`.

Es importante tener en cuenta que nunca debemos de llamar directamente al método `main` de un objeto hilo, sino que debemos de hacerlo indirectamente a través de `start`. Esto se debe a que `start` realiza inicializaciones necesarias para que una aplicación funcione correctamente en modo multihilo.

La tercera forma de crear un hilo Cocoa es usar el siguiente método de instancia de la clase `NSObject`:

```
- (void)performSelectorInBackground:(SEL)aSelector withObject:(id)arg
```

Este método crea un `NSThread` que se ejecuta con `target self`, con el selector dado en `aSelector`, y con el argumento dado en `arg`.

El efecto de ejecutar este método es el mismo que el de llamar al método `factory detachNewThreadSelector:toTarget:withObject:` pasándole como parámetro `target` el `NSObject` sobre el que se ejecuta el método `performSelectorInBackground:withObject::`. Luego este método no es más que una forma abreviada de ejecutar el anterior.

Por último, es importante comentar que usemos la forma que usemos para crear el hilo Cocoa, si no tenemos activada la recogida automática de basura Cocoa creará un autorelease pool antes de ejecutar el hilo y lo liberará al acabar su ejecución. Por esta razón no es necesario que nosotros creamos un autorelease pool al entrar en un hilo Cocoa. Sin embargo, como veremos más abajo, los hilos POSIX no crear un autorelease pool. Además, Cocoa incrementa la cuenta de referencias de los objetos target y argumento y la decrementa cuando acaba la ejecución del hilo.

## 2.2. Crear hilos POSIX

POSIX proporciona una API C para crear pthreads. Como ya indicamos en el apartado 1.1, la principal ventaja de esta API es que puede ser usada desde muchos entornos de programación (incluido Cocoa). Si estamos desarrollando una librería para varias plataformas es más adecuado usar POSIX. Si nuestra librería es sólo para Mac OS X, es más cómodo usar hilos Cocoa.

Los pthreads están declarados en el fichero de cabecera `<pthread.h>`, y se representan como instancias del tipo `pthread_t`. Sus atributos se almacenan en otra estructura de tipo `pthread_attr_t`. Todas las funciones de pthread devuelven un código de error que puede ser cero cuando no hay error o un valor distinto para indicar el tipo de error. El Listado 6.1 muestra un ejemplo de creación de un hilo POSIX.

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

void* HiloTrabajador(void* p) {
    printf("Hola desde el hilo\n");
    return NULL;
```

```

}

int main (int argc, const char * argv[]) {
    pthread_attr_t atributos;
    pthread_t handle_hilo;
    assert(!pthread_attr_init(&atributos));
    assert(!pthread_create(&handle_hilo,
                          &atributos,HiloTrabajador,NULL));
    assert(!pthread_join(handle_hilo, NULL));
    assert(!pthread_attr_destroy(&atributos));
    printf("Acaba el hilo principal\n");
    return 0;
}

```

**Listado 6.1:** Joinable thread POSIX

Lo primero que tenemos que hacer para crear un pthread es inicializar sus atributos con:

```
int pthread_attr_init(pthread_attr_t* attr);
```

Por defecto los hilos pthread son joinable threads. Podemos cambiar su tipo a detached thread con la llamada a:

```
int pthread_attr_setdetachstate(pthread_attr_t* attr,
                               int detachstate);
```

Este cambio deberíamos de hacerlo antes de crear al hilo. Para crear un hilo usamos:

```
int pthread_create(pthread_t* restrict thread,
                  const pthread_attr_t* restrict attr,
                  void* (*start_routine) (void*),
                  void* restrict arg);
```

El parámetro `thread` es un parámetro de salida con el handle del hilo a crear. El parámetro `attr` son sus atributos que deben estar debidamente inicializados. El parámetro `start_routine` es un puntero a la función con el algoritmo del hilo. Por último `arg` es un argumento que podemos pasar a la función con el algoritmo del hilo.

Si creamos un joinable thread, deberemos esperar a la finalización del hilo llamando a:

```
int pthread_join(pthread_t thread, void** value_ptr);
```

## 2.3. Hilos POSIX en Cocoa

Cuando estamos programando en Cocoa, lo más sencillo es usar hilos Cocoa, pero si nuestra aplicación usa código basado en POSIX podemos mezclar ambas tecnologías aunque en este caso debemos de tener en cuenta dos precauciones que vamos a comentar.

En el caso de las aplicaciones multihilo, el framework de Cocoa utiliza mutexes y otras estructuras de sincronización. Sin embargo, Cocoa no crea estas estructuras si la aplicación consta de un sólo hilo. La razón de este diseño es evitar que estas estructuras de sincronización degraden el rendimiento de la aplicación. La primera vez que la aplicación crea un segundo hilo Cocoa, la aplicación pasa a ser multihilo y estas estructuras se activan. Los hilos POSIX no hacen uso ni inicializan estas estructuras. Por eso, la primera precaución que debemos tener en cuenta si nuestra aplicación Cocoa crea hilos POSIX es realizar esta inicialización. Para que el framework de Cocoa sepa que la aplicación es multihilo debemos de lanzar un hilo Cocoa usando uno de los métodos del apartado 2.1. Es suficiente con crear un hilo que simplemente no haga nada y termine. Una forma alternativa es enviar al centro de notificación por defecto el mensaje `NSWillBecomeMultiThreadedNotification`. Podemos consultar si nuestra aplicación ha entrado en modo multithread con el método de clase de `NSThread`:

```
+ (BOOL)isMultiThreaded
```

En el apartado 2.1 indicamos que siempre que creamos un hilo Cocoa y no teníamos activada la recogida automática de basura se crea un autorelease pool para el nuevo hilo que se va a ejecutar. Esto no es cierto cuando se lanza el hilo con POSIX, con lo que la segunda precaución que debemos tener al usar POSIX para crear hilos es crear nosotros este autorelease pool al principio de la ejecución del hilo y destruirlo en su finalización.

## 2.4. Configurar los hilos

Antes de crear un hilo, o bien durante su ejecución, podemos querer configurar el entorno de ejecución del hilo. En los siguientes apartados vamos a ver cómo configurar un hilo tanto Cocoa como POSIX.

### 2.4.1. Tamaño de la pila

Cada vez que creamos un hilo Mac OS X asigna una cantidad de memoria para la pila de memoria de ese hilo<sup>1</sup>. Si queremos cambiar el tamaño de la pila debemos hacerlo antes de lanzar el hilo.

La forma de hacerlo depende de si el hilo es Cocoa o POSIX. En el caso de que el hilo sea Cocoa debemos de crear una instancia de `NSThread` y usar el método `setStackSize:` antes de llamar a su método `start`. En caso de que el hilo sea POSIX, debemos de crear una estructura de tipo `pthread_attr_t` e inicializarla con el tamaño de pila deseado con la función `pthread_attr_setstacksize()` antes de crear el hilo con la función `pthread_create()`.

### 2.4.2. Almacenamiento local al hilo

Tanto los hilos Cocoa como POSIX permiten almacenar información asociada a cada hilo. La forma de almacenar esta información varía dependiendo de si estamos usando un hilo Cocoa o un hilo POSIX, y no es intercambiable, es decir, no podemos usar POSIX para obtener información local a un hilo Cocoa.

En caso de estar usando un hilo Cocoa su información local se almacena en un diccionario que podemos acceder con:

- `(NSMutableDictionary*) threadDictionary`

Obsérvese que el método devuelve un `NSMutableDictionary`, con lo que podemos modificar directamente el contenido de este diccionario.

En caso de estar usando POSIX, podemos usar las funciones:

```
int pthread_setspecific(pthread_key_t key, const void* value);
void* pthread_getspecific(pthread_key_t key);
```

para escribir o leer información asociada a un hilo.

### 2.4.3. Prioridad del hilo

Mac OS X permite asignar más o menos prioridad a los hilos. Un hilo con más prioridad recibe con más regularidad el control, pero a cambio de que otros

---

<sup>1</sup> La documentación de Apple indica que por defecto se asigna una pila de 8MB para el hilo principal y 512KB para los hilos secundarios.

hilos puedan quedar semiparados, con lo que no se recomienda cambiar la prioridad de los hilos a no ser que haya una razón de peso para hacerlo. Un hilo que escribe de buffer a una grabadora de disco es una buena razón para darle más prioridad, un hilo que renderiza vídeo en pantalla no es razón suficiente para modificar la prioridad de los hilos. Si a estos dos hilos les suben la prioridad dos programadores de aplicaciones distintas podríamos estar haciendo perder CDs al usuario que visualiza vídeos mientras graba CDs. Bajar la prioridad de un hilo no es tan problemático, e incluso puede hacer más agradable la interfaz de usuario. Por ejemplo, el proceso `mdworker` de Spotlight tiene reducida la prioridad de sus hilos para no interferir en los tiempos de respuesta de la interfaz de usuario. Una vez advertido esto, vamos a ver que tanto Cocoa como POSIX permiten modificar la prioridad de los hilos.

Para modificar la prioridad de un hilo Cocoa tenemos el método:

```
+ (BOOL) setThreadPriority: (double) priority
```

Este método se puede ejecutar tanto antes de crear el hilo como después de crearlo. El parámetro `priority` deberá ser un número entre 0.0 (mínima prioridad) y 1.0 (máxima prioridad).

Para modificar la prioridad de un hilo POSIX tenemos la función:

```
int pthread_setschedparam(pthread_t thread, int policy,
                           const struct sched_param *param);
```

## 2.5. Terminar los hilos

La forma recomendada de terminar un hilo es dejar que el hilo acabe. Matar un hilo puede implicar que algunos recursos queden mal cerrados y que se produzcan pérdidas de memoria. Sólo en casos muy extremos debemos de matar a los hilos.

Muchas veces la condición de terminación es ajena al hilo (p.e. el hilo debe terminar si el usuario pulsa el botón de cancelar), en este caso lo recomendable es implementar un mecanismo de terminación. Tanto Cocoa como POSIX facilitan el implementar este mecanismo. En Cocoa podemos marcar a un hilo como cancelado con el método:

```
- (void) cancel
```

En todo momento el hilo puede comprobar si está cancelado con el método:

```
- (BOOL) isCancelled
```

y en caso de haber sido cancelado dejar la operación que estaba realizando y terminar.

Cuando pedimos cancelar un hilo, podemos saber si sigue en ejecución con los métodos:

- (BOOL)isExecuting
- (BOOL)isFinished

En POSIX podemos cancelar un hilo con `pthread_cancel()` y comprobar si ha sido cancelado con `pthread_testcancel()`.

La solución más drástica se aplicaría sólo cuando no hemos implementado un mecanismo de cancelación. No está claro si es necesario proporcionar un mecanismo para matar a un hilo o si es responsabilidad del programador el evitar que esta condición se pudiera producir. En base a esta línea de discusión, Cocoa no proporciona un mecanismo para matar a un hilo, con lo que nunca podremos matar hilos Cocoa. Sin embargo, POSIX sí que implementa este mecanismo con la función `pthread_kill()`.

## 3. Programación con objetos operación

---

En el apartado 1.7 introdujimos los objetos operación y motivamos su uso para evitar los efectos laterales inherentes a la programación multihilo. Estos objetos permiten encapsular operaciones a ejecutar. Cada objeto operación incluye unos datos y una operación. Para crear objetos operación creamos instancias de la clase `NSOperation`. En este apartado vamos a aprender cómo se programa con objetos operación.

### 3.1. Ejecución directa frente a colas de operación

Los objetos operación están pensados para que sólo se puedan ejecutar una vez. Si queremos ejecutar una operación varias veces debemos crear nuevas instancias de la misma operación. Al acabar de ejecutarse la operación, está contendrá el resultado de su ejecución. Si se intenta volver a ejecutar una instancia ya ejecutada se producirá una excepción.

Una vez que tenemos una instancia de un objeto operación hay dos formas de ejecutarla:

1. **Ejecución directa.** En este caso debemos ejecutar el método `start` del objeto operación. Este método configura el entorno de ejecución y después ejecuta la operación.
2. **Ejecución en cola de operación.** En este caso añadimos la operación a una cola de operación (representada por una instancia de la clase `NSOperationQueue`). La cola de operación se encargará de ejecutar la operación tan pronto como sea posible.

### 3.2. Operaciones no concurrentes y concurrentes

Los objetos operación pueden ser diseñados para ser operaciones no concurrentes u operaciones concurrentes. Se llama **operación no concurrente** a una operación síncrona, es decir, a una operación que no crea un nuevo hilo para su ejecución. En este caso el método `start` no retorna hasta que la operación haya terminado. Por el contrario las **operaciones concurrentes** son aquellas que se ejecutan de forma asíncrona, es decir, en las que el método `start` retorna antes de que la operación se haya terminado.

La forma en que se implementa la operación concurrente no está definida. La forma más trivial de implementarla sería lanzar un nuevo hilo que lleve a cabo la tarea, pero una operación concurrente podría lanzar también un proceso, o bien podría realizar una llamada a una función asíncrona. Otra razón por la que una operación sería concurrente es porque se queda esperando una

respuesta en un método callback. Incluso la operación concurrente podría hacer cosas tan estrambóticas como crear un temporizador y ejecutar alguna operación cada vez que se dispare el temporizador. En cualquier caso, la característica común de las operaciones concurrentes es que el método `start` retorna antes de que la operación se haya terminado.

Por defecto los objetos operación son no concurrentes. Para indicar que un objeto operación es concurrente debemos de redefinir su método `isConcurrent` para que devuelva YES.

Cuando lanzamos operaciones con una cola de operación es importante indicar si la operación es o no concurrente. Las colas de operación no usan un nuevo hilo para lanzar las operaciones marcadas como concurrentes, sino que llaman a su método `start` y la cola de operación espera que esta llamada retorne rápidamente. Por el contrario, la cola de operación usará un hilo distinto para lanzar cada operación que indique que no es concurrente.

Por último, cuando una operación es concurrente, también debe redefinir los métodos `isExecuting` y `isFinished` para que la podamos preguntar respectivamente si la operación se está ejecutando y si la operación ha acabado.

### 3.3. Operaciones no concurrentes

En este apartado vamos a detallar cómo implementar y ejecutar una operación no concurrente.

#### 3.3.1. Implementar objetos operación no concurrentes

Para ello normalmente nos creamos una derivada de `NSOperation` en la que redefinimos dos métodos: El método de inicialización y el método `main`.

Nuestra clase derivada puede tener variables de instancia y nuevos métodos. En este caso las variables de instancia deben de inicializarse implementando un método de inicialización para tal fin.

A diferencia de las operaciones concurrentes que veremos en el apartado 3.7, en el caso de las operaciones no concurrentes no debemos de redefinir `start`, sino que sólo redefinimos el método `main` para poner la operación a ejecutar. Por defecto `main` no hace nada y se limita a retornar.

Además, si la operación no concurrente que estamos implementando tarda en ejecutarse más de unos milisegundos, nuestro algoritmo `main` debe de ejecutar periódicamente `isCancelled` para comprobar si la operación se ha cancelado, y en este caso debemos retornar de `main` sin acabar la operación.

### 3.3.2. La clase `NSInvocationOperation`

Si la operación que queremos modelar como objeto operación ya está implementada en un método de algún objeto, podemos crear una instancia de `NSInvocationOperation`, que es una derivada de `NSOperation`, para apuntar a este método. En este caso inicialización el objeto con uno de estos métodos:

- `(id)initWithTarget: (id)target selector: (SEL)sel object: (id)arg`
- `(id)initWithInvocation: (NSInvocation*)inv`

El primero pide el selector y parámetro a pasar al método, mientras que el segundo nos permite usar un objeto `NSInvocation` para dar esta información.

### 3.3.3. Ejecutar objetos operación no concurrentes

En el apartado 3.1 indicamos que había dos formas de ejecutar un objeto operación: Directamente y en una cola de operación. En el primer caso ejecutamos el método `start` y, al ser una operación no concurrente, nuestro hilo se quedará bloqueado hasta que la operación acabe. En el segundo caso la cola de operación usa un hilo secundario para ejecutar el método `start`.

### 3.3.4. El entorno de ejecución

Nunca debemos de ejecutar directamente el método `main`, sino que siempre debemos ejecutar las operaciones a través del método `start`. La razón es que el método `start` configura el **entorno de ejecución**. En el entorno de ejecución se lleva la cuenta del estado en el que se encuentra la operación (preparada, ejecutando, terminada, cancelada). La implementación por defecto de los métodos `start`, `isConcurrent`, `isExecuting` e `isFinished` es válida sólo para el entorno de ejecución de operaciones no concurrentes. En el caso de las operaciones concurrentes debemos configurar el entorno de ejecución como veremos en el apartado 3.7.1.

Si `start` detecta un estado incorrecto durante su ejecución, se lanza una `NSInvalidArgumentException`. El método `start` lo primero que hace es llamar al método `isReady` para ver si la operación se puede ejecutar. Si `isReady` devuelve NO, `start` lanza una `NSInvalidArgumentException`. La razón es que nunca debemos ejecutar una operación que no esté preparada. En el siguiente apartado estudiaremos mejor este tema. Después el método `start` pone el estado en ejecutando para que `isExecuting` devuelva YES. Cuando la operación termina, será `isFinished` quien devolverá YES.

En el apartado 3.3.1 indicamos que el algoritmo de la tarea puesta en `main` debe de ejecutar periódicamente `isCancelled` para detectar si se ha pedido cancelar la operación. Si se quiere cancelar una operación (bien sea directamente o a través de una cola de operaciones) podemos ejecutar sobre el objeto de operación el método `cancel`. Esto hace que el método `isCancelled` pase a devolver `YES`, y la operación pueda saber que se ha pedido su cancelación.

### 3.4. Dependencias entre operaciones

Podemos configurar una operación para que dependa de la terminación de otra operación. Para que una operación dependa de la terminación de otra ejecutamos el método:

- `(void)addDependency: (NSOperation*)operation`

indicando en el parámetro `operation` la operación de la que depende nuestra operación.

Cuando una operación depende de la terminación de otras, la operación no podrá ejecutarse hasta que las otras hayan terminado. Para ello, el método `isReady` de una operación con dependencias devuelve `NO`.

Los objetos operación usan KVO<sup>1</sup> para que una operación pueda saber que la operación de la que depende ha terminado:

Cuando llamados a `addDependency:`, este método registra a nuestra operación como observadora KVO de la operación de la que depende. Cuando termina una operación de la que depende nuestra operación, ésta notifica a nuestra operación mediante KVO que su propiedad `isFinished` ha cambiado para que el método `isReady` de nuestra operación devuelva `YES`.

El programador puede crear tantas dependencias como considere, pero es un error de programación el crear dependencias circulares, ya que en este caso nunca se ejecutarían todas las operaciones implicadas.

Las dependencias deben de fijarse antes de ejecutar las operaciones implicadas: bien sea directamente, o bien sea mediante una cola de ejecución. No deberíamos de añadir o eliminar dependencias cuando la ejecución de las operaciones ya se ha iniciado.

---

<sup>1</sup> Key-Value Observing se explica en el tutorial "El lenguaje Objective-C para programadores C++ y Java" que está publicado también en MacProgramadores.

### 3.5. Colas de operación

En programación orientada a objetos, el principio de responsabilidad única (Single Responsibility Principle) recomienda que cada clase tenga sólo una responsabilidad. De esta forma aumentamos la cohesión de la clase. Una clase que incluye tanto la lógica de negocio como la gestión de hilos tendría dos responsabilidades. Para evitar este efecto poco recomendable las librerías de programación han incluido una clase que se encarga de gestionar la ejecución de las operaciones (sin incluir lógica de negocio). En el caso de Java tenemos el Java Executor Framework. En el caso de Cocoa tenemos las colas de operación representadas por la clase `NSOperationQueue`.

En el apartado 3.1 introdujimos las colas de operación. Éstas están representadas por la clase `NSOperationQueue` y nos permiten añadir objetos `NSOperation` para su ejecución. Las operaciones permanecen en la cola hasta que son canceladas o termina su ejecución.

Las colas de operación son objetos con un ciclo de vida largo: suelen dejarse creadas durante toda la vida de la aplicación. Además, aunque se pueden crear varias colas, normalmente sólo se crea una cola para ejecutar todas las operaciones de la aplicación.

Los objetos operación se almacenan en la cola mediante una cola de prioridad que tiene en cuenta las dependencias entre los objetos operación. Tanto las prioridades como las dependencias de los objetos operación se fijan en los propios objetos (no en la cola de operación) mediante los métodos:

- `(void)setQueuePriority:(NSOperationQueuePriority)priority`
- `(void)addDependency:(NSOperation*)operation`

Para ejecutar una operación, simplemente añadimos la operación a la cola con el método:

- `(void)addOperation:(NSOperation*)operation`

Nunca debemos de añadir una operación a más de una cola. La forma en que las colas ejecutan sus operaciones depende de si éstas son concurrentes o no. Si la operación no es concurrente (`isConcurrent` devuelve NO) la cola ejecuta su método `start` en un hilo secundario. Si por el contrario, la operación es concurrente (`isConcurrent` devuelve YES), la cola usa su hilo principal para ejecutar el método `start`, ya que en este caso se supone que `start` terminará rápidamente, y será el objeto operación quien cree un hilo si la operación es larga.

Normalmente la cola de operación ejecuta la operación nada más añadirla, pero si la cola de operación determina que el sistema tiene una carga de trabajo alta, o que la operación tiene una dependencia con otra operación

que todavía no ha terminado, la cola de operación puede dejar la operación esperando hasta que las otras operaciones terminen.

El uso de colas de operaciones tiene varias ventajas: La primera es que permiten definir y gestionar dependencias temporales entre las operaciones a ejecutar. Una segunda ventaja es que la cola de operaciones se comunica directamente con el kernel para optimizar la carga de trabajo. La tercera ventaja es que la cola de operación suele reutilizar un mismo hilo para ejecutar varias operaciones (en vez de crear un hilo para cada operación). Esto hace que las colas de operación sean especialmente útiles en servidores que tienen que lanzar gran cantidad de costosas operaciones.

Nunca deberíamos de modificar una operación una vez que ésta ha sido añadida a la cola. Lo que sí podemos es consultar su estado para saber si la operación está ejecutando o ha terminado ya su ejecución.

Si queremos cancelar una operación podemos llamar al método `cancel` del objeto operación. También podemos usar el método `cancelAllOperations` de la cola de operaciones para cancelar todas las operaciones de la cola.

También podemos suspender temporalmente la ejecución de operaciones de la cola mediante el método de la cola de operaciones:

- `(void)setSuspended:(BOOL)suspend`

Este método no afecta a las operaciones que ya están ejecutando, sólo evita que se inicie la ejecución de nuevas operaciones.

Podemos consultar las operaciones que quedan en la cola con el método:

- `(NSArray*)operations`

También podemos esperar a que acabe la ejecución de todas las operaciones de la cola con el método:

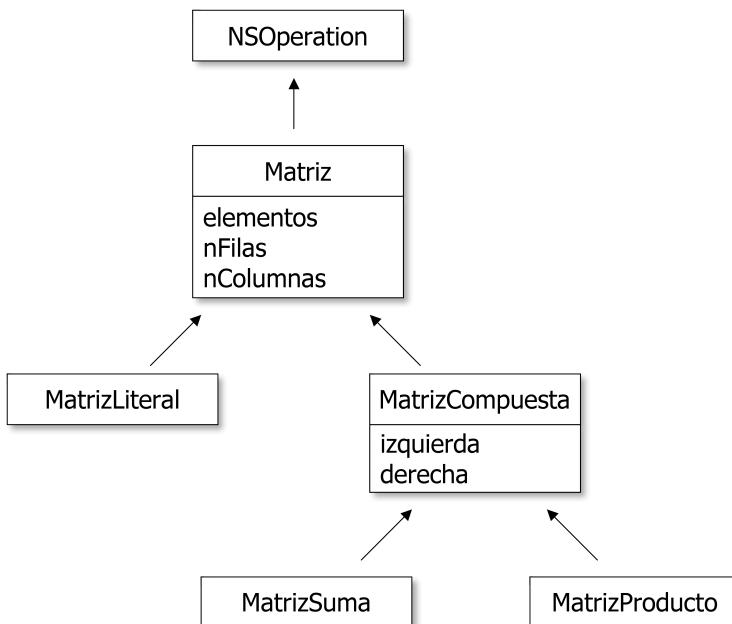
- `(void)waitForAllOperationsAreFinished`

el cual bloquea a nuestro hilo hasta que todas las operaciones de la cola han acabado.

### 3.6. Multiplicador de matrices

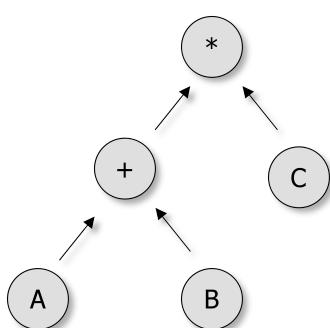
Para demostrar la funcionalidad de los objetos operación que hemos visto hasta ahora, en este apartado vamos a implementar un ejemplo. Las aplicaciones científicas son un escenario típico donde las aplicaciones pueden tener que realizar gran cantidad de cálculos. Las sumas y multiplicaciones de matrices son un tipo de operación muy usada en estas aplicaciones. Los procesa-

dores multicore permiten tener en ejecución varios hilos concurrentemente: uno en cada core. Antes de Mac OS X 10.5, si queríamos tener ejecutando cálculos en varios procesadores, era necesario gestionar la ejecución concurrente de varios hilos. Con los objetos operación podemos modelar cada operación con matrices como un objeto operación, y de esta forma encapsular la operación. Además, las colas de operación nos van a permitir gestionar de forma óptima los recursos del sistema para poder ejecutar tantas operaciones a la vez como sea óptimo para la máquina donde las ejecutemos.



**Figura 6.2:** Clases para operaciones con matrices

La Figura 6.2 muestra cómo podemos modelar estas operaciones con matrices como objetos operación. La clase `Matriz` es una clase base que representa a todos los tipos de matrices. Tiene dos derivadas, la clase `MatrizLiteral`, que representa una matriz no calculada, y la clase `MatrizCompuesta`, que es la base de las operaciones que podemos hacer con una matriz, y representa el resultado de esta operación.



**Figura 6.3:** Ejemplo de árbol de operaciones

Podemos resolver expresiones con matrices combinando las operaciones mediante un árbol de operaciones. Por ejemplo, la Figura 6.3 muestra el árbol de operaciones correspondiente a la expresión  $(A+B) * C$ .

La ventaja de representar una expresión mediante un árbol de operaciones es que permite identificar fácilmente las dependencias entre operaciones. Cuando la operación de la cima del árbol termine tendremos resuelta la expresión.

Además, si una operación falla, esta operación puede cancelar su operación (con el método `cancel`), y esto hará que se cancelen automáticamente todas las operaciones que dependen de la operación cancelada. De esta forma es fácil identificar cuando todas las operaciones han terminado correctamente o no.

El Listado 6.2 muestra el programa principal que resolvería la expresión de la Figura 6.3. La llamada a `waitForAllOperationsAreFinished` sobre la cola hace que el hilo principal espere a que la cola haya terminado de ejecutar todas las operaciones, o bien éstas se cancelen. Tenga en cuenta que el coste computacional de multiplicar matrices es del orden cúbico, lo que significa que en el caso de tener matrices bastante más grandes que las del ejemplo, la ejecución de estas operaciones puede llevar bastante tiempo. La programación multicore puede reducir estos tiempos considerablemente.

```
#import "MatrizSuma.h"
#import "MatrizProducto.h"
#import "MatrizLiteral.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // Crea la expresión  $(A+B) * C$ 
    NSInteger arrayA[3*3] = {1,3,-7
                           ,2,0,-2
                           ,3,5,4};
    Matriz* A = [[MatrizLiteral alloc] initWithDatos:arrayA
                                              nFilas:3 nColumnas:3];
    NSInteger arrayB[3*3] = {5,3,2
                           ,2,0,1
                           ,-1,0,3};
    Matriz* B = [[MatrizLiteral alloc] initWithDatos:arrayB
                                              nFilas:3 nColumnas:3];
    NSInteger arrayC[3*2] = {2,1
                           ,3,0
                           ,1,0};
    Matriz* C = [[MatrizLiteral alloc] initWithDatos:arrayC
                                              nFilas:3 nColumnas:2];
    Matriz* AB = [[MatrizSuma alloc] initIzquierda:A derecha:B];
    Matriz* ABC = [[MatrizProducto alloc]
                  initIzquierda:AB derecha:C];
    // Crea una cola de operaciones
    NSOperationQueue* cola = [NSOperationQueue new];
    [cola addOperation:A];
    [cola addOperation:B];
    [cola addOperation:C];
    [cola addOperation:AB];
```

```

[cola addOperation:ABC];
// Espera a que las operaciones acaben
[cola waitUntilAllOperationsAreFinished];
// Si se ha cancelado la última operación o alguna de las
// anteriores es que ella o alguna operación de las que
// depende ha sido cancelada
if ([ABC isCancelled])
    printf("La operación ha sido cancelada\n");
else {
    printf("A:\n%s\n", [[A description] UTF8String]);
    printf("B:\n%s\n", [[B description] UTF8String]);
    printf("C:\n%s\n", [[C description] UTF8String]);
    printf("AB:\n%s\n", [[AB description] UTF8String]);
    printf("ABC:\n%s\n", [[ABC description] UTF8String]);
}
// Libera memoria
[A release];
[B release];
[C release];
[AB release];
[ABC release];
[cola release];
[pool drain];
return 0;
}

```

**Listado 6.2:** Programa que resuelve una expresión con matrices

La clase base abstracta `Matriz` se muestra en el Listado 6.3 y Listado 6.4. El atributo `elementos` permite almacenar los elementos de longitud variable de la matriz en un objeto `NSData`. Los elementos se almacenan por filas en éste buffer lineal. Para conocer las dimensiones reales de la matriz tenemos los atributos `nFilas` y `nColumnas`.

```

#import <Foundation/Foundation.h>

@interface Matriz : NSOperation {
    const NSData* elementos;
    NSUInteger nFilas;
    NSUInteger nColumnas;
}
@property (retain) const NSData* elementos;
@property (assign) NSUInteger nFilas;
@property (assign) NSUInteger nColumnas;
- (NSUInteger)elementoAtFila:(NSUInteger)fila
                           columna:(NSUInteger)columna;
- (void)setElement:(NSUInteger)valor fila:(NSUInteger)fila
                           columna:(NSUInteger)columna;
- (NSString*) description;
- (void)dealloc;
@end

```

**Listado 6.3:** Interfaz de la clase `Matriz`

```

#import "Matriz.h"

@implementation Matriz

```

```

@synthesize elementos;
@synthesize nFilas;
@synthesize nColumnas;
- (NSInteger)elementoAtFila:(NSUInteger)fila
    columnas:(NSUInteger)columna {
    NSInteger* array = (NSInteger*) [elementos bytes];
    return array[fila*self.nColumnas+columna];
}
- (void)setElement:(NSInteger)valor fila:(NSUInteger)fila
    columnas:(NSUInteger)columna {
    NSInteger* array = (NSInteger*) [elementos bytes];
    array[fila*self.nColumnas+columna] = valor;
}
- (NSString*) description {
    NSString* text = [NSString new];
    for (int f=0;f<self.nFilas;f++) {
        for (int c=0;c<self.nColumnas;c++) {
            text = [text stringByAppendingFormat:@"%i ",
                [self elementoAtFila:f column:c]];
        }
        text = [text stringByAppendingString:@"\n"];
    }
    return text;
}
- (void)dealloc {
    if (elementos!=nil)
        [elementos release];
    [super dealloc];
}
@end

```

**Listado 6.4:** Implementación de la clase Matriz

La clase `MatrizLiteral` del Listado 6.5 y Listado 6.6 simplemente añade funcionalidad para poder inicializar la matriz a partir de un array que pasamos en `paramArray`.

```

#import "Matriz.h"

@interface MatrizLiteral : Matriz {
}
- (id)initWithDatos:(const NSInteger*)paramArray
nFilas:(NSUInteger)paramFilas nColumnas:(NSUInteger)paramColumnas;
@end

```

**Listado 6.5:** Interfaz de la clase `MatrizLiteral`

```

#import "MatrizLiteral.h"

@implementation MatrizLiteral
- (id)initWithDatos:(const NSInteger*)paramArray
    nFilas:(NSUInteger)paramFilas
    nColumnas:(NSUInteger)paramColumnas {
    if (self = [super init]) {
        self.elementos = [NSData dataWithBytes:paramArray
            length:sizeof(NSInteger)*paramFilas*paramColumnas];
        nFilas = paramFilas;
        nColumnas = paramColumnas;
    }
}

```

```

    }
    return self;
}
@end

```

**Listado 6.6:** Implementación de la clase MatrizLiteral

La clase `MatrizCompuesta` del Listado 6.7 y Listado 6.8 añade los atributos izquierda y derecha que permiten apuntar a las matrices que forman la operación a calcular.

```

#import "Matriz.h"

@interface MatrizCompuesta : Matriz {
    const Matriz* izquierda;
    const Matriz* derecha;
}
@property (retain) const Matriz* izquierda;
@property (retain) const Matriz* derecha;
- initIzquierda:(const Matriz*)paramIzquierda
             derecha:(const Matriz*)paramDerecha;
@end

```

**Listado 6.7:** Interfaz de la clase MatrizCompuesta

```

#import "MatrizCompuesta.h"

@implementation MatrizCompuesta
@synthesize izquierda;
@synthesize derecha;
- initIzquierda:(const Matriz*)paramIzquierda
             derecha:(const Matriz*)paramDerecha {
    if (self = [super init]) {
        self.izquierda = paramIzquierda;
        self.derecha = paramDerecha;
        // Añade operaciones de las que depende
        [self addDependency:paramIzquierda];
        [self addDependency:paramDerecha];
    }
    return self;
}
- (void)dealloc {
    if (izquierda!=nil)
        [izquierda release];
    if (derecha!=nil)
        [derecha release];
    [super dealloc];
}
@end

```

**Listado 6.8:** Implementación de la clase MatrizCompuesta

Por último, el cálculo de la operación lo realizan las clases derivadas `MatrizSuma` y `MatrizProducto` redefiniendo el método `main`. Las operaciones anteriores simplemente tienen el método `main` vacío. Obsérvese que las operaciones implementadas en `main` llamar al método `cancel` si detectan que

la operación no se puede realizar. En el Listado 6.2 detectamos esta condición e informamos al usuario.

```
#import "MatrizCompuesta.h"

@interface MatrizSuma : MatrizCompuesta {
}
@end
```

**Listado 6.9:** Interfaz de la clase MatrizSuma

```
#include <stdlib.h>
#import "MatrizSuma.h"

@implementation MatrizSuma
- (void)main {
    if (izquierda.nColumnas!=derecha.nColumnas
        || izquierda.nFilas != derecha.nFilas) {
        [self cancel];
        return;
    }
    // Determina filas y columnas
    nFilas = izquierda.nFilas;
    nColumnas = izquierda.nColumnas;
    // Reserva memoria
    NSUInteger array_length = izquierda.elementos.length;
    NSInteger* array = malloc(array_length);
    // Calcula la suma
    for (int i=0;i<nFilas;i++)
        for (int j=0;j<nColumnas;j++)
            array[i*self.nColumnas+j]
                = [izquierda elementoAtFila:i columna:j]
                + [derecha elementoAtFila:i columna:j];
    // Crea elementos y libera memoria
    self.elementos = [NSData dataWithBytes:array
                                    length:array_length];
    free(array);
}
@end
```

**Listado 6.10:** Implementación de la clase MatrizSuma

```
#import "MatrizCompuesta.h"

@interface MatrizProducto : MatrizCompuesta {
}
@end
```

**Listado 6.11:** Interfaz de la clase MatrizProducto

```
#import "MatrizProducto.h"

@implementation MatrizProducto
- (void)main {
    if (izquierda.nColumnas!=derecha.nFilas) {
        [self cancel];
        return;
    }
}
```

```

// Determina filas y columnas
nFilas = izquierda.nFilas;
nColumnas = derecha.nColumnas;
// Reserva memoria
NSUInteger array_length = sizeof(NSUInteger)*nFilas*nColumnas;
NSUInteger* array = malloc(array_length);
// Calcula la suma m x n x o
for (int m=0;m<nFilas;m++)
    for (int o=0;o<nColumnas;o++) {
        NSUInteger suma = 0;
        for (int n=0;n<izquierda.nColumnas;n++)
            suma += [izquierda elementoAtFila:m columna:n] *
                    [derecha elementoAtFila:n columna:o];
        array[m*self.nColumnas+o] = suma;
    }
// Crea elementos y libera memoria
self.elementos = [NSData dataWithBytes:array
length:array_length];
free(array);
}
@end

```

**Listado 6.12:** Implementación de la clase MatrizProducto

## 3.7. Operaciones concurrentes

En este apartado vamos a ver cómo implementar operaciones concurrentes, que son aquellas que quedan ejecutando una vez que la llamada al método `start` ha retornado.

### 3.7.1. Configurar el entorno de ejecución

En el apartado 3.3.4 indicamos que la clase `NSOperation` manejaba los cambios en el entorno de ejecución automáticamente cuando la operación es no concurrente, en el caso de las operaciones concurrentes debemos de redefinir los métodos `start`, `isConcurrent`, `isExecuting` e `isFinished` para informar del estado correcto de la operación. A estos cambios es a lo que se llama **configurar el entorno de ejecución**.

En concreto el método `start` se debe redefinir para lanzar un hilo, proceso o operación asíncrona. La operación concurrente redefinida puede estar implementada en el método `main`, o en cualquier otro método. En caso de estar en `main`, el método `start` redefinido sería el encargado de ejecutar a `main`, aunque a diferencia del caso no concurrente, en este caso normalmente `start` crea un hilo secundario para ejecutar `main`. Además en este caso no debemos de llamar a `start` de la clase base con `super`, ya que no queremos que el `start` por defecto de `NSOperation` configure el entorno ni ejecute el método `main`.

El método `isConcurrent` debe redefinirse para devolver `YES` indicando que la operación es concurrente. Dado que quién está ejecutando la operación es un hilo secundario, los métodos `isExecuting` e `isFinished` deben redefinirse para informar del estado de ejecución del hilo secundario.

### 3.7.2. Compatibilidad KVO

La implementación por defecto del entorno de ejecución de la clase `NSOperation` realiza notificaciones KVO para las propiedades: `isCancelled`, `isConcurrent`, `isExecuting`, `isFinished`, `isReady`, `dependencies` y `queuePriority`.

De esta forma objetos que se registren como observadores del objeto operación podrán saber cuándo cambian estas propiedades. Por ejemplo, la cola de operación se registra como observadora de `isReady` para saber cuándo pueden lanzar una operación, o como observadora de `isCancelled` e `isFinished` para saber cuándo una operación ha sido cancelada o a terminado, y así poder eliminar a la operación de la cola de operaciones.

En caso de que configuremos el entorno de ejecución debemos de mantener compatibilidad KVO para que se realicen las notificaciones KVO correspondientes a los objetos que estén registrados como observadores de estas propiedades.

## 3.8. Métodos sincronizados

Los métodos de `NSOperation` y `NSQueueOperation` son métodos sincronizados sobre `self`, lo cual permite que una misma instancia sea accedida por varios hilos de forma segura y sin necesidad de mecanismos de sincronización adicionales. Cuando redefinimos los métodos de `NSOperation` debemos de mantener el sincronismo envolviendo su implementación en un bloque `@synchronized` sobre `self`.

## 3.9. Responder a errores

Si una operación puede fallar, es responsabilidad del programador de aplicaciones detectar esta condición y generar el correspondiente tratamiento al error.

La implementación por defecto del entorno de ejecución (la que usan las operaciones no concurrentes) está implementada de forma que `start` llama a `main` y captura cualquier excepción que se produzca en `main`. Esto hace que pueda parecer que la operación ha terminado correctamente cuando en reali-

zada no ha sido así. Para solucionarlo, se recomienda implementar un bloque `@try-@catch` en la función `main` y procesar el error.

Además, si un objeto operación depende de otros objetos operación que pueden fallar, es responsabilidad del programador de aplicaciones el detectar que las operaciones de las que depende nuestra operación han fallado y abortar la operación. Para ello muchas veces se redefine el método `isReady` para comprobar que las operaciones de las que dependemos han terminado correctamente. Hay que tener en cuenta que la operación `isReady` por defecto sólo comprueba que han terminado las operaciones de las que dependemos, independientemente de si han terminado bien o mal. Para redefinir `isReady` primero podemos comprobar que las operaciones de las que dependemos hayan terminado correctamente y después llamar a `isReady` de la clase base con `super`.

# Tema 7

## Técnicas de comunicación y sincronización

---

### **Sinopsis:**

*Existe una gran cantidad de técnicas de comunicación y sincronización entre hilos. Algunas son más apropiadas para interactuar entre hilos del mismo proceso y otras para interactuar entre hilos de distintos procesos. Este tema estudia las técnicas de sincronización más usadas, así como algunas técnicas de comunicación clásicas. Los siguientes temas estudian otras técnicas más avanzadas que también permiten comunicar y sincronizar hilos.*

## 1. Comunicación y sincronización

---

En el apartado 1.2 del Tema 6 ya introdujimos las **técnicas de sincronización** y las dividimos en técnicas de sincronización entre hilos y técnicas de sincronización entre procesos. Por su parte las **técnicas de comunicación** permiten pasar información entre hilos del mismo o de distintos procesos. A diferencia de las técnicas de sincronización, las técnicas de comunicación normalmente sólo se usan para pasar información entre hilos de distintos procesos, ya que el intercambio de información entre hilos del mismo proceso es mucho más sencillo: podemos usar variables globales donde depositar la información compartida.

En los siguientes apartados vamos a estudiar las técnicas de sincronización entre hilos del mismo proceso más básicas con que cuenta Mac OS X: cerrojos, barreras de memoria, operaciones atómicas, condiciones. Después veremos algunas técnicas que permiten comunicar hilos de distintos procesos: pipes y memoria compartida. El Tema 8, Tema 9 y Tema 10 tratan otras técnicas más avanzadas de comunicación y sincronización entre hilos y entre procesos.

En el apartado 1.5 y apartado 1.6 del Tema 6 ya indicamos que la programación multihilo tiene ventajas, pero también inconvenientes. Si hemos decidido usar programación multihilo, también conviene saber que las técnicas de sincronización evitan problemas de sincronización, pero también introducen problemas. Aparte de complicar la lógica del programa, las técnicas de sincronización puede degradar el rendimiento de nuestra aplicación, ya que pueden dejar a hilos bloqueados a la espera de que otros terminen. Para evitar estos problemas, lo mejor es intentar evitar el tener que recurrir a técnicas de sincronización reduciendo al mínimo la interacción entre hilos. Si los hilos ejecutan de forma independiente, no será necesario utilizar técnicas de sincronización entre ellos. Como en la práctica, cuando una aplicación tiene varios hilos, al final tiene que haber un protocolo de comunicación entre los hilos, lo que se recomienda es intentar minimizar este protocolo.

## 2. Los cerrojos

---

Los **cerrojos** (locks) son la forma de sincronización entre hilos más usada. Los cerrojos permiten proteger secciones críticas, que son segmentos de código que sólo pueden ser ejecutados por un hilo a la vez. Una vez que un hilo cierra un cerrojo ningún otro hilo podrá entrar en la sección crítica protegida por el cerrojo hasta que el hilo libere el cerrojo.

Tanto POSIX como Cocoa proporcionan varios tipos de cerrojos. A un cerrojo también se le llama **mutex** (mutual exclusion). De hecho, POSIX se suele

referir a ellos con el nombre mutex, mientras que Cocoa los llama locks. En los siguientes apartados vamos a describir estos tipos.

## 2.1. Cerrojos POSIX

POSIX representa los cerrojos con el tipo `pthread_mutex_t`. Para inicializar el mutex debemos pasarlo a la función:

```
int pthread_mutex_init(pthread_mutex_t* restrict mutex,
                      const pthread_mutexattr_t* restrict attr);
```

Una vez inicializado, un hilo puede cerrar el cerrojo con las funciones:

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_trylock(pthread_mutex_t* mutex);
```

La primera función bloquea al hilo que la llama hasta que el mutex esté libre. La segunda función no bloquea al hilo si el mutex está cogido, sino que retorna el código de error `EBUSY`.

Para abrir un mutex tenemos la función:

```
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

Por último podemos liberar los recursos asociados a un mutex con:

```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

El siguiente trozo de código muestra cómo usar un mutex POSIX:

```
pthread_mutex_t mutex;
void InicializaMutex() {
    pthread_mutex_init(&mutex, NULL);
}
void OperacionCritica() {
    pthread_mutex_lock(&mutex);
    // Sección crítica
    pthread_mutex_unlock(&mutex);
}
```

## 2.2. Cerrojos Cocoa

El tipo de cerrojo más común en Cocoa está representado por la clase `NSLock`. Esta clase envuelve un mutex POSIX, y proporciona las mismas operaciones:

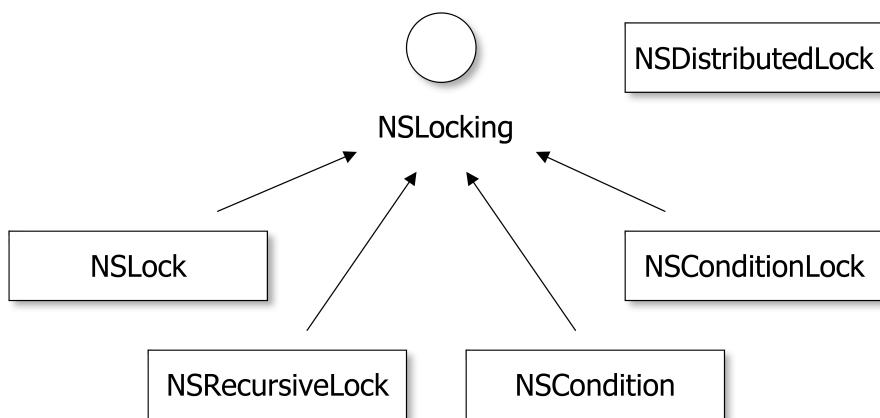
- `(void) lock`

- `(void)unlock`
- `(BOOL)tryLock`

Como muestra la Figura 7.1, Cocoa proporciona una familia más amplia de cerrojos, cuya funcionalidad común está recogida en el protocolo `NSLocking`.

Uno de los problemas más típicos de usar objetos de la clase `NSLock` es que este tipo de objetos sólo se pueden adquirir una vez. Si el hilo que lo adquirió intenta adquirirlo por segunda vez, éste quedaría bloqueado. Además, se produce un deadlock debido a que el hilo que posee el hilo ha quedado bloqueado, con lo que ni el mismo hilo ni ningún otro hilo podrá liberarlo.

Para solucionar este problema tenemos la clase `NSRecursiveLock`, la cual es similar a `NSLock`, pero permite que un mismo hilo ejecute varias veces la operación `lock` o `tryLock` sobre un cerrojo sin quedar bloqueado. Lógicamente, cualquier otro hilo sí que quedaría bloqueado al intentar adquirir un cerrojo bloqueado. Para que el cerrojo quede liberado el hilo deberá ejecutar la operación `unlock` tantas veces como ejecute `lock`.



**Figura 7.1:** Tipos de cerrojos Cocoa

Dos escenarios donde se usa a menudo `NSRecursiveLock` son:

1. Las funciones recursivas donde la función podría adquirir el cerrojo en cada llamada recursiva y liberarlo al retornar de cada llamada.
2. Los cerrojos a nivel de objeto. Es común que un cerrojo se almacene como variable de instancia de un objeto y que los distintos métodos de este objeto deban adquirir el cerrojo antes de acceder a otra variable de instancia. Además es común que los distintos métodos del objeto se llamen entre sí. En este caso el cerrojo se está usando por distintos métodos, pero se garantiza que sólo haya un hilo trabajando con el objeto.

Es importante tener en cuenta que si usamos cerrojos recursivos, una vez que un hilo bloquea un objeto, éste queda bloqueado para los demás hilos hasta que las distintas llamadas recursivas acaban y el hilo termina de trabajar con el objeto. En consecuencia el uso de cerrojos recursivos puede implicar que

otros hilos queden bloqueados bastante tiempo, con lo que los cerrojos recursivos se deben de usar con cautela, es decir, conviene pensar cuando sería mejor usar un cerrojo no recursivo.

La clase `NSDistributedLock` implementa un cerrojo entre hilos de distintos procesos. La forma de implementarlo es en base a un fichero, con lo que deberemos de tener creado un fichero para todas las aplicaciones que van a compartir el cerrojo. La forma de crear el objeto es usando el método factory:

```
+ (NSDistributedLock*) lockWithPath: (NSString*) aPath
```

Donde `aPath` es el path de un fichero existente y sobre el que deberemos tener permiso de escritura.

A diferencia de los demás tipos de cerrojos (ver Figura 7.1) la clase `NSDistributedLock` no adopta el protocolo `NSLocking` y no implementa el método `lock`. La razón que alega Apple para este diseño es que para implementar `lock` sobre un fichero es necesario sondear continuamente el fichero con el consiguiente consumo de recursos<sup>1</sup>. Como alternativa proponen usar el método `tryLock`, el cual sí que existe en `NSDistributedLock`. De esta forma el programador de aplicaciones puede decidir cuándo sondear el fichero. Al igual que con los demás tipos de cerrojos, para liberarlo se usa el método `unlock`.

Debido a que `NSDistributedLock` está implementado en base a un fichero del sistema de ficheros, si la aplicación no libera el fichero asociado a un bloqueo (p.e. porque explota), el correspondiente fichero quedará siempre retenido, y las demás aplicaciones no podrán acceder a él. En este caso las aplicaciones pueden usar el método `breakLock` para liberar el cerrojo sobre el fichero.

## 2.3. Los bloques sincronizados

La directiva del compilador `@synchronized()` permite usar el propio lenguaje Objective-C para crear un mutex sobre una sección crítica. La directiva recibe como parámetro un objeto Objective-C que actúa como mutex de la forma:

```
@synchronized(obj) {
    // Sección crítica
}
```

---

<sup>1</sup> Este argumento deja de tener fundamento en Mac OS X 10.5 ya que se pueden usar los eventos del sistema de ficheros para detectar los cambios en el fichero. Hemos reportado esta circunstancia a Apple para proponer que implementen el método `lock` en la clase `NSDistributedLock`.

En el tutorial "El lenguaje Objective-C para programadores C++ y Java" se describe con más detalle el uso de esta directiva del compilador.

## 2.4. Secciones críticas en colecciones

Cuando varios hilos usan una colección (array, conjunto, diccionario, ...) es muy fácil encontrarnos con race-conditions. El siguiente trozo de programa muestra una situación en la que podría producirse este tipo de errores:

```
NSLock* cerrojoDiccionario; // Asumimos que está creado
NSDictionary* diccionario; // Asumimos que está creado

void FuncionCritica() {
    NSString* nombre;

    // Sección crítica
    [cerrojoDiccionario lock];
    nombre = [diccionario objectForKey:@"nombre"];
    [cerrojoDiccionario unlock];

    printf("El nombre es %s\n", [nombre UTF8String]);
}
```

El problema se debe a que las colecciones de Objective-C envían el mensaje `release` a los objetos cuando los sacan de la colección. La race-condition se produciría si nuestro hilo es interrumpido justo después de salir de la sección crítica, pero antes de imprimir el `nombre`. Si en este momento otro hilo cambiase la entrada `@"nombre"` del `diccionario`, el diccionario enviaría el mensaje `release` al objeto `nombre`, y si la cuenta de referencias de `nombre` alcanzase cero, su memoria se liberaría y se produciría un acceso inválido a memoria cuando éste fuera a ser accedido para imprimirlo.

Creemos que hubiera sido una buena idea el que las colecciones enviaran el mensaje `autorelease` (en vez de `release`) a los objetos que liberan. Ya que esto no se hizo así en su día<sup>1</sup>, lo mejor que podemos hacer es proteger con la sección crítica tanto la colección como los accesos a sus elementos, o bien ejecutar `retain` y `autorelease` sobre los objetos obtenidos de la colección.

---

<sup>1</sup> Hemos enviado un bug report a Apple para sugerirlo pero nos han respondido que cambiar `release` por `autorelease` en las colecciones podría implicar un importante aumento en el consumo de memoria.

### 3. Barreras de memoria y variables volátiles

Si un procesador consta de varios cores, podemos tener varios hilos ejecutando a la vez en distintos cores. Además, si el procesador es segmentado (la mayoría de los actuales lo son) se pueden estar ejecutando más de una instrucción a la vez dentro del mismo core, e incluso se pueden estar ejecutando fuera de orden. Los compiladores actuales también reordenan las instrucciones para optimizar su ejecución simultánea cuando el compilador piensa que es posible su ejecución fuera de orden. Cuando una aplicación consta de un sólo hilo, esta ejecución fuera de orden no introduce ningún riesgo y mejora el rendimiento de las aplicaciones. Sin embargo, cuando la aplicación consta de más de un hilo, los compiladores y procesadores no son capaces de detectar que el orden en que ha pedido ejecutan las instrucciones de acceso a memoria el generador de código del mismo compilador puede dar lugar a resultados incorrectos.

Las **barreras de memoria** son un tipo de sincronización no bloqueante que asegura que los accesos a memoria se producen en el orden correcto. Cuando colocamos una barrera de memoria estamos pidiendo al procesador y al compilador que ejecute completamente todas las instrucciones de acceso a memoria que están delante de la barrera antes de empezar a ejecutar las instrucciones de acceso a memoria que están detrás de la barrera. Ejecute `man barrier` para conocer más sobre las barreras de memoria de Mac OS X. Para colocar una barrera de memoria en Mac OS X, simplemente tenemos que llamar a la función:

```
void OSMemoryBarrier(void);
```

Los cerrojos incluyen barreras de memoria para garantizar que las instrucciones de acceso a memoria anteriores a cerrojo han terminado, con lo que no es necesario colocar una llamada a `OSMemoryBarrier()` antes de entrar en una sección crítica.

Otro tipo de restricción sobre la memoria son las **variables volátiles**. Los compiladores suelen reducir los accesos a memoria manteniendo las variables en registros del procesador. Esto no da ningún problema cuando la aplicación consta de un sólo hilo. Sin embargo, si la aplicación tiene varios hilos y una variable se mantiene en registro (en vez de guardarse en memoria), puede que otro hilo no se entere de que la variable ha cambiado. Para evitarlo se marca a este tipo de variables con el modificador `volatile`, el cual pide al compilador que no deje la variable en registro, sino que siempre guarde la variable en memoria. Es buena idea el marcar como `volatile` a las variables que son compartidas por los hilos que acceden a una sección crítica.

## 4. Operaciones atómicas

---

Adquirir un cerrojo tiene un coste que no debe despreciarse. Muchas veces el cerrojo sólo encierra una sección crítica que actualiza una variable fundamental. En estos casos resulta mucho más eficiente hacer uso de operaciones atómicas las cuales corresponden directamente con operaciones ensamblador que garantizan que una variable de tipo fundamental será actualizada de forma transaccional. Es decir, estas operaciones garantizan que nuestro hilo nunca será interrumpido cuando esté ejecutando una operación atómica, con lo que los demás hilos (que estén ejecutando en el mismo core o en otro core) verán el cambio en la variable o bien terminado, o bien no lo verán en absoluto.

Debido a que estas operaciones quitan temporalmente al sistema operativo la posibilidad de interrumpir un hilo, las operaciones sólo están disponibles para tipos fundamentales. Si queremos crear secciones críticas más complejas que actualicen estructuras, deberemos de usar cerrojos.

Estas operaciones están declaradas en el fichero de cabecera `OSAtomic.h`. Para consultarlas podemos ejecutar `man atomic`. Entre otras, encontramos las típicas operaciones atómicas sobre variables de compare-and-swap, test-and-set y test-and-clear. Por ejemplo, tenemos la operación:

```
int32_t OSAtomicIncrement32(  
    volatile int32_t *theValue);
```

Que incrementa una variable de tipo `int32_t`. Todas estas operaciones atómicas incluyen una variante con barrera de memoria. Por ejemplo:

```
int32_t OSAtomicIncrement32Barrier(  
    volatile int32_t *theValue);
```

La variante con barrera de memoria ejecuta una barrera de memoria antes de realizar la operación atómica. Esto garantiza que otros hilos terminen sus accesos a memoria antes de realizar la actualización atómica. En general, cuando sólo un hilo modifica la variable atómica (y los demás la leen), no es necesario usar la versión con barrera de memoria, pero si más de un hilo puede actualizar la variable atómica, deberíamos usar la versión con barrera de memoria.

## 5. Cerrojos de sondeo

---

Los **cerrojos de sondeo** (spin lock) son cerrojos que no bloquean al hilo, sino que lo dejan ejecutando constantemente un bucle de sondeo hasta que se abre el cerrojo. Este tipo de cerrojos se usa cuando la posibilidad de encontrar el cerrojo cerrado es pequeña y en caso de encontrárnoslo cerrado, el tiempo que tardaría en abrirse también es pequeño. Bajo estas condiciones sería más costoso bloquear al hilo que esperar a que el cerrojo se abra. Para obtener más información sobre este tipo de cerrojos ejecute `man spinlock`.

Las funciones:

```
void OSSpinLockLock(OSSpinLock* lock);  
bool OSSpinLockTry(OSSpinLock* lock);
```

Sirven respectivamente para dejar al hilo sondeando hasta que el cerrojo esté abierto y para preguntar si el cerrojo está abierto. Una vez que retornamos de las funciones sabemos que tenemos cerrado el cerrojo para ejecutar su sección crítica.

Para liberar el cerrojo de sondeo tenemos la función:

```
void OSSpinLockUnlock(OSSpinLock* lock);
```

## 6. Condiciones

---

Los cerrojos permiten que no haya más de un hilo accediendo a unos recursos compartidos, pero en ocasiones este acceso exclusivo a un recurso no es suficiente. ¿Qué ocurre si un hilo desea que una determinada condición sea cierta cuando entre en su sección crítica?

Por ejemplo, imaginemos el clásico escenario de productor-consumidor donde el consumidor ha entrado dentro de la sección crítica y resulta que no hay datos para consumir. Lógicamente quedarse sondeando la condición no sería buena idea ya que se consume recursos. Además, si el hilo se queda dentro de la sección crítica ningún otro hilo podría actualizar estos recursos. En este caso, posiblemente lo mejor que puede hacer el hilo es abandonar la sección crítica y volver más tarde. Pero en este caso, ¿qué hace el hilo consumidor, se duerme y vuelve dentro de tres segundos a ver si el consumidor ha depositado datos? Lógicamente lo óptimo sería volver nada más empezara a ser cierta esta condición.

Las **condiciones** permiten dejar dormido a un hilo (típicamente el consumidor) hasta que otro hilo (típicamente el productor) le despierta. Normalmente le despierta para informarle de que la condición se ha cumplido.

Microsoft Windows llama **eventos de sincronización** a las condiciones. La principal diferencia entre los eventos de sincronización y las condiciones es que los eventos de sincronización son objetos independientes de los cerrojos, mientras que las condiciones son objetos que siempre deben de tener asociado un cerrojo y operar dentro de una sección crítica protegida por ese cerrojo.

Tanto POSIX como Cocoa implementan condiciones. De hecho, las condiciones Cocoa son objetos que envuelven condiciones POSIX en clases que simplifican su uso. En los siguientes apartados estudiaremos tanto las condiciones POSIX como las condiciones Cocoa.

### 6.1. Condiciones POSIX

Cada condición POSIX debe tener asociado un mutex y un predicado. Tanto productor como consumidor deben de usar el mismo par mutex-condición. El **predicado** es una expresión booleana que el productor informa al consumidor que ha pasado a ser cierta y que el consumidor debe comprobar antes de realizar su tarea.

El Listado 7.1 muestra un ejemplo de productor-consumidor que usa una condición y un mutex declarados como variables globales. En nuestro ejemplo

el predicado es `contador>=1000`. Este es el predicado que debe cumplirse para que el consumidor consuma.

Las tareas de la función `main()` son:

1. Inicializar la condición, el mutex y el predicado
2. Crear los hilos productor y consumidor
3. Esperar a que acaben los hilos. Recuérdese que por defecto POSIX crea joinable threads.

Por su parte el consumidor:

1. Cierra el mutex para evitar que el productor modifique las variables compartidas.
2. Espera a que se cumpla el predicado. En caso de no cumplirse el predicado se pone a esperar (al ejecutar la operación de espera el mutex se libera para que el productor pueda modificar el contenido protegido por el mutex). Cuando la operación de espera acaba se vuelve a bloquear el mutex.
3. Consume el contenido. Para ello es importante que el mutex esté cogido por el consumidor.
4. Libera el mutex.

En caso de que no se cumpla el predicado, el consumidor no debe sondear el predicado, sino ponerse a dormir hasta que le despierte el productor. Para ello llama a la función:

```
int pthread_cond_wait(pthread_cond_t* restrict cond,
                      pthread_mutex_t* restrict mutex);
```

Esta función recibe tanto la condición como el mutex asociado. Es importante tener en cuenta que esta función:

- Al entrar en ella se libera el mutex con el fin de que el productor pueda producir contenido.
- Se vuelve a apoderar del mutex al salir con el fin de que el consumidor pueda alterar las variables de la sección crítica de forma segura. Esta es la razón de que el consumidor vuelve a liberar el mutex en el paso 4.

```
#include <stdio.h>
#include <pthread.h>

static pthread_mutex_t mutex;
static pthread_cond_t condicion;
static int contador=0;

void* consumidor(void* p) {
    while(true) {
        // Cierra el mutex
        pthread_mutex_lock(&mutex);
```

```

// Espera a que se cumpla el predicado
while (contador<1000) {
    pthread_cond_wait(&condicion,&mutex)
}
// Consumir contenido
printf("Consumidos %i elementos\n",contador);
contador = 0;
// Libera el mutex
pthread_mutex_unlock(&mutex);
}

void* productor(void* p) {
    while(true) {
        // Produce contenido
        int n_elementos = (int)(200.0*rand()/RAND_MAX);
        for (int i=0;i<n_elementos;i++) {
            pthread_mutex_lock(&mutex);
            contador++;
            pthread_mutex_unlock(&mutex);
        }
        // Si se cumple el predicado activa la condición
        if (contador>=1000)
            pthread_cond_signal(&condicion);
    }
}

int main (int argc, const char * argv[]) {
    // Inicializa el mutex y su condición
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&condicion, NULL);
    // Lanza los hilos
    pthread_t hilo_consumidor;
    pthread_t hilo_productor;
    pthread_create(&hilo_consumidor, NULL,consumidor,NULL);
    pthread_create(&hilo_productor, NULL,productor,NULL);

    // Espera a la terminación de los hilos
    pthread_join(hilo_consumidor, NULL);
    pthread_join(hilo_productor, NULL);
    printf("Acaba el hilo main\n");
    return 0;
}

```

**Listado 7.1:** Ejemplo de productor-consumidor

Por su parte el productor:

1. Debe modificar las variables compartidas con el mutex cogido con el fin de que el consumidor no acceda a estas variables cuando el productor las esté actualizando.
2. Cuando se cumpla el predicado activa la condición para despertar al consumidor.

Para activar la condición se usa la función:

```
int pthread_cond_signal(pthread_cond_t* cond);
```

Para activar la condición no es necesario que el productor tenga el mutex cerrado (aunque puede tenerlo). Lo que sí que sabemos es que el consumidor no saldrá de la función `pthread_cond_wait()` hasta que esta función consiga bloquear el mutex para el consumidor. En este momento el consumidor podrá modificar las variables compartidas de forma segura.

Normalmente, sólo existe un consumidor esperando en una condición, pero es también posible que haya varios consumidores esperando en una condición. En este caso varios hilos podrían lanzarse con la función `consumidor()` del Listado 7.1. Cada vez que un hilo consumidor ejecute la función `pthread_cond_wait()`, éste liberaría el mutex y se pondría a esperar a que se le despierte.

El productor puede elegir entre despertar sólo a un hilo de los que estén esperando, para lo cual ejecuta:

```
int pthread_cond_signal(pthread_cond_t* cond);
```

O bien despertar a todos los hilos que estén esperando, para lo cual ejecuta:

```
int pthread_cond_broadcast(pthread_cond_t* cond);
```

En este último caso no se despertarían todos los consumidores a la vez, ya que para que un consumidor salga de la llamada a `pthread_cond_wait()` debe de tener el mutex, sino que se irían despertando uno a uno, hasta que se despierten todos los consumidores.

## 6.2. Condiciones Cocoa

Las condiciones Cocoa están representadas por la clase `NSCondition`, la cual envuelve un mutex y una condición POSIX. La clase `NSCondition` adopta el protocolo `NSLocking` (ver Figura 7.1), con lo que proporciona los siguientes métodos para cerrar y abrir el mutex:

- `(void)lock`
- `(void)unlock`

Para esperar a la condición tenemos el método:

- `(void)wait`

Y para activar la condición tenemos los métodos:

- `(void)signal`
- `(void)broadcast`

Existe una clase llamada `NSConditionLock` que encierra un mutex POSIX, una condición POSIX y un predicado de tipo entero. Es decir, es un `NSCondition` que, además de un mutex y condición POSIX, encierra un predicado. El predicado se proporciona al objeto cuando se inicializa:

- `(id)initWithCondition:(NSInteger)condition`

El nombre del parámetro `condition` no debe llevar a engaño, ya que realmente se trata de un predicado, y no debe confundirse con la condición POSIX que también encierra el objeto (aunque como explicamos más abajo, la condición POSIX no es accesible desde los métodos de `NSConditionLock`). Para evitar confusiones usaremos el término *condición* para referirnos al predicado de tipo entero y el término *condición POSIX* para referirnos al objeto de sincronización que encierra el objeto.

Se puede usar la condición cuando se va a cerrar el cerrojo:

- `(void)lockWhenCondition:(NSInteger)condition`

En este caso el hilo queda bloqueado hasta que el cerrojo esté abierto y además la condición del objeto tenga el valor pasado como parámetro.

Una vez inicializado el objeto, la única forma de modificar el valor de su condición es usando el método:

- `(void)unlockWithCondition:(NSInteger)condition`

El cual fija el valor de la condición al valor pasado como parámetro y desbloquea el mutex.

La condición se puede usar para comunicarse entre un hilo productor y un hilo consumidor. Para ello podemos inicializar la condición con el valor `SIN_DATOS` y el productor la puede poner a un valor `HAY_DATOS` cuando haya depositado datos en las variables de la sección crítica. El consumidor por su parte puede ejecutar `lockWhenCondition:` con `HAY_DATOS` para quedar bloqueado hasta que se abra el cerrojo y la condición tome el valor `HAY_DATOS`.

La clase `NSConditionLock` (a diferencia de `NSCondition`) no dispone de los métodos `wait` y `signal`. Esto se debe a que la condición POSIX es un valor interno del objeto que se usa en `lockWhenCondition:` para dejar al hilo dormido hasta que se cumpla la condición. Además, la clase `NSConditionLock` no proporciona métodos para acceder a la condición POSIX subyacente.

Dado que `NSConditionLock` implementa `NSLocking`, también dispone del cierre y apertura incondicional del mutex con los métodos `lock` y `unlock`.

## 7. Pipes

---

Los pipes son canales de comunicación unidireccionales usados ampliamente para comunicación entre hilos de distintos procesos, aunque también se pueden usar para comunicar dos hilos del mismo proceso. Al ser canales unidireccionales, un hilo lee datos del pipe y otro hilo escribe datos en el pipe.

Existen dos librerías para programar con pipes: Los **pipes BSD** y los **pipes Cocoa**. Los pipes Cocoa se basan en las librerías de pipes BSD para realizar su trabajo. En los siguientes apartados vamos a estudiar estos tipos de pipes.

### 7.1. Pipes BSD

BSD proporciona dos tipos de pipes, los pipes con nombre y los pipes sin nombre. Los **pipes con nombre** son un tipo de ficheros que se crean con el comando `mkfifo`, y que después se abren como si fueran ficheros con la siguiente función. El entero devuelto representa el pipe.

```
int open(const char *path, int oflag, ...);
```

En `oflag` debemos indicar `O_RDONLY` o `O_WRONLY` dependiendo de si queremos abrir el pipe para lectura o para escritura,

Si queremos abrir un **pipe sin nombre** usamos la función:

```
int pipe(int fildes[2]);
```

La función devuelve en `fildes` dos enteros, uno es el pipe de lectura y otro el de escritura. En este caso se suele crear un nuevo proceso y pasarle uno de los pipes para comunicarnos con él.

Podemos obtener información sobre los pipes (con o sin nombre) existentes con el comando `netstat -f unix`.

Además tenemos los pipes `stdin`, `stdout` y `stderr` que siempre están abiertos.

Para leer y escribir en los pipes se usan las funciones:

```
ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Para cerrar un pipe se usa la función:

```
int close(int fildes);
```

Cuando un pipe se cierra, el otro se queda **viudo** (widowed pipe). Si intentamos escribir en un pipe viudo se produce la señal `SIGPIPE`. Cerrar el pipe de escritura es la forma de indicar el final de stream al pipe de lectura. Cuando el pipe de lectura ha consumido todos los datos del buffer la función `read()` devolverá 0 como número de bytes leídos. En ese momento podemos cerrar el pipe de lectura.

## 7.2. Pipes Cocoa

Los objetos `NSPipe` representan un pipe sin nombre. Para crear un pipe sin nombre disponemos del método de factory:

```
+ (id)pipe
```

La clase `NSPipe` dispone de los métodos:

- `(NSFileHandle*) fileHandleForReading`
- `(NSFileHandle*) fileHandleForWriting`

que dan acceso respectivamente al punto de lectura y escritura en el pipe. Si vamos a comunicar hilos, esto es todo lo que necesitamos.

Podemos comunicarnos con otro proceso de dos formas. La primera forma es acceder a nuestra entrada y salida estándar con los métodos `fileHandleWithStandardInput` y `fileHandleWithStandardOutput` que vimos en el apartado 2.1 del Tema 1.

La segunda forma es lanzar nosotros el proceso (representado por la clase `NSTask`) con el que nos queremos comunicar. Antes de lanzar un proceso podemos modificar su entrada estándar, salida estándar y salida de errores estándar. Para modificarlas usamos los métodos del objeto `NSTask`:

- `(void)setStandardInput:(id)file`
- `(void)setStandardOutput:(id)file`
- `(void)setStandardError:(id)file`

Estos métodos pueden recibir en el parámetro `file` tanto un objeto `NSFileHandle` que representa el fichero a leer como un `NSPipe`. Cuando estos métodos reciben un objeto `NSFileHandle` lo único que pueden hacer es leer o escribir en el fichero. Cuando reciben un objeto `NSPipe` la operación se vuelve más interesante, ya que el proceso puede comunicarse con otro proceso a través del pipe sin nombre.

En caso de que `setStandardInput:` reciba un objeto `NSPipe`, nosotros podremos usar este pipe para escribir, y el nuevo proceso podrá usar este pipe para leer. En caso de que `setStandardOutput:` o `setStandardError:`

reciban un objeto pipe, nosotros podremos usar este pipe para leer y el nuevo proceso podrá usar este pipe para escribir.

El Listado 7.2 muestra una modificación del Listado 5.1 para lanzar el comando `grep`, y en vez de que su salida vaya a la salida estándar, recoger esta salida mediante un pipe.

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSTask* t = [[NSTask alloc] init];
    NSArray* args
        arrayWithObjects:@"hola",@"carta.txt",nil];
    [t setArguments:args];
    [t setLaunchPath:@"/usr/bin/grep"];
    [t setCurrentDirectoryPath:NSUTFHomeDirectory()];
    // Fija un pipe para leer la salida del proceso
    NSPipe* pipe = [NSPipe pipe];
    [t setStandardOutput:pipe];
    [t launch];
    [t waitUntilExit];
    int status = [t terminationStatus];
    if (status==0) {
        NSData* buffer;
        NSFileHandle* handle_lectura = [pipe fileHandleForReading];
        buffer = [handle_lectura availableData];
        NSString* texto = [[NSString alloc] initWithData:buffer
                                                encoding:NSUTF8StringEncoding];
        printf("%s\n",[texto UTF8String]);
    }
    else
        fprintf(stderr,"No se ha encontrado \"hola\" en carta.txt\n");
    [t release];
    [pool drain];
    return 0;
}
```

**Listado 7.2:** Programa que ejecuta un proceso con un pipe

## 8. Memoria compartida

Los hilos de un mismo proceso siempre comparten memoria. La **memoria compartida** es una forma de comunicación entre hilos de distintos procesos, de forma que existe un área de memoria que está mapeada en la memoria de ambos procesos (aunque posiblemente en distintas direcciones de memoria).

Para crear la memoria compartida se usan las páginas de memoria, con lo que normalmente el tamaño de la memoria compartida será un múltiplo del tamaño de página.

El tamaño de página en Mac OS X 10.5 es de 4096 bytes. Podemos usar el comando `vm_stat` para conocer este tamaño.

Mac OS X implementa tanto las primitivas de comparición de memoria de POSIX como las de System V. Apple recomienda usar las primitivas de POSIX por ser más nuevas y flexibles. La única razón para usar las primitivas de System V es que nuestra aplicación deba compilar en sistemas antiguos que no soporten POSIX. En este documento vamos a estudiar sólo las primitivas de POSIX.

El Listado 7.3 muestra un ejemplo con las funciones que intervienen en la creación y uso de memoria compartida.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

int main (int argc, const char * argv[]) {

    // Crea un descriptor de fichero para la memoria compartida
    int fd = shm_open("compartida",O_RDWR|O_CREAT,S_IRUSR|S_IWUSR);
    if (fd == -1) {
        perror("Fallo apertura");
        return EXIT_FAILURE;
    }

    if ( ftruncate( fd, 4046 ) == -1 ) {
        perror("Fallo truncado");
        return EXIT_FAILURE;
    }

    // Mapea el descriptor de fichero en memoria
    char* p = mmap(NULL,4046,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
    if (p == MAP_FAILED) {
        perror("Fallo mapeo");
        return EXIT_FAILURE;
    }
}
```

```

// Escribe en la memoria compartida
strcpy(p,"¿Hay alguien ahi?");

// Desmapea el fichero
if (munmap(p,4096) !=0) {
    perror("Fallo desmapeo");
    return EXIT_FAILURE;
}

// Cierra el descriptor de fichero de la memoria compartida
if (close(fd)!=0) {
    perror("Fallo al cerrar el descriptor de fichero");
    return EXIT_FAILURE;
}

// Libera el recurso memoria compartida
if (shm_unlink("compartida")!=0) {
    perror("Fallo al cerrar memoria compartida");
    return EXIT_FAILURE;
}

return EXIT_SUCCESS;
}

```

**Listado 7.3:** Ejemplo de memoria compartida

Para crear una zona de memoria compartida empezamos usando la función:

```
int shm_open(const char* name, int oflag, ...);
```

Donde `name` es un nombre del **recurso memoria compartida** de hasta 31 caracteres y `oflag` son los permisos de lectura y escritura sobre este objeto. La función devuelve un descriptor de fichero. En el último parámetro debemos de pasar la máscara de permisos sobre este descriptor de fichero (p.e. `S_IRUSR|S_IWUSR`). La función devuelve un descriptor de fichero para el recurso memoria compartida. Una vez tengamos el descriptor de fichero, éste será válido hasta que se reinicie la máquina.

Antes de mapear en memoria el descriptor de fichero es importante dar tamaño a la memoria compartida con la función:

```
int ftruncate(int fildes, off_t length);
```

Para mapear en memoria este descriptor de fichero usamos la función:

```
void* mmap(void* addr, size_t len, int prot, int flags,
          int fildes, off_t offset);
```

Donde `addr` es la dirección de memoria en la que nos gustaría que se mapease. Puede ser `NULL` si no da lo mismo. El parámetro `len` es la longitud de la memoria compartida, y debe ser múltiplo del tamaño de página. El parámetro `prot` indica los permisos de lectura/escritura/ejecución sobre la memoria mapeada. El parámetro `flags` debe tomar el valor `MAP_SHARED`. El parámetro

`fildes` es el descriptor de fichero devuelto por `shm_open()` y `offset` es la posición en fichero dado en `fildes` a partir de la que mapear (normalmente 0).

En este momento ya podemos escribir en la memoria compartida. Tenga en cuenta que las aplicaciones que usan memoria compartida necesitan un mecanismo de sincronización adicional para evitar que dos procesos escriban a la vez en esta memoria.

El uso de memoria compartida da lugar a problemas de seguridad dado que otros programas pueden leer la memoria compartida, con lo que se recomienda usar la memoria compartida sólo para intercambiar datos, nunca código ejecutable. Los permisos que pasamos en `shm_open()` pueden paliar este problema de seguridad. La función `shm_unlink()` descrita más abajo también puede paliar este problema.

Para cerrar el mapeo se usa la función:

```
int munmap(void* addr, size_t len);
```

Finalmente, para liberar el descriptor de fichero asociado a la memoria compartida usamos la función:

```
int close(int fildes);
```

El recurso memoria compartida (cuyo nombre dimos a `shm_open()`) queda creado hasta que se reinicia la máquina (aunque cerremos todos sus descriptores de fichero). Si no queremos que más procesos se puedan unir a este nombre (obtengan descriptores de fichero para este nombre) podemos usar:

```
int shm_unlink(const char* name);
```

Si este nombre se volviera a abrir con `shm_open()` usando el mismo nombre, se crearían descriptores de fichero para otro recurso memoria compartida distinto.

# Tema 8

## Gestión de eventos

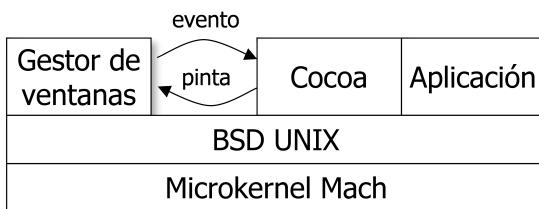
---

### **Sinopsis:**

*Los sistemas operativos como Mac OS X implementan un sistema para procesar los eventos que se producen de forma asíncrona mediante bucles de sondeo. En este tema veremos qué tipos de eventos se pueden procesar en un bucle de sondeo así como la forma en que se procesan estos eventos. Para acabar el tema estudiaremos las notificaciones, un mecanismo para informar de la ocurrencia de eventos cuando no sabemos quién estará interesado en estas notificaciones.*

## 1. El gestor de ventanas de Cocoa

Cuando se desarrolló NeXT y su sucesor NeXTSTEP los ingenieros que lo desarrollaron decidieron usar como base para su implementación el código fuente de BSD, el cual constaba de un microkernel Mach, un conjunto de llamadas al sistema UNIX y el código fuente de un gran número de utilidades. Sin embargo, no quisieron mantener el **gestor de ventanas** (también llamado **servidor de ventanas**) X11 que para entonces ya estaba implementado, sino que construyeron un nuevo gestor de ventanas. Como muestra la Figura 8.1, el gestor de ventanas recibe eventos del usuario y se los envía a las aplicaciones.



**Figura 8.1:** El gestor de ventanas y Cocoa

En tiempos de NeXTSTEP, el gestor de ventanas que se desarrolló estaba basado en PostScript, lo cual permitía que el programador de aplicaciones pudiera desarrollar un código de dibujo una vez, y este mismo código se pudiera usar tanto para pintar en pantalla como para imprimir. En Mac OS X se modificó el gestor de ventanas para que los programadores dibujaran con CoreGraphics (también llamado Quartz) y el mismo código pudiera dibujar en pantalla, en la impresora, o bien pudiera generar un fichero PDF.

Tanto el gestor de ventanas como las aplicaciones Cocoa son procesos UNIX. Como muestra la Figura 8.1, el gestor de ventanas envía eventos a las aplicaciones, y las aplicaciones envían órdenes de dibujo en pantalla al gestor de ventanas.

## 2. Gestión de eventos

Cuando creamos una aplicación Cocoa, Xcode crea un fichero `main.m` con el contenido del Listado 8.1. Normalmente el programador nunca modifica este fichero.

```
#import <Cocoa/Cocoa.h>

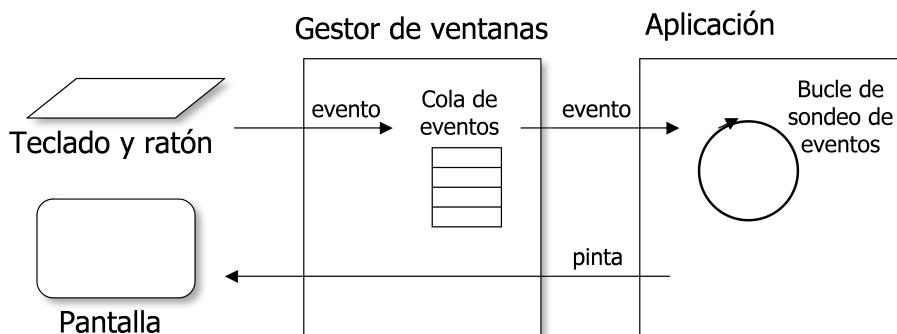
int main(int argc, char *argv[])
{
    return NSApplicationMain(argc, (const char **) argv);
}
```

**Listado 8.1:** Fichero principal de una aplicación Cocoa

La función `main()` se limita a llamar a la función `NSApplicationMain()`. Esta función hace tres cosas importantes:

1. Crea un objeto singleton de la clase `NSApplication` que representa a la aplicación y lo guarda en la variable global `NSApp`, que es un puntero a este objeto.
2. Carga el grafo de objetos de la aplicación desde el fichero nib principal (ver apartado 2.2.1 del Tema 4).
3. Ejecuta el bucle de sondeo de eventos principal (llamando a `[NSApp run]`).

A partir de este momento la aplicación Cocoa queda ejecutando el **bucle de sondeo de eventos principal** (main event loop) hasta que termina. Como muestra la Figura 8.2, el gestor de ventanas almacena los eventos que le llegan del teclado o del ratón en una **cola de eventos**. El bucle de sondeo de eventos principal va pidiendo al gestor de ventanas estos eventos y los va procesando. Por su parte, la aplicación envía órdenes de dibujo al gestor de ventanas, y el gestor de ventanas transmite estas órdenes a la pantalla.



**Figura 8.2:** Bucle de sondeo de eventos principal

En las aplicaciones Cocoa, podemos usar el argumento de la línea de comandos `-NSTraceEvents YES` de la **Tabla 4.1** para que se impriman los eventos del bucle de sondeo de eventos principal:

```
$ /Applications/TextEdit.app/Contents/MacOS/TextEdit  
-NSTraceEvents YES  
2009-08-20 09:59:50.604TextEdit[384:10b] timeout =  
62841456009.395340 seconds, mask = ffffffff, dequeue = 1, mode  
= kCFRunLoopDefaultMode  
2009-08-20 09:59:50.625TextEdit[384:10b] got apple event of  
class 61657674, ID 6f617070  
2009-08-20 09:59:50.667TextEdit[384:10b] still in loop,  
timeout = 62841456009.332939 seconds  
2009-08-20 09:59:50.667TextEdit[384:10b] timeout =  
62841456009.332939 seconds, mask = ffffffff, dequeue = 1, mode  
= kCFRunLoopDefaultMode  
2009-08-20 09:59:54.397TextEdit[384:10b] Received event:  
Kitdefined at: 0.0,0.0 time: 1707647535000 flags: 0 win: 249  
ctxt: 0 subtype: 9, data: 2b,0
```

### 3. Los bucles de sondeo

---

Se llama **bucle de sondeo** (run loop) a un bucle que recoge mensajes de una fuente y los procesa. Cuando no hay mensajes pendientes de procesar, el hilo del bucle de sondeo queda bloqueado. Se llama **bucle de sondeo de eventos** (event loop) a un bucle de sondeo donde el tipo de mensajes que recoge son eventos (normalmente procedentes del gestor de ventanas). Por último, se llama **bucle de sondeo de eventos principal** (main event loop) al bucle de sondeo asociado al hilo principal de la aplicación.

Los bucles de sondeo permiten enviar mensajes a un hilo desde otro hilo o desde otro proceso. Cada hilo Cocoa tiene asociado un **objeto bucle de sondeo** (run loop object), representado por la clase `NSRunLoop`, con lo que no es necesario crearlo, pero sí que tendremos que ponerlo en ejecución si queremos usarlo. En el caso de las aplicaciones Cocoa de interfaz gráfica creadas como muestra el Listado 8.1, el bucle de sondeo lo pone en ejecución el método `run` de la clase `NSApplication` cuando se arranca la aplicación, con lo que no debemos de ejecutarlo nosotros.

En consecuencia, tenemos que poner en ejecución un bucle de sondeo sólo es en las aplicaciones que no crean el objeto `NSApp`: normalmente las aplicaciones Cocoa de consola. El otro caso en el que tenemos que poner en ejecución el bucle de sondeo es cuando queramos usarlo en un hilo secundario. No tenemos porqué ejecutar el bucle de sondeo en todos los hilos secundarios. Por ejemplo, si nuestro hilo secundario está destinado a realizar una tarea larga, posiblemente podamos comunicarnos con él usando alguno de los mecanismos que vimos en el Tema 7, con lo que no sería necesario ejecutar su bucle de sondeo.

El bucle de sondeo de un hilo sólo es necesario ejecutarlo cuando planeamos hacer una de estas cosas:

- Usar fuentes de entrada para comunicarnos con el hilo.
- Asociar un temporizador al hilo para que realice tareas periódicas.
- Usar un método de tipo `performSelector...` para enviar mensajes al hilo desde otro hilo.

En caso de que decidamos usar el bucle de sondeo de un hilo, deberemos de diseñar un mecanismo para terminar el hilo de forma limpia. Tenga en cuenta que siempre es mejor terminar un hilo de forma limpia que forzarlo a terminar. En el apartado 3.4 estudiaremos estos mecanismos.

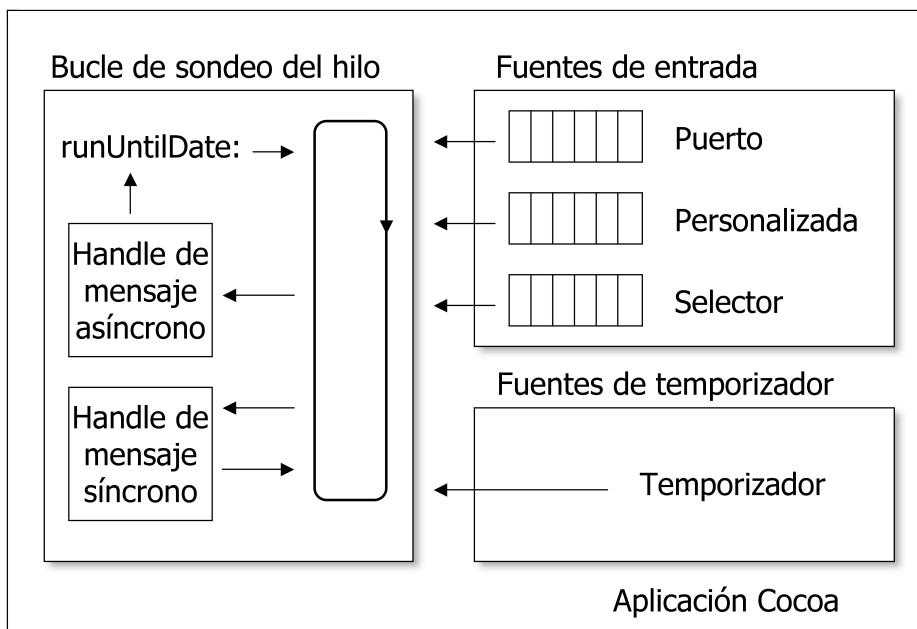
### 3.1. Anatomía de un bucle de sondeo

Un bucle de sondeo recibe mensajes a través de las **fuentes**. Cada fuente tiene asociada un **handle de fuente**, que es el método que ejecuta el bucle de sondeo cuando recibe un mensaje a través de esa fuente. Existen dos tipos de fuentes:

1. Las **fuentes de entrada** (input sources), las cuales permiten enviar mensajes al bucle de sondeo de un hilo. Estos mensajes suelen proceder o bien de otro hilo, o bien de otro proceso, aunque un hilo también se puede enviar mensajes a sí mismo.
2. Las **fuentes de temporizador** (timer sources), las cuales permiten enviar mensajes en determinados momentos de tiempo que se repiten cada cierto intervalo de tiempo.

A los mensajes que llegan por las fuentes de entrada se les llama **mensajes asíncronos**, y a los mensajes que llegan por las fuentes de temporizador se les llama **mensajes síncronos**.

La Figura 8.3 muestra la estructura conceptual del bucle de sondeo y los tipos de fuentes que lo alimentan. El hilo ejecuta un **método bloqueante** de NSRunLoop. Estos métodos tienen nombres de la forma `runXXX`, como por ejemplo `runUntilTime:`, el cual hace que el hilo se bloquee hasta que, o bien llegue un mensaje asíncrono, o bien se cumpla el plazo dado como parámetro a `runUntilTime::`.



**Figura 8.3:** Anatomía del bucle de sondeo y sus fuentes

Cuando llega un mensaje por una fuente siempre se ejecuta el handle de mensaje asociado a esa fuente. La diferencia está (ver Figura 8.3) en que si el mensaje es asíncrono, la llamada bloqueante retorna después de haber ejecutado el handle de mensaje, mientras que si la llamada es síncrona la llamada bloqueante no retorna.

Las fuentes de temporizador tienen prioridad respecto a las fuentes de entrada, y los mensajes de temporizador no se pueden encolar. Aun así, el hecho de que los mensajes de temporizador sean síncronos no impide que haya pequeños retardos desde que se lanza un mensaje de temporizador hasta que este se procesa ejecutando su correspondiente handle de mensaje. En concreto, si el bucle de sondeo está procesando otro mensaje (síncrono o asíncrono), el mensaje de temporizador no se procesará hasta que se termine de procesar el mensaje en curso.

## 3.2. Los modos del bucle de sondeo

Un **modo** del bucle de sondeo define un conjunto de fuentes (de entrada y de temporizador) a ser monitorizadas. Es importante destacar que los modos discriminan por la fuente del mensaje, no por el tipo de mensaje. Por ejemplo, no podemos usar modos para descartar los clicks de ratón, pero sí podemos usarlos para descartar un conjunto de puertos o bien suspender temporalmente un temporizador.

Durante la ejecución de un ciclo del bucle de sondeo sólo se recogen mensajes de las fuentes asociadas con el modo actual. Los posibles mensajes que haya en otras fuentes deben de esperar a que el bucle de sondeo se ejecute en un modo que los recoge.

| Modo  | Descripción   |
|---|---|
| Cocoa:<br>NSDefaultRunLoopMode<br>Core Foundation:<br>kCFRunLoopDefaultMode | Este es el modo que se usa la mayoría de las veces y acepta mensajes de todas las fuentes excepto de fuentes de mensajes distribuidos.  |
| Cocoa:<br>NSConnectionReply   | Este modo se usa para esperar a la respuesta de un mensaje distribuido (tal como veremos en el Tema 10). En este modo no queremos que se procesen otros mensajes hasta que hayamos recibido esta respuesta. |
| Cocoa:<br>NSModalPanelRunLoopMode   | Este modo se usa cuando la aplicación muestra un diálogo modal. En este caso sólo se aceptan mensajes dirigidos al diálogo modal y se descartan los mensajes enviados a otras ventanas.                     |
| Cocoa:<br>NSEventTrackingRunLoopMode  | En este modo se restringen los mensajes a operaciones de tracking de ratón y de drag  |

|   |   |
|---|---|
|   | & drop con el ratón y teclado.  |
| Cocoa:<br>NSRunLoopCommonModes<br>Core Foundation:<br>kCFRunLoopCommonModes | Define un grupo de modos configurables. Al asociar una fuente con este modo, también se asocia a todos los modos del grupo. En Cocoa por defecto este modo incluye el modo por defecto, el modo modal y el modo de event tracking. En Core Foundation <code>CFRunLoopAddCommonMode()</code> se usa para definir los modos asociados a este grupo. |

**Tabla 8.1:** Modos del bucle de sondeo predefinidos por Cocoa

Cuando hacemos una llamada a un método bloqueante de `NSRunLoop` indicamos (implícitamente o explícitamente) el modo en el que queremos entrar. Cocoa tiene varios modos predefinidos que se resumen en la Tabla 8.1. Además nosotros podemos definir nuevos modos. Cada modo se representa mediante un objeto cadena. Si no indicamos modo, por defecto se usa el modo `NSDefaultRunLoopMode`, que es el más común para el bucle de sondeo de eventos principales.

### 3.3. Obtener y ejecutar el bucle de sondeo

Como hemos dicho más arriba, cada hilo tiene asociado un objeto bucle de sondeo, el cual por defecto no está activo. Podemos acceder a este objeto ejecutando el método `currentRunLoop` de la clase `NSRunLoop` desde el hilo.

Tenga en cuenta que los objetos bucle de sondeo no tienen protección multi-hilo (ver apartado 1.8 del Tema 6), con lo que los métodos del objeto bucle de sondeo sólo deben de ser ejecutados desde el hilo al que pertenece el bucle de sondeo.

Por defecto el objeto bucle de sondeo de un hilo no tiene asociada ninguna fuente, con lo que deberemos que asociarle fuentes (como veremos en el apartado 3.5) antes de ponerlo en ejecución. Hay que tener en cuenta que si una llamada bloqueante al objeto bucle de sondeo se hace en un modo que no tiene fuentes asociadas, la llamada retorna inmediatamente, con lo que si creamos modos personalizados, deberemos de asociarles al menos una fuente. Si la llamada bloqueante no indica modo, se usa el modo `NSDefaultRunLoopMode`.

Existen tres formas de poner en ejecución un bucle de sondeo: 1) incondicionalmente, 2) con un límite de tiempo, y 3) en un determinado modo. Si no se indica modo, se usa el modo por defecto.

Para poner el bucle de sondeo en ejecución incondicional, se ejecuta el método `run` del objeto bucle de sondeo. Es decir, hacemos:

```
[ [NSRunLoop currentRunLoop] run];
```

Esta forma de ejecutar el bucle de sondeo de nuestro hilo proporciona poco control sobre el bucle de sondeo, ya que el hilo no se desbloquea hasta que se haya procesado un mensaje asíncrono. En consecuencia, es más difícil parar el bucle de sondeo limpiamente y no hay forma de usar un modo distinto al modo por defecto.

En vez de ejecutar el bucle de sondeo de forma incondicional con `run`, es mejor proporcionar un `timeout`. En este caso el hilo queda bloqueado hasta que se recibe un mensaje asíncrono o el `timeout` se cumple. En este caso podemos aprovechar que el hilo se desbloquea periódicamente para realizar otras tareas de mantenimiento. Para ejecutar la operación bloqueante de esta forma podemos usar el método `runUntilDate:`, por ejemplo así:

```
double segundos = 1.0;
NSDate* timeout =
    [NSDate dateWithTimeIntervalSinceNow:segundos];
[ [NSRunLoop currentRunLoop] runUntilDate:timeout];
```

Además de un `timeout`, podemos especificar un modo con el método `runMode:beforeDate::`. En este caso haríamos:

```
[ [NSRunLoop currentRunLoop] runMode:NSModalPanelRunLoopMode
    beforeDate:timeout];
```

### 3.4. Terminar el bucle de sondeo

A la hora de terminar un bucle de sondeo tenemos que tener en cuenta que sus métodos bloqueantes sólo retornan cuando se recibe un mensaje asíncrono o cuando se cumple el `timeout`. Una forma de parar un bucle de sondeo es enviarle un mensaje asíncrono y cuando la llamada bloqueante retorne comprobar una condición de terminación. Si nuestro bucle de sondeo sólo tiene fuentes de temporizador (como en el siguiente ejemplo), la única forma de poder pararlo es ejecutar la llamada bloqueante con `timeout`, y comprobar la condición de terminación al cumplirse el `timeout`.

El Listado 8.2 muestra un programa en el que hemos creado un bucle de sondeo con un temporizador. El programa imprime un número cada segundo y retorna al haber impreso 10 números, es decir, transcurridos 10 segundos. La condición de salida se comprueba al cumplirse el `timeout`, que se cumple cada segundo. En este caso el bucle de sondeo que usamos es el del hilo principal al que asociamos una fuente de temporizador. Dado que el bucle de sondeo sólo recibe mensajes síncronos, tenemos que ejecutar la llamada

bloqueante con timeout y comprobar la condición de terminación al acabar esta llamada.

```
#import "Contador.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // Crea un temporizador con un contador
    Contador* contador = [Contador new];
    NSTimer* timer = [NSTimer timerWithTimeInterval:1.0
                                              target:contador
                                             selector:@selector(imprime:)
                                           userInfo:nil repeats:YES];
    // Añade el temporizador al bucle de sondeo
    NSRunLoop* rl = [NSRunLoop currentRunLoop];
    [rl addTimer:timer forMode:NSDefaultRunLoopMode];
    // Ejecuta el bucle de sondeo hasta que el temporizador sea 10
    while (contador.contador<10) {
        double segundos = 1.0;
        NSDate* timeout =
            [NSDate dateWithTimeIntervalSinceNow:segundos];
        [rl runUntilDate:timeout];
    }
    [timer invalidate];
    [pool drain];
    return 0;
}
```

#### **Listado 8.2:** Bucle de sondeo con temporizador

Para implementar el handle de mensajes asociado a la fuente de temporizador hemos implementado la clase `Contador`, la cual se muestra en el Listado 8.3 y Listado 8.4. El método `imprime:` es el que actúa como handle de mensajes síncronos para el temporizador.

```
#import <Foundation/Foundation.h>

@interface Contador : NSObject {
    int contador;
}
@property int contador;
- init;
- (void)imprime:(NSRunLoop*) rl;
@end
```

#### **Listado 8.3:** Interfaz de la clase Contador

```
#import "Contador.h"

@implementation Contador
@synthesize contador;
- init {
    if (self = [super init]) {
        contador = 0;
    }
    return self;
}
```

```

- (void)imprime:(NSRunLoop*)rl {
    printf("Contador: %i\n", contador++);
}
@end

```

**Listado 8.4:** Implementación de la clase Contador

### 3.5. Tipos de fuentes

Como indicamos en el apartado 3.1 las fuentes se dividen en fuentes de entrada y fuentes de temporizador. La principal diferencia está en que las fuentes de entrada envían mensajes asíncronos, es decir, que desbloquean al bucle de sondeo, mientras que las fuentes de temporizador envían mensajes síncronos, es decir, que no desbloquean al bucle de sondeo. Las fuentes de entrada, a su vez, se dividen en tres tipos: Las fuentes basadas en puertos, las fuentes personalizadas y las fuentes de selectores. La Figura 8.3 muestra estos tipos de fuentes.

Las **fuentes basadas en puertos** son fuentes que reciben mensajes a través de puertos (representados por la clase `NSPort`). Los puertos permiten recibir mensajes desde otros procesos. Existen dos tipos de puertos:

1. Los **puertos Mach** que están representados por la clase `NSMachPort` y los usa principalmente el servidor de ventanas para enviar mensajes a la aplicación
2. Los **puertos de mensajes**, que están representados por la clase `NSMessagePort` y `NSSocketPort`. Se usan principalmente en programación con objetos distribuidos.

Las **fuentes personalizadas** permiten recibir mensajes desde otros hilos del proceso. Como su nombre indica, el contenido de los mensajes de las fuentes personalizadas no está definido, sino que lo define el programador de aplicaciones. Para crear una fuente personalizada tenemos que definir: 1) una **operación de encolado** que permita introducir un mensaje en la fuente, 2) un **handle de mensaje** que se ejecutará cada vez que se reciba un mensaje por esa fuente, y 3) una **operación de cancelación** que permita eliminar mensajes encolados.

Las **fuentes de selectores** son un tipo de fuentes personalizadas que nos permiten ejecutar un selector en el hilo receptor del mensaje. De esta forma un hilo puede pedir a otro hilo que ejecute un determinado método (selector) sobre un determinado objeto.

Las fuentes basadas en puertos se añaden a un objeto bucle de sondeo con el siguiente método de instancia de `NSRunLoop`:

```
- (void)addPort:(NSPort*)aPort forMode:(NSString*)mode
```

Las fuentes personalizadas y fuentes de selectores tienen que añadirse al objeto bucle de sondeo usando métodos de Core Foundation, ya que `NSRunLoop` no proporciona método para añadir este tipo de fuentes al bucle de sondeo.

Por último las fuentes de temporizador se pueden añadir al objeto bucle de sondeo con el método:

```
- (void)addTimer:(NSTimer*)aTimer forMode:(NSString*)mode
```

### 3.6. Los temporizadores

Las fuentes están pensadas para poder enviar mensajes a un hilo desde otro hilo o proceso. Sin embargo, normalmente las fuentes de temporizador las usan los hilos para poder enviarse a sí mismos mensajes a intervalos regulares de tiempo.

Ya hemos indicado que las fuentes de temporizador tienen prioridad respecto a otras fuentes, pero no son un mecanismo para ejecutar tareas en tiempo real ya que pueden existir pequeños retrasos desde que se dispara un temporizador hasta que se ejecuta su handle de mensaje. Esto se puede deben, o bien a que el bucle de sondeo no está en un modo que acepte el temporizador, o bien a que el bucle de sondeo está procesando otro mensaje cuando se lanza el temporizador. Apple recomienda que la resolución del temporizador no debe de estar por debajo de los 50-100 milisegundos.

Los **temporizadores** son la forma de asociar una fuente de temporizador a un bucle de sondeo. Los temporizadores están representados por la clase `NSTimer`, y hay que usarlos en un hilo que tenga su objeto bucle de sondeo activado. En el Listado 8.2 tenemos un ejemplo de uso de un temporizador.

Existen dos formas principales de crear un temporizador. La primera consiste en crear un temporizador que automáticamente se añade al objeto bucle de sondeo del objeto donde creamos el temporizador. Para ellos usamos uno de los siguientes métodos factory de `NSTimer`:

```
+ (NSTimer*)
scheduledTimerWithTimeInterval:(NSTimeInterval)seconds
                           target:(id)target
                          selector:(SEL)aSelector
                        userInfo:(id)userInfo
                         repeats:(BOOL)repeats
+ (NSTimer*)
scheduledTimerWithTimeInterval:(NSTimeInterval)seconds
                           invocation:(NSInvocation*)invocation
                         repeats:(BOOL)repeats
```

El primero recibe el objeto `target` donde ejecutar el método `aSelector`. El segundo recibe estos parámetros envueltos en un objeto `NSInvocation`.

El selector debe tener un prototipo de la forma:

```
- (void)handleMensajeTimer:(NSTimer*)theTimer
```

Si no queremos que el temporizador se añada automáticamente al bucle de sondeo durante su creación debemos crearlo con los métodos `factory`:

```
+ (NSTimer*)
timerWithTimeInterval:(NSTimeInterval)seconds
    target:(id)target
    selector:(SEL)aSelector
    userInfo:(id)userInfo
    repeats:(BOOL)repeats
+ (NSTimer*)
timerWithTimeInterval:(NSTimeInterval)seconds
    invocation:(NSInvocation*)invocation
    repeats:(BOOL)repeats
```

Y cuando queramos añadirlo al bucle de sondeo ejecutamos el método de instancia de `NSRunLoop`:

```
- (void)addTimer:(NSTimer*)aTimer forMode:(NSString*)mode
```

Obsérvese que si añadimos manualmente el temporizador al objeto bucle de sondeo podemos indicar el modo para el temporizador, mientras que cuando se añade automáticamente siempre se añade para el modo por defecto.

Obsérvese también que al crear el temporizador indicamos tanto su periodo de repetición (parámetro `seconds`) como si queremos que se repita automáticamente cada vez que transcurra ese periodo (parámetro `repeats`). Si `repeats` se fija a `NO`, el temporizador sólo se lanzará una vez.

Si hemos fijado el parámetro `repeats` a `YES`, podemos invalidar el temporizador para que deje de producirse con su método:

```
- (void)invalidate
```

Además en este caso el temporizador esté invalidado (su método `isValid` devuelve `NO`) no podrá volver a ser usado este objeto más tarde para activar otro temporizador.

Una vez creado el temporizador, también podemos indicar el momento de tiempo en que empieza a dispararse con el método:

```
- (void)setFireDate:(NSDate*)date
```

En este caso debemos de crear un temporizador manual, después fijar su tiempo de inicio y por último añadirlo al objeto bucle de sondeo.

Un método de utilidad que proporciona `NSObject` para ejecutar un selector transcurrido un tiempo es:

```
- (void)performSelector:(SEL)aSelector
                  withObject:(id)anArgument
                     afterDelay:(NSTimeInterval)delay
```

Por el momento este método sólo lo incorpora iPhone OS, y para poder usarlo es necesario que se esté ejecutando un bucle de sondeo.

Podemos cancelar la operación anterior antes de que trascurre el intervalo de tiempo dado en `delay`, si antes de transcurrir este intervalo ejecutamos el método de clase de `NSObject`:

```
+ (void)cancelPreviousPerformRequestsWithTarget:(id)aTarget
                                         selector:(SEL)aSelector object:(id)anArgument
```

### 3.7. Ejecutar selectores en otros hilos

Si tenemos un puntero a los objetos `NSThread` de nuestra aplicación, una forma cómoda de comunicar los hilos es que un objeto hilo pida al otro hilo ejecutar un método. Para ello la clase `NSObject` proporciona el método de instancia:

```
- (void)performSelector:(SEL)aSelector
                  onThread:(NSThread*)thread
                  withObject:(id)arg
                 waitUntilDone:(BOOL)wait
```

El `target` es el objeto sobre el que ejecutamos el método. El método a ejecutar se proporciona en `aSelector`. El hilo que lo debe ejecutar se proporciona en el parámetro `thread`. El método `aSelector` a ejecutar puede recibir un parámetro, en cuyo caso su valor se proporciona en `arg`.

La principal restricción para poder usar esta técnica de comunicación es que el hilo receptor del mensaje debe tener en ejecución su bucle de sondeo, con lo que esta técnica se usa principalmente para enviar mensajes desde hilos secundarios al hilo principal, ya que las aplicaciones Cocoa suelen tener un bucle de mensajes principal. Si queremos enviar mensajes usando esta técnica a hilos secundarios, estos deben de tener su bucle de sondeo en ejecución.

Los mensajes llegarán al hilo a través de una fuente de selectores, la cual se añade a su objeto bucle de sondeo cuando pedimos ejecutar el mensaje y se elimina automáticamente del objeto bucle de sondeo cuando acaba de proce-

sarse el mensaje. Tenga en cuenta que en caso de recibir mensajes en un hilo secundario, este hilo secundario deberá tener creadas otras fuentes, ya que (como indicamos en el apartado 3.3) la llamada bloqueante a su bucle de sondeo acaba inmediatamente si el objeto bucle de sondeo no tiene fuentes asociadas.

### 3.8. Bucles de sondeo Core Foundation

Tanto Cocoa como Carbon son implementaciones a alto nivel de la librería de funciones `CFRunLoop`, que son las funciones que implementan los bucles de sondeo Core Foundation al nivel más básico. De hecho existen operaciones con el bucle de sondeo que sólo se pueden acceder mediante Core Foundation. Además otras librerías de programación, como la librería de funciones `CFNetwork`, se basan en las funciones de `CFRunLoop` para implementar su funcionalidad. En este apartado vamos a estudiar cómo se manejan los bucles de sondeo con Core Foundation.

En Cocoa usamos el método `factory singleton currentRunLoop` de `NSRunLoop` para obtener el bucle de sondeo del hilo que ejecuta la operación. En el caso de Core Foundation la función a ejecutar es:

```
CFRunLoopRef CFRunLoopGetCurrent(void);
```

La función devuelve una instancia del tipo opaco `CFRunLoopRef` que podemos usar en otras funciones para referirnos al bucle de sondeo. Esta variable no es un objeto puente (bridged object), con lo que no se puede hacer casting a `NSRunLoop*`, pero podemos usar el método de instancia `getCFRunLoop` de `NSRunLoop` para obtener la correspondiente variable opaca `CFRunLoopRef` que sí podríamos pasar a Core Foundation.

Para ejecutar el bucle de sondeo, el equivalente al método `run` de `NSRunLoop` es:

```
void CFRunLoopRun(void);
```

Pero si queremos más flexibilidad es mejor usar:

```
SInt32 CFRunLoopRunInMode (CFStringRef mode,
                           CFTimeInterval seconds,
                           Boolean returnAfterSourceHandled);
```

Los modos del parámetro `mode` se definen en la Tabla 8.1. El parámetro `returnAfterSourceHandled` puesto a `true` indica que queremos que la función retorne cuando se procese alguna fuente. Si se pone a `false`, la función no retorna hasta que se cumpla el periodo dado en `seconds`.

Podemos crear un temporizador con Core Foundation usando la función:

```
CFRunLoopTimerRef CFRunLoopTimerCreate(
    CFAllocatorRef allocator,
    CFAbsoluteTime fireDate,
    CFTimeInterval interval,
    CFOptionFlags flags,
    CFIndex order,
    CFRunLoopTimerCallBack callout,
    CFRunLoopTimerContext *context);
```

El parámetro `allocator` normalmente toma el valor `NULL`. El parámetro `fireDate` es la fecha en que se lanza. Para obtener la fecha actual podemos usar la función `CFAbsoluteTimeGetCurrent()` y sumarle un número de segundos. Si `interval` es 0 el temporizador sólo se dispara una vez. Sino se dispara cada vez que se cumple ese intervalo de tiempo. Los parámetros `flags` y `order` de momento no tienen valores y deben ser 0. El parámetro `callout` apunta al handle de mensaje que debe tener el prototipo:

```
void handleTemporizador(CFRunLoopTimerRef timer, void* info);
```

El valor puesto en el parámetro `context` se recibe en el parámetro `info` del handle de mensaje.

Para añadir el temporizador al bucle de sondeo se usa la función:

```
void CFRunLoopAddTimer(
    CFRunLoopRef r1,
    CFRunLoopTimerRef timer,
    CFStringRef mode);
```

Y para invalidar el temporizador se usa la función:

```
void CFRunLoopTimerInvalidate(CFRunLoopTimerRef timer);
```

El Listado 8.5 muestra un programa con un bucle de sondeo cuyo temporizador se dispara 10 veces. El programa es similar al del Listado 8.2, pero está implementado con Core Foundation.

```
#import <Foundation/Foundation.h>

void handleTemporizador(CFRunLoopTimerRef timer, void* info) {
    printf("Disparado temporizador\n");
}

int main (int argc, const char * argv[]) {
    // Crea el temporizador
    CFAbsoluteTime timeout = CFAbsoluteTimeGetCurrent() + 1.0;
    CFRunLoopTimerRef temporizador;
    temporizador = CFRunLoopTimerCreate(NULL, timeout, 1.0, 0, 0
                                         , handleTemporizador, NULL);
    // Lo añade al bucle de sondeo y lo ejecuta
```

```
CFRunLoopRef rl = CFRunLoopGetCurrent();
CFRunLoopAddTimer(rl, temporizador, kCFRunLoopDefaultMode);
CFRunLoopRunInMode(kCFRunLoopDefaultMode, 10.0, true);
return 0;
}
```

**Listado 8.5:** Bucle de sondeo con Core Foundation

## 4. Las notificaciones

Una **notificación** es un objeto que encapsula información sobre la ocurrencia de un evento, como por ejemplo que una ventana ha ganado el foco o que una conexión de red se ha cerrado. Cuando el evento se produce el objeto que lo detecta *envía* la notificación a un **centro de notificación**, en este momento el centro de notificación transmite la notificación a todos los objetos que se hayan *registrado* para ser informados de ese evento.

### 4.1. Para qué sirven las notificaciones

La forma estándar de pasar mensajes entre objetos es ejecutando los métodos del receptor del mensaje. Sin embargo, esta forma de pasar mensajes implica que el objeto que envía el mensaje sepa quién es el receptor del mensaje y qué mensaje hay que pasárselo. En ocasiones este acoplamiento no es posible ya que el emisor puede no saber quién desea conocer el mensaje. En este caso se suele aplicar un modelo de broadcast que se suele implementar con el patrón de diseño observador. En este patrón de diseño, los objetos interesados en recibir determinado mensaje se registran en un centro de notificación, y el objeto que produce el mensaje se lo envía al centro de notificación para que éste se lo envíe a los interesados. Cualquier objeto puede registrarse en el centro de notificación y cualquier objeto puede enviar notificaciones al centro de notificación. También es posible que un objeto actúe como emisor y como receptor de notificaciones.

Normalmente el proceso de notificación es un proceso síncrono, es decir, el hilo que envía la notificación queda bloqueado hasta que todos los observadores hayan recibido la notificación. Además de centros de notificación, Cocoa implementa **colas de notificación**. Las colas de notificación difieren de los centros de notificación en que la notificación se transmite de forma asíncrona, es decir, el control vuelve al hilo de notificación antes de que empiece el proceso de información a los observadores de la notificación.

Las notificaciones que implementan los centros de notificación son un proceso de comunicación intraprocess. Sin embargo, Cocoa también implementa **centros de notificación distribuidos**, que son centros de notificación que transmiten una notificación a todos los procesos del sistema.

## 4.2. Cuándo usar notificaciones

El patrón de diseño delegado es un patrón de diseño que permite que un objeto envíe los mensajes que recibe a otro objeto llamado objeto delegado. A diferencia del patrón de diseño observador, el patrón de diseño delegado implica que normalmente haya sólo un delegado, así como que el delegado tenga un papel más activo pudiendo intervenir en la respuesta que se va a dar como retorno a este mensaje pudiendo modificar o rechazar la operación. Sin embargo, los objetos observadores realizan una tarea pasiva en la que simplemente pueden conocer que un evento ha ocurrido, pero no alterar el resultado del evento. De hecho, el método de respuesta a una notificación siempre devuelve `void`. Por otro lado, en el caso de la delegación, el objeto delegado sólo va a recibir un conjunto predefinidos de mensajes, mientras que en las notificaciones un objeto puede recibir cualquier notificación para la que se quiera registrar.

Tanto el mecanismo de notificación KVO como los centros de notificación son una implementación del patrón de diseño observador donde hay una notificación pasiva de un evento. En KVO un objeto puede observar cambios en las propiedades de otros objetos de forma más eficiente que mediante un centro de notificación, ya que no hay un intermediario (centro de notificación) implicado. Sin embargo el mecanismo de notificación KVO implica que el objeto observado implemente el protocolo KVO, con lo que es más adecuado para observar cambios en las propiedades de un objeto. Sin embargo, hay ocasiones en las que no queremos observar cambios en las propiedades de un objeto sino otro tipo de eventos más generales, como por ejemplo conocer que un proceso que hemos lanzado ha terminado. En este caso resulta más apropiado usar centros de notificación. También resulta más idóneo el uso de centros de notificación en ocasiones en las que no sabemos quién va a generar las notificaciones o bien quién está interesado en recibirlas.

Una última consideración es que normalmente el uso de notificaciones es un proceso menos eficiente que la delegación o que KVO, con lo que se recomienda usar notificaciones sólo cuando las técnicas anteriores no se adaptan bien a nuestro problema. Hay que tener en cuenta que nunca podemos saber cuántos objetos van a estar actuando como observadores de una notificación, y como la notificación es un proceso síncrono en que no recibimos el control hasta que se ha notificado a todos los interesados, no sabemos cuándo puede durar el proceso de notificación. Este coste asociado a las notificaciones tiene especial importancia cuando la notificación es distribuida, y en consecuencia debe transmitirse a todos los procesos del sistema.

## 4.3. Los objetos notificación

Un **objeto notificación** almacena la información asociada a una notificación y se representa como una instancia de la clase `NSNotification`. Este objeto encapsula tres informaciones:

1. El nombre de la notificación representado como un objeto cadena.
2. Un objeto asociado la notificación, que normalmente es el objeto que produjo la notificación.
3. Un diccionario con información adicional sobre la notificación.

La clase `NSNotification` tiene los métodos de instancia `name`, `object` y `userInfo` que permiten acceder a estas tres informaciones.

Para crear un objeto `NSNotification` se suele usar su método factory:

```
+ (id)notificationWithName:(NSString*)aName  
                      object:(id)anObject  
                     userInfo:(NSDictionary*)userInfo
```

## 4.4. Los centros de notificación

La clase `NSNotificationCenter` representa a los centros de notificación. Normalmente cada proceso tendrá solamente un centro de notificación que se obtiene con el método factory singleton:

```
+ (id)defaultCenter
```

### 4.4.1. Registrar observadores

Los objetos que actúan como observadores de notificaciones se registran en el centro de notificación usando el siguiente método del centro de notificación:

```
- (void)addObserver:(id)notificationObserver  
              selector:(SEL)notificationSelector  
                 name:(NSString*)notificationName  
                object:(id)notificationSender
```

Los parámetros `notificationObserver` y `notificationSelector` indican respectivamente el objeto observador y el método del objeto observador a ejecutar para informar de que se ha producido una notificación. Para indicar el nombre de la notificación en que está interesado el observador se usa el parámetro `notificationName`. Si queremos recibir notificaciones enviadas al

centro de notificación por un determinado objeto podemos usar el parámetro `notificationSender` para indicar este objeto.

Los únicos parámetros obligatorios son `notificationObserver` y `notificationSelector`. Si el nombre de la notificación y el objeto que envía la notificación se dejan a `nil`, se recibirán todas las notificaciones que se produzcan en el proceso.

Cuando un objeto observador ya no desee recibir más notificaciones se puede desegistrar con los métodos:

- `(void)removeObserver: (id)notificationObserver  
                          name: (NSString*)notificationName  
                          object: (id)notificationSender`
- `(void)removeObserver: (id)notificationObserver`

#### **4.4.2. Enviar notificaciones**

Para enviar una notificación al centro de notificación debemos crear un objeto `NSNotification` usando uno de los siguientes métodos factory:

- + `(id)notificationWithName: (NSString*)aName  
                          object: (id)anObject`
- + `(id)notificationWithName: (NSString*)aName  
                          object: (id)anObject  
                          userInfo: (NSDictionary*)userInfo`

Donde `aName` es el nombre de la notificación, `anObject` es el objeto que envía la notificación y `userInfo` es un diccionario opcional con información adicional.

Después enviamos la notificación al centro de notificación usando su método de instancia:

- `(void)postNotification: (NSNotification*)notification`

Por último conviene comentar que en una aplicación con varios hilos las notificaciones siempre se entregan desde el hilo que envió la notificación, y este hilo puede no coincidir con el hilo que registró al objeto observador.

#### **4.5. Los centros de notificación distribuidos**

Cada proceso tiene un centro de notificación distribuido representado por la clase `NSDistributedNotificationCenter` y al que podemos acceder usando el método factory singleton:

```
+ (id)defaultCenter
```

Hay que tener en cuenta que enviar notificaciones distribuidas es una operación costosa ya que hay que enviar la notificación a todos los procesos registrados en el sistema. Incluso Apple indica que podrían perderse mensajes en caso de que la carga de notificaciones sature las colas del sistema.

### 4.5.1. Registrar observadores

La recepción de mensajes distribuidos se hace a través del bucle de sondeo del hilo principal, con lo que es necesario que el hilo principal tenga en ejecución su bucle de sondeo para recibir este tipo de mensajes. Por defecto las aplicaciones gráficas Cocoa tienen al hilo principal con el bucle de sondeo en ejecución, pero esto no es cierto si, por ejemplo, hacemos una aplicación de consola con Foundation Framework. En este último caso tendríamos que poner al hilo principal a ejecutar el bucle de sondeo.

Para registrar un observador podemos ejecutar el método del centro de notificación distribuido:

```
- (void)addObserver:(id)notificationObserve
              selector:(SEL)notificationSelector
                 name:(NSString*)notificationName
                object:(NSString*)notificationSender
suspensionBehavior:(NSNotificationSuspensionBehavior)
             suspendedDeliveryBehavior
```

Mientras que en los centros de notificación intraprocess el objeto que envía la notificación puede ser cualquier objeto, en los centros de notificación distribuidos este objeto (el parámetro `notificationSender`) siempre debe de ser un objeto cadena y el filtrado para decidir si informar a los observadores se realiza en función del contenido del objeto cadena, no del puntero a este objeto (como ocurre en los centros de notificación intraprocess).

Los centros de notificación distribuidos se pueden suspender usando el siguiente método del objeto centro de notificación distribuido:

```
- (void)setSuspended:(BOOL)suspended
```

El objetivo de suspender el centro de notificación distribuido es no recibir mensajes cuando la aplicación no está activa. El objeto `NSApplication` suspende automáticamente el centro de notificación distribuido cuando la aplicación no está activa (es decir, está oculta o en background).

Cuando un observador se registra debe indicar si desea recibir mensajes del centro de notificación distribuido cuando éste está suspendido. Para ello en el

parámetro `suspendedDeliveryBehavior` debe proporcionar uno de los valores que aparecen en la Tabla 8.2.

|   |   |
|---|---|
| <code>NSNotificationSuspensionBehaviorDrop</code>               | Cuando está suspendido el centro de notificación distribuido no encola notificaciones.  |
| <code>NSNotificationSuspensionBehaviorCoalesce</code>           | Cuando está suspendido el centro de notificación distribuido sólo encola una notificación de cada tipo.   |
| <code>NSNotificationSuspensionBehaviorHold</code>               | Cuando está suspendido el centro de notificación distribuido encola las notificaciones, pero no las entrega hasta que deja de estar suspendido. |
| <code>NSNotificationSuspensionBehaviorDeliverImmediately</code> | El centro de notificación distribuido envía las notificaciones a los observadores aunque esté suspendido  |

**Tabla 8.2:** Comportamiento cuando el centro de notificación distribuido esté suspendido

El Listado 8.6 contiene un programa que durante un minuto captura todas las notificaciones distribuidas y las muestra en pantalla. El objeto que actúa como observador de notificaciones está implementado en el Listado 8.7 y Listado 8.8. Dado que un bucle de sondeo sin puertos termina inmediatamente, hemos añadido un puerto al bucle de sondeo antes de ejecutarlo.

```
#import "LogNotificaciones.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
    [[LogNotificaciones new] autorelease];
    NSRunLoop* rl = [NSRunLoop currentRunLoop];
    [rl addPort:[[NSMachPort new] autorelease]
        forMode:NSDefaultRunLoopMode];
    NSDate* timeout = [NSDate dateWithTimeIntervalSinceNow:60.0];
    [rl runUntilDate:timeout];
    [pool drain];
    return 0;
}
```

**Listado 8.6:** Programa que muestra las notificaciones distribuidas

```
#import <Foundation/Foundation.h>

@interface LogNotificaciones : NSObject {
}
- init;
- (void)notificacionRecibida:(NSNotification*)n;
- (void)dealloc;
@end
```

**Listado 8.7:** Interfaz de LogNotificaciones

```
#import "LogNotificaciones.h"

@implementation LogNotificaciones
- init {
    if (self = [super init]) {
        [[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(notificacionRecibida:) name:nil object:nil];
    }
    return self;
}
- (void)notificacionRecibida:(NSNotification*)n {
    NSLog(@"%@", n);
}
- (void)dealloc {
    [[NSNotificationCenter defaultCenter] removeObserver:self];
    [super dealloc];
}
@end
```

**Listado 8.8:** Implementación de LogNotificaciones

Al ejecutarlo obtenemos notificaciones de la forma:

```
2008-08-07 18:30:16.136 notificaciones[1135:10b]
NSConcreteNotification 0x108540 {name =
com.apple.carbon.core.DirectoryNotification; object =
/.vol/234881029/35239}
2008-08-07 18:30:58.682 notificaciones[1135:10b]
NSConcreteNotification 0x108540 {name =
SafariFeedBookmarksChanged; object = }
```

## 4.5.2. Enviar notificaciones

Podemos enviar notificaciones a un centro de notificación distribuido de forma similar a como se envían a un centro de notificación intraprocess, la diferencia está en que el objeto emisor debe ser un objeto cadena y que en el diccionario de información adicional sólo podemos poner objetos que se puedan serializar. El método del centro de notificación distribuido que usamos en este caso es:

```
- (void)postNotificationName:(NSString*)notificationName
                      object:(NSString*)notificationSender
                     userInfo:(NSDictionary*)userInfo
deliverImmediately:(BOOL)deliverImmediately
```

El parámetro `deliverImmediately` permite que se envíe la notificación a los observadores aunque el centro de notificación esté suspendido. Este paráme-

tro es útil cuando la notificación es urgente (p.e. que la aplicación va a cerrarse).

### 4.5.3. Notificaciones a delegados

Muchos objetos Cocoa permiten que se les asocie un delegado. Si un objeto está actuando como delegado de otro, posiblemente estará interesado en conocer las notificaciones que emite el objeto para el que está actuando como delegado.

Los objetos Cocoa que tienen delegado siguen la convención de avisar a su delegado de las notificaciones que emiten. Apple recomienda que los programadores de aplicaciones también sigan esta convención con sus objetos que tengan asociado un delegado.

Por ejemplo, los objetos `NSWindow` pueden tener asociado un delegado y los objetos `NSWindow` emiten la notificación `NSwindowDidResizeNotification` cada vez que el usuario redimensiona la ventana. De acuerdo a la convención que estamos explicando un objeto `NSWindow` deberá avisar a su delegado, y para hacerlo comprueba si el delegado implementa el método:

```
- (void>windowDidResize:(NSNotification*)n;
```

en cuyo, después de enviar la notificación al centro de notificación, ejecuta este método sobre su delegado. Es decir, gracias a esta convención que siguen los objetos Cocoa, un delegado nunca necesita registrarse como observador de las notificaciones del objeto para el que está actuando como delegado.

La convención que se sigue para el nombre del método de notificación en el delegado es que dado un nombre de notificación, se quite el `NS` del principio, se convierta la primera letra a minúsculas y se quite el `Notification` del final.

Vamos a ver otro ejemplo. `NSApplicationDidResignActiveNotification` es una notificación que envía el objeto `NSApplication` cuando una aplicación deja de estar activa. Como `NSApplication` también puede (y suele) tener asociado un objeto delegado, el objeto `NSApplication` comprobará si su delegado implementa el método:

```
- (void)applicationDidResignActive:(NSNotification*)n;
```

En cuyo ejecutará este método sobre su delegado para avisarle de la notificación que ha enviado al centro de notificación.

## 4.6. Las colas de notificación

Las colas de notificación (representadas por la clase `NSNotificationQueue`) son un interfaz a los centros de notificación (representados por la clase `NSNotificationCenter`) que añade dos servicios:

1. Encola las notificaciones para enviarlas de forma asíncrona a su correspondiente centro de notificación. De esta forma la llamada a la cola de notificación no bloquea al hilo a espera de que todos los observadores hayan sido informados.
2. Permite fusionar mensajes similares, reduciendo así el número de notificaciones que se envían a los observadores.

La clase `NSNotificationQueue` tiene el siguiente método factory que nos permite obtener la cola de notificación por defecto:

```
+ (NSNotificationQueue*) defaultQueue
```

A diferencia de lo que ocurre con los centros de notificación donde hay sólo uno por defecto para todos los hilos del proceso, cada hilo tiene asociada una cola de notificación por defecto, y inicialmente todas las colas de notificación apuntan al centro de notificación por defecto.

Para enviar una notificación a la cola de notificación se usa su método de instancia:

```
- (void)enqueueNotification: (NSNotification*) notification  
postingStyle: (NSPostingStyle) postingStyle
```

Para que las notificaciones encoladas se envíen al centro de notificación el hilo al que corresponde la cola de notificación debe de meterse en su bucle de sondeo. De hecho, la cola de notificación crea un puerto personalizado para el bucle de sondeo del hilo por el que entrega las notificaciones. El parámetro `postingStyle` puede tomar los valores:

- `NSPostWhenIdle`. Las notificaciones encoladas se envían sólo cuando el hilo asociado a la cola de notificación está dentro de su bucle de sondeo, está en el modo `NSDefaultRunLoopMode`, y además no tiene mensajes en ninguna de sus fuentes.
- `NSPostASAP`. Las notificaciones encoladas se envían cuando el hilo asociado a la cola de notificación entra en su bucle de sondeo y está en el modo `NSDefaultRunLoopMode`. En este caso no es necesario que no haya mensajes en las demás fuentes del bucle de sondeo.
- `NSPostImmediately`. En este caso no es necesario que el hilo entre en su bucle de sondeo sino que se envía la notificación de forma síncrona al centro de notificación.

Además, con el método anterior las notificaciones en las que coincide el nombre de la notificación y el emisor de la notificación serán fusionadas de forma que si hay varias iguales, sólo se enviará una notificación.

Al enviar notificaciones a la cola de notificación podemos indicar cómo queremos fusionar las notificaciones y en qué modo del bucle de sondeo queremos que envíen los datos al centro de notificación usando el método:

```
- (void)enqueueNotification:(NSNotification*)notification
                      postingStyle:(NSPostingStyle)postingStyle
                        coalesceMask:(NSUInteger)coalesceMask
                           forModes:(NSArray*)modes
```

El parámetro `postingStyle` puede tomar los mismos valores que en el método anterior. El parámetro `modes` contiene uno o más modos en los que el bucle de sondeo envía la notificación. El parámetro `coalesceMask` indica cómo fusionar las notificaciones y puede tomar los siguientes valores:

- `NSNotificationNoCoalescing` pide que nunca se fusionen notificaciones.
- `NSNotificationCoalescingOnName` pide que se fusionen las notificaciones en las que coincide el nombre.
- `NSNotificationCoalescingOnSender` pide que se fusionen las notificaciones en las que coincide el emisor.

El valor `NSNotificationCoalescingOnName` se puede combinar con un OR binario con `NSNotificationCoalescingOnSender` para que la fusión sólo se realice si coinciden tanto el nombre como el emisor de la notificación.

En la práctica, notificaciones como `NSScreenChangedEventType` se envían con el estilo `NSPostASAP` y `NSNotificationCoalescingOnName` como modo de fusión para que sólo se envíe una notificación al final del proceso de dibujo en una ventana, y no una notificación cada vez que se pinta algo en la ventaja. En los editores de texto se suele usar el estilo `NSPostWhenIdle` y el modo de fusión `NSNotificationCoalescingOnName` para informar de que el texto ha cambiado y hay que actualizar el tamaño de texto en la barra de estado. Esto se debe a que de esta forma reducimos el número de actualizaciones de la barra de estado, y en consecuencia la aplicación no consume tantos recursos.

# Tema 9

# Programación en red

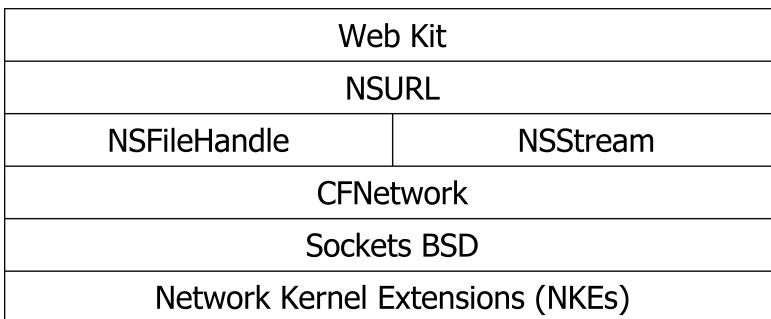
---

## **Sinopsis:**

*En este tema vamos a ver cómo comunicar programas a través de una red de datos. Empezaremos estudiando los socket. Una técnica de programación en red ampliamente usada en muchos sistemas, incluido Mac OS X. Después veremos cómo Cocoa y Core Foundation nos permite mejorar el acceso a red mediante la simplificación de operaciones comunes.*

## 1. Introducción a la programación en red

Mac OS X proporciona un amplio conjunto de APIs para programación en red. Estas APIs se pueden organizar en capas como muestra la Figura 9.1. Las capas de más bajo nivel permiten personalizar el acceso a la red, mientras que las capas de más alto nivel permiten incrementar la productividad del programador en las tareas más comunes.



**Figura 9.1:** Capas de programación en red con Mac OS X

Al más bajo nivel encontramos Network Kernel Extensions (NKE). Esta API permite modificar el comportamiento de la pila de red del kernel de Mac OS X. A continuación encontramos los sockets BSD. La principal ventaja de esta API es que la implementan prácticamente todos los sistemas operativos, con lo que es una API idónea para desarrollar librerías de programación y aplicaciones multiplataforma.

Mac OS X introduce una API de programación de bajo nivel llamada Core Foundation Network (CFNetwork) la cual permite trabajar a nivel de protocolos de red (igual que los sockets BSD). La principal ventaja de CFNetwork respecto a los sockets BSD es que, en vez de disponer de llamadas que bloquean al hilo, nos permite usar el bucle de sondeo para procesar de forma asíncrona los eventos de red.

Las capas de programación anteriores se implementan mediante APIs de programación C. A partir de las capas NSFileStream y NSStream disponemos de una interfaz de programación Objective-C con el aumento de productividad que esto supone. Estas capas nos permiten encapsular sockets y trabajar con flujos de bytes Objective-C. Dado que NSFileHandle y NSStream se basan en CFNetwork, también usan el bucle de sondeo para procesar los eventos de red.

A más alto nivel encontramos NSURL, una familia de clases Objective-C que nos permiten realizar tareas comunes con recursos de red representados como URLs. Por último Web Kit es un conjunto de frameworks de código fuente abierto con los que Apple implementa Safari. Estos frameworks nos

permiten implementar las funcionalidades de un browser de web de forma sencilla.

Network Kernel Extensions (NKEs) y WebKit no se explican en este documento y su estudio se deja para el lector interesado. El resto de APIs de programación en red se explican a lo largo de los distintos apartados de este tema.

## 2. Sockets BSD

---

Los sockets son la API de programación en red por autonomía, así como la más usada hoy en día. La API de programación de sockets fue propuesta por BSD y actualmente los implementan prácticamente todos los sistemas operativos. La idea se basa en crear un **stream de entrada** y otro **stream de salida** que comunica a una aplicación con otra aplicación. Normalmente se hace una analogía con una línea telefónica. El *servidor* es una central de llamadas donde se crea un **socket servidor** (server socket) y se espera a que el cliente llame. A esta operación de naturaleza pasiva se la suele llamar **escuchar en un socket** (listening socket). Cuando el cliente llega en el servidor se crea otro socket, que es el socket que se usa para "hablar" con el cliente. El cliente se conecta al servidor creando un **socket conexión**. Una vez conectados cliente y servidor pueden "hablar" por sus socket usando sus correspondiente stream de entrada y su stream de salida. Aunque normalmente una aplicación actúa sólo como cliente o como servidor, existen algunas aplicaciones, conocidas como **peer-to-peer** en las que la aplicación actúa a la vez como cliente y como servidor.

Para crear un socket se usa la primitiva `socket()`. Un socket se representa mediante un entero al que se llama **descriptor de fichero**, y de hecho este descriptor de fichero es el mismo que usan las famosas primitivas `read()` y `write()` de los sistemas UNIX, con lo que una vez creado el socket podemos procesarlo como si de un fichero se tratase. Las funciones propias de los sockets se encuentran declaradas en el fichero de cabecera `<sys/socket.h>`, mientras que las funciones de lectura y escritura de ficheros (y sockets) se encuentran declaradas en el fichero `<unistd.h>`.

### 2.1. Implementar un cliente

En este apartado vamos a empezar viendo qué funciones usaría un cliente de red básico y luego vamos a implementar un cliente que se conecta a un servidor e imprime en consola el texto que el servidor produce. En el siguiente apartado vamos a implementar el servidor correspondiente.

Lo primero que tiene que hacer un cliente es crear un descriptor de fichero para el socket usando la función:

```
int socket(int domain, int type, int protocol);
```

La función retorna el descriptor de fichero del socket o un número menor de 0 si se produce un error. El parámetro `domain` normalmente siempre va a ser `AF_INET`, que significa que queremos crear un socket IP. El parámetro `type` sirva para indicar si queremos que el socket sea TCP (`SOCK_STREAM`), UDP (`SOCK_DGRAM`) o IP (`SOCK_RAW`). Nosotros sólo vamos a estudiar los socket TCP. El parámetro `protocol` sólo se usa en los socket IP. En los sockets TCP siempre será 0.

En nuestro caso crearemos un socket en la variable `sfd` (socket file descriptor) de la forma:

```
int sfd;
if ((sfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    return 1;
}
```

Una vez que tendamos el socket, podemos pedir establecer una conexión con el servidor usando la función:

```
int connect(int socket, const struct sockaddr* address,
            socklen_t address_len);
```

Donde `socket` es el descriptor de fichero del socket que creó `socket()`. El parámetro `address` es un puntero a una estructura de tipo `sockaddr`. Esta estructura es la misma que usan las funciones `bind()` y `accept()` del servidor (que estudiaremos en el apartado). Curiosamente el programador nunca crea una variable del tipo `sockadd`, sino que este tipo es una especie de clase abstracta para representar información de red en distintos dominios. En el caso de IPv4, el tipo que se utiliza es el tipo `sockadd_in`. Este tipo está declarado en el fichero de cabecera `netinet/in.h` y su definición es:

```
struct sockaddr_in {
    u_char          sin_len;
    u_char          sin_family;
    u_short         sin_port;
    struct in_addr sin_addr;
    char            sin_zero[8];
};
```

El campo `sin_len` deberá contener el tamaño de la estructura tal como devuelve `sizeof()`. El campo `sin_family` en nuestro caso será siempre `AF_INET`. El campo `sin_port` es el puerto del servidor al que queremos conectarnos y el campo `sin_addr` es la dirección IP del servidor representada mediante una estructura de tipo `in_addr`. Esta estructura almacena los bytes de una dirección IP como un número de 32 bits. Por último, el campo

`sin_zero` debe rellenarse siempre con ceros. Para llenar esta estructura podemos hacer:

```
const char* host; // La IP del host
unsigned short puerto; // El puerto a donde conectarnos
struct sockaddr_in dir_servidor;
bzero(&dir_servidor,sizeof(dir_servidor));
dir_servidor.sin_len = sizeof(dir_servidor);
dir_servidor.sin_family = AF_INET;
dir_servidor.sin_port = htons(puerto);
inet_pton(AF_INET,host,&dir_servidor.sin_addr);
```

Es decir, empezamos usando `bzero()` para llenar la estructura con ceros, y luego rellenamos los campos significativos. La función `htons()` (host to network short) devuelve el parámetro en big-endian. En el caso de PowerPC la función no hace nada ya que la representación en memoria de PowerPC es por defecto big-endian. En el caso de Intel convierte `puerto` de little endian a big-endian. Para convertir una cadena de caracteres con una dirección IP (p.e. "127.0.0.1") en una estructura `in_addr` en big-endian usamos la función `inet_pton()` (printable to network).

Una vez que tenemos los parámetros de `connect()`, podemos ejecutar esta función de la forma:

```
if ( connect(sfd,(struct sockaddr*)&dir_servidor,
              sizeof(dir_servidor)) < 0 ) {
    perror("connect");
    return 1;
}
```

Si la operación tiene éxito, el descriptor de fichero dado en `sfd` contendrá un socket abierto con el servidor por el que podremos leer y escribir con las primitivas `read()` y `write()`. Para cerrar el socket disponemos de la primitiva `close()`.

El Listado 9.1 muestra un cliente escrito en C que se conecta al host y puerto que le pasemos como argumentos e imprime en consola lo que recibe por el socket.

```
/* cliente.c */

#include <stdio.h>
#include <strings.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main (int argc, const char * argv[]) {
    int n, sfd;
    const char* host;
    unsigned short puerto;
```

```

char buffer[129];
struct sockaddr_in dir_servidor;
// Recoge argumentos
if (argc<3) {
    fprintf(stderr, "Indique <host> y <punto>\n");
    return 1;
}
host = argv[1];
if (sscanf(argv[2],"%hi",&puerto) < 1) {
    fprintf(stderr, "Puerto %s no valido\n",argv[2]);
    return 1;
}
// Crea el socket
if ((sfd = socket(AF_INET,SOCK_STREAM,0)) < 0) {
    perror("socket");
    return 1;
}
// Se conecta al servidor y puerto recibidos como argumento
bzero(&dir_servidor,sizeof(dir_servidor));
dir_servidor.sin_len = sizeof(dir_servidor);
dir_servidor.sin_family = AF_INET;
dir_servidor.sin_port = htons(puerto);
inet_pton(AF_INET,host,&dir_servidor.sin_addr);
if ( connect(sfd,(struct sockaddr*)&dir_servidor,
              sizeof(dir_servidor)) < 0 ) {
    perror("connect");
    return 1;
}

// Recibe texto
while ( (n=read(sfd,buffer,128)) >0 ) {
    buffer[n]='\0';
    printf(buffer);
}

if (n<0) {
    perror("read");
    return 1;
}

// Cierra socket
if (close(sfd)) {
    perror("close");
    return 1;
}
return 0;
}

```

**Listado 9.1:** Cliente de red usando sockets BSD

Para compilar el programa podemos hacer:

```
$ gcc cliente.c -o cliente
```

Y para ejecutarlo podemos hacer:

```
$ cliente time.nist.gov 13
```

## 2.2. Implementar un servidor

El servidor necesita crear dos sockets: el socket servidor que representaremos como `listen_fd` y que sirve para esperar conexiones, y el socket de conexión que representaremos con `connect_fd` y que sirve para "hablar" con el cliente.

Para crear el socket servidor usamos la misma función `socket()` que usa el cliente para crear su socket:

```
int listen_fd, connect_fd;
if ((listen_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    return 1;
}
```

A continuación tenemos que escuchar en un puerto, lo cual implica usar dos operaciones: `bind()` y `listen()`. A continuación vamos a detallar estas operaciones.

```
int bind(int socket, const struct sockaddr* address,
         socklen_t address_len);
```

En esta operación el parámetro `socket` es el socket de servidor, y el parámetro `address` es una estructura `sockaddr_in` (la misma que usa el cliente) con información sobre el puerto y interfaz de red donde registrarnos. Si en el campo `sin_addr` pasamos `127.0.0.1` sólo estaremos aceptando conexiones locales. Si queremos aceptar conexiones desde cualquier sitio deberemos de pasar el valor `INADDR_ANY` de la forma:

```
dir_servidor.sin_addr.s_addr = htonl(INADDR_ANY);
```

La función `htonl()` (host to network long) garantiza que el número esté escrito en big-endian.

La otra operación a la que tenemos que llamar después es:

```
int listen(int socket, int backlog);
```

En donde `socket` es el socket servidor y `backlog` es el número máximo de peticiones que pueden estar en la cola antes de ser aceptadas por `accept()`. Tenga en cuenta que un servidor puede tener más clientes conectados que el valor de `backlog`, ya que el `backlog` lo que indica es cuántos clientes hay pendientes de aceptar, no cuantos clientes hay ya aceptados.

Luego, la forma de poner al servidor a esperar en un puerto sería:

```
bzero(&dir_servidor, sizeof(dir_servidor));
```

```

dir_servidor.sin_len = sizeof(dir_servidor);
dir_servidor.sin_family = AF_INET;
dir_servidor.sin_port = htons(puerto);
dir_servidor.sin_addr.s_addr = htonl(INADDR_ANY);
if ( bind(listen_fd,(struct sockaddr*)&dir_servidor,
           sizeof(dir_servidor)) < 0 ) {
    perror("bind");
    return 1;
}
if (listen(listen_fd,50) < 0) {
    perror("listen");
    return 1;
}

```

El siguiente paso que tiene que dar el servidor es llamar a la función bloqueante:

```

int accept(int socket, struct sockaddr* restrict address,
           socklen_t* restrict address_len);

```

Esta función bloquea al hilo hasta que se conecta un cliente. El parámetro `socket` es el socket de servidor y la función devuelve el descriptor de fichero del socket de conexión. El parámetro `address` es un parámetro de salida con el host y puerto del cliente aceptado. En `address_len` debemos indicar la longitud de la estructura `address`. Luego la forma de aceptar un cliente sería:

```

struct sockaddr_in dir_cliente;
socklen_t dir_cliente_len = sizeof(struct sockaddr_in);
if ( (connect_fd = accept(listen_fd,
                           (struct sockaddr*)&dir_cliente,&dir_cliente_len)) < 0) {
    perror("accept");
    return 1;
}
printf("Conectado cliente desde el host %s y puerto %i\n",
       inet_ntoa(dir_cliente.sin_addr),
       ntohs(dir_cliente.sin_port));

```

Una vez tenemos en `connect_fd` el socket de conexión, podemos hablar con el cliente usando `read()` y `write()` de forma similar a como hace el cliente.

El Listado 9.2 muestra un programa servidor que a cada cliente que se conecta le sirve un mensaje de texto con el día y la hora actual. Dado que el programa entra en un bucle infinito, la forma de pararlo será con Ctrl+C.

```

/* servidor.c */

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <time.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

```

```
#include <arpa/inet.h>

int main (int argc, const char * argv[]) {
    int listen_fd, connect_fd;
    unsigned short puerto;
    struct sockaddr_in dir_servidor;
    // Recoge argumentos
    if (argc<2) {
        fprintf(stderr, "Indique <puerto> servidor\n");
        return 1;
    }
    if (sscanf(argv[1],"%hi",&puerto) < 1) {
        fprintf(stderr, "Puerto %s no valido\n",argv[1]);
        return 1;
    }
    // Crea el socket servidor
    if ((listen_fd = socket(AF_INET,SOCK_STREAM,0)) < 0) {
        perror("socket");
        return 1;
    }
    int valor_opcion = 1;
    if (setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR,
                   &valor_opcion, sizeof(valor_opcion))) {
        perror("setsockopt");
        return 1;
    }
    // Escucha en el puerto indicado
    bzero(&dir_servidor,sizeof(dir_servidor));
    dir_servidor.sin_len = sizeof(dir_servidor);
    dir_servidor.sin_family = AF_INET;
    dir_servidor.sin_port = htons(puerto);
    dir_servidor.sin_addr.s_addr = htonl(INADDR_ANY);
    if ( bind(listen_fd,(struct sockaddr*)&dir_servidor,
              sizeof(dir_servidor)) < 0 ) {
        perror("bind");
        return 1;
    }
    if (listen(listen_fd,50) < 0) {
        perror("listen");
        return 1;
    }

    while (1) {
        // Acepta un cliente
        struct sockaddr_in dir_cliente;
        socklen_t dir_cliente_len = sizeof(struct sockaddr_in);
        if ( (connect_fd = accept(listen_fd,
                                  (struct sockaddr*)&dir_cliente,&dir_cliente_len)) < 0) {
            perror("accept");
            return 1;
        }
        printf("Conectado cliente desde el host %s y puerto %i\n",
               inet_ntoa(dir_cliente.sin_addr),
               ntohs(dir_cliente.sin_port));
        // Le da la hora
        time_t curtime;
        struct tm *loctime;
        curtime = time(NULL);
        loctime = localtime(&curtime);
        char* msg = asctime(loctime);
        if ( write(connect_fd,msg,strlen(msg)) < 0 ) {
```

```

        perror("write");
        return 1;
    }
    // Cierra el socket
    if (close(connect_fd)) {
        perror("close");
        return 1;
    }
}
return 0;
}

```

**Listado 9.2:** Servidor de red usando sockets BSD

El programa se puede compilar con el comando:

```
$ gcc servidor.c -o servidor
```

Si queremos instalar el servidor en un puerto menor a 1024 debemos ejecutarlo con permiso de administrador. Por ejemplo, si queremos instalarlo en el puerto 13, que es el puerto del servicio de tiempo daytime, podemos hacer:

```
$ sudo ./servidor 13
```

Para ver que el servidor está lanzado en el puerto de `daytime` podemos usar el comando:

```
$ netstat -a -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address (state)
tcp4      0      0 *.daytime          *.*      LISTEN
tcp4      0      0 *.3737            *.*      LISTEN
tcp4      0      0 *.kerberos         *.*      LISTEN
tcp46     0      0 *.vnc-server       *.*      LISTEN
tcp4      0      0 *.49152           *.*      LISTEN
tcp4      0      0 *.27000           *.*      LISTEN
tcp4      0      0 *.ssh              *.*      LISTEN
tcp4      0      0 localhost.ipp      *.*      LISTEN
udp4      0      0 *.rockwell-csp2   *.*      LISTEN
udp4      0      0 192.168.141.158.keeper *.*      LISTEN
udp4      0      0 *.*
udp4      0      0 *.49154           *.*      LISTEN
udp4      0      0 *.*               *.*      LISTEN
udp4      0      0 *.mdns             *.*      LISTEN
```

Y para conectarnos desde el cliente:

```
$ ./cliente localhost 13
Thu Aug 14 19:49:55 2008
```

Un problema típico de los sockets servidor es que cuando cerramos el programa que ha creado un socket servidor, el recurso socket del sistema operativo queda en estado `TIME_WAIT` durante unos segundos, con lo que si vol-

vemos a ejecutar el programa servidor posiblemente obtengamos un mensaje de error indicando que el puerto está en uso. Para evitarlo podemos usar la función `setsockopt()` tal como muestra el Listado 9.2. Esta función fija la opción `SO_REUSEADDR` para que al hacer `listen()` el puerto pueda ser reutilizado aunque el sistema operativo todavía no haya cerrado el recurso.

### 3. Resolución de nombres DNS

---

La clase `NSHost` permite representar un nombre de host así como obtener la IP de un host dado su nombre DNS. Para resolver nombres DNS, `NSHost` no sólo usa los servidores DNS, sino que también usa los servicios de Open Directory.

Para obtener un objeto `NSHost` no debemos de usar `alloc` e `init`, sino que debemos usar uno de estos métodos `factory`:

- + `(NSHost*) currentHost`
- + `(NSHost*) hostWithAddress: (NSString*) address`
- + `(NSHost*) hostWithName: (NSString*) hostname`

El primer método permite obtener el host actual. El segundo método recibe una IP de un host, bien sea IPv4 (p.e. @"127.0.0.1") o IPv6 (p.e. @"fe80::1"). El tercer método recibe un nombre DNS (p.e. @"www.ya.es") y lo resuelve, pero no intenta conectarse a él, es decir, no garantiza que el host esté activo.

Una vez que tenemos una instancia de `NSHost`, podemos obtener la dirección IP y nombre DNS representados por este objeto con los métodos:

- `(NSString*) address`
- `(NSString*) name`

## 4. Objetos stream

---

Los **objetos stream** son objetos Cocoa que representan flujos de bytes que se pueden aplicar para transferir información sobre distintos medios: memoria, fichero, sockets, etc. Los objetos stream están representados por la clase abstracta `NSStream` y por sus clases derivadas `NSInputStream` y `NSOutputStream` que representan respectivamente un stream de entrada y un stream de salida.

El punto de vista desde el que se considera a un stream de entrada o de salida es la memoria principal. En este sentido `NSInputStream` dispone de la operación `read:maxLength:` que nos permite transportar datos desde un fichero o socket a memoria principal. Por su parte `NSOutputStream` dispone de la operación `write:maxLength:` que nos permite escribir datos desde la memoria principal a un fichero o socket. Además, también es posible usar estas clases para transportar datos desde una zona de memoria representada por una instancia de la clase `NSInputStream` a otra zona de memoria representada por una instancia de la clase `NSOutputStream`.

Los streams Cocoa se basan en Core Foundation, y de hecho los objetos `NSInputStream` y `NSOutputStream` tienen sus correspondientes tipos puente `CFReadStreamRef` y `CFWriteStreamRef`. La clase abstracta `NSStream` no tiene correspondencia en Core Foundation.

### 4.1. Políticas de bloqueo

Los métodos `read:maxLength:` y `write:maxLength:` por defecto son síncronos, es decir, dejan al hilo bloqueado hasta que se pueden leer o escribir datos. En el caso de los streams en memoria o en fichero estos bloqueos no son largos, pero en el caso de los accesos a red los bloqueos pueden dejar al hilo de la aplicación paralizado.

La primera política que puede seguir nuestro programa es **asumir los bloqueos**. En este caso simplemente sabemos que el bloqueo se va a producir, con lo que normalmente debemos crear un hilo dedicado exclusivamente a procesar el stream. De esta forma el hilo principal no se verá afectado por estos bloqueos. En general, asumir el bloqueo es perfectamente válido cuando estemos leyendo de streams de memoria o de fichero, ya que en este tipo de streams los bloqueos son pequeños. Sin embargo no es aconsejable cuando estemos leyendo streams que representan sockets de red.

La otra política que podemos seguir es **evitar el bloqueo**. Para evitar este bloqueo se han propuesto dos técnicas: El polling y el uso de un bucle de sondeo.

El **polling** consiste en preguntar si existen datos para leer antes de leer de un stream de entrada, así como en preguntar si es posible escribir sin quedar bloqueados en el stream de salida antes de ejecutar la operación de escritura. Para ello la clase `NSInputStream` dispone del método `hasBytesAvailable` y la clase `NSOutputStream` dispone del método `hasSpaceAvailable`. En caso de no poder leer o escribir, el hilo puede dedicarse a otras tareas y preguntar más tarde.

La técnica de polling es la más sencilla, y es una técnica muy usada cuando estamos portando código que usa polling. Los sockets BSD son un ejemplo donde muchas veces se asume el bloqueo y otras veces se usa el polling para procesar los sockets.

La segunda forma de evitar el bloqueo es mediante un **bucle de sondeo**. Para ello registramos el stream como una fuente de entrada del objeto bucle de sondeo del hilo. Cuando se reciban datos, se producirá un evento, y en este momento realizaremos la lectura de datos. Del mismo modo, cuando haya espacio para escribir en el stream de salida se producirá un evento y realizaremos la operación de escritura.

Los objetos stream pueden tener asociado un objeto delegado. Este delegado entra en juego cuando estemos usando el bucle de sondeo para leer o escribir en un stream. Si el objeto stream no tiene delegado, como es costumbre en Cocoa, el objeto stream es delegado de sí mismo. En este caso sus derivadas podrían actuar como objetos delegado. El único método de delegación que tienen los objetos derivados de `NSStream` es:

- `(void)stream: (NSStream*)theStream handleEvent: (NSStreamEvent)streamEvent`

Este método se ejecuta cuando el bucle de sondeo dispone de datos para leer en el stream y cuando se puede escribir en el stream. En este momento el delegado puede leer o escribir del stream sin que el método de lectura o escritura quede bloqueado hasta entonces. La Tabla 9.1 muestra los posibles valores que puede tomar el parámetro `streamEvent` del método anterior.

| <b>Evento</b>                               | <b>Descripción</b>   |
|---|--|
| <code>NSStreamEventOpenCompleted</code>     | Se ha terminado de abrir el stream de entrada o salida       |
| <code>NSStreamEventHasBytesAvailable</code> | Hay datos disponibles en el stream de entrada                |
| <code>NSStreamEventHasSpaceAvailable</code> | Se puede escribir en el stream de salida sin que se bloquee. |
| <code>NSStreamEventErrorOccurred</code>     | Se ha producido un error en un stream                        |
| <code>NSStreamEventEndEncountered</code>    | Se ha alcanzado el final de un stream                        |

|  |             |
|--|-------------|
|  | de entrada. |
|--|-------------|

**Tabla 9.1:** Eventos de red del bucle de sondeo

## 4.2. Procesar el stream

En este apartado vamos a ver cómo se procesa un stream de entrada y de salida cuando estamos usando el bucle de sondeo para procesar el stream. En caso de estar procesando un stream en el que asumimos bloqueos o usamos polling su procesamiento es secuencial, en consecuencia más sencillo, y se puede entender como una parte del procesamiento con bucle de sondeo. La forma de procesar los streams de lectura y escritura es muy similar con lo que la vamos a ir comentando en paralelo.

El procesamiento de un stream en un bucle de sondeo se puede reducir a tres pasos fundamentales que vamos a detallar en los siguientes apartados:

1. Preparar el stream.
2. Procesar los eventos del stream en su objeto delegado.
3. Cerrar el stream cuando no haya más datos que procesar.

### 4.2.1. Preparar el stream

Preparar un stream consta de cuatro operaciones: crear el objeto stream, fijar su delegado, registrar el stream como un flujo de entrada del bucle de sondeo de un hilo, y finalmente abrir el stream.

Podemos crear un objeto `NSInputStream` a partir de un objeto dato (representado por `NSData`), o bien a partir de un fichero de entrada o de una conexión de red. Para crear el objeto disponemos de los métodos factory:

```
+ (id)inputStreamWithData:(NSData*)data
+ (id)inputStreamWithPath:(NSString*)path
```

Por su parte, los objetos `NSOutputStream` se pueden crear a partir de un buffer de memoria, de un fichero de salida o de una conexión de red. Para crear el objeto disponemos de los métodos factory:

```
+ (id)outputStreamToMemory
+ (id)outputStreamToBuffer:(uint8_t*)buffer
                    capacity:(NSUInteger)capacity
+ (id)outputStreamToFileAtPath:(NSString*)path
                           append:(BOOL)shouldAppend
```

Mientras que `outputStreamToMemory` crea un buffer de memoria interno para depositar los datos que escribamos en el stream, el método

`outputStreamToBuffer:capacity:`: usa el buffer pasado en el parámetro `buffer`. El parámetro `shouldAppend` del tercer método sirve para indicar si queremos escribir al final del fichero o al principio.

Si lo que queremos es crear una conexión de red, normalmente crearemos un socket con dos streams: de entrada y de salida. En este caso debemos usar el método de la clase base `NSStream`:

```
+ (void)getStreamsToHost:(NSHost*)host
                  port:(NSInteger)port
                 inputStream:(NSInputStream**)inputStream
                outputStream:(NSOutputStream**)outputStream
```

Este método recibe el host y puerto al que conectarnos y usa los parámetros de salida `inputStream` y `outputStream` para devolvernos los streams. Si sólo queremos un stream, podemos poner a `nil` el parámetro del stream que no nos interese.

Una vez creado el objeto stream, lo siguiente que tenemos que hacer es fijar su delegado con el siguiente método de la clase base `NSStream`:

- `(void)setDelegate:(id)delegate`

La clase `NSStream` también proporciona los siguientes métodos que nos permiten registrar y desregarstrar un objeto stream en un bucle de sondeo:

- `(void)scheduleInRunLoop:(NSRunLoop*)aRunLoop
 forMode:(NSString*)mode`
- `(void)removeFromRunLoop:(NSRunLoop*)aRunLoop
 forMode:(NSString*)mode`

Finalmente deberemos abrir el stream con el método de `NSStream`:

- `(void)open`

En este momento empezarán a producirse eventos que se enviarán al objeto delegado.

Aunque la forma más potente de procesar un objeto stream es mediante un bucle de sondeo, también podemos asumir el bloqueo, en cuyo caso no hace falta implementar un delegado ni registrar el stream en un bucle de sondeo. El Listado 9.3 muestra un programa que se conecta al servidor daytime (que implementamos en el Listado 9.2) y escribe la respuesta en el fichero `daytime.txt`. En este caso la operación de lectura es bloqueante. El método `streamStatus` de la clase base `NSStream` se usa para saber cuando hemos llegado al final del fichero.

Una vez que el programa obtiene un `NSInputStream` al servidor tiene que abrirlo con `open`. Como estamos usando el modo de polling, tenemos que esperar a que el método `streamStatus` deje de devolver el valor `NSStreamStatusOpening` y devuelva el valor `NSStreamStatusOpen` indicando que la conexión está abierta. Si obtenemos `NSStreamStatusError` es que la conexión ha fallado.

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // Crea los streams
    NSHost* host = [NSHost hostWithName:@"127.0.0.1"];
    NSInputStream* is = nil;
    [NSStream getStreamsToHost:host port:13 inputStream:&is
                           outputStream:nil];
    NSOutputStream* os = [NSOutputStream
                           outputStreamToFileAtPath:@"daytime.txt"
                           append:NO];
    // Abre los streams con polling
    [is open];
    [os open];
    int status;
    while (1) {
        status = [is streamStatus];
        if (status == NSStreamStatusOpening)
            continue;
        else if (status == NSStreamStatusError) {
            NSError* err = [is streamError];
            NSLog(@"Error de conexión:%@",
                  [err localizedDescription]);
            return 1;
        } else if (status==NSStreamStatusOpen) {
            printf("Socket abierto correctamente\n");
            break;
        }
    }
    // Procesa su contenido
    while ([is streamStatus] != NSStreamStatusAtEnd) {
        uint8_t buffer[128];
        NSInteger n = [is read:buffer maxLength:128];
        if (n<0) {
            NSError* err = [is streamError];
            NSLog(@"Error leyendo entrada:%@"
                  , [err localizedDescription]);
            continue;
        }
        [os write:buffer maxLength:n];
    }
    // Cierra los streams
    [is close];
    [os close];
    [pool drain];
    return 0;
}
```

**Listado 9.3:** Cliente de red que pide la hora a un servidor daytime

Una vez que la conexión está abierta, debemos de leer el stream hasta que `streamStatus` devuelva `NSStreamStatusAtEnd`, indicando que hemos llegado al final del stream de entrada, es decir, que el servidor ha cerrado el socket.

#### 4.2.2. Procesar los eventos del stream

Una vez que hayamos añadido un stream al bucle de sondeo y hayamos ejecutado `open` sobre el objeto `stream`, empezará a enviarse al delegado el mensaje `stream:handleEvent:`.

En el caso del stream de entrada los mensajes que deberán tratarse son `NSStreamEventHasBytesAvailable`, que indica que han llegado datos y podemos leerlos con el método `read:maxLength:`, y el mensaje `NSStreamEventEndEncountered`, para indicar que se ha llegado al final del stream. Por su parte, en el stream de salida el mensaje que se debe procesar es `NSStreamEventHasSpaceAvailable`, que indica que debemos escribir en el stream. La forma de indicar que no queremos escribir más datos es llamar a `write:maxLength:` con `maxLength:0`. Cuando hagamos esto, recibiremos el evento `NSStreamEventEndEncountered` indicando que se ha terminado el stream de salida.

Tanto stream de entrada como de salida pueden recibir los mensajes `NSStreamEventOpenCompleted`, que suele ser el primer mensaje que se recibe e indica que se ha terminado de abrir el stream, y el mensaje `NSStreamEventErrorOccurred`, que se recibe cuando se produce un error. En caso de producirse un error podemos obtener un objeto `NSError` con información del error llamando al método:

- `(NSError*)streamError`

En el evento de lectura `NSStreamEventHasBytesAvailable` y el evento de escritura `NSStreamEventHasSpaceAvailable`, se nos informa de que podemos leer o escribir, pero no se nos dice cuantos bytes. Esto puede producir un bloqueo si intentamos leer o escribir más bytes de los realmente disponibles. En caso de estar usando polling, el problema es similar: Si los métodos `hasBytesAvailable` (de la clase `NSInputStream`) o `hasSpaceAvailable` (de la clase `NSOutputStream`) devuelven `NO`, definitivamente las operaciones de lectura o escritura bloquearán al hilo, pero si devuelven `YES`, no se garantiza que el hilo no sea bloqueado durante la lectura o escritura. De nuevo el bloqueo se producirá dependiendo de cuántos bytes pidamos leer o escribir.

### 4.2.3. Cerrar el stream

Cuando recibimos el evento `NSStreamEventEndEncountered` sabemos que hemos llegado al final del stream de entrada, o bien que hemos finalizado un stream de salida llamando a `write:length:` con `maxLength:0`. Una vez terminado de procesar un stream, debemos de quitarlo del bucle de sondeo ejecutando sobre el objeto stream los métodos de `NSStream`:

- `(void)removeFromRunLoop:(NSRunLoop*)aRunLoop  
forMode:(NSString*)mode`
- `(void)close`

Por último, conviene comentar que si hemos creado un objeto `NSOutputStream` a partir de un buffer con el método `factory`:

- + `(id)outputStreamToBuffer:(uint8_t*)buffer  
capacity:(NSUInteger)capacity`

En `buffer` se nos habrá depositado el contenido del fichero, pero si lo creamos con el método de `factory`:

- + `(id)outputStreamToMemory`

Para obtener el puntero al buffer debemos ejecutar sobre el objeto stream de salida el método:

- `(id)propertyForKey:(NSString*)key`

Pasando en `key` la clave `NSSstreamDataWrittenToMemoryStreamKey`. Como retorno obtendremos un puntero al buffer.

El Listado 9.4, Listado 9.5 y Listado 9.6 muestran cómo se puede implementar un programa que obtiene la respuesta de un servidor daytime. Este programa es funcionalmente similar al del Listado 9.3, pero en este caso en vez de usar polling, usamos un bucle de sondeo.

En el Listado 9.6 creamos una instancia de la clase `ProcesaDaytime`. Este objeto es el que actúa como objeto delegado. Su variable de instancia `finPeticion` es el que nos indica cuando hemos recibido el final del stream de salida, es decir, el evento `NSStreamEventEndEncountered`. En este momento el bucle de sondeo deja de repetirse.

Tenga en cuenta que el uso simultáneo de un stream de entrada y otro de salida suele resultar muy problemático, ya que si en algún momento nos quedamos sin datos que poder escribir en el stream de salida llamaríamos a `write:length:` con `maxLength:0`, lo cual indicaría la terminación del stream de salida. En nuestro programa de ejemplo hemos usado un

`NSInputStream` con bucle de sondeo para leer del socket, pero las escrituras en fichero las hacemos usando un objeto `NSFileHandle` (visto en el apartado 2 del Tema 1). Dado que las lecturas de red son mucho más lentas que las escrituras en fichero, los bloqueos que se producirán en nuestro programa durante las escrituras son bastante tolerables.

```
#import <Foundation/Foundation.h>

@interface ProcesaDaytime : NSObject {
    NSFileHandle* ficheroSalida;
    BOOL finPeticion;
}
- init;
- (void)stream:(NSInputStream*)theStream
handleEvent:(NSStreamEvent)streamEvent;
- (BOOL) finPeticion;
@end
```

**Listado 9.4:** Interfaz de `ProcesaDaytime`

```
#import "ProcesaDaytime.h"

@implementation ProcesaDaytime
- init {
    if (self = [super init]) {
        ficheroSalida = [NSFileHandle
                          fileHandleForWritingAtPath:@"/daytime.txt"];
        finPeticion = NO;
    }
    return self;
}
- (void)stream:(NSInputStream*)theStream
handleEvent:(NSStreamEvent)streamEvent {
    switch(streamEvent) {
        case NSStreamEventOpenCompleted: {
            NSLog(@"Abierto %@", theStream);
            break;
        }
        case NSStreamEventHasBytesAvailable: {
            uint8_t buffer[512];
            NSInteger n = [theStream read:buffer maxLength:512];
            NSData* datos = [NSData dataWithBytes:buffer length:n];
            [ficheroSalida writeData:datos];
            break;
        }
        case NSStreamEventEndEncountered: {
            finPeticion = YES;
            [ficheroSalida closeFile];
            NSLog(@"Terminado %@", theStream);
            break;
        }
        case NSStreamEventErrorOccurred: {
            NSError* err = [theStream streamError];
            NSLog(@"Error %@ procesando stream %@", err, theStream);
            finPeticion = YES;
            break;
        }
    }
}
```

```

    }
}
- (BOOL) finPeticion {
    return finPeticion;
}
@end

```

**Listado 9.5:** Implementación de ProcesaDaytime

En el Listado 9.5 hemos declarado el parámetro `theStream` de tipo `NSInputStream` (en vez de `NSStream`). Esto podemos hacerlo porque ambos parámetros son punteros del mismo tamaño. Además, de esta forma la llamada al método `read:maxLength:` no producirá un warning indicando que el método `read:maxLength:` podría no encontrarse en el objeto `theStream`.

```

#import "ProcesaDaytime.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    ProcesaDaytime* p = [ProcesaDaytime new];
    NSHost* host = [NSHost hostWithName:@"127.0.0.1"];
    NSInputStream* is;
    [NSStream getStreamsToHost:host port:13
        inputStream:&is
        outputStream:nil];
    [is setDelegate:p];
    NSRunLoop* rl = [NSRunLoop currentRunLoop];
    [is scheduleInRunLoop:rl forMode:NSDefaultRunLoopMode];
    // Abre el stream de entrada con bucle de sondeo
    [is open];
    while (![p finPeticion]) {
        NSDate* timeout = [NSDate dateWithTimeIntervalSinceNow:1.0];
        [rl runUntilDate:timeout];
    }
    [is removeFromRunLoop:rl forMode:NSDefaultRunLoopMode];
    [is close];
    [pool drain];
    return 0;
}

```

**Listado 9.6:** Cliente de red que usa un bucle de sondeo para leer de un servidor daytime

## 5. Sockets con objetos Cocoa

---

Las clases `NSInputStream` y `NSOutputStream` resultan muy útiles para implementar un cliente de red, sin embargo, no disponen de funcionalidad para implementar un servidor. Los sockets BSD son una tecnología potente para procesar sockets, pero su programación es en C, no en Objective-C. Cocoa permite envolver sockets BSD en objetos `NSFileHandle`. De esta forma podemos aprovechar toda la potencia de los sockets y toda la productividad de programar con Objective-C.

### 5.1. Representar descriptores de fichero

En el apartado 2 vimos que la primitiva `socket()` devolvía un descriptor de fichero. Una vez abierto el socket, podemos usar su descriptor de fichero para llamar a las primitivas `read()` y `write()`. La clase `NSFileHandle` no proporciona ningún método para crear un socket, pero si tenemos creado un descriptor de fichero (p.e. con `socket()`) podemos crear un `NSFileHandle` con sus métodos de instancia:

- `(id)initWithFileDescriptor:(int)fileDescriptor`
- `(id)initWithFileDescriptor:(int)fileDescriptor  
closeOnDealloc:(BOOL)flag`

El segundo de ellos nos permite indicar si queremos que se cierre el descriptor de fichero cuando el objeto `NSFileHandle` reciba el mensaje `dealloc`.

A partir de este momento podemos usar métodos como `availableData`, `readDataOfLength:`, `writeData:` o `closeFile` para procesar el descriptor de fichero con un objeto Cocoa. El uso de estos métodos lo estudiamos en el apartado 2 del Tema 1.

### 5.2. Operaciones asíncronas

Las operaciones `availableData`, `readDataOfLength:`, `writeData:` de la clase `NSFileHandle` son síncronas. Esto puede implicar largos bloqueos cuando estamos trabajando con sockets. Para evitar este problema la clase `NSFileHandle` proporciona otro conjunto de operaciones asíncronas, las cuales indican que queremos recibir una notificación cuando llegue una nueva conexión, se reciban datos, o se puedan escribir datos. De esta forma el hilo de la aplicación puede estar realizando otras tareas mientras que llega la notificación. Para poder usar estas operaciones el hilo debe de estar ejecutando su bucle de sondeo. En este apartado vamos a ver estas operaciones.

### 5.2.1. Aceptar una conexión

Como ya hemos dicho, la clase `NSFileHandle` no proporciona ningún método para crear un socket, con lo que si queremos crear un socket servidor debemos empezar usando `socket()` para crear el socket servidor, y ponerlo a escuchar con las primitivas `bind()` y `listen()`. Una vez tengamos el socket servidor escuchando, podemos crear un objeto `NSFileHandle` que lo represente de la forma:

```
int listen_fd; // Ya creado y escuchando
NSFileHandle* fh_listen = [[NSFileHandle alloc]
                           initWithFileDescriptor:listen_fd];
```

Ahora en vez de ejecutar la primitiva `accept()`, y dejar al hilo bloqueado, podemos pedir que se nos notifique cuando se reciba una conexión. Para ello debemos de:

1. Registrarnos en el centro de notificación como observadores de la notificación `NSFileHandleConnectionAcceptedNotification`.
2. Ejecutar el método `acceptConnectionInBackgroundAndNotify` de nuestro objeto `NSFileHandle` para indicarle que queremos que se ponga a aceptar conexiones de forma asíncrona.

Si el método en el que vamos a recibir la notificación de nueva conexión es:

- `(void)nuevaConexion:(NSNotification*)n`

Podemos registrarnos y pedir que se acepte conexión en background de la forma:

```
NSNotificationCenter* nc;
nc = [NSNotificationCenter defaultCenter];
[nc addObserver:self selector:@selector(nuevaConexion:)
           name:NSFileHandleConnectionAcceptedNotification
         object:nil];
[fh_listen acceptConnectionInBackgroundAndNotify];
```

Cuando se produzca la notificación, el socket de conexión que usa el servidor para comunicarse con el cliente se nos pasará en la clave `NSFileHandleNotificationFileHandleItem` del diccionario asociado a la conexión. Para obtenerlo podemos hacer:

```
// n es la notificación que se nos envía
NSDictionary* user_info = [n userInfo];
NSFileHandle* fh_conexion = [user_info
                             objectForKey:NSFileHandleNotificationFileHandleItem];
```

Si hay un error, en este diccionario también se nos pasará un objeto `NSNumber` con el código de error producido en la propiedad con clave `@"NSFileHandleError"`. Podemos consultar esta condición de la forma:

```
NSNumber* n_err = [user_info
                     objectForKey:@"NSFileHandleError"];
if (n_err) {
    perror("accept");
    return;
}
```

El importante tener en cuenta que una vez que se acepta una conexión se deja de esperar nuevas conexiones. Si queremos seguir aceptando clientes debemos volver a ejecutar `acceptConnectionInBackgroundAndNotify` sobre el objeto `NSFileHandle` que representa a nuestro socket servidor.

El Listado 9.7 muestra un programa que actúa como servidor y acepta conexiones de forma asíncrona con notificaciones desde `NSFileHandle`. El programa registra una fuente en el bucle de sondeo (necesaria para que el bucle de sondeo se pueda poner a esperar) y pone en ejecución el bucle de sondeo.

```
#import <Foundation/Foundation.h>
#import "Servidor.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    // Recoge argumentos
    unsigned short puerto;
    if (argc<2) {
        fprintf(stderr, "Indique <puerto> del servidor\n");
        return 1;
    }
    if (sscanf(argv[1],"%hi",&puerto) < 1) {
        fprintf(stderr, "Puerto %s no valido\n",argv[1]);
        return 1;
    }

    // Crea el objeto servidor
    Servidor* s;
    if ( (s=[[Servidor alloc] initConPuerto:puerto]) == nil )
        return 1;

    // Ejecuta el bucle de sondeo
    NSRunLoop* rl = [NSRunLoop currentRunLoop];
    [rl addPort:[[[NSMachPort alloc] init] autorelease]
        forMode:NSDefaultRunLoopMode];
    [rl run];

    [pool drain];
    return 0;
}
```

**Listado 9.7:** Programa servidor daytime con `NSFileHandle`

El Listado 9.8 y Listado 9.9 muestran la implementación de una clase Servidor la cual, durante su inicialización, crea un socket servidor y lo pone a aceptar conexiones. La clase también implementa el método que recibe notificaciones de conexión, y cuando las recibe devuelve la hora al cliente. Para escribir la hora usa el método síncrono `writeData:` y cierra la conexión con el método `closeFile`. Después vuelve a pedir que se le notifique cuando llegue otro cliente: Para ello tiene que volver a ejecutar el método `acceptConnectionInBackgroundAndNotify`.

```
#import <Foundation/Foundation.h>

@interface Servidor : NSObject {
    unsigned short puerto;
    NSFileHandle* fhListen;
}
- initConPuerto:(unsigned short)paramPuerto;
- (void)nuevaConexion:(NSNotification*)n;
@end
```

#### Listado 9.8: Interfaz de Servidor

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#import "Servidor.h"

@implementation Servidor
- initConPuerto:(unsigned short)paramPuerto {
    if (self = [super init]) {
        puerto = paramPuerto;
        // Crea el socket servidor
        int listen_fd;
        if ((listen_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            perror("socket");
            return nil;
        }
        int valor_opcion = 1;
        if (setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR,
                      &valor_opcion, sizeof(valor_opcion))) {
            perror("setsockopt");
            return nil;
        }
        // Escucha en el puerto indicado
        struct sockaddr_in dir_servidor;
        bzero(&dir_servidor, sizeof(dir_servidor));
        dir_servidor.sin_len = sizeof(dir_servidor);
        dir_servidor.sin_family = AF_INET;
        dir_servidor.sin_port = htons(puerto);
        dir_servidor.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(listen_fd, (struct sockaddr*)&dir_servidor,
                  sizeof(dir_servidor)) < 0) {
            perror("bind");
            return nil;
        }
        if (listen(listen_fd, 50) < 0) {
            perror("listen");
```

```

        return nil;
    }

    // Crea un NSFfileHandle para escuchar
    fhListen = [[NSFileHandle alloc]
                initWithFileDescriptor:listen_fd];

    // Se registra como observador y acepta conexión
    NSNotificationCenter* nc;
    nc = [NSNotificationCenter defaultCenter];
    [nc addObserver:self selector:@selector(nuevaConexion:)
               name:NSFileHandleConnectionAcceptedNotification
             object:nil];
    [fhListen acceptConnectionInBackgroundAndNotify];
}

return self;
}
- (void)nuevaConexion:(NSNotification*)n {
    // Recoge la conexión
    NSDictionary* user_info = [n userInfo];
    NSFileHandle* fh_conexion = [user_info
        objectForKey:NSFileHandleNotificationFileHandleItem];
    NSNumber* n_err = [user_info objectForKey:@"NSFileHandleError"];
    if (n_err) {
        perror("accept");
        return;
    }
    // Procesa la conexión
    time_t curtime;
    struct tm *loctime;
    curtime = time(NULL);
    loctime = localtime(&curtime);
    char* msg = asctime(loctime);
    NSData* respuesta = [NSData dataWithBytes:msg
                                         length:strlen(msg)];
    [fh_conexion writeData:respuesta];
    [fh_conexion closeFile];

    // Pide otra notificación
    [fhListen acceptConnectionInBackgroundAndNotify];
}
@end

```

**Listado 9.9:** Implementación de Servidor

### 5.2.2. Leer datos en background

Aunque siempre podemos usar los métodos de lectura síncrona que proporciona `NSFileHandle` para leer los datos que nos llegan del cliente, esta forma de trabajar implica dejar bloqueado al hilo hasta que los datos llegan por la red. Al igual que pasaba con la recepción de conexiones, una forma más eficiente de leer es usar el bucle de sondeo para saber cuándo llegan datos por el socket. Podemos pedir una lectura asíncrona con el método de instancia de `NSFileHandle`:

- `(void)waitForDataInBackgroundAndNotify`

y registrar un objeto para que se le informe cuando por la red llegue un bloque de datos. Para ello debemos registrar a nuestro objeto como observador de la notificación `NSFileHandleDataAvailableNotification`. Para leer los datos que han llegado usamos los métodos síncronos de lectura `readDataOfLength:` y `readDataToEndOfFile`.

De nuevo, esto hace que se nos informe cuando llegue el primer bloque de datos. Si queremos que se nos vuelva a notificar de la llegada de más datos debemos de volver a ejecutar `waitForDataInBackgroundAndNotify`.

Una alternativa al método de lectura anterior es pedir que se nos informe cuando lleguen datos, depositándolos en un objeto dato. Para ello pedimos ser notificados con el método:

- `(void)readInBackgroundAndNotify`

En este caso debemos registramos como observadores de la notificación `NSFileHandleReadCompletionNotification`. Cuando llegue la notificación, los datos que han llegado se encontrarán en un objeto `NSData` del diccionario de la notificación. Para acceder a ellos usamos la clave `NSFileHandleNotificationDataItem`. Es decir, si `n` es la notificación recibida, hacemos:

```
NSDictionary* user_info = [n userInfo];
NSData* data = [user_info
                objectForKey: NSFileHandleNotificationDataItem];
```

Si no esperamos recibir muchos datos del cliente, podemos usar el método:

- `(void)readToEndOfFileInBackgroundAndNotify`

El cual pide que no se nos informe hasta que el cliente nos haya enviado todos los datos y haya cerrado su socket. En este caso debemos registrar a uno de nuestros objetos como observador de la notificación `NSFileHandleReadToEndOfFileCompletionNotification`.

### 5.2.3. Escribir datos en background

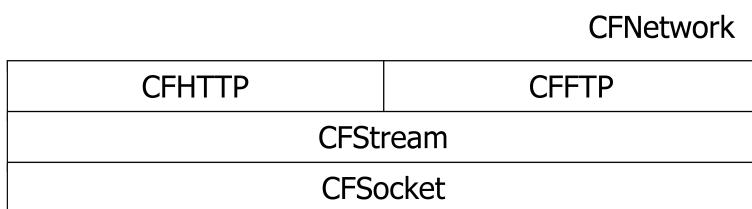
La clase `NSFileHandle` no proporciona métodos para escribir datos en background. Esto se debe a que cuando llamamos a `writeData:` los datos se almacenan en la pila de TCP/IP del sistema operativo y el método no suele producir bloqueos.

Sin embargo, si escribimos rápidamente muchos datos en el socket, la pila de TCP/IP se podría llenar, y en este caso, la llamada al método `writeData:` bloquearía al hilo hasta que la pila disponga de espacio. Esta posibilidad de

bloqueo de nuestro hilo es importante tenerla en cuenta cuando estamos implementando una aplicación que transmite gran cantidad de datos.

## 6. Programación en red con Core Foundation

En el apartado 1 introdujimos Core Foundation Network (CFNetwork), un framework de alto rendimiento que extendía los sockets BSD para que, en vez de realizar llamadas bloqueantes, realicemos las operaciones de red desde el bucle de sondeo, y de esta forma evitemos la necesidad de crear hilos dedicados a los accesos a red. Con CFNetwork simplemente registramos funciones callback que se ejecutan cuando se producen los eventos de red.



**Figura 9.2:** Organización en capas de CFNetwork

Como muestra la Figura 9.2, CFNetwork dispone de varias capas que vamos a comentar en los siguientes apartados. Los nombres de las capas corresponden con el prefijo de las funciones que las implementan, es decir, encontraremos funciones de la forma `CFSocketXXX()`, `CFStreamXXX()`, `CFHTTPXXX()`, etc.

### 6.1. CFSocket

El tipo `CFSocketRef` se usa para representar un socket BSD mediante un objeto de tipo opaco Core Foundation.

Para crear un socket debemos usar las funciones `CFSocketCreate()` o `CFSocketCreateNative()`. Después debemos de crear una fuente para el bucle de sondeo con `CFSocketCreateRunLoopSource()` y añadir esta fuente al bucle de sondeo con `CFRunLoopAddSource()`.

#### 6.1.1. Crear un socket

Antes de ver cómo conectarnos a un servidor o cómo aceptar conexiones desde un servidor, vamos a empezar viendo cómo se crea un objeto `CFSocketRef`. Los prototipos de las funciones para crear un socket Core Foundation son:

```

CFSocketRef CFSocketCreate (
    CFAllocatorRef allocator,
    SInt32 protocolFamily,
    SInt32 socketType,
    SInt32 protocol,
    CFOptionFlags callBackTypes,
    CFSocketCallBack callout,
    const CFSocketContext *context);

CFSocketRef CFSocketCreateWithNative (
    CFAllocatorRef allocator,
    CFSocketNativeHandle sock,
    CFOptionFlags callBackTypes,
    CFSocketCallBack callout,
    const CFSocketContext *context);

```

En la primera función debemos pasar el protocolo y tipo de socket a crear, y la función crea un socket BSD representado mediante un socket Core Foundation. Por el contrario, a la segunda función debemos pasarla un descriptor de fichero de un socket ya creado.

El parámetro `callout` es un puntero a una función callback que se ejecutará cuando se produzca un determinado evento sobre el socket. La función callback deberá tener el siguiente prototipo:

```

typedef void (*CFSocketCallBack) (
    CFSocketRef s,
    CFSocketCallBackType callbackType,
    CFDataRef address,
    const void *data,
    void *info
);

```

Los eventos en los que estamos interesados se pasan como flags en `callBackTypes`, y sus posibles valores se describen en la Tabla 9.2.

| <b>Flag</b>                           | <b>Descripción</b>  |
|---------------------------------------|---|
| <code>kCFSocketAcceptCallBack</code>  | Usado por los servidores para pedir que se acepten las conexiones entrantes y se ejecutará la función de callback pasándola el descriptor de fichero del socket de la nueva conexión. |
| <code>kCFSocketConnectCallBack</code> | Usado por los clientes para pedir que se informe a la función callback cuando termine el proceso de conexión con el servidor.   |
| <code>kCFSocketReadCallBack</code>    | Usado para indicar que se ejecute la función callback cuando podamos leer del socket. En este caso uno de los parámetros de la función callback será el descriptor de fichero del     |

|                        |   |
|------------------------|---|
|                        | que leer los datos que han llegado.   |
| kCFSocketDataCallBack  | Usado para indicar que cuando se reciban datos se ejecute la función callback, y a la función callback se pasen los datos leídos en un parámetro de tipo <code>CFData</code> .                |
| kCFSocketWriteCallBack | Usado para indicar que se ejecute la función callback cuando podamos escribir en el socket, y que a la función callback se le pase el descriptor de fichero en el que podemos escribir datos. |

**Tabla 9.2:** Flags de callback

Los flags `kCFSocketReadCallBack`, `kCFSocketAcceptCallBack`, y `kCFSocketDataCallBack` son mutuamente exclusivos.

### 6.1.2. Conectar desde un socket cliente

Una vez creado el objeto `CFSocketRef`, podemos conectarnos desde un socket cliente a un servidor con la función:

```
CFSocketError CFSocketConnectToAddress (
    CFSocketRef s,
    CFDataRef address,
    CFTimeInterval timeout);
```

La función recibe un objeto dato en el parámetro `address`, y en este objeto dato debemos meter una variable `struct sockaddr` con la dirección del servidor al que queremos conectarnos. El parámetro `timeout` indica el tiempo máximo que puede durar el intento de conexión. Si hemos creado el socket con el flag `kCFSocketConnectCallBack`, se informará a la función callback cuando el proceso de conexión termine.

### 6.1.3. Crear una fuente para el bucle de sondeo

Para que el mecanismo de callback anterior funcione no basta con que creamos el socket, debemos de crear una fuente y añadirla al bucle de sondeo de la aplicación. Para crear la fuente usamos:

```
CFRunLoopSourceRef CFSocketCreateRunLoopSource (
    CFAllocatorRef allocator,
    CFSocketRef s,
    CFIndex order);
```

El parámetro `order` indica la prioridad de esta fuente respecto a otras fuentes. Cuanto más bajo sea el número más prioridad tiene la fuente.

Una vez creada la fuente (variable de tipo opaco `CFRunLoopSourceRef`), añadimos la fuente al bucle de sondeo con la función:

```
void CFRunLoopAddSource (
    CFRunLoopRef rl,
    CFRunLoopSourceRef source,
    CFStringRef mode);
```

### 6.1.4. Leer y escribir en el socket

La lectura y escritura del socket se realizará cuando se reciban en la función callback los eventos `kCFSocketReadCallBack`, `kCFSocketDataCallBack` y `kCFSocketWriteCallBack`.

En caso de haber pedido ser notificados con `kCFSocketDataCallBack`, en el parámetro `data` de la función callback recibiremos un `CFDataRef` con los datos recibidos. En caso contrario, para leer y escribir en el socket usaremos las primitivas `read()` y `write()`. Para obtener el descriptor de fichero del socket podemos usar la función:

```
int CFSocketGetNative(CFSocketRef s);
```

### 6.1.5. Terminar la conexión

Para terminar la conexión debemos cerrar el socket con la función:

```
void CFSocketInvalidate(CFSocketRef s);
```

En este momento se nos garantiza que la función callback no volverá a ser ejecutada. Además, por defecto se cierra el descriptor de fichero asociado al socket. En la Tabla 9.3 aparece un flag que podemos usar para pedir que no se cierre el descriptor de fichero del socket cuando se invalide el socket Core Foundation.

También debemos eliminar la fuente del bucle de sondeo con la función:

```
void CFRunLoopRemoveSource (
    CFRunLoopRef rl,
    CFRunLoopSourceRef source,
    CFStringRef mode);
```

Por último, conviene recordar que las referencias obtenidas con funciones del tipo `CFCREATEXXX()` o `CFCOPYXXX()` las poseemos nosotros, y al acabar de trabajar con ellas debemos liberarlas con `CFRELEASE()`.

Como ejemplo de uso de CFNetwork para crear un cliente, el Listado 9.10 muestra un cliente de daytime. La función `ProcesaRed()` es la función que actúa como función callback. En ella se reciben las notificaciones de que nos hemos conectado y de dónde podemos leer datos.

```
#import <Foundation/Foundation.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

void ProcesaRed(CFSocketRef s, CFSocketCallBackType callbackType,
                 CFDataRef address, const void *data, void *info) {
    switch (callbackType) {
        case kCFSocketReadCallBack: {
            int s_fd = CFSocketGetNative(s);
            char buffer[128];
            int n;
            do {
                n = read(s_fd,buffer,sizeof(buffer)-1);
                if (n== -1) {
                    perror("read");
                    break;
                }
                buffer[n] = '\0';
                printf(buffer);
            } while (n>0);
            break;
        }
        case kCFSocketConnectCallBack: {
            SInt32 err = (SInt32) data;
            if (err==0)
                printf("Conectado al servidor\n");
            else {
                printf("Fallo durante la conexión al servidor\n");
            }
            break;
        }
        default:
            printf("Evento no contemplado\n");
            break;
    }
}

int main (int argc, const char * argv[]) {
    // Recoge argumentos
    if (argc<3) {
        fprintf(stderr, "Indique <host> <puerto>\n");
        return 1;
    }
    const char* host = argv[1];
    unsigned short puerto;
    if (sscanf(argv[2],"%hi",&puerto) < 1) {
        fprintf(stderr,"Puerto %s no valido\n",argv[2]);
        return 1;
    }
    // Crea el socket
    CFSocketRef s = CFSocketCreate(NULL, PF_INET, SOCK_STREAM
        , IPPROTO_TCP,kCFSocketConnectCallBack|kCFSocketReadCallBack
        ,ProcesaRed,NULL);
}
```

```

// Se intenta conectar al servidor
struct sockaddr_in dir_servidor;
bzero(&dir_servidor,sizeof(dir_servidor));
dir_servidor.sin_len = sizeof(dir_servidor);
dir_servidor.sin_family = AF_INET;
dir_servidor.sin_port = htons(puerto);
inet_nton(AF_INET,host,&dir_servidor.sin_addr);
CFDataRef datos = CFDataCreate(NULL,(uint8_t*)&dir_servidor,
                               sizeof(dir_servidor));
CFRelease(datos);
CFSocketError err = CFSocketConnectToAddress(s, datos, 0.0);
if(err!=kCFSocketSuccess) {
    printf("Fallo intentando conectar al servidor\n");
    return 1;
}
// Se mete en un bucle de sondeo
CFRunLoopRef rl = CFRunLoopGetCurrent();
CFRunLoopSourceRef rls = CFSocketCreateRunLoopSource(NULL,s,0);
CFRunLoopAddSource(rl, rls, kCFRunLoopDefaultMode);
while (!recibidaPeticion) {
    CFRunLoopRunInMode(kCFRunLoopDefaultMode, 1.0, YES);
}
// Cierra el socket y lo quita del bucle de sondeo
CFSocketInvalidate(s);
CFRunLoopRemoveSource(rl, rls, kCFRunLoopDefaultMode);
CFRelease(s);
CFRelease(rls);
return 0;
}

```

**Listado 9.10:** Programa cliente con CFNetwork

### 6.1.6. Aceptar conexiones desde un socket servidor

Si queremos crear un servidor primer debemos crear un socket servidor y ponerle a escuchar en un puerto. Para crear el socket servidor podemos usar las funciones del apartado 6.1.1. Una vez creado el socket servidor con las funciones anteriores lo ponemos a escuchar con la función:

```
CFSocketError CFSocketSetAddress (
    CFSocketRef s,
    CFDataRef address);
```

En `address` debemos pasar un `struct sockaddr` con la interfaz y puerto donde queremos escuchar.

Conviene recordar que en el servidor hay dos tipos de sockets. El socket servidor y el socket conexión (véase apartado 2). Hasta ahora hemos creado el socket servidor. Cuando en la función callback se reciba el mensaje `kCFSocketReadCallBack` o `kCFSocketWriteCallback` obtendremos el socket conexión.

Cada socket (socket servidor y socket conexión) tendrá su propia función callback, y deberemos registrar cada socket como una fuente diferente del bucle de sondeo del hilo. El Listado 9.11 muestra un servidor daytime donde el evento `kCFSocketAcceptCallBack` se procesa en la función callback `ProcesaSocketServidor()` y el evento `kCFSocketWriteCallBack` se procesa en `ProcesaSocket()`. La primera función recibe el socket servidor, y la segunda el socket conexión.

```
#import <Foundation/Foundation.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <time.h>

void ProcesaSocket(CFSocketRef s, CFSocketCallBackType callbackType,
                    CFDataRef address, const void *data, void *info) {
    switch (callbackType) {
        case kCFSocketWriteCallBack: {
            int connect_fd = CFSocketGetNative(s);
            // Le da la hora
            time_t curtime;
            struct tm *loctime;
            curtime = time(NULL);
            loctime = localtime(&curtime);
            char* msg = asctime(loctime);
            if ( write(connect_fd,msg,strlen(msg)) < 0 ) {
                perror("write");
                exit(1);
            }
            // Invalida el socket para no recibir más mensajes
            CFSocketInvalidate(s);
            break;
        }
        default:
            printf("Evento no contemplado\n");
            break;
    }
}

void ProcesaSocketServidor(CFSocketRef ss, CFSocketCallBackType
callbackType,
                            CFDataRef address, const void *data, void *info) {
    switch (callbackType) {
        case kCFSocketAcceptCallBack: {
            int connect_fd = *((int*)data);
            // Además de crear un CFSocket, pone la función callback
            // para el socket
            CFSocketRef s = CFSocketCreateWithNative(NULL,connect_fd,
                kCFSocketReadCallBack|kCFSocketWriteCallBack,
                ProcesaSocket,NULL);
            // Añade el socket como fuente del bucle de sondeo
            CFRUNLoopRef rl = CFRUNLoopGetCurrent();
            CFRUNLoopSourceRef rls;
            rls = CFSocketCreateRunLoopSource(NULL, s, 0);
            CFRUNLoopAddSource(rl, rls, kCFRunLoopDefaultMode);
            CFRUNRelease(rls);
            // Informa sobre la conexión recibida
            CFDataRef datos = CFSocketCopyPeerAddress(s);
        }
    }
}
```

```

        struct sockaddr_in dir_cliente;
        CFRRange r = CFRRangeMake(0,CFDataGetLength(datos));
        CFDataGetBytes(datos, r, (uint8_t*)&dir_cliente);
        printf("Nueva conexión desde el host %s y puerto %i\n",
               inet_ntoa(dir_cliente.sin_addr),
               ntohs(dir_cliente.sin_port));
        CFRelease(datos);
        CFRelease(s);
        break;
    }
    default:
        printf("Evento no contemplado\n");
        break;
}
}

int main (int argc, const char * argv[]) {
    // Recoge argumentos
    unsigned short puerto;
    if (argc<2) {
        fprintf(stderr, "Indique <puesto> servidor\n");
        return 1;
    }
    if (sscanf(argv[1],"%hi",&puerto) < 1) {
        fprintf(stderr, "Puerto %s no valido\n",argv[1]);
        return 1;
    }
    // Crea el socket servidor
    CFSocketRef ss;
    ss= CFSocketCreate(NULL,PF_INET,SOCK_STREAM,IPPROTO_TCP,
                       kCFSocketAcceptCallBack,
                       ProcesaSocketServidor,NULL);
    // Escucha en el puerto indicado
    struct sockaddr_in dir_servidor;
    bzero(&dir_servidor,sizeof(dir_servidor));
    dir_servidor.sin_len = sizeof(dir_servidor);
    dir_servidor.sin_family = AF_INET;
    dir_servidor.sin_port = htons(puerto);
    dir_servidor.sin_addr.s_addr = htonl(INADDR_ANY);
    CFDataRef datos = CFDataCreate(NULL, (uint8_t*)&dir_servidor,
                                   sizeof(dir_servidor));
    CFSocketSetAddress(ss, datos);
    CFRelease(datos);
    // Añade el socket como fuente del bucle de sondeo
    CFRunLoopRef rl = CFRunLoopGetCurrent();
    CFRunLoopSourceRef rls = CFSocketCreateRunLoopSource(NULL,ss,0);
    CFRunLoopAddSource(rl,rls,kCFRunLoopDefaultMode);
    CFRelease(rls);
    // Se mete en un bucle de sondeo
    CFRunLoopRun();
    return 0;
}

```

**Listado 9.11:** Programa servidor con CFNetwork

### 6.1.7. Habilitar y deshabilitar las funciones callback

Los eventos `kCFSocketReadCallBack`, `kCFSocketDataCallBack` y `kCFSocketAcceptCallBack`, por defecto se reactivan continuamente. Por el contrario, `kCFSocketWriteCallBack` y `kCFSocketConnectCallBack` son eventos que una vez producidos, por defecto se desactivan.

Este comportamiento se diseñó así para que cada vez que se pueda leer datos, o cada vez que aceptemos una nueva conexión al servidor, se nos pida realizar su operación correspondiente. El evento `kCFSocketWriteCallBack` no se reactiva continuamente porque para poder escribir necesitamos disponer de datos, y si a una aplicación se la pide escribir muchos datos, la aplicación podría no disponer de datos para escribir en el socket.

Para activar o desactivar un evento podemos usar la función:

```
void CFSocketEnableCallbacks (
    CFSocketRef s,
    CFOptionFlags callBackTypes);
```

En el parámetro `callBackFlags` indicamos qué evento queremos activar o desactivar de acuerdo a los flags vistos en la Tabla 9.2.

Conviene comentar que la posibilidad de deshabilitar un evento no existe en los objetos `NSStream` que estudiamos en el apartado 4. Por esta razón en el ejemplo del Listado 9.4 y Listado 9.5 implementamos las lecturas de forma asíncrona, pero implementamos las escrituras de forma síncrona. El hecho de que Core Foundation incluya la posibilidad de deshabilitar eventos nos permite implementar tanto las lecturas como las escrituras de forma asíncrona.

Por último, conviene comentar que aunque el comportamiento por defecto a la hora de reactivar los eventos es bastante razonable, podemos cambiar este comportamiento con la función:

```
void CFSocketSetSocketFlags (
    CFSocketRef s,
    CFOptionFlags flags);
```

Los valores que puede tomar `flags` se describen en la Tabla 9.3. En este caso, lo que indicamos no es si queremos que se active hasta el próximo disparo, sino si queremos que se reactive un evento automáticamente o bien haya que activarlo manualmente con `CFSocketEnableCallbacks()`.

|   |
|---|
| <code>kCFSocketAutomaticallyReenableReadCallBack</code> |
|---|

Cuando el flag está activo indica que queremos que el evento de lectura se reactive automáticamente. Si no está activo indica que no queremos que se reactive automáticamente.

|  |
|--|
| kCFSocketAutomaticallyReenableAcceptCallBack   |
| Cuando el flag está activo indica que queremos que el evento de aceptar clientes se reactive automáticamente. Si no está activo indica que no queremos que se reactive automáticamente.        |
| kCFSocketAutomaticallyReenableDataCallBack   |
| Cuando el flag está activo indica que queremos que el evento de lectura con objeto dato se reactive automáticamente. Si no está activo indica que no queremos que se reactive automáticamente. |
| kCFSocketAutomaticallyReenableWriteCallBack  |
| Cuando el flag está activo indica que queremos que el evento de escritura se reactive automáticamente. Si no está activo indica que no queremos que se reactive automáticamente.               |
| kCFSocketCloseOnInvalidate   |
| Indica que queremos que se cierre el descriptor de fichero cuando se invalide el socket Core Foundation. Por defecto está activo. Ver apartado 6.1.5.  |

**Tabla 9.3:** Flags de socket Core Foundation

## 6.2. CFStream

En el apartado 4 vimos la clase `NSStream` y sus derivadas `NSInputStream` y `NSOutputStream`. Estas clases tienen sus correspondientes objetos puentes representados por los tipos opacos `CFReadStreamRef` y `CFWriteStreamRef`. Existe una correspondencia casi directa entre los métodos de `NSStream` y las funciones Core Foundation de `CFStream`, con lo que no vamos a repetirlas.

En este apartado tan sólo vamos a añadir que `CFStream` nos permite crear objetos `CFStreamRef` a partir de un descriptor de fichero (p.e. de un socket), cosa que `NSStream` no nos permite. Para crearlo podemos usar:

```
void CFStreamCreatePairWithSocket (
    CFAllocatorRef alloc,
    CFSocketNativeHandle sock,
    CFReadStreamRef *readStream,
    CFWriteStreamRef *writeStream);
```

En el parámetro `sock` pasamos el descriptor de fichero, y en los parámetros de salida `readStream` y `writeStream` obtenemos los objetos `CFReadStreamRef` y `CFWriteStreamRef` correspondientes. Dado que son objetos puente, podemos hacerles casting respectivamente a `NSInputStream*` y `NSOutputStream*`.

## 6.3. CFHTTP y CFFTP

A través de los socket sólo recibimos streams de bytes. Protocolos de red muy usados como HTTP o FTP transmiten mensajes sencillos de texto a través del

socket. Sin embargo, Cocoa no proporciona clases para facilitar la construcción y parseo de estos mensajes. Afortunadamente Core Foundation sí que proporciona dos grupos de funciones con este fin: CFHTTP y CFFTP. Como es habitual, estas funciones tienen respectivamente los prefijos `CFHTTPXXX()` y `CFFTPXXX()`. Además, existen objetos puente Cocoa como `NSURL` o `NSData` que se pueden transformar en objetos `CFURLRef` y `CFDataRef` mediante un simple casting. En este documento sólo vamos a estudiar las funciones de CFHTTP. El lector puede completar el estudio de CFFTP en la documentación de Apple.

El protocolo HTTP suele constar de dos mensajes: Una petición (request) que envía el cliente al servidor, y una respuesta (response) que envía el servidor al cliente. En cualquier caso, todos los mensajes CFHTTP están representados por el tipo opaco `CFHTTPMessageRef`. En los siguientes apartados vamos a ver cómo componen y parsear estos mensajes el cliente y el servidor.

### 6.3.1. Cliente de HTTP

El cliente suele tener asignadas básicamente dos tareas: Componer la petición y parsear la respuesta del servidor.

Para crear la petición se empieza usando la función:

```
CFHTTPMessageRef CFHTTPMessageCreateRequest (
    CFAllocatorRef alloc,
    CFStringRef requestMethod,
    CFURLRef url,
    CFStringRef httpVersion);
```

El parámetro `requestMethod` suele ser GET o POST, y el parámetro `url` contiene la URL para la que estamos componiendo la petición.

Después debemos fijar las cabeceras con:

```
void CFHTTPMessageSetHeaderValue (
    CFHTTPMessageRef message,
    CFStringRef headerField,
    CFStringRef value);
```

Algunas peticiones, como POST, tienen también un cuerpo de mensaje, en cuyo caso fijamos este cuerpo con:

```
void CFHTTPMessageSetBody (
    CFHTTPMessageRef message,
    CFDataRef bodyData);
```

Finalmente podemos obtener un objeto dato `CFDataRef` con el contenido del mensaje `CFHTTPMessageRef` usando la función:

```
CFDataRef CFHTTPMessageCopySerializedMessage (
    CFHTTPMessageRef request);
```

El objeto dato devuelto contendrá el stream de bytes que se enviará por la red.

Por ejemplo, dada una `url`, para obtener los bytes de un mensaje GET a un servidor web podemos hacer:

```
CFHTTPMessageRef msg = CFHTTPMessageCreateRequest(NULL,
    CFSTR("GET"), url, kCFHTTPVersion1_1);
CFHTTPMessageSetHeaderValue(msg, CFSTR("host"),
    url_texto);
NSData* dato = (NSData*)
    CFHTTPMessageCopySerializedMessage(msg);
```

Después debemos enviar esta petición al servidor (p.e. usando un socket). Una vez enviada la petición, obtendremos un stream de bytes con la respuesta del servidor. La forma más sencilla de parsear esta respuesta es creando un objeto `CFHTTPMessageRef` vacío con la función:

```
CFHTTPMessageRef CFHTTPMessageCreateEmpty (
    CFAllocatorRef alloc,
    Boolean isRequest);
```

Y después rellenamos este mensaje con los bytes que vayan llegando mediante la función:

```
Boolean CFHTTPMessageAppendBytes (
    CFHTTPMessageRef message,
    const UInt8 *newBytes,
    CFIndex numBytes);
```

Debemos comprobar que esta función siempre devuelva `TRUE`. En caso de devolver `FALSE` significaría que el mensaje que estamos recibiendo no es un mensaje HTTP válido.

Para recoger todo el mensaje, debemos usar la función anterior hasta que hayamos parseado todo el mensaje. Por ejemplo, podemos convertir el stream de bytes de un `NSInputStream` representado por la variable `is` en un mensaje de respuesta `CFHTTPMessageRef` de la forma:

```
// Parsea la respuesta
msg = CFHTTPMessageCreateEmpty(NULL, FALSE);
do {
    uint8_t buffer[512];
    [is open];
    NSInteger n_bytes = [is read:buffer maxLength:512];
    if (n_bytes== -1) {
        fprintf(stderr,"Error leyendo del servidor\n");
```

```

        exit(1);
    }
    // Detecta el fin de mensaje
    if (n_bytes==0)
        break;
    if (!CFHTTPMessageAppendBytes(msg, buffer, n_bytes)) {
        fprintf(stderr,
            "El mensaje recibido no es HTTP valido\n");
        exit(1);
    }
} while (TRUE);

```

Una vez tengamos el mensaje parseado, podemos obtener sus distintas partes con las funciones:

```

CFStringRef CFHTTPMessageCopyResponseStatusLine (
    CFHTTPMessageRef response);

CFDictionaryRef CFHTTPMessageCopyAllHeaderFields (
    CFHTTPMessageRef message);

CFStringRef CFHTTPMessageCopyHeaderValue (
    CFHTTPMessageRef message,
    CFStringRef headerField);

CFDataRef CFHTTPMessageCopyBody (
    CFHTTPMessageRef message);

```

Conviene recordar que dado que los objetos `CFURLRef`, `CFHTTPMessageRef` y `CFDataRef` los hemos creado con funciones Core Foundation con el prefijo `Create` o `Copy`, debemos liberarlos con `CFRelease()`.

El Listado 9.12 muestra la implementación de un cliente de HTTP de acuerdo con los pasos previamente explicados. El programa crea unos objetos `NSInputStream` y `NSOutputStream` para hablar con el servidor de forma síncrona.

```

#import <Foundation/Foundation.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // Recoge argumentos
    if (argc<2) {
        fprintf(stderr, "Indique <url>\n");
        return 1;
    }
    CFStringRef url_texto = CFStringCreateWithCString(NULL,
        argv[1], kCFStringEncodingUTF8);
    CFURLRef url = CFURLCreateWithString(NULL, url_texto, NULL);
    CFStringRef nombre_host = CFURLCopyHostName(url);
    NSHost* host = [NSHost hostWithName: (NSString*)nombre_host];

```

```

// Abre una conexión al servidor
NSInputStream* is;
NSOutputStream* os;
[NSSetStream getStreamsToHost:host port:80 inputStream:&is
                           outputStream:&os];

// Compone el mensaje
CFHTTPMessageRef msg = CFHTTPMessageCreateRequest(NULL,
                                                    CFSTR("GET"), url, kCFHTTPVersion1_1);
CFHTTPMessageSetHeaderFieldValue(msg, CFSTR("host"),
                                 nombre_host);
NSData* dato = (NSData*) CFHTTPMessageCopySerializedMessage(msg);

// Envía el mensaje
[os open];
if ([os write:[dato bytes] maxLength:[dato length]] == -1) {
    fprintf(stderr,"Error escribiendo en el servidor\n");
    exit(1);
}
CFRelease(url_texto);
CFRelease(url);
CFRelease(nombre_host);
CFRelease(msg);
[dato release];

// Parsea la respuesta
msg = CFHTTPMessageCreateEmpty(NULL, FALSE);
do {
    uint8_t buffer[512];
    [is open];
    NSInteger n_bytes = [is read:buffer maxLength:512];
    if (n_bytes== -1) {
        fprintf(stderr,"Error leyendo del servidor\n");
        exit(1);
    }
    // Detecta el fin de mensaje
    if (n_bytes==0)
        break;
    if (!CFHTTPMessageAppendBytes(msg, buffer, n_bytes)) {
        fprintf(stderr,
                "El mensaje recibido no es HTTP valido\n");
        exit(1);
    }
} while (TRUE);

// Imprime la respuesta
NSString* str = (NSString*)
    CFHTTPMessageCopyResponseStatusLine(msg);
printf("Linea de estado:%s\n", [(NSString*)str UTF8String]);
[str release];
NSDictionary* dict =
    (NSDictionary*)CFHTTPMessageCopyAllHeaderFields(msg);
printf("Cabeceras:\n");
for (NSString* clave in dict) {
    NSString* valor = [dict valueForKey:clave];
    printf("%s:%s\n", [clave UTF8String], [valor UTF8String]);
}
[dict release];
printf("\nCuerpo:\n");
dato = (NSData*)CFHTTPMessageCopyBody(msg);

```

```

str = [NSString stringWithCString:[dato bytes]
                           length:[dato length]];
printf("%s\n", [str UTF8String]);
[dato release];

// Libera referencias
[is close];
[os close];
CFRelease(msg);
[pool drain];
return 0;
}

```

**Listado 9.12:** Cliente de HTTP

## 6.4. Servidor de HTTP

El servidor suele tener asignadas básicamente dos tareas: parsear la petición del cliente y componer la respuesta.

Parsear la petición es similar a parsear la respuesta: empezamos llamando a `CFHTTPMessageCreateEmpty()` y luego vamos añadiendo al mensaje los bytes que recibimos usando `CFHTTPMessageAppendBytes()`. Si sólo nos interesa recibir las cabeceras de la petición, pero no el cuerpo podemos usar la siguiente función para saber cuando hemos recibido las cabeceras:

```
Boolean CFHTTPMessageIsHeaderComplete (
    CFHTTPMessageRef message);
```

Una vez que terminamos de recibir el mensaje podemos usar las siguientes funciones para obtener el método de petición y la URL pedida:

```
CFStringRef CFHTTPMessageCopyRequestMethod (
    CFHTTPMessageRef request);

CFURLRef CFHTTPMessageCopyRequestURL (
    CFHTTPMessageRef request);
```

Y podemos obtener las cabeceras y cuerpo del mensaje con las funciones que vimos en el apartado anterior `CFHTTPMessageCopyHeaderFieldValue()` y `CFHTTPMessageCopyBody()`.

Para componer la respuesta empezamos llamando a:

```
CFHTTPMessageRef CFHTTPMessageCreateResponse (
    CFAllocatorRef alloc,
    CFIndex statusCode,
    CFStringRef statusDescription,
    CFStringRef httpVersion);
```

Y después rellenamos las cabeceras y cuerpo de la respuesta con:

```
CFHTTPMessageRef CFHTTPMessageCreateResponse (  
    CFAllocatorRef alloc,  
    CFIndex statusCode,  
    CFStringRef statusDescription,  
    CFStringRef httpVersion);  
  
void CFHTTPMessageSetBody (  
    CFHTTPMessageRef message,  
    CFDataRef bodyData);
```

En el Listado 9.7, Listado 9.8 y Listado 9.9 implementamos un servidor de daytime. En el Listado 9.13 hemos cambiado la implementación de la clase Servidor para implementar un servidor de HTTP. Dado que sólo varía el método `nuevaConexion`, en el Listado 9.13 sólo aparece este método. En este método primero parseamos la petición con las funciones que acabamos de estudiar, y luego componemos un mensaje de respuesta.

```

- (void)nuevaConexion:(NSNotification*)n {
    // Recoge la conexión
    NSDictionary* user_info = [n userInfo];
    NSFfileHandle* fh_conexion = [user_info
        objectForKey:NSFileHandleNotificationFileHandleItem];
    NSNumber* n_err = [user_info objectForKey:@"NSFileHandleError"];
    if (n_err) {
        perror("accept");
        return;
    }
    // Parsea la petición hasta el fin de las cabeceras
    CFHTTPMessageRef msg = CFHTTPMessageCreateEmpty(NULL, TRUE);
    do {
        NSData* dato = [fh_conexion availableData];
        CFHTTPMessageAppendBytes(msg, [dato bytes], [dato length]);
    } while(!CFHTTPMessageIsHeaderComplete(msg));
    NSString* method =
        (NSString*) CFHTTPMessageCopyRequestMethod(msg);
    NSString* url = (NSString*) CFHTTPMessageCopyRequestURL(msg);
    NSDictionary* dict =
        (NSDictionary*) CFHTTPMessageCopyAllHeaderFields(msg);

    // Compone la respuesta
    CFRelease(msg);
    msg = CFHTTPMessageCreateResponse(NULL, 200, CFSTR("Ok"),
                                      kCFHTTPVersion1_1);
    CFHTTPMessageSetHeaderValue(msg, CFSTR("Content-Type"),
                               CFSTR("text/html"));

    NSString* cuerpo_str = [NSString stringWithFormat:
    @"<html>" 
    "<head>" 
    "<title>Servidor HTTP basico</title>" 
    "<body>" 
    "<p>Method:<tt>%@</tt>" 
    "<p>URL:<tt>%@</tt>" 
    "<p>Cabeceras:" 
    "<p><tt>%@</tt>" 
    "</body>" 
    "</html>",

```

```

method,url,dict
];
NSData* cuerpo =
[cuerpo_str dataUsingEncoding:kCFStringEncodingISOLatin1
    allowLossyConversion:YES];
CFHTTPMessageSetBody(msg, (CFDataRef)cuerpo);
NSData* respuesta =
(NSData*) CFHTTPMessageCopySerializedMessage(msg);
[fh_conexion writeData:respuesta];
[fh_conexion closeFile];

// Libera referencias
CFRelease(msg);
[respuesta release];

// Pide otra notificación
[fhListen acceptConnectionInBackgroundAndNotify];
}

```

**Listado 9.13:** Método que implementa el servidor de HTTP

## 7. Sistema de carga de URLs

Dado que el acceso a URLs es una operación muy común, Mac OS X incluye un conjunto de clases que permiten realizar estas operaciones de forma sencilla.

Los protocolos que actualmente implementan el sistema de carga de URLs de Mac OS X son: file:///, http://, https:// y ftp://.

La forma más sencilla de acceder a contenido web es usando el método `factory stringWithContentsOfURL:encoding:error:` que proporcionan la clase `NSString`. La clase `NSData` también tiene un método `factory` similar. Por ejemplo:

```

NSError* error;
NSURL* url = [NSURL URLWithString:
    @"http://www.macprogramadores.org"];
NSString* texto_html = [NSString stringWithContentsOfURL:url
    encoding:NSUTF8StringEncoding error:&error];
if (error != nil) {
    NSLog(@"Error %@ obteniendo %@", [error description], url);
}

```

Este método, sin embargo, es limitado ya que no nos permite manejar cabeceras ni acceder de forma asíncrona a la red.

Para acceder a URLs de forma personalizada, en el resto de esta sección vamos a estudiar las clases `NSURLConnection` y `NSURLDownload`. La clase `NSURLConnection` se utiliza cuando queremos obtener el stream de bytes de

la conexión. En caso de que sólo queramos almacenar el contenido de la URL en un fichero local, resulta más sencillo usar `NSURLDownload`.

La clase `NSURL` se utiliza para representar un URL y acceder a las distintas partes de la URL: protocolo, host, recurso, etc. Los programadores Java encontrarán una similitud con la clase `java.net.URL`.

Las clases `NSURLRequest` y `NSURLResponse` se usan para representar respectivamente la petición que se haría al servidor y la respuesta del servidor. Además estas clases permiten almacenar las cabeceras del protocolo (en caso de usarse cabeceras en el protocolo).

## 7.1. Conexión síncrona

La forma más sencilla de obtener el contenido de una URL es mediante una conexión síncrona, la cual bloquea al hilo hasta obtener la respuesta o bien obtener un error. Para ello `NSURLConnection` proporciona el método de clase:

```
+ (NSData*)sendSynchronousRequest:(NSURLRequest*)request
                           returningResponse:(NSURLResponse**)response
                             error:(NSError**)error
```

Este método recibe en el parámetro `request` un objeto `NSURLRequest` con la URL del recurso a acceder y retorna un objeto `NSData` con los datos obtenidos de la conexión o `nil` si se produce un error. El método también retorna por referencia un objeto `NSURLResponse` con las cabeceras de respuesta.

El Listado 9.14 muestra un programa que carga de forma síncrona una URL. Las cabeceras que obtenemos varían dependiendo del protocolo de la URL a la que accedemos.

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
    NSURL* url = [NSURL URLWithString:@"http://www.google.es"];
    NSURLRequest* peticion = [NSURLRequest requestWithURL:url];
    NSURLResponse* respuesta;
    NSError* error;
    NSData* dato = [NSURLConnection sendSynchronousRequest:peticion
                                               returningResponse:&respuesta error:&error];
    if (dato==nil) {
        printf("Error: %s\n", [[error localizedDescription]
                               UTF8String]);
    } else {
        printf("Tipo MIME: %s\n", [[respuesta MIMEType] UTF8String]);
        printf("Longitud estimada: %lld\n",
               [respuesta expectedContentLength] );
        printf("Nombre fichero sugerido: %s\n",
               [respuesta suggestedFilename] );
    }
}
```

```

        [[respuesta suggestedFilename] UTF8String]);
printf("Encoding: %s\n",
       [[respuesta textEncodingName] UTF8String]);
printf("\n%s", [dato bytes]);
}
[pool drain];
return 0;
}

```

**Listado 9.14:** Carga síncrona de una URL

## 7.2. Conexión asíncrona

`NSURLConnection` nos permite obtener información sobre los datos recibidos según se van recibiendo. Para ello debemos de proporcionar un delegado al objeto `NSURLConnection`, y el delegado irá recibiendo notificaciones de la evolución de la conexión.

Para crear la conexión usamos el método `factory` de `NSURLConnection`:

```
+ (NSURLConnection*) connectionWithRequest:
    (NSURLRequest*) request delegate: (id) delegate
```

El cual crea la conexión y empieza a bajar datos. Este método además añade un puerto al bucle de sondeo del hilo que llama al método.

Cuando se obtenga la respuesta, en el delegado se ejecutará el siguiente método de respuesta, y en el parámetro `response` se recibirá la respuesta obtenida:

```
- (void) connection: (NSURLConnection*) connection
    didReceiveResponse: (NSURLResponse*) response
```

Los datos se obtendrán mediante una o más llamadas al siguiente método de respuesta del delegado:

```
- (void) connection: (NSURLConnection*) connection
    didReceiveData: (NSData*) data
```

Si se produce algún error durante la conexión se ejecutará el siguiente método de respuesta del delegado. En el parámetro `error` recibiremos una descripción del error:

```
- (void) connection: (NSURLConnection*) connection
    didFailWithError: (NSError*) error
```

En caso de no producirse errores, cuando la conexión acabe el delegado recibirá el mensaje:

```
- (void)connectionDidFinishLoading:
    (NSURLConnection*)connection
```

El Listado 9.15 muestra un programa que crea un objeto `NSURLConnection` para cargar una URL de forma asíncrona. El objeto delegado está implementado en el Listado 9.16 y Listado 9.17.

```
#import <Foundation/Foundation.h>
#import "Delegado.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSURL* url = [NSURL URLWithString:
        @"http://localhost/index.html"];
    NSURLRequest* peticion = [NSURLRequest requestWithURL:url];
    Delegado* d = [Delegado new];
    [NSURLConnection connectionWithRequest:peticion delegate:d];
    NSRunLoop* rl = [NSRunLoop currentRunLoop];
    [rl addPort:[[NSMachPort new] autorelease]
        forMode:NSTimerMode];
    while (![d termino]) {
        NSDate* timeout = [NSDate dateWithTimeIntervalSinceNow:1.0];
        [rl runUntilDate:timeout];
    }
    [pool drain];
    return 0;
}
```

### Listado 9.15: Carga asíncrona de una URL

```
#import <Foundation/Foundation.h>

@interface Delegado : NSObject {
    BOOL termino;
}
@property BOOL termino;
- init;
- (void)connection:(NSURLConnection*)connection
    didReceiveResponse:(NSURLResponse*)response;
- (void)connection:(NSURLConnection*)connection
    didReceiveData:(NSData*)data;
- (void)connectionDidFinishLoading:(NSURLConnection*)connection;
- (void)connection:(NSURLConnection*)connection
    didFailWithError:(NSError*)error;
@end
```

### Listado 9.16: Interfaz del delegado de carga asíncrona de URLs

```
#import "Delegado.h"

@implementation Delegado
@synthesize termino;
- init {
    if (self = [super init]) {
        termino = NO;
    }
    return self;
}
```

```

- (void)connection:(NSURLConnection*)connection
    didReceiveResponse:(NSURLResponse*)response {
    printf("Recibida respuesta:\n");
    printf("Tipo MIME: %s\n", [[response MIMEType] UTF8String]);
    printf("Longitud estimada: %lld\n",
           [response expectedContentLength] );
    printf("Nombre fichero sugerido: %s\n",
           [[response suggestedFilename] UTF8String]);
    printf("Encoding: %s\n\n",
           [[response textEncodingName] UTF8String]);

}
- (void)connection:(NSURLConnection*)connection
    didReceiveData:(NSData*)data {
    printf("%s", [data bytes]);
}
- (void)connectionDidFinishLoading:(NSURLConnection*)connection {
    termino = YES;
}
- (void)connection:(NSURLConnection*)connection
    didFailWithError:(NSError*)error {
    printf("Error:%s\n", [[error localizedDescription] UTF8String]);
    termino = YES;
}
@end

```

**Listado 9.17:** Implementación del delegado de carga asíncrona de URLs

### 7.3. Bajar ficheros

Cocoa proporciona la clase `NSURLDownload` especialmente pensada para bajar ficheros. El objeto se inicializa con el método:

- `(id)initWithRequest:(NSURLRequest*)request  
delegate:(id)delegate`

Al igual que `NSURLConnection`, también le pasamos la petición que queremos hacer y un objeto delegado. El método también registra el objeto en el bucle de sondeo del hilo. En ese momento se empieza el proceso de descarga.

En caso de producirse un error, el delegado recibirá el mensaje:

- `(void)download:(NSURLDownload*)download  
didFailWithError:(NSError*)error`

En caso contrario, el delegado recibirá mensajes de respuesta en el siguiente orden:

- `(void)downloadDidBegin:(NSURLDownload*)download`

Es el primer mensaje que se recibe nada más empezar el proceso de descarga. Este mensaje lo recibe el delegado incluso cuando el recurso no existe en la URL especificada.

Cuando se recibe la respuesta, el delegado recibe el siguiente mensaje con la respuesta obtenida:

- `(void)download: (NSURLDownload*) download  
didReceiveResponse: (NSURLResponse*) response`

A continuación el objeto `NSURLDownload` pedirá al delegado un path para el fichero local usando el siguiente método. En el parámetro `filename` se propone un nombre de fichero en base a la URL utilizada.

- `(void)download: (NSURLDownload*) download  
decideDestinationWithSuggestedFilename: (NSString*) filename`

Para indicar el path del fichero a recibir, el objeto delegado envía al objeto `NSURLDownload` el mensaje:

- `(void)setDestination: (NSString*) path  
allowOverwrite: (BOOL)allowOverwrite`

Cada vez que se reciba un bloque de datos, el delegado recibirá el siguiente mensaje. De esta forma podemos indicar al usuario el porcentaje de fichero recibido.

- `(void)download: (NSURLDownload*) download  
didReceiveDataOfLength: (NSUInteger) length`

En este caso los datos recibidos se escriben directamente a fichero, con lo que el delegado nunca es informado de qué datos se reciben, sólo de su tamaño.

Cuando el proceso de descarga acaba, el delegado recibe el mensaje:

- `(void)downloadDidFinish: (NSURLDownload*) download`

El Listado 9.18 muestra un programa que crea un objeto `NSURLDownload` para descargar un fichero. El Listado 9.19 y Listado 9.20 muestran la interfaz e implementación de su objeto delegado.

```
#import <Foundation/Foundation.h>
#import "Delegado.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSURL* url = [NSURL
        URLWithString:@"http://localhost/index.html"];
    NSURLRequest* peticion = [NSURLRequest requestWithURL:url];
```

```

Delegado* delegado = [Delegado new];
[[NSURLDownload alloc] initWithRequest:peticion
                                delegate:delegado];
NSRunLoop* rl = [NSRunLoop currentRunLoop];
[rl addPort:[[NSMachPort new] autorelease]
    forMode:NSDefaultRunLoopMode];
while (![delegado termino]) {
    NSDate* timeout = [NSDate dateWithTimeIntervalSinceNow:1.0];
    [rl runUntilDate:timeout];
}
[pool drain];
return 0;
}

```

**Listado 9.18:** Programa que descarga un fichero

```

#import <Foundation/Foundation.h>

@interface Delegado : NSObject {
    BOOL termino;
}
@property BOOL termino;
- (void)downloadDidBegin:(NSURLDownload*)download;
- (void)download:(NSURLDownload*)download
    didReceiveResponse:(NSURLResponse*)response;
- (void)download:(NSURLDownload*)download
    decideDestinationWithSuggestedFilename:(NSString*)filename;
- (void)download:(NSURLDownload*)download
    didCreateDestination:(NSString*)path;
- (void)download:(NSURLDownload*)download
    didReceiveDataOfLength:(NSUInteger)length;
- (void)downloadDidFinish:(NSURLDownload*)download;
- (void)download:(NSURLDownload*)download
didFailWithError:(NSError*)error;
@end

```

**Listado 9.19:** Interfaz de delegado que baja ficheros de URLs

```

#import "Delegado.h"

@implementation Delegado
@synthesize termino;
- init {
    if (self = [super init]) {
        termino = NO;
    }
    return self;
}
- (void)downloadDidBegin:(NSURLDownload*)download {
    printf("Comenzo la descarga\n");
}
- (void)download:(NSURLDownload*)download
    didReceiveResponse:(NSURLResponse*)response {
    printf("Recibida respuesta:\n");
    printf("Tipo MIME: %s\n", [[response MIMEType] UTF8String]);
    printf("Longitud estimada: %lld\n",
           [response expectedContentLength] );
    printf("Nombre fichero sugerido: %s\n",
           [[response suggestedFilename] UTF8String]);
}

```

```
    printf("Encoding: %s\n\n",
           [[response textEncodingName] UTF8String]);
}
- (void)download:(NSURLDownload*)download
decideDestinationWithSuggestedFilename:(NSString*)filename {
printf("Se nos propone el nombre de fichero: %s\n",
       [filename UTF8String]);
NSString* path = NSHomeDirectory();
path = [NSString stringWithFormat:@"%@/%@", path, filename];
[download setDestination:path allowOverwrite:YES];
printf("Proponemos el path:%s\n", [path UTF8String]);
}
- (void)download:(NSURLDownload*)download
didCreateDestination:(NSString*)path {
printf("Se creó el fichero %s\n", [path UTF8String]);
}
- (void)download:(NSURLDownload*)download
didReceiveDataOfLength:(NSUInteger)length {
printf("Se han recibido %i bytes\n", length);
}
- (void)downloadDidFinish:(NSURLDownload*)download {
termino = YES;
}
- (void)download:(NSURLDownload*)download
didFailWithError:(NSError*)error {
printf("Error:%s\n", [[error localizedDescription] UTF8String]);
termino = YES;
}
@end
```

**Listado 9.20:** Implementación de delegado que baja ficheros de URLs

# Tema 10

## Objetos distribuidos

---

### **Sinopsis:**

*Para acabar este tutorial en este tema vamos a estudiar los objetos distribuidos. Empezaremos con un ejemplo sencillo de objeto distribuido para luego ir profundizando en los detalles de su funcionamiento.*

## 1. Introducción

---

En los temas anteriores hemos visto varios mecanismos de comunicación entre hilos del mismo y de distintos procesos. El runtime de Objective-C permite implementar **objetos distribuidos**: un mecanismo de comunicación interprocess que permite ver y interactuar con objetos de otros procesos como si estuviesen en el espacio de memoria de nuestro proceso. Los objetos remotos pueden estar desplegados, o bien en otro proceso de nuestra propia máquina, o bien en otras máquinas.

Los objetos distribuidos se introdujeron el tutorial "El lenguaje Objective-C para programadores C++ y Java". Este tema extiende las explicaciones que allí introdujimos.

## 2. Crear y acceder a objetos remotos

---

Podemos dividir los objetos en **objetos locales**, que son objetos situados en nuestro propio proceso y **objetos remotos**, que son objetos situados en otros proceso. Hasta ahora sólo hemos creado objetos locales, en este tema vamos a ver cómo crear y acceder a objetos remotos.

### 2.1. Definir el objeto remoto

Vamos a empezar viendo un ejemplo sencillo de un objeto que publicamos como objeto remoto. Todo objeto remoto debe de adoptar un protocolo con las operaciones que exporta, es decir, que son accesibles remotamente. En el Listado 10.1 hemos definido en el fichero `ProtocoloCalculadora.h` un protocolo `Calculadora` con las operaciones de nuestro objeto. Este fichero de definición de protocolo es el único fichero que debe existir tanto en el proceso que actúa como servidor (es decir, que exporta el objeto remoto) como en el proceso cliente que accede al objeto remoto.

```
#import <Foundation/Foundation.h>

@protocol Calculadora
- (double) suma:(double)a y:(double)b;
- (double) resta:(double)a y:(double)b;
- (double) multiplica:(double)a y:(double)b;
- (double) divide:(double)a y:(double)b;
@end
```

**Listado 10.1:** Protocolo del objeto distribuido

En el proceso servidor es donde se implementa un objeto que adopta el protocolo. El Listado 10.2 y Listado 10.3 muestran la interfaz e implementación

de este objeto. Observe que el objeto `Calculadora` adopta el protocolo `Calculadora` para indicar que implementa sus operaciones. Recuérdese que en Objective-C el espacio de nombres de los protocolos es distintos al de las clases, con lo que protocolo y clase pueden tener el mismo nombre sin producirse conflictos. El proceso cliente no tiene porqué tener acceso a esta implementación, basta con conocer los métodos del protocolo `Calculadora`, es decir, con importar el fichero `ProtocoloCalculadora.h`.

```
#import "ProtocoloCalculadora.h"

@interface Calculadora : NSObject <Calculadora> {
}
-(double) suma:(double)a y:(double)b;
-(double) resta:(double)a y:(double)b;
-(double) multiplica:(double)a y:(double)b;
-(double) divide:(double)a y:(double)b;
@end
```

**Listado 10.2:** Interfaz del objeto remoto `Calculadora`

```
#import "Calculadora.h"

@implementation Calculadora
-(double) suma:(double)a y:(double)b {
    return a+b;
}
-(double) resta:(double)a y:(double)b {
    return a-b;
}
-(double) multiplica:(double)a y:(double)b {
    return a*b;
}
-(double) divide:(double)a y:(double)b{
    return a/b;
}
@end
```

**Listado 10.3:** Implementación del objeto remoto `Calculadora`

## 2.2. Exportar un objeto remoto

Como veremos en el apartado 3, cuando un cliente accede a un objeto remoto situado en otro proceso, se crean un par de objetos de tipo `NSConnection` que mantienen la conexión entre el proceso cliente y el proceso servidor: uno en el proceso cliente y el otro en el proceso servidor. En el Listado 10.4 hemos implementado el proceso servidor. Éste crea un objeto `Calculadora` y lo registra en la conexión por defecto del hilo principal. Cada hilo tiene una conexión por defecto que se obtiene con el método `factory singleton`:

```
+ (NSConnection*) defaultConnection
```

Cada conexión tiene asociado un objeto al que se llama **objeto raíz** (root object). Cuando la conexión recibe mensajes se los pasa a su objeto raíz. En nuestro ejemplo el objeto raíz es una instancia de `Calculadora`. Podemos fijar el objeto raíz de un objeto `NSConnection` con el método:

- `(void)setRootObject:(id)anObject`

Una vez que una conexión tiene asociado un objeto, debemos publicar la conexión para que otros objetos puedan encontrarla. En el apartado 4.2 veremos que hay varios registros de nombres. Para publicar un objeto simplemente le damos un nombre usando el método de `NSConnection`:

- `(BOOL)registerName:(NSString*)name`

El método devuelve si la operación de registro ha tenido éxito. Si ya existe otro objeto publicado con ese nombre, la operación fallaría.

Una vez que ejecutamos el método `factory singleton defaultConnection` de `NSConnection`, se crea un puerto y se registra en el bucle de sondeo del hilo. Este puerto es el lugar por donde se reciben mensajes remotos. Dado que el puerto ya está registrado en el bucle de sondeo del hilo, lo único que queda para que se empiecen a recibir estemos mensajes es ejecutar el método `run` del bucle de sondeo.

```
#import <Foundation/Foundation.h>
#import "Calculadora.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // Crea un objeto calculadora
    Calculadora* cal = [[Calculadora new] retain];
    // Obtiene la conexión por defecto
    NSConnection* conn = [NSConnection defaultConnection];
    // Fija al objeto raiz de la conexión por defecto
    [conn setRootObject:cal];
    // Publica la conexión por defecto con el nombre "calculadora"
    if ([conn registerName:@"calculadora"] == NO) {
        printf("No se puede registrar la calculadora\n");
        exit(1);
    }
    printf("Objeto remoto calculadora esperando conexiones\n");
    // Espera en un bucle de sondeo donde estara registrado el
    // puerto de la conexión al objeto remoto
    [[NSRunLoop currentRunLoop] run];
    [pool drain];
    return 0;
}
```

**Listado 10.4:** Proceso servidor que exporta un objeto remoto

## 2.3. Acceder al objeto remoto

El cliente que va a acceder a un objeto remoto sólo necesita el protocolo del objeto remoto (p.e. el protocolo `Calculadora` del Listado 10.1) y el nombre con el que está registrado (p.e. `@"calculadora"`). El Listado 10.5 muestra cómo implementar este proceso cliente.

Cuando el cliente va a conectarse a un objeto remoto no usa la conexión por defecto de su hilo, sino que crea un objeto `NSConnection` con el método `factory`:

```
+ (id)connectionWithRegisteredName:(NSString*)name
                           host:(NSString*)hostName
```

El cual recibe el nombre con el que está registrado el objeto remoto y el host donde está registrado. Para acceder a `@"localhost"` podemos pasar `nil` en el parámetro `hostName`.

```
#import <Foundation/Foundation.h>
#import "ProtocoloCalculadora.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // Obtiene una conexión a la calculadora
    NSConnection* conn = [NSConnection
        connectionWithRegisteredName:@"calculadora" host:nil];
    if (conn==nil) {
        printf("No se pudo conectar\n");
        exit(1);
    }
    id proxy = [conn rootProxy];
    [proxy setProtocolForProxy:@protocol(Calculadora)];
    double c = [proxy suma:3.0 y:4.5];
    printf("El objeto remoto responde:%lf",c);
    [pool drain];
    return 0;
}
```

**Listado 10.5:** Proceso cliente que accede a un objeto remoto

En el tutorial "El lenguaje Objective-C para programadores C++ y Java" explicamos que el proceso cliente no accede directamente al objeto remoto, sino que en su espacio de memoria se crea un **objeto proxy** que envía y recibe mensajes a y desde el objeto remoto. Para obtener el proxy de la conexión podemos usar el método:

```
- (NSDistantObject*)rootProxy
```

El objeto proxy devuelto es un objeto al que vamos a poder enviar mensajes, y estos mensajes se transmiten por la conexión hasta el objeto remoto. Dado que en principio el proxy no sabe qué mensajes acepta el objeto remoto, se tiene que realizar un intercambio de mensajes por la conexión para pasar esta

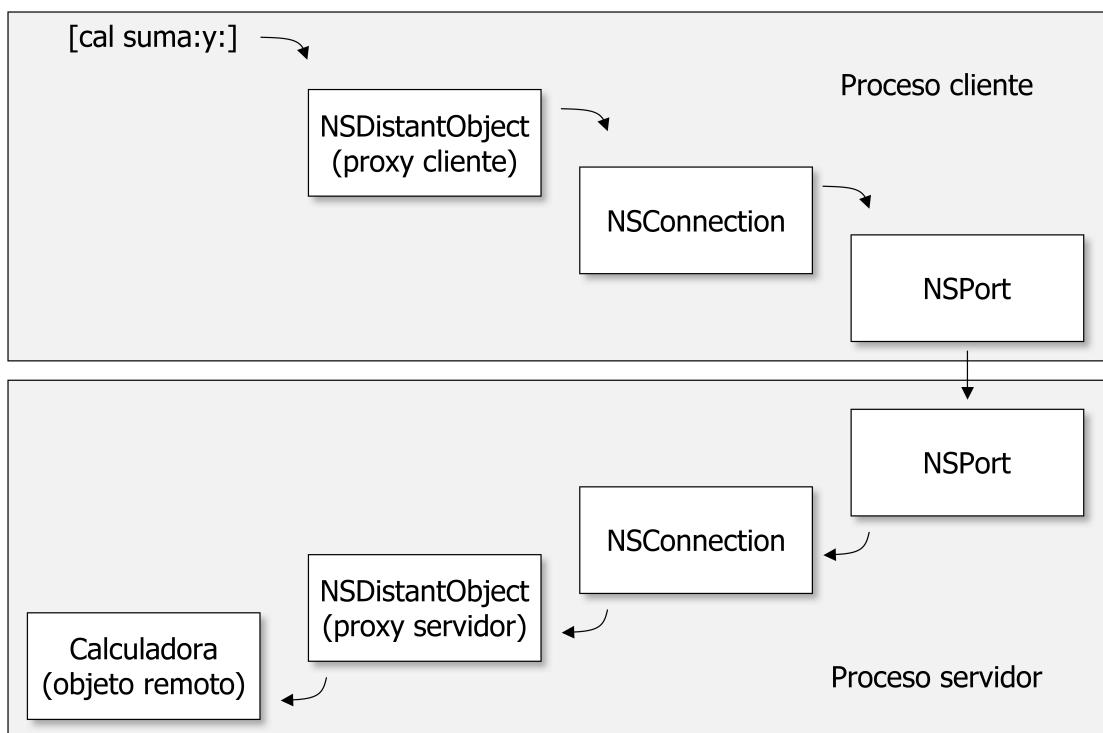
información. Podemos evitar este intercambio de mensajes, y optimizar así el rendimiento de la aplicación, informando al proxy de forma local sobre los mensajes que se le van a enviar con el método:

```
- (void)setProtocolForProxy:(Protocol*)aProtocol
```

En el Listado 10.5 vemos que podemos enviar el mensaje `suma:y:` al objeto proxy, y este se encarga de enviarlo por la conexión y obtener el resultado.

### 3. Arquitectura, conexiones y proxies

La Figura 10.1 muestra los tipos de objetos implicados en una llamada a un método de un objeto remoto.



**Figura 10.1:** Objetos implicados en un mensaje a un objeto remoto

Al más bajo nivel, la conexión entre cliente y servidor se realiza a través de un puerto situado en el cliente y otro situado en el servidor. En el apartado 4 veremos que Cocoa permite la comunicación a través de tres tipos de puertos: puertos Mach, puertos de mensajes y sockets.

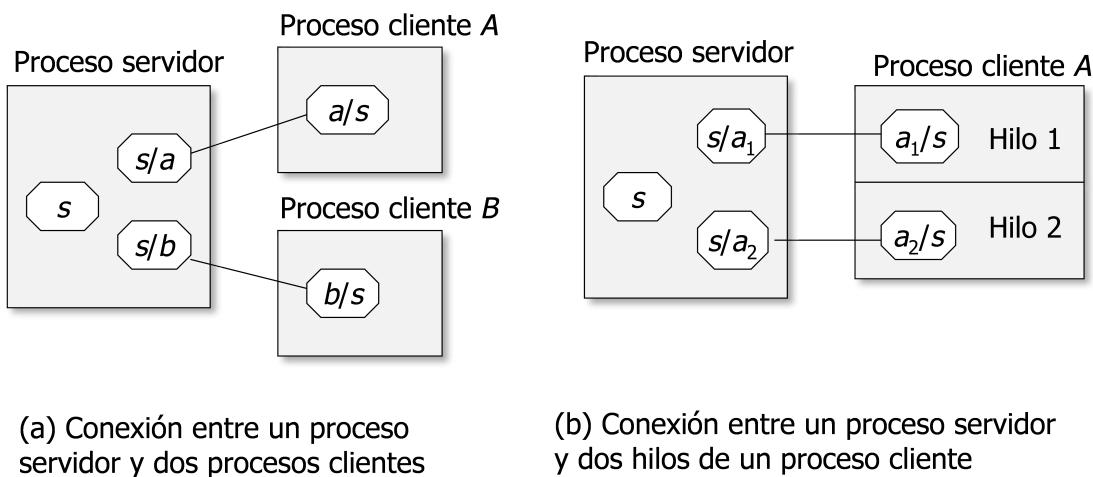
Al nivel más alto encontramos los proxies, que están representados como instancias de `NSDistantObject`. En el proceso cliente el proxy se encarga de recibir mensajes Objective-C y enviarlos a través de la conexión. En el proceso servidor el proxy se encarga de recoger los mensajes de la conexión y de enviar el mensaje Objective-C correspondiente al objeto remoto.

### 3.1. Las conexiones

El acceso a un objeto remoto se realiza mediante dos objetos `NSConnection`, uno situado en el cliente y otro situado en el servidor. La existencia de estos objetos nos aísla de conocer el tipo de puerto que se está usando para conectar el proxy del cliente con el proxy del servidor. Normalmente el programador de aplicaciones nunca enviará o recibirá datos por la conexión, esta tarea se reserva a los proxies. El único momento en el que el programador de aplicaciones tiene acceso al objeto conexión es durante el establecimiento de la conexión. A partir de ese momento el proxy se encargará de realizar el resto del trabajo.

La Figura 10.2 muestra cómo las conexiones siempre se realizan entre pares de objetos. El objeto `s` es la conexión por defecto del hilo servidor que obtiene el proceso servidor con el método `defaultConnection`. Esta conexión es donde se registra el objeto y es la única que no tiene pareja.

Una vez que el proceso cliente se conecta al proceso servidor se crea un par de objetos conexión: uno en el cliente y otro en el servidor. En la Figura 10.2 (a) se muestran dos conexiones con dos procesos cliente distintos `A` y `B`. Estas conexiones están etiquetadas respectivamente como `s/a-a/s` y `s/b-b/s`. En la Figura 10.2 (b) se muestra un proceso cliente `A` con dos hilos. En este caso cada hilo ha establecido una conexión con el proceso remoto distinta `s/a1-a1/s` y `s/a2-a2/s`. Hasta Mac OS X 10.5 las conexiones no se podían compartir aunque ambas conexiones estuviesen en el mismo proceso. A partir de Mac OS X 10.5 se cambio este comportamiento para que sólo se cree una conexión en cada proceso compartida por todos los hilos.



**Figura 10.2:** Conexiones remotas

A veces se crean conexiones en las que el cliente y el servidor están en el mismo proceso. Esta forma de conectar los objetos locales a través de objetos remotos permite a un objeto recibir mensajes desde distintos hilos, es

decir crea una protección multihilo. Concepto que explicamos en el apartado 1.8 del Tema 6.

Es importante tener en cuenta que siempre que se crea un objeto conexión, éste registra su puerto asociado en el bucle de sondeo del hilo que lo crea. Para poder recibir mensajes el hilo que creó la conexión deberá activar su bucle de sondeo llamando a algún método `runXXX`. En el caso del hilo que envía mensajes a través del proxy de cliente no es necesario que su hilo active el bucle de mensajes siempre que no pasemos al proceso servidor objetos por referencia, a los cuales luego el proceso servidor podría enviar mensajes.

## 3.2. Los proxies

En el tutorial "El lenguaje Objective-C para programadores C++ y Java" introdujimos las responsabilidades de los objetos proxy e indicamos sus diferencias con los objetos delegado: La principal responsabilidad de los objetos proxy es recibir mensajes y transmitirlos al objeto real que están representando.

Los proxies están representados por la clase abstracta `NSProxy`. Esta clase se usa como base para todos los tipos de proxies, y es una de las pocas clases que no deriva de la clase `NSObject` (aunque implementa el protocolo `NSObject`). `NSProxy` implementa los métodos básicos del runtime de Objective-C, y por eso `NSProxy` es una clase raíz. Sin embargo, como `NSProxy` es una clase abstracta, no implementa ningún método de inicialización (no se puede instanciar), sino que deja esta responsabilidad a sus clases derivadas.

El sistema de objetos distribuidos usa dos derivadas de `NSProxy`: La clase `NSDistantObject` y la clase `NSProtocolChecker`. Ambas clases redefinen los siguientes métodos, aunque con propósitos diferentes:

- `(void)forwardInvocation: (NSInvocation*) anInvocation`
- `(NSMethodSignature*) methodSignatureForSelector: (SEL)aSelector`
- + `(BOOL)respondsToSelector: (SEL)aSelector`

`NSDistantObject` redefine estos métodos para transmitir o recibir llamadas a métodos. `NSProtocolChecker` redefine estos métodos para permitir o denegar el acceso a estos métodos. Vamos a ver a continuación con más detalle cada una de estas clases.

### 3.2.1. La clase `NSDistantObject`

La clase `NSDistantObject` representa dos tipos de proxies (que aparecen en la Figura 10.1): El proxy cliente y el proxy servidor. El **proxy cliente** se en-

carga de recibir mensajes y retransmitirlos al otro extremo a través de su conexión. el **proxy servidor** se encarga de recibir peticiones y transmitirlas a un objeto local, que es el objeto remoto instalado en el servidor.

El proxy servidor se inicializa ejecutando en el proceso servidor el método:

- `(id) initWithLocal: (id) anObject  
connection: (NSConnection*) aConnection`

Donde `anObject` es el objeto remoto del espacio de memoria del proceso servidor (p.e. una instancia de `Calculadora`).

Por su parte, el proxy cliente se inicializa con el método:

- `(id) initWithTarget: (id) remoteObject  
connection: (NSConnection*) aConnection`

Este método se suele ejecutar también en el proceso servidor y el parámetro `remoteObject` es el objeto que actúa como proxy servidor.

Una vez que el proxy cliente se crea en el servidor, se transporta hasta el cliente. Esta es la razón por la que `NSDistantObject` implementa el protocolo `NSCoding`, para poder ser archivado y transportado hasta el proceso cliente. El objeto que se encarga de realizar este archivado es una instancia de `NSPortCoder`.

Normalmente el programador de aplicaciones nunca usará estos métodos de inicialización para crear los proxies, sino que el proceso servidor simplemente registrará el objeto remoto tal como muestra el Listado 10.4 y cuando se reciba una petición en el bucle de sondeo se creará un proxy cliente que se trasmítirá hasta el proceso cliente. Por su parte el proceso cliente, tal como muestra el Listado 10.5, se limitará a llamar al método `rootProxy` de la conexión, y este método se encarga de transportar el proxy cliente hasta el proceso cliente y de retornarlo en la llamada a `rootProxy`.

El resto de mensajes (instancias de `NSInvocation`) también se transmiten archivadas usando un objeto `NSPortCoder`, pero este proceso lo realizan los objetos `NSConnection` y también es un proceso transparente al programador de aplicaciones.

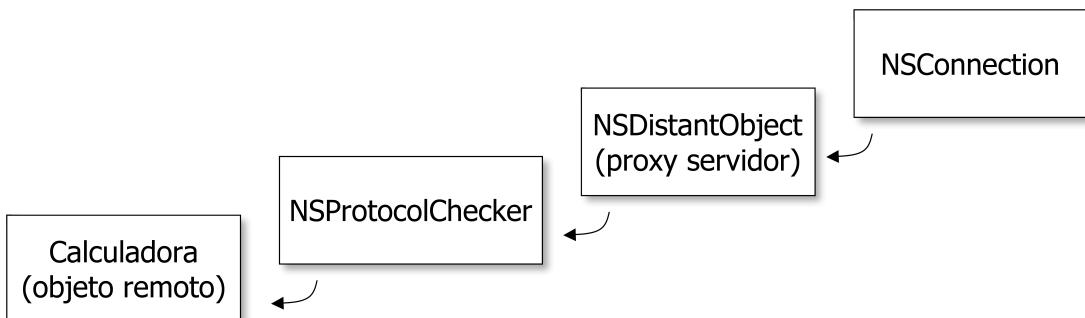
Por último conviene recordar que la clase `NSDistantObject` dispone del método:

- `(void) setProtocolForProxy: (Protocol*) aProtocol`

El cual evita tener que intercambiar información sobre la signatura de los métodos, con lo que es muy conveniente que el proceso cliente use este método antes de empezar a enviar mensajes a través del proxy cliente.

### 3.2.2. La clase NSProtocolChecker

Cuando se publica un objeto remoto todos sus métodos (heredados o añadidos) son accesibles. A veces queremos limitar los métodos de un objeto que pueden ser accedidos. En principio, esto se puede querer hacer tanto en local como en remoto, pero normalmente un programa no suele limitar los métodos de sus objetos locales. La Figura 10.3 muestra cómo una instancia de `NSProtocolChecker` se ha colocado entre el proxy servidor y el objeto remoto para controlar los métodos que se ejecutan del objeto servidor.



**Figura 10.3:** Control de métodos accesibles remotamente

Para crear un `NSProtocolChecker` podemos usar su método factory:

```
+ (id)protocolCheckerWithTarget: (NSObject*)anObject
                           protocol: (Protocol*)aProtocol
```

El método recibe en `anObject` el objeto remoto y en `aProtocol` los métodos que se permiten ejecutar.

Después, en vez de que el proceso remoto use el método de la conexión `setRootObject:` para registrar directamente el objeto remoto, registraría la instancia de `NSProtocolChecker`.

Ahora si el cliente intentase ejecutar un método que no está en el protocolo, recibiría una excepción de tipo `NSInvalidArgumentException`.

## 4. Los puertos

---

Hilos situados en distintos procesos se comunican a través de puertos, los cuales están representados por la clase abstracta `NSPort`. Esta clase tiene tres derivadas que representan los tres tipos de puertos que soporta Cocoa para la programación distribuida:

Los **puertos de mensajes Core Foundation** `NSMessagePort` sólo permiten enviar mensajes entre procesos de la misma aplicación. La aplicación debe de usar Core Foundation y los mensajes son del tipo `CFMessagePortRef`. Dado que esta forma de comunicación sólo sirve para aplicaciones Core Foundation, Apple empezó a desaconsejar esta forma de comunicación en Mac OS X 10.5.

Los **puertos de mensajes Mach** `NSMachMessage`) son puertos Mach que sólo permiten intercambiar mensajes entre procesos de la misma máquina. Dado que los puertos Mach son una forma básica de comunicación que proporciona el núcleo Mach, todos los tipos de aplicaciones Mac OS X pueden enviar y recibir estos mensajes.

Los **puertos de sockets** `NSSocketPort` son los únicos que permiten enviar mensajes entre procesos situados en distintas máquinas y están implementados mediante sockets.

Normalmente se crea un puerto local y se registra con un nombre. El proceso remoto crea otro puerto que se conecta al primero proporcionado el nombre con el que se registró.

### 4.1. Envío y recepción de mensajes

Una vez establecida la conexión se pueden enviar mensajes por el puerto usando el método:

```
- (BOOL)sendBeforeDate:(NSDate*)limitDate  
                  msgid:(NSUInteger)msgID  
             components:(NSMutableArray*)components  
                 from:(NSPort*)receivePort  
            reserved:(NSUInteger)headerSpaceReserved
```

Cada derivada de `NSPort` implementa este método de una forma, pero el objetivo es enviar antes de la fecha dada en `limitDate`, el mensaje cuyo código es `msgid` y cuyo valor se proporciona en `components`. Si este mensaje tiene respuesta, la respuesta deberá obtenerse por el puerto dado en `receivePort`.

Para recibir mensajes se suele registrar el puerto en el bucle de sondeo usando el método:

```
- (void)scheduleInRunLoop:(NSRunLoop*)runLoop  
forMode:(NSString*)mode
```

Cuando el objeto `NSPort` recibe un mensaje, informa a su delegado. El delegado se fija enviando al objeto puerto el mensaje:

```
- (void)setDelegate:(id)anObject
```

Después, en el caso de `NSMachPort`, esta clase envía a su delegado el mensaje:

```
- (void)handleMachMessage:(void*)machMessage
```

En `machMessage` se pasa una estructura de tipo `msg_header_t` con el mensaje Mach recibido.

En el caso de los objetos de tipo `NSMessagePort` o un `NSSocketPort`, informan a su delegado con el mensaje:

```
- (void)handlePortMessage:(NSPortMessage*)portMessage
```

Y en `portMessage` se pasa un objeto de tipo `NSPortMessage` con el mensaje (no confundir con `NSMessagePort`).

Si no hubiéramos fijado delegado, el puerto actúa como delegado de sí mismo.

Normalmente el delegado es un objeto de tipo `NSConnection`, y el programador de aplicaciones no tiene que interactuar con objetos `NSPort` para conectar aplicaciones. En cualquier caso conocer de su existencia ayuda a entender mejor la forma en que funcionan los objetos distribuidos.

## 4.2. Registrar los puertos

Cuando registramos un objeto distribuido, realmente lo que estamos haciendo es registrar su puerto.

Existe un registro para cada tipo de puerto. La clase `NSPortNameServer` es una clase abstracta donde sus derivadas representan los distintos puertos. La clase `NSMessagePortNameServer` representa el registro de puertos Core Foundation. La clase `NSMachBootstrapServer` representa el registro de puertos Mach. La clase `NSSocketPortNameServer` representa el registro de

puertos de socket. Para obtener un registro de un determinado tipo simplemente ejecutamos sobre su clase el método factory singleton:

```
+ (id)sharedInstance
```

Independientemente del registro con el que estemos trabajando, los siguientes métodos son los que permiten registrar y desregarstrar puertos:

- (BOOL)registerPort:(NSPort\*)aPort name:(NSString\*)portName
- (BOOL)removePortForName:(NSString\*)portName

Para buscar puerto en el registro se usan los métodos:

- (NSPort\*)portForName:(NSString\*)portName
- (NSPort\*)portForName:(NSString\*)portName host:(NSString\*)hostName

Ambos métodos reciben el nombre del puerto a buscar. El segundo método también recibe el host donde buscarlo, y sólo se usa para puertos de sockets.

## 5. Autorizar conexiones

---

Un objeto servidor puede limitar las conexiones que recibe. Cuando la conexión por defecto del hilo servidor (el objeto *s* en la Figura 10.2) recibe un cliente que pide una nueva conexión, la conexión por defecto envía a su delegado (que deberá haber sido fijado con `setDelegate:`) el mensaje:

- `(BOOL)connection:(NSConnection*)parentConnection  
shouldMakeNewConnection:(NSConnection*)newConnnection`

Este mensaje sólo se envía al delegado si éste acepta el mensaje. La conexión por defecto del hilo servidor puede consultar si su delegado acepta el mensaje ejecutando sobre su delegado el método `respondsToSelector::`.

El método anterior sólo se recomienda usarlo cuando queremos poder aceptar o rechazar conexiones. Si lo único que queremos es saber cuando llega un nuevo cliente, es mejor pedir al centro de notificación por defecto ser informados de la notificación `NSConnectionDidInitializeNotification`. Esta notificación la lanza la conexión por defecto del hilo servidor nada más aceptar una nueva conexión. Es decir, podemos hacer:

```
Delegado* delegado = [Delegado new];  
[[NSNotificationCenter defaultCenter]  
addObserver:delegado  
selector:@selector(nuevaConexion:)  
name:NSConnectionDidInitializeNotification  
object:conn];
```

De esta forma se ejecutaría el método `nuevaConexion:` en el objeto delegado cuando la conexión por defecto del hilo servidor `conn` acepte una nueva conexión.

## 6. Manejo de errores en la conexión

Dado que en cualquier momento se puede romper la conexión de red, las aplicaciones que quieran ser robustas a errores deben de contemplar esta posibilidad.

La primera medida que pueden tomar las aplicaciones es envolver sus llamadas a métodos remotos en bloques `@try-@catch`. En concreto, una llamada a un método remoto puede lanzar una de estas excepciones:

`NSPortTimeoutException`, se produce cuando el puerto por el que se envía el mensaje no está roto, pero tarda mucho en responder.

`NSInvalidSendPortException`, se produce cuando se va a llamar a un método y se detecta que el puerto de envío está roto.

`NSInvalidReceivePortException`, se produce cuando se va a llamar a un método y se detecta que el puerto de recepción está roto.

Por ejemplo, la llamada a método remoto del Listado 10.5 la podemos proteger de la forma:

```
double c;
@try {
    c = [proxy suma:3.0 y:4.5];
} @catch (NSException* ex) {
    if ([[ex name] isEqualToString:NSPortTimeoutException]
        || [[ex name] isEqualToString:NSInvalidSendPortException]
        || [[ex name] isEqualToString:NSInvalidReceivePortException]) {
        printf("%s\n", [[ex reason] UTF8String]);
        exit(1);
    } else {
        [ex raise];
    }
}
```

Dado que envolver cada llamada a método remoto en un bloque `@try-@catch` resulta tedioso, siempre podemos crear un sólo bloque `@try-@catch` a un nivel superior.

La segunda alternativa consiste en escuchar las notificaciones del objeto conexión. Cuando los objetos `NSConnection` detectan que la conexión con su peer se ha perdido envian al centro de notificación por defecto la notificación `NSConnectionDidDieNotification`. Cualquier otra parte de nuestro programa puede pedir al centro de notificación por defecto que le informe sobre esta notificación. La forma de pedir esta notificación podría ser:

```
Delegado* delegado = [Delegado new];
[[NSNotificationCenter defaultCenter] addObserver:delegado
    selector:@selector(conexionHaMuerto:)]
```

```
name : NSConnectionDidDieNotification  
object : conn];
```

Donde la clase `Delegado` implementa el método `conexionHaMuerto:` que es donde recibirá la notificación lanzada por el objeto `conn`.

Usando estas técnicas un cliente puede detectar y manejar errores de forma robusta. Por desgracia, no ocurre lo mismo con los servidores. El servidor es un objeto pasivo con lo que no puede detectar que la conexión con un cliente se ha perdido, es decir, cuando se pierde la conexión con un cliente no se lanza en el servidor la notificación `NSConnectionDidDieNotification`. Aunque el servidor no reaccione ante la ocurrencia de errores en las conexiones con sus clientes, los clientes sí que se enterarán de esta circunstancia usando las técnicas que hemos visto en este apartado.