Sistemas Distribuidos - Lab

Tobías Díaz, José Luis Segura

Apr 15, 2023

CONTENTS

Esta documentación contiene material e información necesaria para realizar las prácticas de la asignatura Sistemas Distribuidos del curso 2022/2023.

Esta documentación también está disponible en PDF

Profesores

Tobias Díaz B2 - Lunes 17:00h - Edsger W. Dijkstra (0.05 + 6) - Edificio Politécnico

A1 - Martes 17:00h - John Carmack (LD4) - Escuela Superior de Informática

A2 - Martes 18:30h - John Carmack (LD4) - Escuela Superior de Informática

José Luis Segura C2 - Martes 18:30 - Dennis Ritchie (LD2) - Escuela Superior de Informática

C1 - Martes 20:00h - John Carmack (LD4) - Escuela Superior de Informática

B1 - Jueves 18:30h - Dennis Ritchie (LD2) - Escuela Superior de Informática

Contenido

Las sesiones de laboratorio se centrarán en conocer y aprender la tecnología de **invocación remota de métodos**, o **RMI** (*Remote Method Invocation*) utilizando para ello el *middleware* ZeroC Ice.

En todas las sesiones se presentarán ejemplos de diferente dificultad para ser implementados en clase. También servirán para familiarizarnos con algunos de los servicios proporcionados por el *middleware*, que se irán viendo en sesiones independientes.

La asistencia a las clases de laboratorio **no es obligatoria**, aunque se valorará la participación en clase y posibles actividades optativas para mejorar la nota.

Entorno de ejecución

El entorno de laboratorio estará basado en GNU/Linux y todos los ejemplos de código se realizarán utilizando el lenguaje Python en su versión **3.10**.

Evaluación

- La nota de prácticas corresponde al 35% de la nota de la asignatura (35 puntos de 100).
- Las prácticas se consideran de caracter obligatorio, lo que obliga a sacar un 40% de la nota (14 puntos).

CHAPTER

ONE

INTRODUCCIÓN A PYTHON 3

1.1 ¿Qué es Python?

Es un lenguaje de programación interpretado, desarrollado por Guido Van Rossum. Se trata de un lenguaje que soporta varios paradigmas de programación: programación orientada a objetos, programación imperativa, programación funcional y, recientemente, programación asíncrona.

1.1.1 Versiones de Python

Como muchos otros lenguajes y software en general, la versión de Python viene determinada por la versión *major* y la *minor*. Se distinguen de forma muy marcada Python 2 y Python 3. La versión 2 dejó de tener soporte de seguridad desde hace unos años, por lo que nos centraremos en la versión 3.

Dentro de cada versión, la versión *minor* nos indica si están soportadas nuevas características del lenguaje. A la escritura de este manual, la versión más reciente es **Python 3.10.7**.

1.1.2 Especificación del estándar

El desarrollo de Python está fuertemente dirigido por la comunidad a través de documentos llamados *Python Enhancement Proposals* o **PEP**. En ellos se documentan los acuerdos llegados para definir diferentes aspectos del lenguaje, como el PEP 100 que especifica la integración de Unicode en PYthon, y algunos en clave de humor, como el PEP 20, The Zen of Python.

1.1.3 El intérprete de Python

Llamamos **intérprete** a cualquier programa que sea capaz de, tomando como entrada un programa escrito en Python, reproducir su ejecución. El intérprete por defecto y más habitual de encontrar es **CPython**, el cuál se distribuye bajo una licencia libre.

CPython es la implementación que es distribuída desde la web oficial de Python y la que suele ser distribuida por las diferentes distribuciones de GNU/Linux en sus sistemas de empaquetado.

Al ser software libre, **CPython** soporta multitud de plataformas, como Linux, Windows, BSD, Darwin, MacOS e incluso algunas plataformas como PS3, XBMC o Nintendo DS. Las plataformas soportadas están documentadas en el PEP 11.

También existen intérpretes alternativos que pueden ser instalados y utilizados:

- Iron Python: implementado en C# e integrado en la plataforma .NET.
- Jython: implementado en Java, permite ejecutar programas Python en la JVM.
- PyPy: intérprete implementado en… ¡Python!

- Nuitka: compilador (no intérprete), escrito en Python y que permite generar ficheros ejecutables binarios para la plataforma deseada.
- MicroPython: desarrollado específicamente para ser utilizado en microcontroladores.

1.2 Introducción al lenguaje

En este apartado no vamos a hablar de cómo desarrollar en Python desde 0, ya que es un lenguaje que ya se ha utilizado con anterioridad en la carrera. Sin embargo, si que haremos hincapié en algunas características del lenguaje que nos serán imprescindibles para su uso en la asignatura de Sistemas Distribuidos.

1.3 Definición de clases

Como mencionamos anteriormente, Python es un lenguaje que permite utilizar la programación orientada a objetos.

Para definir una clase:

```
class MyClass:
    def method1(self):
        pass

    def method2(self, argument):
        pass
```

Con el código anterior se definirá una clase llamada `MyClass`, con 2 métodos diferentes. El primero no aceptará ningún argumento, y el segundo, aceptando un argumento. Como puede verse, ninguno de los 2 métodos define ni el tipo de retorno del mismo ni, en el caso del segundo, el tipo que debe tener el parámetro `argument`. Ésto se debe a que en Python no se comprueban en ningún momento (ni en tiempo de compilación ni de ejecución) los valores devueltos por una función o método, ni los tipos de los argumentos que se van a pasar. Hay que ser cuidadoso con ello, ya que de utilizar un parámetro de un tipo diferente al que el método espera recibir puede provocar errores en tiempo de ejecución.

1.3.1 Creando una instancia

Para crear una instancia de la clase definida en el ejemplo anterior, tenemos que realizar lo siguiente:

```
instance = MyClass()
```

A partir de ese momento, la variable `instance` contendrá una referencia a un objeto de tipo `MyClass`, por lo que podemos invocar a cualquiera de sus métodos o utilizar sus atributos.

Siempre que instanciamos un objeto, se llevan a cabo 2 procesos diferentes:

- Creación de la nueva instancia, a través del método especial `__new__`.
- Inicialización de la nueva instancia, a través del método especial `__init__`.

Mientras que el primero nos devuelve una instancia vacía, el segundo se encarga de añadir los atributos necesarios a la instancia, siendo éste el que normalmente utilizaremos en nuestras clases para definir sus atributos.

1.3.2 El argumento `self`

Si nos fijamos bien en el ejemplo anterior, ambos métodos tienen definido un primer argumento denominado `self`. De forma muy general, podemos considerar a `self` el equivalente al `this` de Java: es la referencia a la instancia de la que vamos a ejecutar el código.

1.3.3 Atributos de una clase

Las clases, además de métodos, pueden tener atributos. Éstos atributos son los que harán que 2 diferentes instancias sean, en efecto, diferentes. Los atributos de una clase pueden ser estáticos (creados en la definición de la clase) o dinámicos (creados en el inicializador de la clase).

En cuanto a la visibilidad, en Python no existen los atributos privados como tal, aunque si que existen algunos mecanismos y convenciones que permiten, hasta cierto punto, hacer privados los atributos:

- Si un atributo comienza por un _ (guión bajo o *underscore*), se considera que el atributo no debe ser utilizado directamente por los usuarios de la instancia. Se trata de una simple convención, por lo que el intérprete de Python no realiza ninguna acción.
- Si un atributo comienza por __ (doble *underscore*), Python realiza una modificación denominada "*name mangling*", en la cual se modifica el nombre del atributo de manera que los usuarios de las instancias no tengan ningún tipo de acceso sobre él de manera directa. Si se conoce bien el mecanismo, es posible acceder a los atributos declarados de esta manera, pero al igual que en el caso anterior, **no se debe**.

A continuación se muestra una modificación de nuestra `MyClass` para añadir diferentes atributos:

```
class MyClass:
    def __init__(self):
        self.public = "this is a normal attribute"
        self._dont_use = "this is a normal attribute, but please don't use it"
        self.__hidden = "this attribute will have name mangling"

    def show_attributes(self):
        print(f"Internal access to attributes in {self.__class__.__name__}")
        print(f"normal = {self.normal}")
        print(f"don't use = {self._dont_use}")
        print(f"hidden = {self.__hidden}")

a = MyClass()
a.show_attributes()

print(f"Direct access to attributes in {a.__class__.__name__}")
print(f"public={a.public}")
print(f"don't use={a._dont_use}")
print(f"internal={a._MyClass__hidden}")
```

Cómo se puede ver en el ejemplo, en el método `show_attributes` la instancia es capaz de acceder a los atributos con el nombre con el que han sido definidos, mientras que fuera de ese ámbito necesitamos hacer uso del nombre "privado" del atributo `__hidden` para poder imprimir su valor.

1.3.4 Herencia y polimorfismo

Como lenguaje orientado a objetos, la herencia es un concepto muy utilizado en Python. De hecho, una clase Python puede heredar de varias clases padres simultáneamente, aunque es una funcionalidad que no se utiliza demasiado.

Para definir que una clase hereda de nuestra `MyClass` haremos lo siguiente:

```
class ChildrenClass(MyClass):
    pass
```

El concepto de polimorfismo en Python es algo diferente al mismo en lenguajes de tipado estático como Java. Al no haber ningún tipo de verificación de tipos en tiempo de compilación, se basará simplemente en lo que denominamos *duck typing*.

El concepto de *duck typing* se puede resumir en

Note: Si anda como un pato y grazna como un pato, entonces debe ser un pato

Esta frase resume muy bien el concepto de polimorfismo en Python: si un objeto es capaz de responder a una serie de invocaciones a métodos pasándole una serie de parámetros válidos, independientemente de que esté definido formalmente como una herencia de clase o no, podremos utilizar ese objeto de esa manera.

1.4 Estructura típica de un programa Python

Un programa Python está definido en uno o varios ficheros de texto plano que cumplen con la sintaxis de Python. Cuando ejecutamos el intérprete de Python pasándole un fichero como entrada, el intérprete cargará dicho fichero y tratará de ejecutarlo.

1.4.1 Entrypoints

En otros lenguajes estamos habituados a que se introduce un método o función con un nombre determinado que actuará como punto de entrada de la ejecución del programa. Por ejemplo, la función `main` en lenguajes como C, C++ o Golang, un método público y estático llamado `main` en una clase Java…

En Python sin embargo no existe ninguna convención al respecto: el código que se ejecutará será el que se encuentre en el fichero que se pase al intérprete sin ninguna identación.

```
print("Mi primer programa Python")
```

Si escribiéramos en un fichero el código anterior y pasáramos el fichero al intérprete de Python:

```
$ python3 miprograma
Mi primer programa Python
```

Sin embargo, este tipo de programas, aunque nos puedan servir para empezar a trabajar en Python, no son muy mantenibles a largo plazo, ya que el código sin identar se ejecutará también cuando otro fichero Python quisiera importar alguna de las funciones que tuviéramos definidas. Por ello es habitual añadir el código que consideramos parte del *entrypoint* a una función y ejecutarlo únicamente cuando el fichero se está ejecutando desde un intérprete de forma directa:

```
def entrypoint():
    print("Mi primer programa Python")


if __name__ == "__main__":
    entrypoint()
```

El código anterior cargará la función `entrypoint`, sin ejecutarla, y ejecutará el bloque `if`. La variable `__name__` es una variable especial que el intérprete de Python inyecta a nuestro programa. Si el módulo donde estamos escribiendo esto ha sido ejecutado directamente por el intérprete, y no como parte de una secuencia de importación, el valor será la cadena `"__main__"`, por lo que de ser así, ejecutaremos la función `entrypoint`.

1.4.2 *Shebangs*

Un *shebang* es una catacterística típica de los terminales que cumplen con el estándar POSIX que permite indicar, desde el propio fichero, el intérprete que debe ser utilizado para ejecutar el fichero. Debe ser especificado en la primera línea del fichero y comenzar con un símbolo de `#` seguido de una admiración `!` y la ruta al intérprete que debe ejecutarlo.

```
#!/usr/bin/python3

def entrypoint():
    print("Mi primer programa Python")


if __name__ == "__main__":
    entrypoint()
```

Al hacerlo de esta manera, si damos permisos de ejecución al fichero, podríamos lanzarlo sin necesidad de invocar al intérprete.

Note: Es bastante habitual que cuando queremos lanzar nuestro programa Python, tras haberle dado permisos de ejecución, que el sistema se quede "congelado" y el cursor se convierta en una cruceta. Si ésto te ocurre, se debe a que has olvidado definir el *shebang* y el sistema está intentando ejecutar tu programa como si se tratara de un script en Bash. El cambio de cursor se debe a que casi siempre las primeras líneas de nuestros programas Python incluyen `import`, que además de ser la palabra reservada en Python para importar otros módulos, es un comando de línea de comandos que permite hacer capturas de pantalla.

1.4.3 Modularización en Python

Como se ha mencionado de pasada anteriormente, Python permite importar funcionalidad disponible en otros ficheros Python a través de dos elementos de su sintaxis: `import` y `from ... import ...`.

- `import module` permite añadir al espacio de nombres de nuestro módulo el módulo `module` y utilizar cualquier función definida en él a través de `module.something`
- `from module import function` permite añadir al espacio de nombres de nuestro módulo únicamente un símbolo de los definidos en `module`, en este caso, el símbolo `function`.

La modularización en Python es muy importante ya que nos permite utilizar tanto funcionalidades de la librería estándar de Python como de librerías de terceros. También nos permite organizar nuestro código mejor, manteniendo una estructura en nuestro programa que permita mejorar la legibilidad y el mantenimiento de nuestro software.

1.5 La librería estándar de Python

El lenguaje Python proporciona una extensísima librería de módulos que nos sirven para construir nuestro software sin tener que reimplementar una y otra vez algunas funcionalidades que suelen ser comunes a todo tipo de programas.

Entre esas librerías podemos mencionar algunas de las más comunes:

- `os` incluye muchas funciones y variables útiles para interactuar con aspectos específicos del sistema operativo y el entorno de ejecución: variables de entorno, rutas de ficheros, descriptores de ficheros de entrada, salida y error estándar…
- `sys` nos permite acceder a la línea de comandos de nuestro programa (`sys.argv`) y también a la función `sys.exit` que nos permite terminar la ejecución e indicar el código de salida.
- `re` incluye todas las funcionalidades relativas a expresiones regulares.
- `socket` contiene toda la librería que da acceso de bajo nivel a sockets de comunicación.
- `math` incluye una buena colección de funciones matemáticas y valores para constantes.

Esto es sólo una pequeña muestra de toda la librería estándar de Python.

En este apartado vamos a ver un par de librerías que deberéis utilizar en vuestra práctica y que os facilitarán la vida: `argparse` y `logging`

1.5.1 argparse

Casi todos los programas que se ejecutan desde la línea de comandos suelen necesitar leer una serie de parámetros y argumentos de la misma para decidir qué acciones deben realizarse. A través de la librería `sys` podemos acceder fácilmente a la línea que ha sido ejecutada, usando la variable `sys.argv`.

Aunque en algunas ocasiones es posible que con lo anterior sea suficiente, en otras muchas no lo será. Además, también es recomendable proporcionar una ayuda cuando se solicite o se comenta algún error en la entrada, e implementar todo eso a mano suele ser bastante tedioso.

Para ello, Python proporciona la librería `argparse`. Básicamente esta librería funciona definiendo un objeto `ArgumentParser` al cuál le vamos añadiendo los diferentes argumentos que queremos que soporte nuestro programa. Cada uno de estos argumentos están ligados una opción en la línea de comandos, unos argumentos y puede definir un texto de ayuda que permita generar la salida del comando de ayuda de forma automática.

```
#!/usr/bin/env python3
import argparse

def parse_commandline():
    parser = argparse.ArgumentParser(description="Commandline calculator")
    parser.add_argument('value1', type=float, help='First number')
    parser.add_argument('value2', type=float, help='Second number')
    return parser.parse_args()

def main():
    user_options = parse_commandline()
    print(user_options.value1 + user_options.value2)

if __name__ == '__main__':
    main()
```

En el ejemplo anterior se puede observar que nuestro *entrypoint* va a ejecutar primero la función `parse_commandline`. En dicha función se crea el objeto `argparse.ArgumentParser` y se añaden dos

argumentos a la línea de órdenes. El valor devuelto es el resultado de la llamada a `parse_args`, que analizará la línea de argumentos que ha recibido nuestro programa y devolverá, si es correcta, un `Namespace` que contendrá los valores recibidos. En este caso, el programa no debe recibir ninguna opción y aceptará obligatoriamente 2 valores de tipo `float`. Si se pasaran menos o más de 2, automáticamente el programa fallaría.

```
#!/usr/bin/env python3

import argparse
import sys

_ALLOWED_OPERATIONS_ = ['sum', 'sub', 'mul', 'div']

def parse_commandline():
    parser = argparse.ArgumentParser(description="Commandline calculator")
    parser.add_argument('value1', type=float, help='First number')
    parser.add_argument('value2', type=float, help='Second number')
    parser.add_argument(
        '-o',
        '--operation',
        choices=_ALLOWED_OPERATIONS_,
        default='sum',
        dest='op',
        help='Operation to execute',
    )

    return parser.parse_args()


def main():
    user_options = parse_commandline()
    if user_options.op == 'sum':
        print(user_options.value1 + user_options.value2)
    elif user_options.op == 'sub':
        print(user_options.value1 - user_options.value2)
    elif user_options.op == 'mul':
        print(user_options.value1 * user_options.value2)
    elif user_options.op == 'div':
        print(user_options.value1 / user_options.value2)

if __name__ == '__main__':
    main()
    sys.exit(0)
```

En éste otro ejemplo se ha añadido un argumento de opción (`-o` o `--operation`) que permite pasar un valor dentro de una lista de valores permitidos y que hace que nuestro método `main` se comporte de manera diferente según el valor especificado.

1.5.2 `logging`

En todo proyecto suele ser necesario añadir diferentes mensajes que indiquen qué está haciendo nuestro programa, tanto a nivel de tener informes de lo que está sucediendo como, en tiempo de desarrollo, para depurar el funcionamiento del programa.

Todos los lenguajes permiten escribir mensajes en el terminal de modo que podamos ver dichos mensajes y saber qué está ocurriendo, pero esa es una solución muy poco práctica, ya que en ocasiones podemos olvidarnos de quitar mensajes de depuración y terminan llegando al producto final, o que muestran información que no debería ser vista por los usuarios de nuestra aplicación.

En ocasiones también es muy práctico poder volcar, según el tipo o la gravedad, los mensajes a la salida estándar, de error o a un fichero, o incluso yendo más allá, utilizar algún sistema de monitorización de logs en la nube. Evidentemente, para este tipo de situaciones un simple `print` se nos queda muy corto.

Python proporciona la librería `logging` que permite toda esta serie de operaciones:

- Definir diferentes "*loggers*", que tengan diferentes configuraciones cada uno de ellos.
- Cada mensaje puede ser enviado con un nivel de "severidad" diferente, yendo desde mensajes de depuración (debug) a mensajes criticos (critical).
- Desacoplar la escritura de cada mensaje de su configuración, no teniendo que preocuparnos en nuestro código de si el mensaje va a ser enviado a la salida estándar, fichero o un servicio en la nube remoto.
- Desacoplar el formato con el que se mostrará o almacenará el mensaje de la escritura del mismo.

Veamos algunos ejemplos:

```
import logging

logging.debug("This is a debug message")
logging.info("This is an info message")
logging.warning("This is a warning message")
logging.error("This is an error message")
logging.critical("This is a critical message")
```

El ejemplo anterior muestra el uso más básico de la librería, ya que todos los mensajes se enviarán al *logger* denominado "root", al no haber especificado otro. Cada mensaje se envía con un nivel diferente. Si lo ejecutamos, veremos que sólo se imprimirán por la salida de error estándar los mensajes de nivel "warning", "error" y "critical", ya que el *logger* por defecto está configurado para emitir a su salida únicamente los mensajes de nivel superior a **"info"**.

Además, se imprimen con un formato muy concreto:

```
$ python3 log_example.py
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

Ese formato por defecto utilizado incluye en primer lugar el nivel, luego el nombre del *logger* y por último el mensaje.

Para configurar la salida podemos utilizar la siguiente función:

```
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s [%(levelname)s]:
 ↪%(message)s', force=True)
logging.debug("This is a debug message")
logging.info("This is an info message")
```

(continues on next page)

(continued from previous page)

```
logging.warning("This is a warning message")
logging.error("This is an error message")
logging.critical("This is a critical message")
```

En este caso, estamos proporcionando una configuración por defecto diferente, especificando que queremos imprimir aquellos mensajes de nivel "info" o superior y modificando el formato del mensaje. Si volvemos a ejecutarlo ahora, obtendremos:

```
$ python3 log_example.py
2022-09-25 14:09:42,317 [INFO]:This is an info message
2022-09-25 14:09:42,317 [WARNING]:This is a warning message
2022-09-25 14:09:42,317 [ERROR]:This is an error message
2022-09-25 14:09:42,317 [CRITICAL]:This is a critical message
```

Como se puede observar, con sólo una línea se ha modificado por completo la salida del programa sin tener que modificar el código como tal de la emisión de mensajes, mostrando el enorme desacoplamiento entre configuración y mensajes que permite la librería `logging`.

Para más información sobre el uso de esta librería, Python proporciona unos tutoriales básico y avanzado en su documentación.

1.6 Organización del código Python

Cuando escribimos cualquier software es muy importante cómo organizamos el código. Cada lenguaje permite realizar esta modularización de maneras diferentes. En el caso de Python, tenemos a nuestra disposición un mecanismo de paquetes y módulos que nos permite organizar nuestros ficheros de la forma que mejor nos permita reutilizar código y su mantenimiento posterior.

También es importante, de cara a organizar nuestro código, saber de qué manera se va a distribuir, ya que nos evitará tener que reorganizar nuestro código a posteriori cuando nos enfrentemos a la tarea de distribuirlo.

1.6.1 Módulos, paquetes y subpaquetes

El código de un software escrito en Python estará organizado en uno o varios ficheros con definiciones de Python. Cada uno de estos archivos es un módulo. Desde cada módulo podemos importar otros módulos.

Para organizar los módulos, Python proporciona el concepto de "paquetes" y "subpaquetes".

Un paquete Python es un directorio del sistema de archivos en el que existe un fichero `__init__.py`. Dentro del directorio puede haber uno o varios módulos de código Python, y también otros paquetes. En este caso, se suelen denominar subpaquetes.

La manera habitual de distribuir el código Python es la de crear un paquete que contiene todos los módulos con toda la funcionalidad asociada.

1.6.2 Distribución de software en Python

El propio ecosistema de Python proporciona un comando para la instalación de paquetes llamado `pip`, lo que indica que la forma preferida de distribución de software escrito en Python es en forma de paquetes.

Dentro de cada paquete es habitual escribir, al menos, un módulo en el que se encuentre la funcionalidad principal. Si es necesario, se puede dividir el código en varios módulos o incluso subpaquetes, pero eso dependerá en gran medida de las necesidades de cada proyecto.

Para la creación de paquetes Python se suele utilizar la librería `setuptools`, que aunque no es para de la librería estándar, se ha convertido en un estándar de facto.

Dicha librería reutiliza algunos conceptos e ideas de la librería `distutils`, que si pertenece a la librería estándar. Entre esas ideas que se mantienen está la de la utilización de un fichero `setup.py` o un fichero `setup.cfg` para especificar la información del paquete, así como su ubicación en el sistema de archivos, qué módulos deben incluirse o excluirse e incluso la autgeneración de manejadores de comandos.

Todo eso y más lo veremos con más detenimiento a través del repositorio plantilla que utilizaremos para la elaboración de los entregables de la asignatura.

1.7 Gestión de dependencias en Python

Cuando vamos a trabajar en un software Python, uno de los problemas a los que tenemos que enfrentarnos es la gestión de dependencias. Cada programa o librería tiene sus propias dependencias, y en algunas ocasiones, puede ocurrir que no sean compatibles entre ellas.

Por ello, en Python se proporciona el mecanismo de entornos virtuales, más conocidos como `virtualenv`, que permite crear entornos aislados con su propia instalación de dependencias.

1.7.1 Creación de un `virtualenv`

Para crear un `virtualenv` ejecutaremos:

```
python3 -m venv DIR
```

Donde `DIR` será la ruta del directorio donde queramos crear el entorno virtual.

1.7.2 Flujo de trabajo con un `virtualenv`

Cuando tenemos ya creado un entorno virtual, lo primero que debemos hacer es activarlo con:

```
$ source DIR/bin/activate.sh
```

Tras eso, nuestro intérprete Python utilizado será el que se ha copiado dentro del entorno virtual, y los paquetes que instalemos a través de `pip` se instalarán de manera local en este `virtualenv`.

Una vez que hayamos terminado de trabajar en ese entorno virtual o queramos cargar otro, deberemos ejecutar primero `deactivate` para que el intérprete de comandos deje de considerar el `virtualenv`.

CHAPTER

TWO

DE PROGRAMACIÓN ORIENTADA A OBJETOS A LA INVOCACIÓN REMOTA DE MÉTODOS

La programación orientada a objetos es un paradigma de programación muy extendido. En éste paradigma se hace hincapié en la definición de **clases**, que indica qué atributos tiene una determinada entidad de nuestro programa, y sus **métodos**, que indican la interfaz a través de la cual podemos interactuar con los objetos o instancias de esa clase.

En sistemas distribuidos, la contraparte de la orientación a objetos es lo que conocemos como **invocación de métodos remotos**, o **RMI**, el acrónimo en inglés de *Remote Method Invocation*.

2.1 Ejemplo de orientación a objetos

Imaginemos que tenemos que hacer un software que sea capaz de representar un mensaje cualquiera en algún medio. Para ello, podríamos definir una interfaz que indique cuál es el nombre del método, qué parámetros aceptará y qué valor devolverá.

Para nuestro ejemplo, hagamos una interfaz muy simple. Dado que en Python no existe el concepto de interfaz diferenciado del concepto de clase, definiremos una clase la cuál no tenga implementado ningún comportamiento específico:

```
class Printer:
    def write(self, message):
        pass
```

A partir de aquí, cualquier clase que tenga un método `write` que acepte un argumento tendrá la misma interfaz que nuestra clase `Printer`. De ese modo, podríamos implementar diferentes clases con el comportamiento que queramos.

```
class ConsolePrinter:
    def write(self, message):
        print(message, flush=True)
```

El ejemplo anterior muestra una implementación de la interfaz la cuál imprimirá el mensaje en la salida estándar del proceso, sin hacer ningún tipo de *buffering* (`flush=True` indica precisamente eso).

```
class FilePrinter:
    def __init__(self, path):
        self._filepath = path
        self._file_handler = open(path, "w")

    def write(self, message):
        self._file_handler.write(message)
```

(continues on next page)

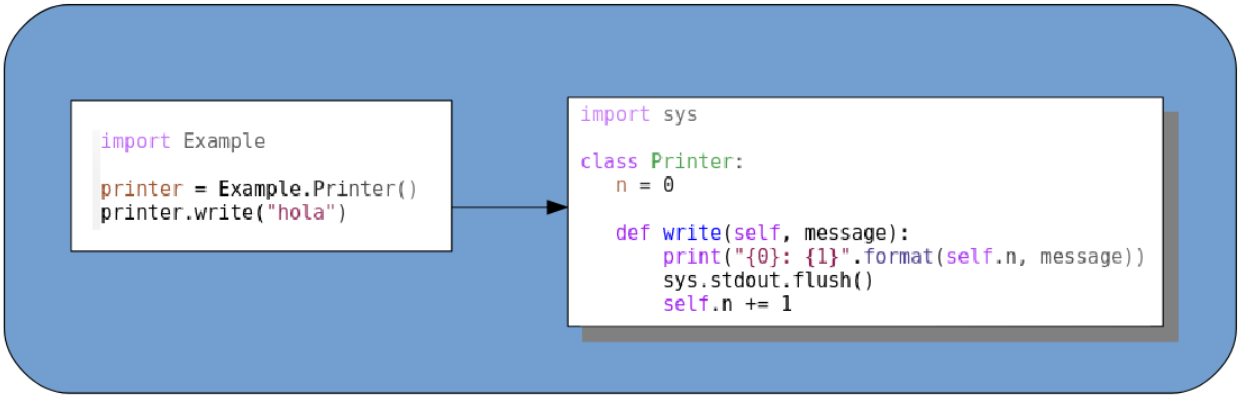
(continued from previous page)

```
    def close(self):
        self._file_handler.close()
```

Esta segunda implementación, además del método `write`, tiene un método de inicialización, que acepta la ruta a un fichero, y un método `close` que provocaría la termiación de la escritura en un fichero en el disco duro.

A pesar de tener esos métodos añadidos, la clase `FilePrinter` sigue cumpliendo con la misma interfaz al mantener el método `write` y un argumento.

De este modo, un usuario de nuestras clases podría utilizar, de manera casi indistinta, estas dos clases, modificando sólo la creación del objeto.

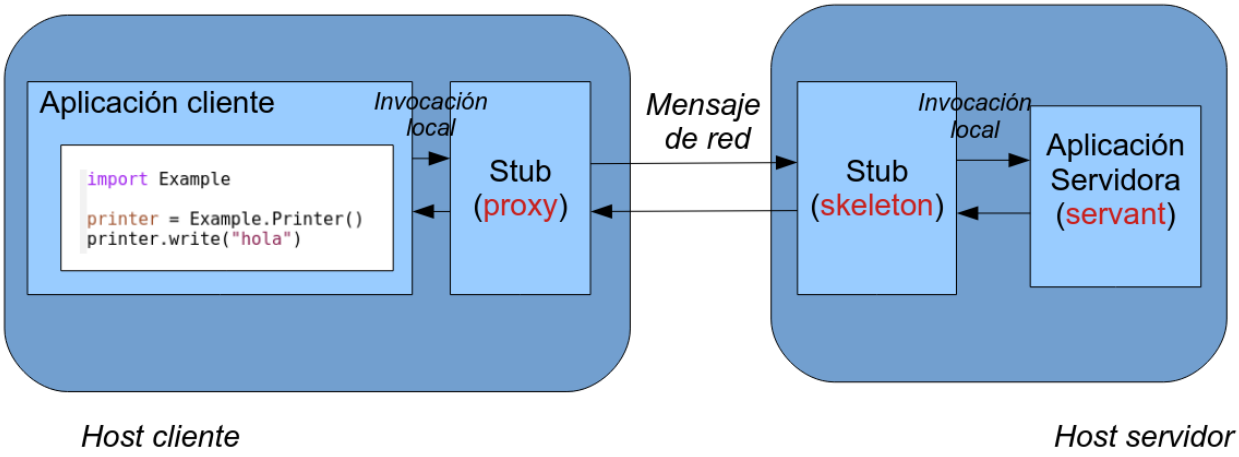


Como se ve en la imagen anterior, una vez que tenemos la referencia al objeto, simplemente podemos realizar invocaciones sobre sus métodos, en este caso, sobre `write`.

2.2 Extendiendo a RMI

Como se ha dicho con anterioridad, **RMI** es la respuesta para llevar la programación orientada a objetos al mundo de los sistemas distribuidos. La idea es poder realizar una invocación a un objeto local en la memoria de nuestro programa de manera que ésta sea traducida, en el otro extremo de una red de computadores, en una llamada local al objeto real que contiene el comportamiento del método.

Desde el punto de vista de los programadores, la invocación es vista exactamente de la misma manera a una invocación normal en orientación a objetos, pero sin embargo, detrás de las escenas ocurren múltiples cosas de manera transparente al programador que provocan que lo que, desde el punto de vista del diseño, es una invocación a un objeto normal y corriente, se traduzca en una invocación a través de la red.



Para que esta invocacion ocurra, debe existir una serie de librerías que se encarguen de diferentes aspectos necesarios:

- Una manera de especificar la API entre el cliente del objeto y el objeto propiamente dicho, normalmente a través de la definición de una interfaz.
- Una librería en la memoria del cliente que sea capaz de crear un objeto con la misma interfaz que el objeto remoto y que se encargue de realizar toda la conversión de parámetros entre la llamada local y la llamada a través de la red.
- Un proceso de ***marshalling***, que tome los parámetros que deben ser pasados al método remoto y convertirlos en bytes que puedan viajar a través de la red con un formato bien conocido.
- Un proceso de ***unmarshalling***, que reciba una serie de bytes de la red y sea capaz de transformarlos en la invocación correcta, al objeto correcto y con los parámetros correctos.
- La librería que, una vez realizado el *unmarshalling*, realiza la invocación sobre el objeto destinatario de la misma y recoge el valor de retorno, si lo hubiera.

Todos estos pasos ocurren sin que el usuario de la librería tenga que ser consciente de ello.

2.3 Un ejemplo de RMI: ZeroC Ice

ZeroC Ice, o en adelante, **ICE**, es un middleware de comunicación orietado a objetos, multiplataforma y multilenguaje, que realiza todo lo descrito en el apartado anterior.

ICE es software libre, por lo que está portado a infinidad de plataformas y sistemas operativos. Además, de manera nativa soporta múltiples lenguajes: C++, Java, Python, Ruby, JavaScript…

2.3.1 Un poco de nomenclatura

En ICE se utilizan algunos nombres que es **muy conveniente** conocer para poder hablar con propiedad:

- Servidor: cualquier programa en ejecución en un entorno distribuido con ICE.
- Sirviente: objeto inicializado en la memoria de un servidor ICE que es capaz de recibir invocaciones remotas.
- Adaptador de objetos: utilidad proporcionada por ICE para manejar la red: se encarga de definir qué protocolos, direcciones, puertos… se van a utilizar en un determinado servidor ICE. Los objetos que se añadan al adaptador de objetos son los que pueden actuar como sirvientes.
- Communicator: es el punto de entrada principal de la librería ICE. Proporciona toda una serie de utilidades para crear todo lo necesario para hacer funcionar un servidor ICE.

- Proxy: es un objeto "vacío" que se crea en un servidor ICE que **representa** a un objeto remoto y que, cuando recibe una invocación a uno de sus métodos, es capaz de realizar los pasos necesarios para que el sirviente al que representa reciba dicha invocación.
- Interfaz: es uno de los conceptos principales en sistemas distribuidos. Representa el **contrato entre un sirviente y sus clientes**. Dado que ICE soporta múltiples lenguajes, las interfaces deben definirse en un IDL (Interface Definition Language) llamado **Slice**.
- Translators: son programas proporcionados con ICE que convierten un fichero Slice con unas interfaces definidas en código de un lenguaje en concreto, de modo que pueda ser utilizado como librería por programas en dicho lenguaje.

2.3.2 Definiendo la interfaz

Como se menciona anteriormente, en ICE se utiliza un IDL particular llamado Slice. La sintaxis está en un punto intermedio entre C++ y Java. Tiene soporte para bastantes tipos básicos y permite la definición de estructuras básicas como listas y mapas, así como la definición de nuevos tipos compuestos.

```
module Example {
  interface Printer {
    void write(string message);
  };
};
```

En este caso, se define una interfaz con un único método (`write`), que aceptará una cadena como argumento. Este método no devolverá ningún valor.

Las interfaces escritas usando Slice siempre deben estar definidas dentro de un módulo, y estos a su vez pueden estar anidados. En nuestro caso únicamente definimos el módulo `Example` para alojar nuestra interfaz.

2.3.3 Traduciendo la interfaz a código Python

Evidentemente el código Slice no es compatible con ningún lenguaje de programación en concreto, por lo que debemos traducirlo al lenguaje de implementación que vayamos a utilizar. En éste caso, utilizaremos Python, por lo que ICE nos brinda dos posibilidades diferentes para traducir desde la interfaz:

#. Usando el *translator* `slice2py`, proporcionado por el propio middleware, y que traduce la interfaz desde Slice a un módulo Python listo para ser importado desde nuestro programa. #. Usando el método `Ice.loadSlice`, proporcionado por la propia librería Python para ICE, y que traduce el Slice a un módulo Python de manera dinámica.

La diferencia principal entre ambos es que mientras que el *translator* genera un directorio con el código del módulo que podemos inspeccionar, el segundo lo carga de forma dinámica en memoria. Por regla general y facilidad, optaremos por usar el segundo.

```
#!/usr/bin/env python3

import sys

import Ice
Ice.loadSlice("Printer.ice")

try:
    import Example
except ImportError:
    print("Error importing Example from Printer.ice file")
```

(continues on next page)

(continued from previous page)

```
    sys.exit(1)

print("Everything went well. Congratulations.")
sys.exit(0)
```

2.3.4 Implementando la interfaz en Python

Cualquiera de las clases que hemos visto con anterioridad en el ejemplo de orientación a objetos cumple con la interfaz definida en el Slice. Pero para que puedan ser utilizadas como una implementación de la interfaz definida en el Slice, debemos hacer algunos cambios. Partamos de la base del código del `ConsolePrinter`:

```
class ConsolePrinter:
    def write(self, message):
        print(message, flush=True)
```

Lo primero de todo, la clase debe **heredar** de la interfaz para que ICE sea capaz de reconocerla como una implementación válida:

```
class ConsolePrinter(Example.Printer):
    def write(self, message):
        print(message, flush=True)
```

Como se puede observar, el traductor de Slice a Python traduce los módulos como módulos Python y las interfaces, como classes Python. Sin embargo, el código anterior nos arrojaría algunos errores a la hora de recibir invocaciones remotas.

Debido a que puede ser necesario extraer información relativa a la propia invocación en un momento dado, el middleware nos pedirá que añadamos a cada uno de los métodos un argumento adicional. Dicho argumento contendrá información relativa al entorno de ejecución. Por ahora, no ahondaremos más en ello.

```
class ConsolePrinter(Example.Printer):
    def write(self, message, current=None):
        print(message, flush=True)
```

El argumento `current` contendrá la ya mencionada información adicional en forma de un objeto de tipo `Ice.Current`. Poniéndole un valor por defecto, esta misma implementación nos puede servir tanto para atender invocaciones locales como para atender invocaciones remotas.

2.4 Hola Mundo con ZeroC Ice

En este capítulo vamos a ver, tras lo aprendido en el capítulo anterior, como integrar la clase anterior en un servidor Ice y como implementar un cliente para la misma.

2.4.1 Interfaz

Como comentamos, es imprescindible definir una interfaz de comunicación para que los clientes sepan qué métodos pueden invocar y qué argumentos son necesarios y cuáles esperar como valor de retorno.

```
module Example {
  interface Printer {
    void write(string message);
  };
};
```

Descarga

2.4.2 Servidor de Hola Mundo

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
import Ice
Ice.loadSlice("Printer.ice")
import Example


class ConsolePrinter(Example.Printer):
    def write(self, message, current=None):
        print(message, flush=True)


class Server(Ice.Application):
    def run(self, argv):
        broker = self.communicator()
        servant = ConsolePrinter()

        adapter = broker.createObjectAdapter("PrinterAdapter")
        proxy = adapter.add(servant, broker.stringToIdentity("printer1"))

        print(proxy, flush=True)

        adapter.activate()
        self.shutdownOnInterrupt()
        broker.waitForShutdown()

        return 0


if __name__ == "__main__":
    server = Server()
    sys.exit(server.main(sys.argv))
```

Descarga

Además del sirviente, ya visto y explicado en profundidad en el apartado anterior, haremos uso de `Ice.Application`.

Esta clase es una utilidad que ofrece ZeroC ICE a los desarrolladores de aplicaciones para poder crear servidores de manera mucho más sencilla.

Desde el punto de vista del desarrollador, lo único que hay que hacer es lo siguiente:

1. Crear una clase que **herede** de `Ice.Application`.
2. Implementar el método `run`, que acepta un argumento. El código dentro del método será el que se ejecute cuando la aplicación se ejecute en el siguiente paso.
3. Crear un objeto de nuestra nueva clase y llamar a su método `main`, pasándole la línea de argumentos (o una variable equivalente).

Internamente, el método `main`, que es heredado de la `Ice.Application`, realizará una serie de inicializaciones de ICE y finalmente llamará al método `run` que hemos definido.

En nuestro ejemplo, el código realiza las 4 operaciones que cualquier servidor ICE realizara:

1. Inicializar y activar el adaptador de objetos.
2. Crear el sirviente (o sirvientes) y añadirlos al adaptador de objetos.
3. Configurar el modo de salida (`shutdownInInterrupt`). En este caso se realizará un `shutdown` del *broker* de comunicación cuando se reciba una interrupción.
4. Activar el modo de espera del broker: `waitForShutdown`.

2.4.3 Cliente de Hola Mundo

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
import Ice
Ice.loadSlice('Printer.ice')
import Example


class Client(Ice.Application):
    def run(self, argv):
        proxy = self.communicator().stringToProxy(argv[1])
        printer = Example.PrinterPrx.checkedCast(proxy)

        if not printer:
            raise RuntimeError('Invalid proxy')

        printer.write('Hello World!')

        return 0


sys.exit(Client().main(sys.argv))
```

Descarga

Para el cliente hemos vuelto a utilizar la misma `Ice.Application`, aunque en esta ocasión los pasos que realiza el método `run` son algo diferentes, al tratarse de un cliente puro:

1. Generamos un *proxy* genérico a partir del *stringfied proxy* que recibimos por la línea de argumentos.
2. Tratamos de hacer una conversión desde ese *proxy* genérico a un `Example.PrinterPrx`, utilizando el método `checkedCast`.
3. Si la conversaión falla, el método anterior devolverá un `None`, y saldremos.

4. Si la conversión fue bien, usaremos el *proxy* para realizar una invocación remota al método `write`.

CHAPTER

THREE

ICEFLIX: DISEÑO DE MICROSERVICIOS - PARCIAL 1

El objetivo principal del proyecto es **desarrollar un sistema distribuido basado en microservicios**. Para ello se tomará como modelo la popular plataforma de streaming **NetFlix** para crear una pequeña plataforma de streaming bajo demanda, utilizando algunos de los servicios disponibles en ZeroC ICE. Los objetivos principales de la práctica son:

- Familiarizarse con la invocación a métodos remotos.
- Control de eventos.
- Diseñar algoritmos tolerantes a fallos comunes en sistemas distribuidos.
- Fomentar el trabajo en equipos multidisciplinares.

3.1 Introducción

Conocida por todos es la plataforma de *streaming* bajo demanda **Netflix**, lo que quizás no es tan popular es la gran competencia técnica que existe detrás, el increíble sistema distribuido que soporta el uso simultáneo de la plataforma por millones de usuarios en todo el mundo. No todos los detalles de su funcionamiento han sido publicados por la empresa, sin embargo, sí los suficientes para poder hacernos una idea general de su funcionamiento.

Una colección de microservicios soporta el funcionamiento de la plataforma: autenticación, facturación, catalogado, recodificación, análisis y profiling de clientes (big data), etc. Estos microservicios se ejecutan en varias regiones de AWS. Por otro lado, Netflix dispone de sus propios servidores físicos (OCA's) que almacenan los archivos de vídeo y están diseñados específicamente para ofrecer un gran rendimiento en tareas de *streaming*.

Obviamente no se pretende crear una arquitectura de una complejidad semejante, pero sí intentaremos implementar una pequeña maqueta con una serie de microservicios que cooperarán entre sí para ofrecer video bajo demanda de una forma similar a como lo hace la conocida plataforma. No vamos a utilizar mecanismos de caché ni a implementar muchos de los servicios disponibles en la plataforma real: nos centraremos en autenticación, catalogado de medios y transferencia de fichero (en lugar de *streaming*).

3.2 El microservicio como base.

El sistema estará dividido en diferentes microservicios, cada uno encargado de realizar una de las diferentes tareas que tiene que realizar el sistema en su conjunto.

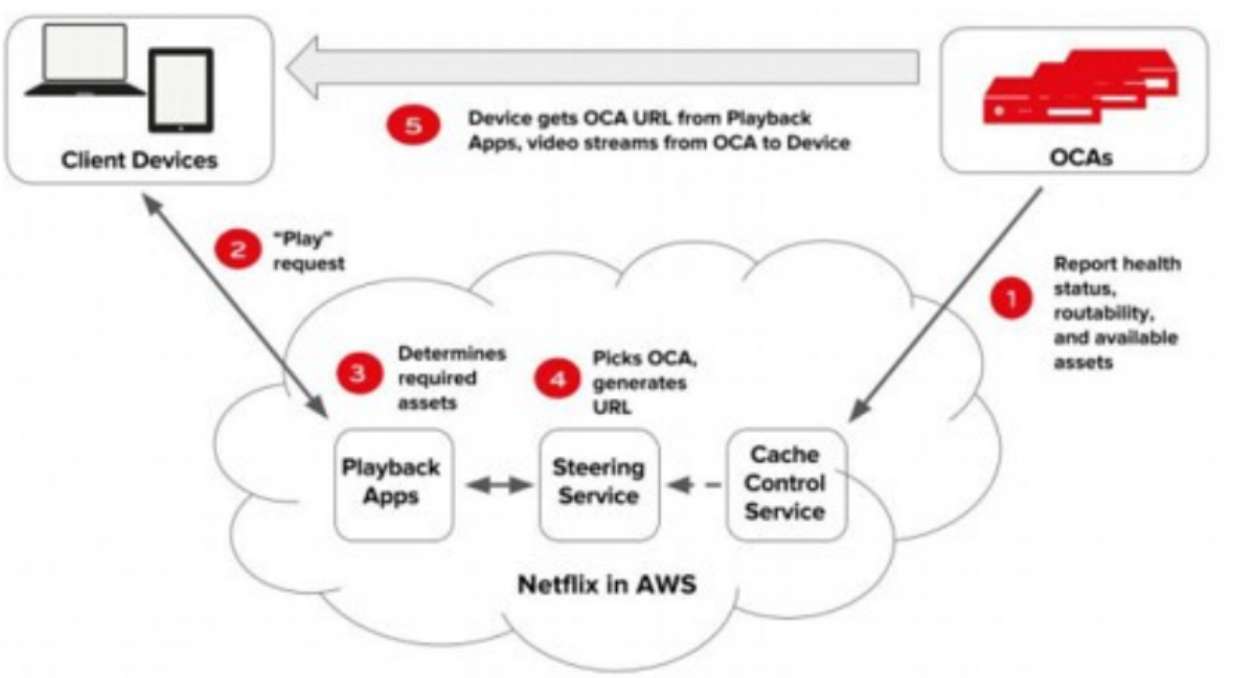


Fig. 3.1: Arquitectura de servicios de Netflix

3.2.1 La interfaz Slice

Cómo ya sabéis, el alma de la comunicación entre clientes y servicios en un entorno RMI es la interfaz. En ZeroC ICE esa interfaz se define en el IDL específico del middleware, Slice.

El código de la interfaz podéis encontrarlo *aquí*. Se recomienda leer la descripción de los servicios con el fichero Slice, para identificar las interfaz, métodos y argumentos aquí especificados.

3.2.2 Servicio principal

El primer microservicio actúa de entrada al sistema para los clientes. A través de su interfaz podrá obtener referencias a otros servicios.

Así mismo, el servicio debe recibir anunciamientos de los servicios de autenticación y catálogo para mantener las referencias actualizadas. Al arrancar, cada servicio enviará `newService` al servicio principal, y continuará enviando periódicamente un `announce`. Si pasados más de 30 segundos un servicio dejara de enviar sus `announce`, éste deberá ser borrado para no ser devuelto a los usuarios del sistema.

Este servicio debe implementar la interfaz `Main` del fichero Slice, la cual tiene estás funciones:

- `getAuthenticator()` que devuelve un proxy a un servicio de autenticación.
- `getCatalog()` que devuelve un proxy a un servicio de catalogo.
- `getFileService()` que devuelve un proxy a un servicio de ficheros.
- `newService()` acepta el proxy a un objeto. Debe comprobar de qué tipo de servicio se trata y almacenarlo en la caché.
- `announce()`: igual que el anterior, sirve para renovar la vigencia de un servicio. Si un servicio pasara demasiado tiempo sin enviar su `announce`, deberá ser eliminado de la caché.

3.2.3 Servicio de autenticación.

La autenticación en el servicio se gestiona a través de este servicio. El usuario debe enviar un usuario y contraseña periódicamente, obteniendo un token de autenticación que tendrá una validez limitada en el tiempo (2 minutos). También se encarga de realizar la comprobación de si un token es válido o no para el resto de servicios del sistema.

Para arrancar este servicio debe proporcionarse un token de administración, que deberá ser almacenado en memoria para poder realizar la comprobación de su validez en aquellas operaciones que requieren de dicho token. Éste token se pasará a través del fichero de configuración.

El servicio debe almacenar en una *base de datos* persistente entre reinicios (no necesariamente debe ser SQL) las credenciales de los usuarios. Los tokens, por su naturaleza temporal, no deben aparecer en ese almacenamiento persistente.

Adicionalmente proporciona al administrador la funcionalidad de añadir y eliminar usuarios.

Dicho servicio implementa la interfaz `Authenticator`:

- `refreshAuthorization()`: crea un nuevo token de autorización de usuario si las credenciales son válidas.
- `isAuthorized()`: indica si un token dado es válido o no.
- `whois()`: permite descubrir el nombre del usuario a partir de un token válido.
- `isAdmin()`: devuelve un valor booleano para comprobar si el token proporcionado corresponde o no con el administrativo.
- `addUser()`: función administrativa que permite añadir unas nuevas credenciales en el almacén de datos si el token administrativo es correcto.
- `removeUser()`: función administrativa que permite eliminar unas credenciales del almacén de datos si el token administrativo es correcto.

Envío de anunciamientos

Conforme a lo explicado en el apartado del *servicio principal* el servicio de autenticación deberá enviar una invocación `newService()` al arrancar, y continuar enviando periódicamente cada 25 segundos una invocación de `announce()`.

3.2.4 Servicio de catálogo

El servicio de catálogo proporciona a los usuarios acceso al catálogo y a la descarga de los medios disponibles en la aplicación.

Para ello, debe disponer de un almacén de datos persistente, dónde se almacenará información relativa a los medios: identificador, nombre asignado al medio por el administrador, y los tags asignados por cada usuario del sistema, si las hubiera.

El servicio debe recibir información desde los proveedores de streaming (*explicados más adelante*). para poder ofrecer a la aplicación de usuario un proxy para poder acceder a cada archivo.

El servicio debe implementar la interfaz `MediaCatalog`:

- `getTile()`: permite realizar la búsqueda de un medio conocido su identificador.
- `getTilesByName()`: permite realizar una búsqueda de medios usando su nombre.
- `getTilesByTags()`: permite realizar búsquedas de medios en función de los tags definidos por el usuario.
- `addTags()`: permite añadir una lista de tags a un medio concreto.
- `removeTags()`: permite eliminar una lista de tags de un medio concreto. Los tags que no existan se ignorarán.

- `renameTile()`: operación de administración que permite renombrar un determinado medio en la base de datos.
- `newMedia()`: operación invocada por el servicio de ficheros para informar de un nuevo fichero que puede ser ofrecido a los clientes.
- `removedMedia()`: operación invocada por el servicio de ficheros para informar de que un fichero ha dejado de estar disponible.

Envío de anunciamientos

Conforme a lo explicado en el apartado del *servicio principal* el servicio de catálogo deberá enviar una invocación `newService()` al arrancar, y continuar enviando periódicamente cada 25 segundos una invocación de `announce()`.

3.2.5 Servicio de ficheros

En lugar de implementar un servicio de *streaming* como se la plataforma real, en esta práctica se va a implementar un servicio de descargas.

El servicio de ficheros se encarga de enviar al usuario el fichero para que pueda visualizarlo. El servicio debe leer de un directorio en el disco duro, que será pasado al servicio a través de su configuración, los ficheros que serán compartidos.

El servicio deberá anunciar al servicio de catálogo cada archivo alojado en el directorio.

- `newMedia()`: será emitido por el servicio cuando se encuentre un nuevo fichero o sea subido uno nuevo por el administrador.
- `removedMedia()`: se emitirá cuando un fichero sea eliminado del servicio por el administrador.

El identificador de un medio, utilizado tanto por el servicio de ficheros como por el catálogo, se calculará en función del contenido del fichero (función hash). Para ello, puede utilizarse la suma SHA256 del fichero.

El servicio implementa la interfaz `FileService`:

- `openFile()`: dado el identificador del medio devolverá un proxy al manejador del archivo (`FileHandler`), que permitirá descargarlo.
- `uploadFile()`: dado el token de administrador y un proxy para la subida del archivo, lo guardará en el directorio y devolverá su identificador.
- `removeFile()`: dado un identificador y el token de administrador, borrará el archivo del directorio.

Cuando un usuario solicite abrir un fichero, el servicio creará un sirviente para manejar su posible descarga. Ésta se manejará a través de la interfaz `FileHandler`, que para todas las operaciones solicitará el token del usuario, que debe ser siempre el mismo que solicitó su apertura

Las operaciones disponibles en la interfaz `FileHandler` son:

- `receive()`: recibe el número indicado de bytes del archivo.
- `close()`: indica al servidor que el proxy para este fichero ya no va a ser usado y puede ser eliminado.

Envío de anunciamientos

Conforme a lo explicado en el apartado del *servicio principal* el servicio de archivos deberá enviar una invocación `newService()` al arrancar, y continuar enviando periódicamente cada 25 segundos una invocación de `announce()`.

Descarga de ficheros

Cuando un usuario abre un fichero a través de `openFile` en el servicio, el manejador de fichero debe ser consciente en todo momento de a qué usuario pertenece el token y, si éste expírase en algún momento, devolver la excepción adecuada.

Si el cliente recibiera dicha excepción durante la descarga, deberá renovar su token y utilizar el nuevo para poder continuar la descarga.

Subida de ficheros

Si el usuario es, además, administrador de la plataforma, puede utilizar el método `uploadFile()` del servicio para realizar una descarga.

Para ello el cliente deberá proporcionar un proxy a un sirviente de la interfaz `FileUploader`, a través del cuál el servicio irá solicitando la subida del fichero al cliente.

3.3 Cliente IceFlix

A continuación, se especificarán las características que debe implementar el cliente.

3.3.1 Conexión y autenticación

El cliente debe permitir especificar el proxy del servicio `Main` que va a utilizar. En caso de capturar un error `TemporaryUnavailable`, podría informar al usuario, pero debe reintentar automáticamente la conexión hasta un máximo determinado de veces que podrá definir el usuario y que por defecto serán 3 intentos, con una pausa de 5.0 segundos entre ellos.

Cuando la conexión esté establecida (se haya comprobado que el proxy es válido) se debe permitir al usuario autenticarse en el sistema. Esta autenticación no debe ser obligatoria puesto que se permite realizar búsquedas en el catálogo de forma anónima (únicamente por nombre).

En ningún caso el cliente debe mandar la contraseña introducida por el usuario (que tampoco se debe mostrar nunca en el cliente) sino que mandará la representación hexadecimal de la suma SHA256 del password.

También debe permitir cerrar la sesión del usuario para poder cambiar las credenciales si el usuario lo desea.

Por último, el cliente mantendrá informado al usuario en todo momento de que la conexión está establecida y de si existe una sesión iniciada.

3.3.2 Búsqueda en el catálogo

Cuando el cliente se encuentre on-line, permitirá al usuario buscar títulos por nombre o por tags. Las búsquedas por nombre se pueden realizar por término exacto o simplemente que el título incluya la palabra de búsqueda. La búsqueda por tags funcionará de manera similar: cuando el usuario introduzca la lista de tags, podrá indicar si quiere los títulos cuyos tags tengan alguno en común o tengan que estar todos.

Una vez realizada la búsqueda, si se obtienen identificadores de vuelta, el cliente mostrará el nombre y los tags del stream referenciado por cada identificador.

El último resultado de búsqueda que haya obtenido títulos se almacenará en memoria de manera que el usuario pueda seleccionar alguno de los títulos obtenidos en cualquier momento. Cuando el usuario seleccione un título, el programa cliente deberá informar al usuario en todo momento de su selección activa.

3.3.3 Edición del catálogo

Las opciones de edición del catálogo sólo podrán utilizarse si el cliente tiene una sesión activa en ese momento, en caso contrario se informará al usuario que debe autenticarse. Además de tener un token válido, el usuario deberá haber seleccionado alguno de los títulos que haya encontrado en una búsqueda anterior; si no tiene seleccionado ningún título, el error deberá informar al usuario de que necesita uno.

Una vez se disponga de autorización y del título que se desea editar, el cliente debe permitir añadir tags (uno o varios) o eliminarlos.

3.3.4 Reproducción de medios

Si el cliente dispone de conexión, autorización y título seleccionado, debe permitir al usuario iniciar la descarga del archivo. Si el cliente recibe durante la descarga la excepción `Unautheorized`, deberá solicitar al servicio de autenticación un nuevo token y continuar la descarga.

3.3.5 Operaciones administrativas

Las diferentes operaciones administrativas que están definidas para los servicios deben poder ser utilizadas, bien en el mismo cliente a través de un menú específico, bien en un cliente específico aparte. En ambos casos, el token administrativo se tratará como una contraseña, no debiéndose mostrar en pantalla. Puede ser suministrado como parámetro de configuración.

Para la operación `FileService.uploadMedia()` será necesario implementar un sirviente de la interfaz `FileUploader`:

- `receive()`: el servicio de stream provider llamará a esta operación para recibir un bloque de bytes, de como máximo, el tamaño indicado.
- `close()`: el servicio de stream provider llamará a esta operación para cancelar la subida o cuando el número de bytes recibido en el último `receive` sea 0, para indicar que ya se ha terminado la transferencia.

3.4 La entrega

Cada alumno deberá implementar uno de los servicios descritos en el enunciado (servicio principal, autenticación, catálogo o servicio de ficheros) o el cliente para la plataforma IceFlix descrita. El servicio o cliente a implementar por cada alumno será asignado por los profesores a través de sorteo, que se publicará en **Campus Virtual**.

Se entregará únicamente el enlace al repositorio en GitHub o cualquier otro repositorio accesible a través de Internet en la tarea habilitada para tal efecto en la plataforma de **Campus Virtual**. Dicho repositorio **debe ser privado**, con el fin de evitar plagios, y únicamente se invitará a poder ver el repositorio a los profesores de la asignatura.

Se considerará como entregable el *commit* etiquetado como **`ordinaria-p1`**.

Trabajo en equipo

Aunque la práctica es individual, se **recomienda encarecidamente** la colaboración entre alumnos, de modo que cada uno con vuestro servicio o cliente podáis ayudar y ser ayudados por otro compañero que esté implementando otro servicio o el cliente.

De ese modo podréis probar si vuestro servicio se comporta como el cliente o los servicios de vuestros compañeros esperan y estudiar si el comportamiento de la interfaz es el esperado.

3.4.1 Uso de la VPN

La Universidad de Castilla la Mancha pone a disposición del personal y los alumnos una red privada virtual (VPN). Una vez configurada, permite la conectividad entre dispositivos dentro de la misma, sin importar que éstos estén en diferentes redes físicas.

La VPN puede ser incluso configurada dentro de una máquina virtual, permitiendo que un programa que sea ejecutado en ella sea accesible a pesar de las limitaciones que suelen tener este tipo de instalaciones en cuanto al uso de la red.

Esta funcionalidad puede ser utilizada para realizar trabajo colaborativo y poder realizar pruebas entre diferentes servicios, siempre y cuando se utilice el endpoint correcto.

En el siguiente ejemplo podéis veir una ejecución del ejemplo de "Hola mundo" que tan trillado tenemos: al lanzar el servidor se muestra el proxy del sirviente y, separados por `:`, todos los endpoints del adaptador de objetos.

```
$ ./Server.py --Ice.Config=Server.config
printer1 -t -e 1.1:tcp -h 192.168.1.57 -p 7070 -t 60000:tcp -h 192.168.1.106 -p 7070 -
 ↪t 60000:tcp -h 192.168.122.1 -p 7070 -t 60000:tcp -h 192.168.130.1 -p 7070 -t␣
 ↪60000:tcp -h 172.24.178.57 -p 7070 -t 60000
```

En éste caso, el equipo estaba conectado a 5 redes diferentes. Sólo uno de los endpoints pertenece a la VPN, por lo que averiguar cúal es para utilizarlo es crucial, ya que si usamos cualquier otro, el endpoint no será accesible si estamos en otra red.

En el ejemplo, el endpoint correspondiente a la red de la VPN es `tcp -h 172.24.178.57 -p 7070 -t 60000`. Para podernos conectar al proxy remoto, deberemos ejecutar el cliente de la siguiente manera:

```
$ ./Client.py "printer1 -t -e 1.1:tcp -h 172.24.178.57 -p 7070 -t 60000"
```

¿Cuál es el endpoint relacionado con la VPN?

La forma más eficaz es conocer la IP de nuestra interfaz en la VPN: al conectarnos a una VPN el sistema operativo crea una interfaz de red virtual (no asociada a un dispositivo físico) que permite "tunelar" el tráfico a través de ella. Por defecto, en sistemas GNU/Linux **suele** llamarse `tun0`, por lo que para averiguar su IP usaremos:

```
$ ip address show dev tun0
7: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1422 qdisc fq_codel state␣
 ↪UNKNOWN group default qlen 500
    link/none
    inet 172.24.178.57/32 scope global tun0
       valid_lft forever preferred_lft forever
    inet6 fe80::2163:8856:aabd:5c22/64 scope link stable-privacy
       valid_lft forever preferred_lft forever
```

Una vez conocida la IP de nuestra interfaz, fácilmente averiguaremos qué endpoint está asociado a la misma.

3.4.2 Estructura del repositorio

Para evitar en la medida de lo posible problemas a la hora de crear la estructura del repositorio, se proporcionará un repositorio plantilla que podrá ser utilizado por los alumnos.

El repositorio de entrega deberá cumplir al menos con la siguiente estructura:

- `run_service`: debe estar en la raíz del directorio del proyecto y debe ejecutar la instancia del microservicio implementado. El fichero debe ser ejecutable. En caso de implementar el cliente, este fichero no debe existir en el repositorio..
- `run_client`: debe estar en la raíz del directorio del proyecto y debe lanzar el programa cliente. El fichero debe ser ejecutable. En caso de implementar un servicio, este fichero no debe existir en el repositorio.
- `README.md`: debe estar en la raíz del directorio del proyecto. Será un pequeño documento en lenguaje Markdown o texto plano que explicará paso a paso cómo se debe realizar la configuración del servicio o cliente y cómo lanzarlo usando uno de los scripts listados anteriormente.

  También debe incluir, bajo el título, la URL al repositorio del proyecto, así como cualquier otra información relevante del proyecto que el alumno considere oportuna.

Para los alumnos que tengan que implementar el servicio de ficheros, se puede incluir un archivo de no más de 5MB a modo de ejemplo. De ser así, dicho archivo **deberá estar en el directorio `resources`**.

Es muy importante seguir las indicaciones sobre la entrega de manera exacta ya que parte del proceso de evaluación se realiza de forma automática, por eso los nombres de fichero deben ser exactamente los indicados anteriormente.

Para facilitar la creación de esta estructura, se proporciona un repositorio plantilla que incluye todos los ficheros anteriores, además de algunos ficheros de configuración, ejemplos de implementación y utilidades para crear un programa instalable Python.

Adicionalmente a la evaluación automática se realizará un análisis estático de código con `pylint`. Se considera que el estilo es correcto si obtiene como mínimo un **9.0** de puntuación global.

Note: ATENCIÓN: Si usas Windows, ten en cuenta que los retornos de línea deben ser TIPO UNIX. Si se utilizan los retornos tipo DOS la práctica no funcionará en el entorno de pruebas.

3.4.3 Documentación de entrega

No se solicitará memoria de prácticas ni documentación adicional en la entrega. La evolución del proyecto se consultará con el historial de commits de GIT. Por tanto, no se recomienda subir la práctica en uno o dos commits.

3.4.4 Defensa de prácticas

La defensa de la práctica es **obligatoria**. Si algún alumno no se presentara la práctica se considerará **no presentada**.

Cada alumno presente la práctica será convocado por uno de los profesores de prácticas a una hora específica para revisar la práctica con ellos y ejecutarla.

Se hará especial hincapié en ello en los siguientes casos:

- Repositorio con muy pocos commits.
- El buscador de plagios alerte sobre dos o más entregas.

Una vez publicadas las notas, los alumnos tendrán derecho a revisión si no estuvieran de acuerdo.

3.5 Interfaz Slice de IceFlix

```
//
// P1 version
//
[["ice-prefix"]] module IceFlix {

    ////////////// Errors /////////////
    // Raised if provided authentication token is wrong
    // Also raised if invalid user/password
    exception Unauthorized { };

    // Raised if provided media ID is not found
    exception WrongMediaId { string mediaId; };

    // Raised if some item is requested but currently unavailable
    exception TemporaryUnavailable { };

    ////////////// Custom Types /////////////
    // List of bytes
    sequence<byte> Bytes;

    // List of strings
    sequence<string> StringList;

    ////////////// File server related interfaces /////////////
    // Handle file transfer
    interface FileHandler {
        Bytes receive(int size, string userToken) throws Unauthorized;
        void close(string userToken);
    };

    // Handle administrative file upload
    interface FileUploader {
        Bytes receive(int size);
        void close();
    };

    // File service
    interface FileService {
        FileHandler* openFile(string mediaId, string userToken) throws Unauthorized,␣
↪WrongMediaId;
        string uploadFile(FileUploader* uploader, string adminToken) throws␣
↪Unauthorized;
        void removeFile(string mediaId, string adminToken) throws Unauthorized,␣
↪WrongMediaId;
    };

    ////////////// Catalog service related structs and interfaces /////////////
    // Media info
    struct MediaInfo {
        string name;
        StringList tags;
     };

    // Media location
```

(continues on next page)

(continued from previous page)

```
    struct Media {
        string mediaId;
        FileService *provider;
        MediaInfo info;
    };

    // MediaCatalog service
    interface MediaCatalog {
        Media getTile(string mediaId, string userToken) throws WrongMediaId,␣
↪TemporaryUnavailable, Unauthorized;

        StringList getTilesByName(string name, bool exact);
        StringList getTilesByTags(StringList tags, bool includeAllTags, string␣
↪userToken) throws Unauthorized;

        void newMedia(string mediaId, FileService* provider);
        void removeMedia(string mediaId, FileService* provider);
        void renameTile(string mediaId, string name, string adminToken) throws␣
↪Unauthorized, WrongMediaId;

        void addTags(string mediaId, StringList tags, string userToken) throws␣
↪Unauthorized, WrongMediaId;
        void removeTags(string mediaId, StringList tags, string userToken) throws␣
↪Unauthorized, WrongMediaId;
    };

    ////////////// Auth server /////////////
    interface Authenticator {
        string refreshAuthorization(string user, string passwordHash) throws␣
↪Unauthorized;
        bool isAuthorized(string userToken);
        string whois(string userToken) throws Unauthorized;
        bool isAdmin(string adminToken);

        void addUser(string user, string passwordHash, string adminToken) throws␣
↪Unauthorized, TemporaryUnavailable;
        void removeUser(string user, string adminToken) throws Unauthorized,␣
↪TemporaryUnavailable;
    };

    ////////////// Main server /////////////
    interface Main {
        Authenticator* getAuthenticator() throws TemporaryUnavailable;
        MediaCatalog* getCatalog() throws TemporaryUnavailable;
        FileService* getFileService() throws TemporaryUnavailable;

        void newService(Object* service, string serviceId);
        void announce(Object* service, string serviceId);
    };
};
```

Descarga

3.6 Lista de requisitos

3.6.1 Servicio principal

- Responder con una referencia válida, cuando la hubiera, el método `getAuthenticator`.
- Responder con una referencia válida, cuando la hubiera, el método `getCatalog`.
- Responder con una referencia válida, cuando la hubiera, el método `getFileService`.
- Lanzar la excepción correspondiente cuando no haya servicio de uno de los anteriores disponible.
- Recibir los anunciamientos `newService` y `announce` de manera correcta.
- Ignorar los `announce` cuando vengan de un servicio no anunciado previamente con `newService`.
- Ignorar cualquier tipo de anunciamiento de objetos que no sea de las interfaces esperadas.
- Los servicios que dejen de anunciarse durante 30s son eliminados y nunca más pueden ser devueltos por ninguna de las 3 operaciones de acceso.
- Comprobar que, tras eliminar un servicio "caducado", no se puede recuperar de nuevo a través de la función de acceso.
- Comprobar que si se recibe un `newService` de un servicio existente previamente, ese identificador queda invalidado (ni el servicio antiguo ni el nuevo se devuelven)

3.6.2 Servicio de autenticación

- El servicio se anuncia siguiendo la secuencia correcta.
- El servicio identifica correctamente al administrador a través de su token.
- El servicio permite crear un usuario.
- El servicio permite eliminar un usuario.
- El servicio guarda los datos de los usuarios de manera persistente.
- El servicio permite identificar correctamente a un usuario válido, y rechaza usuarios inexistentes/incorrectos.
- El servicio identifica correctamente si un token es válido a través de `isAuthorized`.
- El servicio identifica correctamente al usuario a través de su token usando `whois`.
- El servicio revoca automáticamente los token pasados 2 minutos.
- El servicio revoca un token si el usuario se identifica de nuevo, aunque no hayan pasado los 2 minutos.

3.6.3 Servicio de catálogo

- El servicio se anuncia siguiendo la secuencia correcta.
- Un usuario válido puede recuperar el objeto `Media` de una película existente y disponible.
- Un usuario válido puede recuperar el objeto `Media` de una película existente, aunque no esté disponible.
- Un usuario válido puede modificar las "tags" de un archivo existente.
- Un usuario válido puede hacer búsquedas por etiquetas.
- Cualquier usuario puede hacer búsquedas por nombre.

- El administrador puede renombrar un archivo.
- Se puede notificar la disponibilidad de un archivo a través de `newMedia`.
- Se puede notificar que un fichero deja de estar disponible a tavés de `removeMedia`
- Los archivos nuevos que no existieran previamente y los tags de usuarios se almacenan de manera persistente.

3.6.4 Servicio de archivos

- El servicio se anuncia siguiendo la secuencia correcta.
- El servicio notifica al catálogo de todos los archivos en su directorio inicial.
- El servicio permite que el administrador pueda subir un archivo.
- El servicio sólo guarda el fichero subido en disco cuando llega completo. En cualquier otro caso, el fichero se descarta.
- El servicio permite abrir un archivo existente a un usuario válido.
- El servicio permite descargar un archivo a usuarios autenticados.
- El servicio detecta cuando expira el token de un usuario y rechaza continuar la descarga hasta recibir un token nuevo y válido.
- El servicio permite eliminar un archivo al administrador.
- El servicio calcula los "media id" utilizando el algoritmo especificado.
- El programa crea objetos de tipo `FileHandler` sólo cuando son necesarios, bajo demanda.

3.6.5 Cliente

- El programa reintenta conectarse en caso de fallo.
- El programa permite a un usuario identificarse.
- El programa permite realizar búsquedas anónimas por nombre.
- El programa permite realizar búsquedas por "tags".
- El programa solicita automáticamente un nuevo token en caso de revocación del existente.
- El programa permite al administrador añadir y eliminar usuarios.
- El programa permite al administrador renombrar ficheros en el catálogo.
- El programa permite al administrador subir archivos al servidor de archivos
- El programa permite al administrador eliminar archivos del servidor de archivos.
- El programa permite a un usuario válido descargar un archivo.

CHAPTER

FOUR

COMUNICACIÓN INDIRECTA CON ZEROC ICE: ICESTORM

En esta sesión vamos a utilizar el servicio de canales de eventos del middleware para realizar comunicaciones indirectas de uno a muchos.

En ICE existen varios mecanismos de comunicación indirecta:

- Endpoints IP multicast, que utilizan las capacidades multicast de las redes IP para poder enviar una determinada invocación remota a varios sirvientes que se han suscrito previamente a ese endpoint multicast.
- El servicio **IceStorm**, que permite la creación de diferentes canales de eventos (**topics**), a los cuáles los sirvientes pueden suscribirse para recibir las invocaciones.

En ambos casos la comunicación cuenta con una restricción: sólo pueden usarse comunicaciones de una sola dirección, por lo que no se permite que las invocaciones tengan valor de retorno o lancen posibles excepciones.

Ésto es así porque al tratarse de invocaciones remotas a métodos, si tuvieran valor de retorno o una excepción, ¿cómo debería manejar el cliente los diferentes valores o errores recibidos para obtener un único valor?

4.1 Mecanismo de funcionamiento de IceStorm

Cuando un servicio quiere realizar invocaciones a través de IceStorm para que varios sirvientes puedan recibir la misma invocación, primero debe obtener una referencia a un topic.

En ICE, todo se representa a través de interfaces Slice y de métodos remotos, por lo que desde el punto de vista de ese programa, el topic será un proxy de tipo `IceStorm.TopicPrx`.

El proxy al topic puede recuperarse si conocemos su versión "string", como siempre, pero lo más habitual es recuperar dicha referencia desde el propio servicio IceStorm, a través de la interfaz `IceStorm.TopicManager`.

El topic, entre otras operaciones, nos permite solicitar un *publisher* o subscribir un proxy al topic.

- El *publisher* es un objeto de tipo `Ice.Object` que permite mandar invocaciones al topic. Dicho objeto puede ser convertido al tipo que necesitemos a través de llamadas a `uncheckedCast`.
- Los *subscribers* serán nuestros sirvientes, a los que subscribiremos al topic.

El programa que quiere realizar la invocación, realizará los siguientes pasos:

1. Recuperará el proxy al topic desde el *topic manager*.
2. Solicitará un objeto *publiser* al topic.
3. Realizará una conversión del *publisher* al tipo deseado con `uncheckedCast`.
4. Comanzará a usar el proxy del *publisher* ya convertido, realizando las invocaciones que desee.

Los programas que quieran subscribirse a un topic, necesitarán hacer lo siguiente:

1. Recuperará el proxy al topic desde el *topic manager*.

2. Subscribirá el proxy del sirviente, previamente añadido al adaptador de objetos, al topic.

4.2 Arranque del servicio

IceStorm es un servicio que es parte del middleware y está implementado utilizando las propias herramientas disponibles en él.

En este caso, la implementación está realizada en C++, utilizando IceBox, que sin entrar en mucho detalle, es un framework que permite desarrollar servicios en forma de librerías dinámicas, que luego son cargadas a través de configuración por IceBox.

Para arrancarlo, necesitaremos tener tanto IceBox como IceStorm instalados. Esto se debe realizar a nivel de sistema operativo:

En distribuciones basadas en Debian, deben instalarse los paquetes `zeroc-icebox` y `libzeroc-icestorm3.7`. Para la instalación en otros sistemas operativos o distribuciones, consultad la página oficial de instalación de IceStorm.

Una vez instalada la librería, debe arrancarse `icebox` pasándole la configuración adecuada.

```
IceBox.Service.IceStorm=IceStormService,37:createIceStorm --Ice.Config=config/
↪icestorm.config
```

Como se puede observar, el fichero de configuración elige la librería y versión a cargar, pasándole después los argumentos, que en este caso incluyen el fichero de configuración del servicio:

```
IceStorm.TopicManager.Endpoints=tcp -p 10000
IceStorm.Publish.Endpoints=tcp -p 10001
IceStorm.LMDB.Path=/tmp/icestorm

IceStorm.Trace.TopicManager=2
IceStorm.Trace.Topic=1
IceStorm.Trace.Subscriber=1

IceStorm.Flush.Timeout=2000

Ice.Admin.InstanceName=icestorm
IceMX.Metrics.Debug.GroupBy=id
IceMX.Metrics.ByParent.GroupBy=parent
```

Con estos dos ficheros ubicados dentro de un mismo directorio `config`, podremos lanzar el servicio:

```
$ icebox --Ice.Config=config/icebox.config
```

4.3 Publicador de eventos

Para el primer ejemplo de uso de IceStorm, vamos a utilizar un viejo conocido: nuestro querido ejemplo de *"Hello World"*.

La interfaz Slice ya la conocemos muy bien:

```
module Example {
  interface Printer {
    void write(string message);
```

(continues on next page)

(continued from previous page)

```
  };
};
```

4.3.1 Modificando el cliente

Para este ejemplo, modificaremos el cliente que hemos utilizado anteriormente, añadiendo soporte para IceStorm.

Primero, debemos proporcionar al programa el proxy al servicio **IceStorm/TopicManager**. Como su propio nombre indica, ese servicio se encarga de la gestión de "topics": creación, destrucción y recuperación del proxy a los mismos.

Para ello, añadiremos unas líneas como las siguientes:

```
topic_manager_str_prx = "IceStorm/TopicManager -t:tcp -h localhost -p 10000"
topic_manager = IceStorm.TopicManagerPrx.checkedCast(
    self.communicator().stringToProxy(topic_manager_str_prx),
)
```

Una vez hecho lo siguiente, podemos realizar operaciones de gestión sobre los topics, cómo haremos a continuación:

```
topic_name = "PrinterTopic"
try:
    topic = topic_manager.create(topic_name)
except IceStorm.TopicExists:
    topic = topic_manager.retrieve(topic_name)
```

Por último, antes de realizar invocaciones tenemos que recuperar un ***publisher*** del topic, con el fin de enviar nuestras invocaciones al mismo:

```
publisher = topic.getPublisher()
printer = Example.PrinterPrx.uncheckedCast(publisher)
```

A partir de este punto, ya estamos listos para utilizar el objeto `printer`. Cada invocación al mismo resultará en un mensaje enviado al topic, y si hay algún sirviente de la interfaz subscrito al mismo, recibirá las invocaciones que realicemos.

4.3.2 Ejemplo completo

Para ejecutar el siguiente ejemplo no es necesario ningún fichero de configuración adicional.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys

import Ice
import IceStorm

Ice.loadSlice('Printer.ice')
import Example


class Publisher(Ice.Application):
    def run(self, argv):
```

(continues on next page)

(continued from previous page)

```
        topic_manager_str_prx = "IceStorm/TopicManager -t:tcp -h localhost -p 10000"
        topic_manager = IceStorm.TopicManagerPrx.checkedCast(
            self.communicator().stringToProxy(topic_manager_str_prx),
        )

        if not topic_manager:
            raise RuntimeError("Invalid TopicManager proxy")

        topic_name = "PrinterTopic"
        try:
            topic = topic_manager.create(topic_name)
        except IceStorm.TopicExists:
            topic = topic_manager.retrieve(topic_name)

        publisher = topic.getPublisher()
        printer = Example.PrinterPrx.uncheckedCast(publisher)

        if not printer:
            raise RuntimeError("Invalid publisher proxy")

        printer.write('Hello World!')

        return 0


sys.exit(Publisher().main(sys.argv))
```

4.4 Subscriptor de eventos

Para continuar con éste ejemplo, realizaremos modificaciones en el sirviente del ejemplo de *"Hello World"*.

4.4.1 Cambios en el sirviente

Una de las grandes ventajas que proporciona el sistema propuesto por ZeroC Ice es que no necesitamos realizar ninguna modificación en el código del sirviente.

4.4.2 Cambios en el servidor

Para subscribir nuestro sirviente al topic de nuestra elección, debemos realizar las siguientes modificaciones:

- Añadir el soporte para utilizar el *"topic manager"*
- Recuperar el proxy al mismo *topic* que utilizarán los publicadores.
- Subscribir nuestro proxy a dicho *topic*.

Para los dos primeros cambios, podemos reutilizar lo que hemos visto en el *publicador*.

```
topic_manager_str_prx = "IceStorm/TopicManager -t:tcp -h localhost -p 10000"
topic_manager = IceStorm.TopicManagerPrx.checkedCast(
    self.communicator().stringToProxy(topic_manager_str_prx),
)
```

(continues on next page)

(continued from previous page)

```
if not topic_manager:
    raise RuntimeError("Invalid TopicManager proxy")

topic_name = "PrinterTopic"
try:
    topic = topic_manager.create(topic_name)
except IceStorm.TopicExists:
    topic = topic_manager.retrieve(topic_name)
```

Para el último, debemos utilizar el método `subscribeAndGetPublisher` disponible en los topics. Éste método permite subscribir un proxy a un topic, especificando una calidad de servicio (QoS). El método devolverá un *publisher* específico para enviar invocaciones indirectas únicamente a éste subscriber, que en este caso ignoraremos, ya que no vamos a utilizarlo desde éste programa.

```
qos = {}
topic.subscribeAndGetPublisher(qos, proxy)
```

Tras estos cambios, cada subscriptor que lancemos se unirá al topic, por lo que cada vez que el publicador se ejecute, todas esas instancias recibirán el mensaje enviado.

4.4.3 Ejemplo completo

Para ejecutar el siguiente ejemplo no es necesario ningún fichero de configuración adicional.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys

import Ice
import IceStorm

Ice.loadSlice("Printer.ice")
import Example


class ConsolePrinter(Example.Printer):
    def write(self, message, current=None):
        print(message, flush=True)


class Server(Ice.Application):
    def run(self, argv):
        broker = self.communicator()
        servant = ConsolePrinter()

        adapter = broker.createObjectAdapter("PrinterAdapter")
        proxy = adapter.addWithUUID(servant)

        print(proxy, flush=True)

        adapter.activate()
```

(continues on next page)

(continued from previous page)

```
        topic_manager_str_prx = "IceStorm/TopicManager -t:tcp -h localhost -p 10000"
        topic_manager = IceStorm.TopicManagerPrx.checkedCast(
            self.communicator().stringToProxy(topic_manager_str_prx),
        )

        if not topic_manager:
            raise RuntimeError("Invalid TopicManager proxy")

        topic_name = "PrinterTopic"
        try:
            topic = topic_manager.create(topic_name)
        except IceStorm.TopicExists:
            topic = topic_manager.retrieve(topic_name)

        qos = {}
        topic.subscribeAndGetPublisher(qos, proxy)

        self.shutdownOnInterrupt()
        broker.waitForShutdown()

        return 0


if __name__ == "__main__":
    server = Server()
    sys.exit(server.main(sys.argv))
```

4.5 Ejemplo de monitorización usando IceStorm

4.5.1 Interfaz

```
module Monitoring {
    interface AgentControl {
        void publishMetrics();
        void setPublishInterval(float newInterval);
        void shutdown();
    };

    interface Agents {
        void announce(AgentControl* proxy, string nodeId);
    };

    interface Metrics {
        void cpu(float percent, string nodeId);
        void mem(long total, long free, string nodeId);
    };

};
```

Descarga

CHAPTER

FIVE

ICEFLIX: DISEÑO DE MICROSERVICIOS - ENTREGA FINAL

El objetivo de esta práctica es extender la implementación realizada durante la *primera entrega*

Los objetivos de este segundo entregable son:

- Utilización del servicio de distribución de eventos **IceStorm**.
- Aumentar la cardinalidad de cada servicio, pudiendo lanzarse varias instancias del mismo.
- Utilización de los canales de eventos para mantener el estado de la ejecución sincronizado.
- Desacoplar la comunicación entre servicios a través de comunicación indirecta de uno a muchos.
- Fomentar el trabajo en equipos multidisciplinares.

5.1 Introducción

Después de la entrega del primer parcial, debemos haber obtenido una plataforma basada en microservicios en los que cada uno de ellos tiene una labor encomendada muy acotada.

En ésta segunda entrega, se buscará aumentar la disponibilidad de nuestra plataforma, permitiendo que se pueda escalar el número de instancias de cada tipo de servicio de manera independiente.

Debido a este cambio, nos enfrentaremos a un nuevo conjunto de problemas, como por ejemplo mantener la coherencia del estado entre las diferentes instancias de cada servicio, o cómo comunicar ciertos cambios entre servicios que antes se comunicaban 1-1.

5.2 Uso de canales de eventos

Como se ha mencionado en la intrudicción, en esta nueva versión de la práctica nos encontramos dos nuevos problemas:

- Los servicios que antes se comunicaban a través de comunicación directa ahora lo harán de manera indirecta, de uno a muchos.
- Las distintas instancias de un mismo servicio deberán comunicarse entre ellas para mantener la coherencia entre ellas.

En esta sección analizaremos dónde están las comunicaciones que se verán afectadas por estos nuevos requisitos.

5.2.1 Anunciamientos

En la primera parte de esta práctica, el único servicio interesado en recibir anunciamientos era el "Main". Ahora, todos los servicios deberán recibir anunciamientos, ya que necesitan saber que servicios son legítimos y se están anunciando correctamente para aceptar mensajes que vengan de ellos.

Cada uno de nuestros servicios estará interesado en descubrir a otros, pero quizá no esté interesado en los anunciamientos de todos los servicios.

Para ello se definirá una nueva interfaz llamada `Announcement` que tendrá un método ya conocido: `announce`:

```
interface Announcement {
    void announce(Object* service, string serviceId);
};
```

Nombre del topic

El topic para publicar y recibir los anunciamientos de servicios deberá llamarse `Announcements`. Recuerda utilizar ese nombre para recuperar el topic para asegurar la compatibilidad entre diferentes implementaciones de los servicios.

Todos los servicios deberán enviar sus anunciamientos de la siguiente manera:

- Utilizando la interfaz `Announcement` en lugar de `Main`.
- Enviándolo al topic de nombre `"Announcements"` disponible en IceStorm.
- Únicamente enviarán invocaciones del método `announce`.
- En lugar de cada 30 segundos como máximo, deberán ser cada 10 segundos como máximo.

A su vez, todos los servicios, y no sólo el `Main`, deberán subscribirse a ese mismo topic para recibir todos los anunciamientos.

Anunciamientos no relevantes

Que un servicio reciba los anunciamientos de todos los demás no significa que sean relevantes. Algunos servicios simplemente ignorarán los anunciamientos si vienen de servicios que no le interesen para nada.

Utilizando los anunciamientos como fuente de "servicios conocidos"

Como se menciona anteriormente, las diferentes instancias de un mismo servicio utilizarán un topic de IceStorm para comunicarse y mantener la coherencia de los datos.

Es importante que sepamos que los datos, para ser aceptados por nuestra instancia, deberían venir siempre de una instancia que ya conociéramos anteriormente.

El canal de eventos de anunciamientos será nuestra fuente de "verdad" respecto a servicios conocidos: si recibimos una actualización de datos de nuestro servicio, pero el "serviceId" del mensaje no lo reconocemos, deberemos ignorar el mensaje.

5.2.2 Comunicación entre `Authenticators`

Las diferentes instancias de `Authenticator`, de manera independiente, son capaces de tener sus propias bases de datos de usuarios registrados y los tokens de usuario activos. Para esta entrega, se pedirá que ambas cosas se compartan entre las diferentes instancias a través de un topic IceStorm.

Por ello, el servicio de autenticación deberá convertirse también en un publicador y subscriptor de mensajes en dicho canal de eventos, de acuerdo con esta interfaz:

```
    // Interface to be used in the topic for user related updates
    interface UserUpdate {
        void newToken(string user, string token, string serviceId);
        void revokeToken(string token, string serviceId);

        void newUser(string user, string passwordHash, string serviceId);
        void removeUser(string user, string serviceId);
    };
```

Nombre del topic

El topic para publicar y recibir los cambios ocurridos en una instancia de `Authenticator` deberá llamarse `UserUpdates`. Recuerda utilizar ese nombre para recuperar el topic para asegurar la compatibilidad entre diferentes implementaciones de los servicios.

Actualizaciones no relevantes

Cada vez que se reciba un mensaje desde este topic, el servicio deberá cerciorarse de que el servicio que lo envió es un servicio conocido con anterioridad a través de su `serviceId`.

5.2.3 Comunicación entre `MediaCatalogs`

Los `MediaCatalog` pueden recibir, desde un cliente, cambios en el estado de sus datos: actualizaciones de tags o renombrado de archivos.

Estos cambios en los datos que maneja el servicio deben ser difundidos al resto de instancias a través de un topic en IceStorm, por lo que el servicio se convertirá también en un publicador y subscriptor de ese topic.

Los mensajes transmitidos a través de esa

```
    // Interface to be used in the topic for catalog media changes
    interface CatalogUpdate {
        void renameTile(string mediaId, string newName, string serviceId);
        void addTags(string mediaId, string user, StringList tags, string serviceId);
        void removeTags(string mediaId, string user, StringList tags, string␣
↪serviceId);
    };
```

Nombre del topic

El topic para publicar y recibir los cambios ocurridos en una instancia de `MediaCatalog` deberá llamarse `CatalogUpdates`. Recuerda utilizar ese nombre para recuperar el topic para asegurar la compatibilidad entre diferentes implementaciones de los servicios.

Actualizaciones no relevantes

Cada vez que se reciba un mensaje desde este topic, el servicio deberá cerciorarse de que el servicio que lo envió es un servicio conocido con anterioridad a través de su `serviceId`.

5.2.4 Anunciamiento de archivos disponibles desde el `FileService`

En el caso de los servicios de ficheros, no tenemos el problema de la consistencia de datos que hemos descrito en el resto de servicios, debido a que este tipo de servicio no tiene una base de datos local de ningún tipo.

Aún así, las instancias de los servidores de ficheros necesitarán notificar a un número indeterminado de `MediaCatalog` sobre sus archivos disponibles, para lo que utilizarán también IceStorm.

Para ello se publicará periodicamente en un topic la lista de identificadores de archivo que están disponibles en cada instancia, de modo que todos los `MediaCatalog` podrán saber qué cambios ha sufrido cada servidor de archivos.

```
// Interface to be used in the topic for file availability announcements
interface FileAvailabilityAnnounce {
    void announceFiles(StringList mediaIds, string serviceId);
};
```

Nombre del topic

El topic para publicar y recibir los cambios ocurridos en una instancia de `FileService` deberá llamarse `FileAvailabilityAnnounce`. Recuerda utilizar ese nombre para recuperar el topic para asegurar la compatibilidad entre diferentes implementaciones de los servicios.

Actualizaciones no relevantes

Cuando un servidor de archivos envíe un anuncio, los catálogos que estén subscritos deberán comprobar si ese servicio, identificado por su `serviceId`, ya estaba siendo anunciado antes.

Además, la invocación no contiene el proxy al servicio, por lo que éste deberá ser deducido por el servicio de catálogo de los *anunciamientos de servicio*.

5.3 Servicios y cliente

En esta sección se describirán los cambios más relevantes que deberán introducirse en los servicios y en el cliente de la práctica.

5.3.1 Servicio principal

- El método `newService` desaparece.
- El método `announce` dejará de estar disponible en la interfaz `Main` para estarlo en la interfaz `Announcement`.
- En lugar de recibir los anunciamientos de manera individual, los recibirá subscribiéndose al topic IceStorm adecuado.
- Además de recibir anunciamientos, también deberá enviarlos para que el resto de servicios y los clientes puedan descubrir su proxy.
- El tiempo de espera antes de eliminar un servidor por no anunciarse debidamente pasa de 30s a 10s.
- No se considera necesario el mensaje de `newService` existente en la versión anterior.

5.3.2 Servicio de autenticación

- El servicio deberá subscribirse al canal de anunciamientos de servicios, de modo que sólo aquellos que se estén anunciado correctamente se considerarán servicios válidos.
- El servicio se anunciará a ese mismo canal de eventos, invocando el método `announce` cada 10s como máximo.
- Cada identificación exitosa por parte de un usuario a través de `refreshAuthorization` deberá ir acompañada de un envío de `newToken` a través del canal de eventos dedicado a la comunicación entre instancias de este tipo de servicio.
- Todos los tokens de usuario, ya sean generados por la propia instancia o recibidos a través del topic de IceStorm deberán ser revocados cada 2 minutos.
- Cuando un token expire, todas las instancias enviarán el `revokeToken` al topic de IceStorm.
- Cuando se cree o se elimine a un usuario en una instancia, ésta deberá notificar al resto a través de una llamada a `newUser` o `removeUser` en el topic de IceStorm, según corresponda.
- En todo caso, cualquier mensaje que se reciba a través del topic de comunicación de servicios de autenticación incluye un `serviceId`, que deberá ser comprobado como un identificador válido.

5.3.3 Servicio de catálogo

- El servicio deberá subscribirse al canal de anunciamientos de servicios, de modo que sólo aquellos que se estén anunciado correctamente se considerarán servicios válidos.
- El servicio se anunciará a ese mismo canal de eventos, invocando el método `announce` cada 10s como máximo.
- Cada actualización en los tags de un determinado usuario deberá ser enviada al resto de instancias a través del topic IceStorm pertienente, con una llamada a `addTags` o a `removeTags`.
- Cada vez que se renombre un fichero por parte de un administrador en una instancia, ésta deberá enviar el mensaje `renameTile` a través del topic IceStorm adecuado.
- El servicio se deberá subscribir al topic donde los servidores de archivos anunciarán sus archivos disponibles periódicamente, actualizando la disponibilidad en base al proxy al servicio descubierto a través del canal de anunciamientos.
- En todo caso, cualquier mensaje que se reciba a través de un topic de comunicación incluye un `serviceId`, que deberá ser comprobado como un identificador válido.

5.3.4 Servicio de ficheros

- El servicio deberá subscribirse al canal de anunciamientos de servicios, de modo que sólo aquellos que se estén anunciado correctamente se considerarán servicios válidos.
- El servicio se anunciará a ese mismo canal de eventos, invocando el método `announce` cada 10s como máximo.
- El servicio deberá enviar al topic pertinente de manera periódica la lista de `mediaId` disponibles.
- El servicio deberá enviar siempre un mensaje de `announceFiles` cuando reciba alguna modificación por parte de un administrador (tanto añadir como eliminar un archivo).

5.3.5 Cliente

- El cliente deberá descubrir el servicio principal a través del topic de IceStorm de anunciamientos.
- En modo administrador, el cliente deberá ser capaz de subscribirse a cualquiera de los *canales de eventos* descritos en el apartado anterior y mostrar por pantalla, de un modo legible, los eventos que se vayan recibiendo.

5.4 Arranque y sincronización de los servicios

Una de las nuevas problemáticas que aparecen en la práctica con los nuevos requisitos es que cuando un servicio arranca es posible que ya haya otras instancias de ese mismo servicio funcionando.

Si nuestra instancia únicamente se encarga de cargar su última configuración conocida y aplicar los cambios "delta" que va recibiendo por los canales de anunciamientos, es muy posible que las diferentes instancias de un mismo servicio terminen con "bases de datos" diferentes.

Para ejemplificar esto, imagina que cuando arrancamos una segunda instancia del sevicio de autenticación, el sistema se encuentra en éste estado:

- Hay una instancia del servicio de autenticación funcionando.
- Se han creado 5 usuarios en dicha instancia.
- Hay 2 usuarios con un token activo.

Cuando la segunda instancia se conecta al sistema:

- No tiene conocimiento de los 5 usuarios creados, por lo que si alguno intenta identificarse en estainstancia, fallará.
- No tiene conocimiento de los tokens activos, por lo que si algń servicio intenta validar uno de los tokens, se le dirá que ha expirado.

Ésto también puede ocurrir con otros servicios, por lo que se propone un mecanismo de arranque y sincronización de servicios.

5.4.1 Algoritmo de arranque y sincronización de servicios

1. Cuando el servicio arranca, se subscribe al canal de anunciamiento de servicios.
2. Espera al menos 12s para recibir:
    - Un anunciamiento de una instancia del servicio `Main`
    - Un anunciamiento de una instancia de un servicio de su mismo tipo.
3. Si en 12s no ha recibido ningún anunciamiento de `Main`, el programa terminará.

4. Si en 12s ha recibido un anunciamiento de `Main`:
    - Si no ha recibido ningún anunciamiento de otra instacia de su mismo tipo, asumirá que es el primero en arrancar y dará por válida la base de datos persistente que tenga el servicio.
    - Si recibe un anunciamiento de otra instancia, invocará el método de actualización de la misma.

5.4.2 Método `bulkUpdate`

Éste método será utilizado por cualquier instancia del servicio de autenticación que, al arrancar, descubra que hay otras instancias de su mismo tipo funcionando.

El método deberá devolver el tipo de datos definido en la interfaz, dándole a la nueva instancia todos los datos disponibles en ese momento respecto a "tokens" activos, usuarios existentes y la contraseña de administrador.

La instancia al recibir esta estructura deberá actualizar todos los valores en consecuencia.

5.4.3 Método `getAllDeltas`

Éste método será utilizado por cualquier instancia del servicio de catálogo que, al arrancar, descubra que hay otras instancias de `MediaCatalog` funcionando.

El método se invocará sobre la primera instancia de la que se reciba un `announce`, y provocará que esta instancia automáticamente envíe toda la información de renombrado de archivo y de tags de usuarios disponibles mediante el envío de los respectivos mensajes al canal de eventos `CatalogUpdates`.

5.5 Interfaz Slice de IceFlix

```
//
// P1 version
//
[["ice-prefix"]] module IceFlix {

    ////////////// Errors /////////////
    // Raised if provided authentication token is wrong
    // Also raised if invalid user/password
    exception Unauthorized { };

    // Raised if provided media ID is not found
    exception WrongMediaId { string mediaId; };

    // Raised if some item is requested but currently unavailable
    exception TemporaryUnavailable { };

    ////////////// Custom Types /////////////
    // List of bytes
    sequence<byte> Bytes;

    // List of strings
    sequence<string> StringList;

    // Dictionary with str keys and str values
    dictionary<string, string> DictStrToStr;
```

(continues on next page)

(continued from previous page)

```
    ////////////// File server related interfaces //////////////
    // Handle file transfer
    interface FileHandler {
        Bytes receive(int size, string userToken) throws Unauthorized;
        void close(string userToken);
    };

    // Handle administrative file upload
    interface FileUploader {
        Bytes receive(int size);
        void close();
    };

    // File service
    interface FileService {
        FileHandler* openFile(string mediaId, string userToken) throws Unauthorized,␣
↪WrongMediaId;
        string uploadFile(FileUploader* uploader, string adminToken) throws␣
↪Unauthorized;
        void removeFile(string mediaId, string adminToken) throws Unauthorized,␣
↪WrongMediaId;
    };

    // Interface to be used in the topic for file availability announcements
    interface FileAvailabilityAnnounce {
        void announceFiles(StringList mediaIds, string serviceId);
    };

    ////////////// Catalog service related structs and interfaces //////////////
    // Media info
    struct MediaInfo {
        string name;
        StringList tags;
     };

    // Media location
    struct Media {
        string mediaId;
        FileService *provider;
        MediaInfo info;
    };

    // MediaCatalog service
    interface MediaCatalog {
        Media getTile(string mediaId, string userToken) throws WrongMediaId,␣
↪TemporaryUnavailable, Unauthorized;

        StringList getTilesByName(string name, bool exact);
        StringList getTilesByTags(StringList tags, bool includeAllTags, string␣
↪userToken) throws Unauthorized;

        void renameTile(string mediaId, string name, string adminToken) throws␣
↪Unauthorized, WrongMediaId;

        void addTags(string mediaId, StringList tags, string userToken) throws␣
↪Unauthorized, WrongMediaId;
```

(continues on next page)

(continued from previous page)

```
        void removeTags(string mediaId, StringList tags, string userToken) throws␣
↪Unauthorized, WrongMediaId;

        void getAllDeltas();
    };

    // Interface to be used in the topic for catalog media changes
    interface CatalogUpdate {
        void renameTile(string mediaId, string newName, string serviceId);
        void addTags(string mediaId, string user, StringList tags, string serviceId);
        void removeTags(string mediaId, string user, StringList tags, string␣
↪serviceId);
    };

    ////////////// Auth server /////////////
    class AuthenticatorData {
        string adminToken;
        DictStrToStr currentUsers; // users: passwords
        DictStrToStr activeTokens; // users: tokens
    };

    interface Authenticator {
        string refreshAuthorization(string user, string passwordHash) throws␣
↪Unauthorized;
        bool isAuthorized(string userToken);
        string whois(string userToken) throws Unauthorized;
        bool isAdmin(string adminToken);

        void addUser(string user, string passwordHash, string adminToken) throws␣
↪Unauthorized, TemporaryUnavailable;
        void removeUser(string user, string adminToken) throws Unauthorized,␣
↪TemporaryUnavailable;

        AuthenticatorData bulkUpdate();
    };

    // Interface to be used in the topic for user related updates
    interface UserUpdate {
        void newToken(string user, string token, string serviceId);
        void revokeToken(string token, string serviceId);

        void newUser(string user, string passwordHash, string serviceId);
        void removeUser(string user, string serviceId);
    };

    ////////////// Main server /////////////
    interface Main {
        Authenticator* getAuthenticator() throws TemporaryUnavailable;
        MediaCatalog* getCatalog() throws TemporaryUnavailable;
        FileService* getFileService() throws TemporaryUnavailable;
    };

    interface Announcement {
        void announce(Object* service, string serviceId);
    };
};
```

Descarga

5.6 Lista de requisitos

5.6.1 Servicio principal

- El microservicio envía `announce` con una frecuencia máxima de 10s.
- El microservicio descubre al resto de microservicios recibiendo sus `announce`.
- El mircroservicio descarta aquellos microservicios que han dejado de mandar `announce`.
- El microservicio ignora los anunciamientos de aquellos microservicios que no necesita.
- En caso de haber varios `Authenticator` anunciándose disponibles, no se devuelve siempre la misma referencia.
- En caso de haber varios `MediaCatalog` anunciándose disponibles, no se devuelve siempre la misma referencia.
- En caso de haber varios `FileService` anunciándose disponibles, no se devuelve siempre la misma referencia.
- En caso de que no haya ningún `Authenticator`, se lanza `TemporaryUnavailable`.
- En caso de que no haya ningún `MediaCatalog`, se lanza `TemporaryUnavailable`.
- En caso de que no haya ningún `FileService`, se lanza `TemporaryUnavailable`.

5.6.2 Servicio de autenticación

- El microservicio envía `announce` con una frecuencia máxima de 10s.
- El microservicio descubre al resto de microservicios recibiendo sus `announce`.
- El microservicio descarta aquellos microservicios que han dejado de mandar `announce`.
- El microservicio ignora los anunciamientos de aquellos microservicios que no necesita.
- El microservicio no arranca si en 12 segundos no descubrió ningún `Main`.
- El microservicio detecta si es la primera instancia o cualquiera de las siguientes.
- Si **no** es la primera instancia, es capaz de sincronizar desde otra con `bulkUpdate`.
- El microservicio es capaz de responder a la petición de `bulkUpdate` de una nueva instancia que lo solicitara.
- La instancia genera eventos delta cuando es necesario.
- La instancia procesa los eventos delta adecuadamente
- La instancia ignora eventos de origen desconocido

5.6.3 Servicio de catálogo

- El microservicio envía `announce` con una frecuencia máxima de 10s.
- El microservicio descubre al resto de microservicios recibiendo sus `announce`.
- El microservicio descarta aquellos microservicios que han dejado de mandar `announce`.
- El microservicio ignora los anunciamientos de aquellos microservicios que no necesita.
- El microservicio no arranca si en 12 segundos no descubrió ningún `Main`.
- El microservicio detecta si es la primera instancia o cualquiera de las siguientes.
- Si **no** es la primera instancia, es capaz de solicitar los deltas mediante `getAllDeltas`.
- El microservicio genera los eventos delta cuando se recibe `getAllDeltas`.
- El microservicio actualiza correctamente los `FileProvider` de los archivos acorde a los anuncios recibidos.
- La instancia genera eventos delta cuando es necesario.
- La instancia procesa los eventos delta adecuadamente.
- La instancia ignora los eventos de origen desconocido.

5.6.4 Servicio de archivos

- El microservicio envía `announce` con una frecuencia máxima de 10s.
- El microservicio descubre al resto de microservicios recibiendo sus `announce`.
- El microservicio descarta aquellos microservicios que han dejado de mandar `announce`.
- El microservicio no arranca si en 12 segundos no descubrió ningún `Main`.
- El microservicio ignora los anunciamientos de aquellos microservicios que no necesita.
- La instancia publica la lista de ficheros disponibles periódicamente.
- La instancia publica la lista de ficheros cuando ésta se altera.
- Se para una descarga cuando su token asociado expira, y continúa al renovarlo.

5.6.5 Cliente

- En caso de fallo de conexión al Topic Manager, el cliente intenta reconectarse.
- Espera hasta que descubre un `Main` y conecta automáticamente con él.
- Si descubre varios servicios `Main`, las peticiones no irán siempre a la misma instancia.
- Sólo en modo administrador se permite la monitorización de canales de eventos.
- El monitor es capaz de mostrar los anunciamientos de servicios.
- El monitor es capaz de mostrar actualizaciones delta de instancias `Authenticator`.
- El monitor es capaz de mostrar actualizaciones delta de instancias `MediaCatalog`.
- El monitor es capaz de mostrar los anunciamuentos de ficheros realizados por instancias de `FileService`.

5.7 Notas comunes a todos los servicios y cliente

- Cuando un servicio reciba un anunciamiento, deberá **siempre** imprimir un mensaje en el terminal, de modo que en el mensaje se diga si el anunciamiento ha sido almacenado o ignorado.
- Cuando un servicio reciba una actualización delta deberá **siempre** imprimir un mensaje en el terminal, de modo que en el mensaje se diga si la actualización ha sido tenida en cuenta o se ha ignorado.

5.7.1 Puntos extras para todos los servicios y cliente

- Utilización de la librería `logging` de Python de manera adecuada (hasta 1 punto):
  - Uso de los diferentes niveles de log.
  - Nivel de log configurable.
- El proyecto utiliza integración continua (hasta 1 punto):
  - **No** cuenta la integración contínua ya proporcionada, a no ser que se aporte algo adicional.
- El proyecto es instalable con `pip install` (hasta 1 punto):
  - Nombre del paquete actualizado.
  - Únicamente se instalan los "entry points" necesarios, no todos.
- El proyecto se puede desplegar utilizando Docker (hasta 2 puntos)
- El código obtiene de media en pylint >9.0 (hasta 2 puntos):
  - Si se abusa de los comentarios de "disable" no se considerará apta para éste extra.
- El proyecto dispone de pruebas con cobertura >80% (hasta 5 puntos).

5.8 La entrega

Cada alumno deberá implementar el servicio o cliente que le fuera asignado en **Campus Virtual**.

Se entregará únicamente el enlace al repositorio en GitHub o cualquier otro repositorio accesible a través de Internet en la tarea habilitada para tal efecto en la plataforma de **Campus Virtual**. Dicho repositorio **debe ser privado**, con el fin de evitar plagios, y únicamente se invitará a poder ver el repositorio a los profesores de la asignatura.

Se considerará como entregable el *commit* etiquetado como **`ordinaria-final`**.

Trabajo en equipo

Aunque la práctica es individual, se **recomienda encarecidamente** la colaboración entre alumnos, de modo que cada uno con vuestro servicio o cliente podáis ayudar y ser ayudados por otro compañero que esté implementando otro servicio o el cliente.

De ese modo podréis probar si vuestro servicio se comporta como el cliente o los servicios de vuestros compañeros esperan y estudiar si el comportamiento de la interfaz es el esperado.

5.8.1 Formato de la entrega

El repositorio debe incluir los siguientes ficheros:

- `run_service`: debe estar en la raíz del directorio del proyecto y debe ejecutar la instancia del microservicio implementado. El fichero debe ser ejecutable. En caso de implementar el cliente, este fichero no debe existir en el repositorio..
- `run_client`: debe estar en la raíz del directorio del proyecto y debe lanzar el programa cliente. El fichero debe ser ejecutable. En caso de implementar un servicio, este fichero no debe existir en el repositorio.
- `README.md`: debe estar en la raíz del directorio del proyecto. Será un pequeño documento en lenguaje Markdown o texto plano que explicará paso a paso cómo se debe realizar la configuración del servicio o cliente y cómo lanzarlo usando uno de los scripts listados anteriormente.

  También debe incluir, bajo el título, la URL al repositorio del proyecto, así como cualquier otra información relevante del proyecto que el alumno considere oportuna.

Además, tanto los servicios como el cliente **deberán** aceptar como entrada el proxy al *TopicManager* de IceStorm en forma de **propiedad** en el fichero de configuración.

La propiedad deberá llamarse **`IceStorm.TopicManager`**. Ésto implica hacer un pequeño cambio en el código para que sea capaz de leer el Topic Manager de esa manera:

```
topic_manager = IceStorm.TopicManagerPrx.checkedCast(
    self.communicator().propertyToProxy("IceStorm.TopicManager")
)
```

No se admitirá que el proxy al objeto Topic Manager se proporcione como una cadena escrita directamente en el código, por línea de órdenes o en una propiedad con nombre diferente al descrito en el párrafo anterior. A continuación, algunos ejemplos de soluciones que **no serán admitidas**:

```
topic_manager = IceStorm.TopicManagerPrx.checkedCast(
    self.communicator().stringToProxy("IceStorm/TopicManager -t: tcp -h localhost -p␣
 ↪10000")
)
```

```
topic_manager = IceStorm.TopicManagerPrx.checkedCast(
    self.communicator().stringToProxy(argv[1])
)
```

```
topic_manager = IceStorm.TopicManagerPrx.checkedCast(
    self.communicator().propertyToProxy("TopicMgr")
)
```

5.8.2 Documentación de entrega

No se solicitará memoria de prácticas ni documentación adicional en la entrega. La evolución del proyecto se consultará con el historial de commits de GIT. Por tanto, no se recomienda subir la práctica en uno o dos commits.

5.8.3 Defensa de prácticas

La defensa de la práctica es **obligatoria** si los profesores lo requieren. En ese caso, si algún alumno no se presentara, la práctica se considerará **no presentada**.

Cada alumno requerido para realizar la defensa será convocado por uno de los profesores de prácticas a una hora específica para revisar la práctica con ellos y ejecutarla.

Se hará especial hincapié en ello en los siguientes casos:

- Repositorio con muy pocos commits.
- El buscador de plagios alerte sobre dos o más entregas.

Una vez publicadas las notas, los alumnos tendrán derecho a revisión si no estuvieran de acuerdo.

CHAPTER

SIX

ICEFLIX: DISEÑO DE MICROSERVICIOS - ENTREGA FINAL DE LA CONVOCATORIA EXTRAORDINARIA

El objetivo principal del proyecto es **desarrollar un sistema distribuido basado en microservicios**. Para ello se tomará como modelo la popular plataforma de streaming **NetFlix** para crear una pequeña plataforma de streaming bajo demanda, utilizando algunos de los servicios disponibles en ZeroC ICE. Los objetivos principales de la práctica son:

- Familiarizarse con la invocación a métodos remotos.
- Control de eventos.
- Diseñar algoritmos tolerantes a fallos comunes en sistemas distribuidos.
- Utilización del servicio de distribución de eventos **IceStorm**.
- Aumentar la cardinalidad de cada servicio, pudiendo lanzarse varias instancias del mismo.
- Utilización de los canales de eventos para mantener el estado de la ejecución sincronizado.
- Desacoplar la comunicación entre servicios a través de comunicación indirecta de uno a muchos.
- Fomentar el trabajo en equipos multidisciplinares.

6.1 Introducción

Conocida por todos es la plataforma de *streaming* bajo demanda **Netflix**, lo que quizás no es tan popular es la gran competencia técnica que existe detrás, el increíble sistema distribuido que soporta el uso simultáneo de la plataforma por millones de usuarios en todo el mundo. No todos los detalles de su funcionamiento han sido publicados por la empresa, sin embargo, sí los suficientes para poder hacernos una idea general de su funcionamiento.

Una colección de microservicios soporta el funcionamiento de la plataforma: autenticación, facturación, catalogado, recodificación, análisis y profiling de clientes (big data), etc. Estos microservicios se ejecutan en varias regiones de AWS. Por otro lado, Netflix dispone de sus propios servidores físicos (OCA's) que almacenan los archivos de vídeo y están diseñados específicamente para ofrecer un gran rendimiento en tareas de *streaming*.

Obviamente no se pretende crear una arquitectura de una complejidad semejante, pero sí intentaremos implementar una pequeña maqueta con una serie de microservicios que cooperarán entre sí para ofrecer video bajo demanda de una forma similar a como lo hace la conocida plataforma. No vamos a utilizar mecanismos de caché ni a implementar muchos de los servicios disponibles en la plataforma real: nos centraremos en autenticación, catalogado de medios y transferencia de fichero (en lugar de *streaming*).

Además de lo anterior, debemos enfrentarnos a problemas como mantener la coherencia del estado entre las diferentes instancias de cada servicio, o cómo comunicar ciertos cambios entre los servicios.

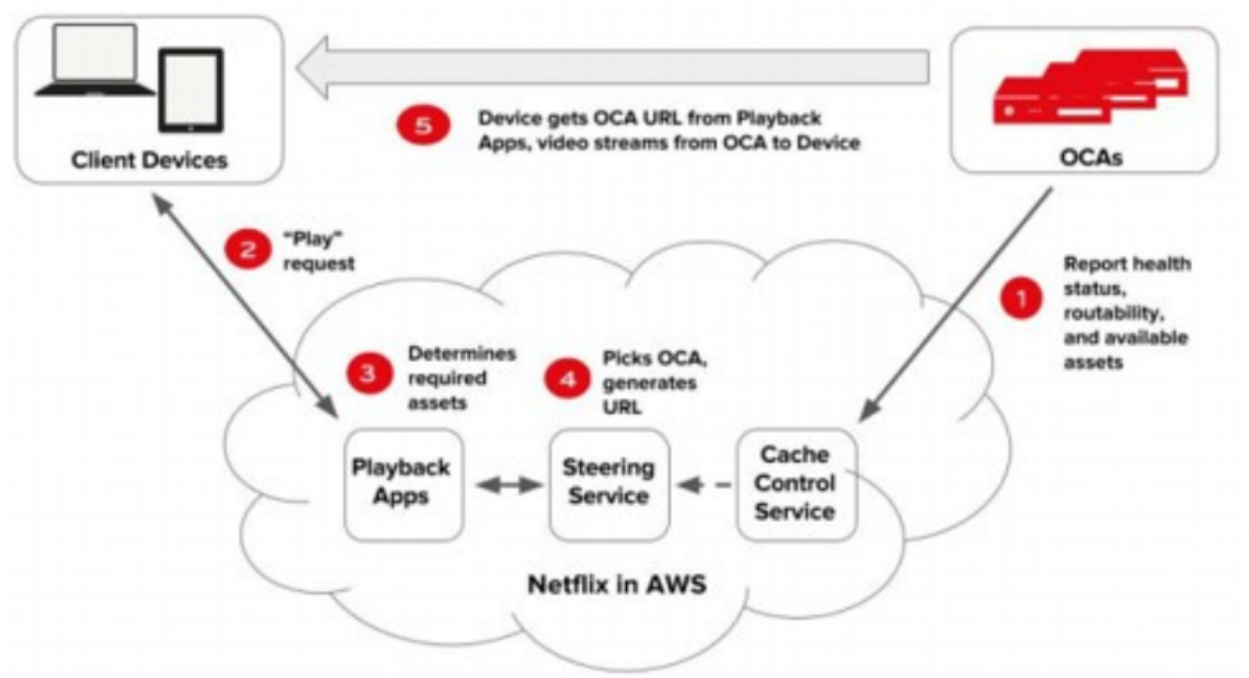


Fig. 6.1: Arquitectura de servicios de Netflix

6.2 El microservicio como base.

El sistema estará dividido en diferentes microservicios, cada uno encargado de realizar una de las diferentes tareas que tiene que realizar el sistema en su conjunto.

6.2.1 La interfaz Slice

Cómo ya sabéis, el alma de la comunicación entre clientes y servicios en un entorno RMI es la interfaz. En ZeroC ICE esa interfaz se define en el IDL específico del middleware, Slice.

El código de la interfaz podéis encontrarlo *aquí*. Se recomienda leer la descripción de los servicios con el fichero Slice, para identificar las interfaz, métodos y argumentos aquí especificados.

6.2.2 Servicio de autenticación.

La autenticación en el servicio se gestiona a través de este servicio. El usuario debe enviar un usuario y contraseña periódicamente, obteniendo un token de autenticación que tendrá una validez limitada en el tiempo (2 minutos). También se encarga de realizar la comprobación de si un token es válido o no para el resto de servicios del sistema.

Para arrancar este servicio debe proporcionarse un token de administración, que deberá ser almacenado en memoria para poder realizar la comprobación de su validez en aquellas operaciones que requieren de dicho token. Éste token se pasará a través del fichero de configuración.

El servicio debe almacenar en una *base de datos* persistente entre reinicios (no necesariamente debe ser SQL) las credenciales de los usuarios. Los tokens, por su naturaleza temporal, no deben aparecer en ese almacenamiento persistente.

Adicionalmente proporciona al administrador la funcionalidad de añadir y eliminar usuarios.

Dicho servicio implementa la interfaz `Authenticator`:

- `refreshAuthorization()`: crea un nuevo token de autorización de usuario si las credenciales son válidas.
- `isAuthorized()`: indica si un token dado es válido o no.
- `whois()`: permite descubrir el nombre del usuario a partir de un token válido.
- `isAdmin()`: devuelve un valor booleano para comprobar si el token proporcionado corresponde o no con el administrativo.
- `addUser()`: función administrativa que permite añadir unas nuevas credenciales en el almacén de datos si el token administrativo es correcto.
- `removeUser()`: función administrativa que permite eliminar unas credenciales del almacén de datos si el token administrativo es correcto.

6.2.3 Servicio de catálogo

El servicio de catálogo proporciona a los usuarios acceso al catálogo y a la descarga de los medios disponibles en la aplicación.

Para ello, debe disponer de un almacén de datos persistente, dónde se almacenará información relativa a los medios: identificador, nombre asignado al medio por el administrador, y los tags asignados por cada usuario del sistema, si las hubiera.

El servicio debe recibir información desde los proveedores de streaming (*explicados más adelante*). para poder ofrecer a la aplicación de usuario un proxy para poder acceder a cada archivo.

El servicio debe implementar la interfaz `MediaCatalog`:

- `getTile()`: permite realizar la búsqueda de un medio conocido su identificador.
- `getTilesByName()`: permite realizar una búsqueda de medios usando su nombre.
- `getTilesByTags()`: permite realizar búsquedas de medios en función de los tags definidos por el usuario.
- `addTags()`: permite añadir una lista de tags a un medio concreto.
- `removeTags()`: permite eliminar una lista de tags de un medio concreto. Los tags que no existan se ignorarán.
- `renameTile()`: operación de administración que permite renombrar un determinado medio en la base de datos.
- `newMedia()`: operación invocada por el servicio de ficheros para informar de un nuevo fichero que puede ser ofrecido a los clientes.
- `removedMedia()`: operación invocada por el servicio de ficheros para informar de que un fichero ha dejado de estar disponible.

6.2.4 Servicio de ficheros

En lugar de implementar un servicio de *streaming* como se la plataforma real, en esta práctica se va a implementar un servicio de descargas.

El servicio de ficheros se encarga de enviar al usuario el fichero para que pueda visualizarlo. El servicio debe leer de un directorio en el disco duro, que será pasado al servicio a través de su configuración, los ficheros que serán compartidos.

El servicio deberá anunciar a los posibles catálogos de forma periódica una lista de los archivos alojados en el directorio. Para ello usará la interfaz `FileAvailabilityAnnounce` como se verá en el *apartado siguiente*.

- `newMedia()`: será emitido por el servicio cuando se encuentre un nuevo fichero o sea subido uno nuevo por el administrador.
- `removedMedia()`: se emitirá cuando un fichero sea eliminado del servicio por el administrador.

El identificador de un medio, utilizado tanto por el servicio de ficheros como por el catálogo, se calculará en función del contenido del fichero (función hash). Para ello, puede utilizarse la suma SHA256 del fichero.

El servicio implementa la interfaz `FileService`:

- `openFile()`: dado el identificador del medio devolverá un proxy al manejador del archivo (`FileHandler`), que permitirá descargarlo.
- `uploadFile()`: dado el token de administrador y un proxy para la subida del archivo, lo guardará en el directorio y devolverá su identificador.
- `removeFile()`: dado un identificador y el token de administrador, borrará el archivo del directorio.

Cuando un usuario solicite abrir un fichero, el servicio creará un sirviente para manejar su posible descarga. Ésta se manejará a través de la interfaz `FileHandler`, que para todas las operaciones solicitará el token del usuario, que debe ser siempre el mismo que solicitó su apertura

Las operaciones disponibles en la interfaz `FileHandler` son:

- `receive()`: recibe el número indicado de bytes del archivo.
- `close()`: indica al servidor que el proxy para este fichero ya no va a ser usado y puede ser eliminado.

Descarga de ficheros

Cuando un usuario abre un fichero a través de `openFile` en el servicio, el manejador de fichero debe ser consciente en todo momento de a qué usuario pertenece el token y, si éste expírase en algún momento, devolver la excepción adecuada.

Si el cliente recibiera dicha excepción durante la descarga, deberá renovar su token y utilizar el nuevo para poder continuar la descarga.

Subida de ficheros

Si el usuario es, además, administrador de la plataforma, puede utilizar el método `uploadFile()` del servicio para realizar una descarga.

Para ello el cliente deberá proporcionar un proxy a un sirviente de la interfaz `FileUploader`, a través del cuál el servicio irá solicitando la subida del fichero al cliente.

6.3 Comunicación entre servicios

En algunas ocasiones, los servicios tienen diferentes métodos que pueden ser utilizados por los clientes o por otros servicios.

El principal problema subyacente para poder comunicarse con un servicio y realizar una invocación remota es averiguar su *proxy*. Para ello, los servicios anunciarán su *proxy* en un canal de eventos.

A su vez, es posible que haya más de 1 servicio de cada tipo ejecutándose a la vez: por ejemplo, podemos tener varios Authenticator o varios MediaCatalog, incluso utilizando diferentes implementaciones.

Dichas instancias de un mismo servicio, en muchas ocasiones, necesitarán comunicarse entre ellas para actualizar sus estados. Para ello, también utilizaremos canales de eventos a los que los servicios se subscribirán para recibir actualizaciones de estado del resto de instancias.

A su vez, el cliente deberá subscribirse al canal de eventos de anunciamientos para ser capaz de descubrir los servicios a los que necesita conectarse para realizar las diferentes tareas.

En el siguiente apartado, se detallarán las comunicaciones que se llevarán a cabo a través de canales de eventos.

6.4 Uso de canales de eventos

Como se ha mencionado en el apartado anterior, la comunicación entre servicios presenta los siguientes problemas:

- Los servicios se deben comunicar de manera indirecta, de uno a muchos, en lugar de de comunicación directa.
- Las distintas instancias de un mismo servicio deberán comunicarse entre ellas para mantener la coherencia de estado entre ellas.

En esta sección analizaremos dónde están las comunicaciones que se verán afectadas por estos nuevos requisitos.

6.4.1 Anunciamientos

Todos los servicios deberán enviar y recibir anunciamientos. De ese modo los servicios recopilarán qué servicios son legítimos y se están anunciando correctamente.

Esa validación será la que haga que un servicio, al recibir un mensaje, lo acepte o lo ignore.

Cada uno de nuestros servicios estará interesado en descubrir a otros, pero quizá no esté interesado en los anunciamientos de todos los servicios.

Para ello se definirá una nueva interfaz llamada `Announcement` que tendrá un método ya conocido: `announce`:

```
interface Announcement {
    void announce(Object* service, string serviceId);
};
```

Nombre del topic

El topic para publicar y recibir los anunciamientos de servicios deberá llamarse `Announcements`. Recuerda utilizar ese nombre para recuperar el topic para asegurar la compatibilidad entre diferentes implementaciones de los servicios.

Todos los servicios deberán enviar sus anunciamientos de la siguiente manera:

- Utilizando la interfaz `Announcement`.
- Enviándolo al topic de nombre `"Announcements"` disponible en IceStorm.
- Únicamente enviarán invocaciones del método `announce`.
- Deberán enviarse cada 10 segundos como máximo.

A su vez, todos los servicios deberán subscribirse a ese mismo topic para recibir todos los anunciamientos.

Anunciamientos no relevantes

Que un servicio reciba los anunciamientos de todos los demás no significa que sean relevantes. Algunos servicios simplemente ignorarán los anunciamientos si vienen de servicios que no le interesen para nada.

Utilizando los anunciamientos como fuente de "servicios conocidos"

Como se menciona anteriormente, las diferentes instancias de un mismo servicio utilizarán un topic de IceStorm para comunicarse y mantener la coherencia de los datos.

Es importante que sepamos que los datos, para ser aceptados por nuestra instancia, deberían venir siempre de una instancia que ya conociéramos anteriormente.

El canal de eventos de anunciamientos será nuestra fuente de "verdad" respecto a servicios conocidos: si recibimos una actualización de datos de nuestro servicio, pero el "serviceId" del mensaje no lo reconocemos, deberemos ignorar el mensaje.

6.4.2 Comunicación entre `Authenticators`

Las diferentes instancias de `Authenticator`, de manera independiente, son capaces de tener sus propias bases de datos de usuarios registrados y los tokens de usuario activos. Para esta entrega, se pedirá que ambas cosas se compartan entre las diferentes instancias a través de un topic IceStorm.

Por ello, el servicio de autenticación deberá convertirse también en un publicador y subscriptor de mensajes en dicho canal de eventos, de acuerdo con esta interfaz:

```
    // Interface to be used in the topic for user related updates
    interface UserUpdate {
        void newToken(string user, string token, string serviceId);
        void revokeToken(string token, string serviceId);

        void newUser(string user, string passwordHash, string serviceId);
        void removeUser(string user, string serviceId);
    };
```

Nombre del topic

El topic para publicar y recibir los cambios ocurridos en una instancia de `Authenticator` deberá llamarse `UserUpdates`. Recuerda utilizar ese nombre para recuperar el topic para asegurar la compatibilidad entre diferentes implementaciones de los servicios.

Actualizaciones no relevantes

Cada vez que se reciba un mensaje desde este topic, el servicio deberá cerciorarse de que el servicio que lo envió es un servicio conocido con anterioridad a través de su `serviceId`.

6.4.3 Comunicación entre `MediaCatalogs`

Los `MediaCatalog` pueden recibir, desde un cliente, cambios en el estado de sus datos: actualizaciones de tags o renombrado de archivos.

Estos cambios en los datos que maneja el servicio deben ser difundidos al resto de instancias a través de un topic en IceStorm, por lo que el servicio se convertirá también en un publicador y subscriptor de ese topic.

Los mensajes transmitidos a través de esa

```
    // Interface to be used in the topic for catalog media changes
    interface CatalogUpdate {
        void renameTile(string mediaId, string newName, string serviceId);
        void addTags(string mediaId, string user, StringList tags, string serviceId);
        void removeTags(string mediaId, string user, StringList tags, string␣
↪serviceId);
    };
```

Nombre del topic

El topic para publicar y recibir los cambios ocurridos en una instancia de `MediaCatalog` deberá llamarse `CatalogUpdates`. Recuerda utilizar ese nombre para recuperar el topic para asegurar la compatibilidad entre diferentes implementaciones de los servicios.

Actualizaciones no relevantes

Cada vez que se reciba un mensaje desde este topic, el servicio deberá cerciorarse de que el servicio que lo envió es un servicio conocido con anterioridad a través de su `serviceId`.

6.4.4 Anunciamiento de archivos disponibles desde el `FileService`

En el caso de los servicios de ficheros, no tenemos el problema de la consistencia de datos que hemos descrito en el resto de servicios, debido a que este tipo de servicio no tiene una base de datos local de ningún tipo.

Aún así, las instancias de los servidores de ficheros necesitarán notificar a un número indeterminado de `MediaCatalog` sobre sus archivos disponibles, para lo que utilizarán también IceStorm.

Para ello se publicará periodicamente en un topic la lista de identificadores de archivo que están disponibles en cada instancia, de modo que todos los `MediaCatalog` podrán saber qué cambios ha sufrido cada servidor de archivos.

```
    // Interface to be used in the topic for file availability announcements
    interface FileAvailabilityAnnounce {
        void announceFiles(StringList mediaIds, string serviceId);
    };
```

Nombre del topic

El topic para publicar y recibir los cambios ocurridos en una instancia de `FileService` deberá llamarse `FileAvailabilityAnnounce`. Recuerda utilizar ese nombre para recuperar el topic para asegurar la compatibilidad entre diferentes implementaciones de los servicios.

Actualizaciones no relevantes

Cuando un servidor de archivos envíe un anuncio, los catálogos que estén subscritos deberán comprobar si ese servicio, identificado por su `serviceId`, ya estaba siendo anunciado antes.

Además, la invocación no contiene el proxy al servicio, por lo que éste deberá ser deducido por el servicio de catálogo de los *anunciamientos de servicio*.

6.5 Arranque y sincronización de los servicios

Una de las nuevas problemáticas que aparecen en la práctica con estos requisitos es que cuando un servicio arranca es posible que ya haya otras instancias de ese mismo servicio funcionando.

Si nuestra instancia únicamente se encarga de cargar su última configuración conocida y aplicar los cambios "delta" que va recibiendo por los canales de anunciamientos, es muy posible que las diferentes instancias de un mismo servicio terminen con "bases de datos" diferentes.

Para ejemplificar esto, imagina que cuando arrancamos una segunda instancia del sevicio de autenticación, el sistema se encuentra en éste estado:

- Hay una instancia del servicio de autenticación funcionando.
- Se han creado 5 usuarios en dicha instancia.
- Hay 2 usuarios con un token activo.

Cuando la segunda instancia se conecta al sistema:

- No tiene conocimiento de los 5 usuarios creados, por lo que si alguno intenta identificarse en estainstancia, fallará.
- No tiene conocimiento de los tokens activos, por lo que si algń servicio intenta validar uno de los tokens, se le dirá que ha expirado.

Ésto también puede ocurrir con otros servicios, por lo que se propone un mecanismo de arranque y sincronización de servicios.

6.5.1 Algoritmo de arranque y sincronización de servicios

1. Cuando el servicio arranca, se subscribe al canal de anunciamiento de servicios.
2. Espera al menos 12s para recibir un anunciamiento de una instancia de un servicio de su mismo tipo.
3. Si en 12s:
    - No ha recibido ningún anunciamiento de otra instacia de su mismo tipo, asumirá que es el primero en arrancar y dará por válida la base de datos persistente que tenga el servicio.
    - Si recibe un anunciamiento de otra instancia, invocará el método de actualización de la misma.

6.5.2 Método `bulkUpdate`

Éste método será utilizado por cualquier instancia del servicio de autenticación que, al arrancar, descubra que hay otras instancias de su mismo tipo funcionando.

El método deberá devolver el tipo de datos definido en la interfaz, dándole a la nueva instancia todos los datos disponibles en ese momento respecto a "tokens" activos, usuarios existentes y la contraseña de administrador.

La instancia al recibir esta estructura deberá actualizar todos los valores en consecuencia.

6.5.3 Método `getAllDeltas`

Éste método será utilizado por cualquier instancia del servicio de catálogo que, al arrancar, descubra que hay otras instancias de `MediaCatalog` funcionando.

El método se invocará sobre la primera instancia de la que se reciba un `announce`, y provocará que esta instancia automáticamente envíe toda la información de renombrado de archivo y de tags de usuarios disponibles mediante el envío de los respectivos mensajes al canal de eventos `CatalogUpdates`.

6.6 Interfaz Slice de IceFlix

```
//
// P1 version
//
[["ice-prefix"]] module IceFlix {

    ////////////// Errors /////////////
    // Raised if provided authentication token is wrong
    // Also raised if invalid user/password
    exception Unauthorized { };

    // Raised if provided media ID is not found
    exception WrongMediaId { string mediaId; };

    // Raised if some item is requested but currently unavailable
    exception TemporaryUnavailable { };

    ////////////// Custom Types /////////////
    // List of bytes
    sequence<byte> Bytes;

    // List of strings
    sequence<string> StringList;

    // Dictionary with str keys and str values
    dictionary<string, string> DictStrToStr;

    ////////////// File server related interfaces /////////////
    // Handle file transfer
    interface FileHandler {
        Bytes receive(int size, string userToken) throws Unauthorized;
        void close(string userToken);
    };

    // Handle administrative file upload
    interface FileUploader {
        Bytes receive(int size);
        void close();
    };

    // File service
    interface FileService {
        FileHandler* openFile(string mediaId, string userToken) throws Unauthorized,␣
↪WrongMediaId;
        string uploadFile(FileUploader* uploader, string adminToken) throws␣
↪Unauthorized;
```

(continues on next page)

(continued from previous page)

```
        void removeFile(string mediaId, string adminToken) throws Unauthorized,␣
↪WrongMediaId;
    };

    // Interface to be used in the topic for file availability announcements
    interface FileAvailabilityAnnounce {
        void announceFiles(StringList mediaIds, string serviceId);
    };

    ////////////// Catalog service related structs and interfaces ////////////
    // Media info
    struct MediaInfo {
        string name;
        StringList tags;
     };

    // Media location
    struct Media {
        string mediaId;
        FileService *provider;
        MediaInfo info;
    };

    // MediaCatalog service
    interface MediaCatalog {
        Media getTile(string mediaId, string userToken) throws WrongMediaId,␣
↪Unauthorized;

        StringList getTilesByName(string name, bool exact);
        StringList getTilesByTags(StringList tags, bool includeAllTags, string␣
↪userToken) throws Unauthorized;

        void renameTile(string mediaId, string name, string adminToken) throws␣
↪Unauthorized, WrongMediaId;

        void addTags(string mediaId, StringList tags, string userToken) throws␣
↪Unauthorized, WrongMediaId;
        void removeTags(string mediaId, StringList tags, string userToken) throws␣
↪Unauthorized, WrongMediaId;

        void getAllDeltas();
    };

    // Interface to be used in the topic for catalog media changes
    interface CatalogUpdate {
        void renameTile(string mediaId, string newName, string serviceId);
        void addTags(string mediaId, string user, StringList tags, string serviceId);
        void removeTags(string mediaId, string user, StringList tags, string␣
↪serviceId);
    };

    ////////////// Auth server ////////////
    class AuthenticatorData {
        string adminToken;
        DictStrToStr currentUsers; // users: passwords
        DictStrToStr activeTokens; // users: tokens
```

(continues on next page)

(continued from previous page)

```
    };

    interface Authenticator {
        string refreshAuthorization(string user, string passwordHash) throws␣
↪Unauthorized;
        bool isAuthorized(string userToken);
        string whois(string userToken) throws Unauthorized;
        bool isAdmin(string adminToken);

        void addUser(string user, string passwordHash, string adminToken) throws␣
↪Unauthorized, TemporaryUnavailable;
        void removeUser(string user, string adminToken) throws Unauthorized,␣
↪TemporaryUnavailable;

        AuthenticatorData bulkUpdate();
    };

    // Interface to be used in the topic for user related updates
    interface UserUpdate {
        void newToken(string user, string token, string serviceId);
        void revokeToken(string token, string serviceId);

        void newUser(string user, string passwordHash, string serviceId);
        void removeUser(string user, string serviceId);
    };

    interface Announcement {
        void announce(Object* service, string serviceId);
    };
};
```

Descarga

6.7 La entrega

Cada alumno deberá implementar el servicio o cliente que le fuera asignado en **Campus Virtual**.

Se entregará únicamente el enlace al repositorio en GitHub o cualquier otro repositorio accesible a través de Internet en la tarea habilitada para tal efecto en la plataforma de **Campus Virtual**. Dicho repositorio **debe ser privado**, con el fin de evitar plagios, y únicamente se invitará a poder ver el repositorio a los profesores de la asignatura.

Se considerará como entregable el *commit* etiquetado como **`extraordinaria`**.

Trabajo en equipo

Aunque la práctica es individual, se **recomienda encarecidamente** la colaboración entre alumnos, de modo que cada uno con vuestro servicio o cliente podáis ayudar y ser ayudados por otro compañero que esté implementando otro servicio o el cliente.

De ese modo podréis probar si vuestro servicio se comporta como el cliente o los servicios de vuestros compañeros esperan y estudiar si el comportamiento de la interfaz es el esperado.

6.7.1 Formato de la entrega

El repositorio debe incluir los siguientes ficheros:

- `run_service`: debe estar en la raíz del directorio del proyecto y debe ejecutar la instancia del microservicio implementado. El fichero debe ser ejecutable. En caso de implementar el cliente, este fichero no debe existir en el repositorio..
- `run_client`: debe estar en la raíz del directorio del proyecto y debe lanzar el programa cliente. El fichero debe ser ejecutable. En caso de implementar un servicio, este fichero no debe existir en el repositorio.
- `README.md`: debe estar en la raíz del directorio del proyecto. Será un pequeño documento en lenguaje Markdown o texto plano que explicará paso a paso cómo se debe realizar la configuración del servicio o cliente y cómo lanzarlo usando uno de los scripts listados anteriormente.

  También debe incluir, bajo el título, la URL al repositorio del proyecto, así como cualquier otra información relevante del proyecto que el alumno considere oportuna.

Además, tanto los servicios como el cliente **deberán** aceptar como entrada el proxy al *TopicManager* de IceStorm en forma de **propiedad** en el fichero de configuración.

La propiedad deberá llamarse **`IceStorm.TopicManager`**. Ésto implica hacer un pequeño cambio en el código para que sea capaz de leer el Topic Manager de esa manera:

```
topic_manager = IceStorm.TopicManagerPrx.checkedCast(
    self.communicator().propertyToProxy("IceStorm.TopicManager")
)
```

No se admitirá que el proxy al objeto Topic Manager se proporcione como una cadena escrita directamente en el código, por línea de órdenes o en una propiedad con nombre diferente al descrito en el párrafo anterior. A continuación, algunos ejemplos de soluciones que **no serán admitidas**:

```
topic_manager = IceStorm.TopicManagerPrx.checkedCast(
    self.communicator().stringToProxy("IceStorm/TopicManager -t: tcp -h localhost -p␣
 ↪10000")
)
```

```
topic_manager = IceStorm.TopicManagerPrx.checkedCast(
    self.communicator().stringToProxy(argv[1])
)
```

```
topic_manager = IceStorm.TopicManagerPrx.checkedCast(
    self.communicator().propertyToProxy("TopicMgr")
)
```

6.7.2 Documentación de entrega

No se solicitará memoria de prácticas ni documentación adicional en la entrega. La evolución del proyecto se consultará con el historial de commits de GIT. Por tanto, no se recomienda subir la práctica en uno o dos commits.

6.7.3 Defensa de prácticas

La defensa de la práctica es **obligatoria** si los profesores lo requieren. En ese caso, si algún alumno no se presentara, la práctica se considerará **no presentada**.

Cada alumno requerido para realizar la defensa será convocado por uno de los profesores de prácticas a una hora específica para revisar la práctica con ellos y ejecutarla.

Se hará especial hincapié en ello en los siguientes casos:

- Repositorio con muy pocos commits.
- El buscador de plagios alerte sobre dos o más entregas.

Una vez publicadas las notas, los alumnos tendrán derecho a revisión si no estuvieran de acuerdo.