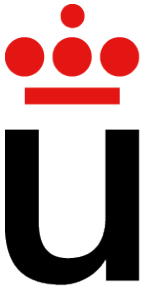


Universidad
Rey Juan Carlos

INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

Curso Académico 2020/2021

Trabajo Fin de Grado

IMPLEMENTACIÓN DE INTEGRACIÓN CONTINUA PARA LAS PRÁCTICAS DE PTAVI

Autor : Sergio de la Fuente García

Tutor : Dr. Gregorio Robles

Trabajo Fin de Grado

Implementación de Integración Continua para las Prácticas de PTAVI

Autor : Sergio de la Fuente García

Tutor : Dr. Gregorio Robles

La defensa del presente Proyecto Fin de Carrera se realizó el día de julio de 2021, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de julio de 2021

Dedicado a

mi abuelo

II

Agradecimientos

En primer lugar quiero agradecer todo el esfuerzo de toda una vida a mis padres, Nuria y Carlos. Es todo lo que ellos me han inculcado lo que me ha permitido llegar aquí. Gracias mamá por el coraje que siempre me has enseñado, gracias papá por la serenidad que me has transmitido.

En segundo lugar, quiero dar las gracias a dos personas sin las que su apoyo me habría hecho zozobrar, gracias Belén, gracias Ana, habéis sido el apoyo y el consuelo en mis momentos más duros.

Seguidamente quiero dar las gracias a Gregorio, ya no por ser el tutor en este proyecto, sino porque fuiste mi profesor en PTAVI, me enseñaste Python y fuiste fundamental para mi recorrido actual.

Quiero dar las gracias especialmente a mi tío Álvaro, la primera persona que me enseño lo que era una derivada, y que me mostró la belleza de las carreras técnicas. Gracias a mis abuelas que han sufrido lo mismo que yo con esta carrera. Gracias a toda mi familia, una familia que siempre ha estado pendiente y que me ha apoyado en este proceso. Gracias a Luis, porque hay veces en que el apoyo no es suficiente. Gracias a Sergio, por todos los trabajos que hemos hecho juntos, me llevo un compañero de noches en vela y sobre todo un gran amigo. Gracias a Bea e Irene que nunca dudaron en ayudarme cuando se me atragantó alguna asignatura. Gracias a Carlos y a Álvaro, que siempre entendieron lo importante que era esto para mí. Gracias a Raquel y a Nerea, que siempre se mostraron dispuestas a echarme una mano. Gracias a Álvaro que nunca dudó en cubrirme unas horas para que pudiera realizar este proyecto. Gracias a mis profesores y a la universidad por darme todas las armas para enfrentarme a esta carrera.

Y por último, gracias a mi abuelo, porque me enseñaste a ser trapero del tiempo.

Resumen

Este trabajo de fin de grado tiene como finalidad la implementación de integración continua en la asignatura de Protocolos para la Transmisión de Audio y Vídeo en Internet (PTAVI) del tercer curso del Grado en Ingeniería en Sistemas Audiovisuales y Multimedia. Durante todo el documento explicaremos el proceso que se ha seguido para intentar conseguir el objetivo. Se ofrecerá una visión del estado actual de las prácticas y el concepto que queremos conseguir.

Se diseñará además todo un ecosistema para ayudar al alumno y al profesor en este camino. Se abordarán diferentes tecnologías y cómo ha ido evolucionando el trabajo de fin de grado con ellas. Utilizaremos tests unitarios y tests *endtoend* para generar toda una suite que valide nuestras prácticas. Trabajaremos con guías de estilo y las automatizaremos para mejorar la calidad del código.

El trabajo de fin de grado está diseñado para que no solo sea un proyecto sino para que se pueda convertir en una herramienta dentro de la asignatura de PTAVI. Para ello se han realizado las configuraciones correspondientes y los desarrollos de los tests que fueran convenientes.

Por último se han generado versiones estables de todas las implementaciones. Se explicarán los resultados y se analizarán, concluyendo si se ha conseguido, o no, el resultado que esperábamos.

Summary

This final degree proyect has the finality of the implementation of continous integration into the subject of PTAVI. Along all the document we will explain all the process we have followed for trying to achieve the objective. We will offer a vision of the actual state of the PTAVI exercises and the concept we want to perform.

A design of a new environment will be provide to help the student and the professor on their path into the CI implementation. We will abord different technologies and how has the proyect evolved with them. Unit and end to end tests will be used to create all of a suite that validates the exercises. We will work with guide styles and automate them for increasing the quality of the code.

This proyect is not only design for being the final proyect of a degree, but hopefully, to be a tool that someday integrates into the PTAVI subject. For this we have made the necessary configurations and the development of the convenient tests.

At last, stable versions of all the implementations are made. We will see the results and analyse them, concluding if finally the project has achieve the results we were expecting.

Índice general

Índice de figuras

Capítulo 1

Introducción

"Juntarse es un comienzo. Seguir juntos es un progreso. Trabajar juntos es un éxito"

Henry Ford

Los seres humanos llevan trabajando juntos desde hace milenios, ya sea la caza de un mamut o la elaboración de una complicadísima aplicación, el ser humano siempre ha trabajado en grupo. Este trabajo de fin de grado habla de esto, del trabajo en grupo, de cómo el trabajo de un individuo puede ser crucial en todo un sistema de millones. Habla de cómo integrar ese trabajo, de metodologías modernas que implantamos para mejorar la productividad y la calidad.

En este trabajo veremos que la implantación de ciertas prácticas en una asignatura de universidad puede ser comparable a las implementaciones actuales en sistemas de producción profesionales. Además intentaremos involucrar no solo al alumnado sino también al profesorado. Tratando entonces a la asignatura como un sistema de producción optimizado de aprendizaje.

A lo largo de los capítulos veremos cómo se ha implementado un sistema de integración continua en las prácticas de la asignatura de Protocolos para la Transmisión de Audio y Vídeo en Internet (PTAVI) del tercer curso del Grado Ingeniería de Sistemas Audiovisuales y Multimedia; y qué significa esto.

1.1. Motivaciones

Cuando se me presentó el primer boceto de este trabajo de fin de grado me pareció una oportunidad para poder profundizar en un concepto que conocía levemente de mi etapa profesional

y que quería saber aplicar y encontrar por qué era tan popular. Además juntaba una de mis asignaturas preferidas de el grado con mi lenguaje preferido de programación, lo que desde el primer momento me hizo aceptar el reto. Consideré asimismo que la posibilidad de implementar algo como esto desde cero me daría perspectiva cuando yo tuviese que seguir una metodología continua o diseñarla en el mundo profesional. Y, por último, aportar mi granito de arena a una asignatura como PTAVI, u otras parecidas, que me fascinó desde el primer momento.

1.2. Estructura de la memoria

La estructura de esta memoria es la siguiente:

- Introducción: Este capítulo contiene una breve introducción a todo el concepto del trabajo en grupo y mis motivaciones personales para hacer este proyecto.
- Objetivos: En este capítulo se explican los objetivos de una manera general y de una manera especifica enumerando uno por uno todos los objetivos de este proyecto.
- Estado del arte: En este capítulo se explicará en profundidad todas las tecnologías y conceptos utilizados en este proyecto.
- Diseño e implementación: En este capítulo veremos cómo se ha implementado todo, desde la arquitectura que se ha seguido, a la implementación en las prácticas una por una.
- Experimentos, validaciones y resultado: En este capítulo veremos como ha sido el proceso de validar la implementación en las prácticas y que resultado hemos obtenido.
- Conclusiones: Aquí veremos que objetivos se han conseguido y cuales no, cómo mejorar el proyecto y cómo continuarlo.
- Apéndice: Se detallarán en el apéndice los nuevos guiones después de la implementación del proyecto. También se podrá consultar la nueva estructura de carpetas diseñada.

Capítulo 2

Objetivos

2.1. Objetivo general

El objetivo de este proyecto es generar un ecosistema de integración continua en la asignatura de PTAVI. En consecuencia, se pretende alcanzar dos metas: i) enseñar al alumno una ruta de aprendizaje y un sistema de trabajo más interesante y más cercano a la realidad, y ii) facilitar al profesor la corrección de las prácticas de la asignatura.

El objetivo a su vez se centra no solo en generar este ecosistema, sino en decidir de que manera se podría implementar reduciendo el impacto total en la actual arquitectura de las prácticas y en ninguna manera aumentar la curva de dificultad del alumno al realizar las mismas.

2.2. Objetivos específicos

Este proyecto tiene como objetivo generar un ecosistema de integración continua, sin embargo, este concepto resulta ciertamente amplio como para abarcarlo de una manera general. Es por ello que se han definido una serie de objetivos específicos enunciados a continuación:

1. Decidir de que manera y bajo que entorno se va a realizar el ecosistema. En esta fase se decide cómo se va a afrontar el problema y que tecnologías se van a usar.

2. Incluir tests [1] unitarios en las prácticas para controlar la calidad de las mismas.

[1] Según la R.A.E Nueva gramática de la lengua española.3.4p [10]. No se recomienda el uso de *tests* como plural de test, sin embargo se ha decidido que la ambigüedad que esto podía acarrear era demasiada.

3. Incluir tests de integración o tests *end to end* en algunas prácticas en las que los tests unitarios no pueden asegurar el correcto desarrollo de las mismas o que resultan demasiado restrictivos para los alumnos.

4. Incluir una guía de estilo y su posterior comprobación para ensenar al alumno buenas prácticas en lo que a la escritura de código se refiere.

5. Verificar y experimentar el sistema con repositorios de alumnos seleccionados al azar.

6. Modificar algunos enunciados de las prácticas de la asignatura de una manera que no impacte o impacte lo mínimo posible tanto a profesor como alumno.

2.3. Planificación temporal

A continuación, se puede visualizar el cronograma de este Trabajo Fin de Grado.

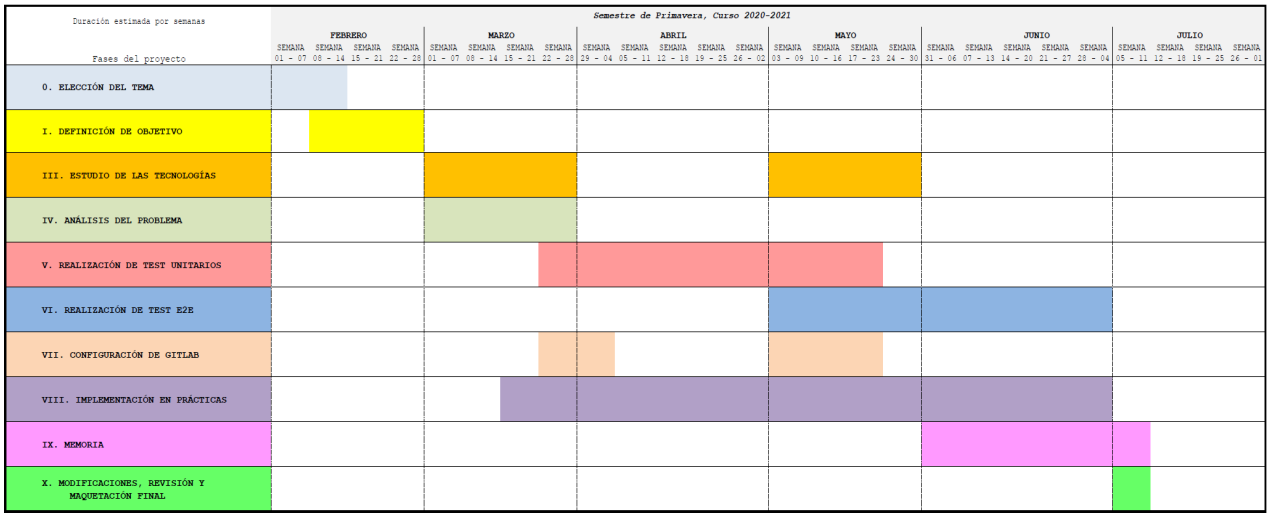


Figura 2.1: Cronograma de la realización del proyecto.

Capítulo 3

Estado del arte

3.1. Tests

En todo este proyecto se va continuamente a hacer referencia a los llamados tests. En este apartado se va a explicar qué es un test. La Real Academia de la Lengua Española define la palabra test de la siguiente manera:

> *Test. M. Prueba destinada a evaluar conocimientos o aptitudes, en la cual hay que elegir la respuesta correcta entre varias opciones previamente fijadas.*
>
> [5]

Esta definición se ajusta bastante bien a nuestro sentido de test desde un punto de vista tecnológico. El objetivo del mismo será evaluar ciertas aptitudes de una determinada funcionalidad o funcionalidades. Imaginémonos que diseñásemos una calculadora, ¿podríamos saber si en todas las operaciones diseñadas se está resolviendo correctamente el resultado?, rápidamente se nos viene a la mente el probar las operaciones introduciendo números y (conociendo previamente el resultado) comprobar si el resultado es correcto. Básicamente esto es lo que hace un test. Dada una funcionalidad y conociendo previamente el resultado o el comportamiento que queremos conseguir, comprobamos que la funcionalidad realiza el comportamiento esperado. Esto puede parecer plausible en el ejemplo de la calculadora, sin embargo, según crecen las funcionalidades el poder comprobar manualmente un resultado esperado se torna realmente complicado. Lo ideal sería poder de alguna manera automatizar todo este proceso, generando baterías de pruebas en todas las funcionalidades. Esto es un sistema de testing o un test suite.

Dentro de este ámbito hay que tener en cuenta que se puede probar tanto aplicaciones grandes como funcionalidades pequeñas y por lo tanto existen distintos tipos dentro de los tests más adecuados para unas tareas u otras. Dentro del testing existen tres grandes tipos:

- Tests unitarios: Encargados de evaluar funcionalidades concretas, así pues, un test que controle que el resultado de una suma es correcto en una calculadora es un ejemplo de test unitario. Fundamentales a la hora de comprobar las funcionalidades son el primer eslabón del testing.

- Tests de integración: Son los encargados de evaluar que nuestra funcionalidad trabaje en sintonía con el resto de funcionalidades ya desarrolladas. Así, por ejemplo, una función que acceda a una base de datos no debe entrar en conflicto con el resto de funciones o la propia base de datos. Son más costosos de evaluar que los tests unitarios ya que se debe tener control sobre nuestra funcionalidad y el resto de funciones con las que podría fallar.

- Tests *end-to-end*: Son los encargados de evaluar el funcionamiento de todas las funcionalidades replicando un entorno de ejecución completo. Estas pruebas deben ser capaces de evaluar el flujo total de la aplicación. Son pruebas muy costosas ya que generalmente hay que construir un entorno completo antes de realizarlas. Además deben ser modificadas siempre que cambie en algo el flujo de la aplicación.

Además de estos tres tipos puede darse la prueba funcional o de negocio. Realmente estas pruebas serían una mezcla de las tres anteriores, evaluando que se cumple una cierta funcionalidad establecida por un equipo de decisión.

Los tests deberían ser una parte fundamental del código. Funcionan como un control de calidad y hacen que nuestro código sea mantenible, evitando así ciertos *bugs* que se dan por una mala calidad del código. Realizar tests a nuestros desarrollos, además, nos ayuda a mejorar como desarrolladores, ya que podemos ver la funcionalidad desde un prisma diferente y tener un código más robusto.

3.2. Integración y Despliegue Continuos

El titulo de este proyecto es *"Implementación de integración continua para las prácticas de PTAVI"*, sin embargo, ¿a qué nos referimos con integración continua?

El CI/CD [2] es un concepto que significa *"Continous Integration, Continous Deployment"*, oséase, integración y despliegue continuos. Para ello tenemos que situarnos en un entorno de metodología continuo como podría ser *agile* [1], las metodologías ágiles abogan por adaptar la forma de trabajo a las condiciones del proyecto y transformarse con él, y es aquí precisamente donde entra la filosofía del CI/CD, el poder integrar nuestro desarrollo en todo momento de una manera continua según vamos creciendo con el proyecto.

Para entender el concepto lo mejor es situar dos equipos con un mismo fin siguiendo dos metodologías diferentes. El fin podría ser el desarrollo de una aplicación Web, que se compondrá de un *frontend,* y de un *backend* a la vez dividido en dos, una parte que se ocupe de gestionar y mantener una base de datos y otra parte encargada del desarrollo de una *API-REST* para la comunicación con el frontal. Los dos equipos deciden desarrollar la aplicación con los mismos medios, el mismo número de personas y las mismas subdivisiones.

El primer equipo decide que en una primera fase se completará el desarrollo de cada parte y en una segunda fase, una vez hayan terminado todas las subdivisiones, pondrán en común todo lo desarrollado por cada una de ellas. Finalmente se testearan todas las funcionalidades y se ensamblará todo en el proyecto final.

El segundo equipo, mientras tanto, decide que según vayan desarrollando se va a ir subiendo a un repositorio común. Realizarán tests automáticos de la funcionalidad propia y de la integración en la aplicación. Ninguna funcionalidad que no pase con éxito los tests podrá ser incluida en el repositorio de la aplicación, manteniendo así, la calidad del código. Según se vayan añadiendo funcionalidades al repositorio, se irán generando versiones de la aplicación final. Esto produce que las fases mencionadas en el otro equipo se acaben entremezclando en este.

En este caso hipotético, el segundo equipo sería el que implementa CI/CD. A primera vista podemos apreciar ciertas diferencias entre un equipo y el otro. Dado un problema de integración entre subdivisiones, como pudiera ser el fallo en un servicio de comunicación entre el *front* y la *API*, el primer equipo lo detectaría en su fase 2 o en la fase de testeo, mientras que el segundo equipo lo detectaría en el momento en el que se intentase integrar, que ocurriría acto seguido al desarrollo del mismo.

Otro problema común es la ampliación de funcionalidades en una aplicación. Esto con un sistema de CI/CD no supone ningún cambio, a excepción del tiempo que se tarde en desarrollar

esa funcionalidad. Sin embargo, en el caso del primer equipo, si esa funcionalidad solo tuviera que impactar en una subdivisión, es probable que haya que rehacer funcionalidades en todas las subdivisiones.

En conclusión, en CI/CD iniciaríamos con una definición de una nueva funcionalidad y según fuésemos desarrollando lo iríamos integrando en el proyecto. De esta forma nos permitiría atajar cualquier cambio de impacto de una manera más sencilla, podríamos controlar cómo se integra el código nuevo y maximizar su calidad mediante los bancos de pruebas o tests ya mencionados. En definitiva, podríamos automatizar todo el proceso generando coherencia y cohesión en el código.

A continuación, se muestra un esquema que nos permitirá explicar los procesos CI/CD que se han seguido de una manera gráfica. Se ha extraído la base del esquema de la documentación de GitLab, ya que además de ser muy descriptivo, parecía adecuado ya que es el devOps que se va a utilizar en este proyecto.

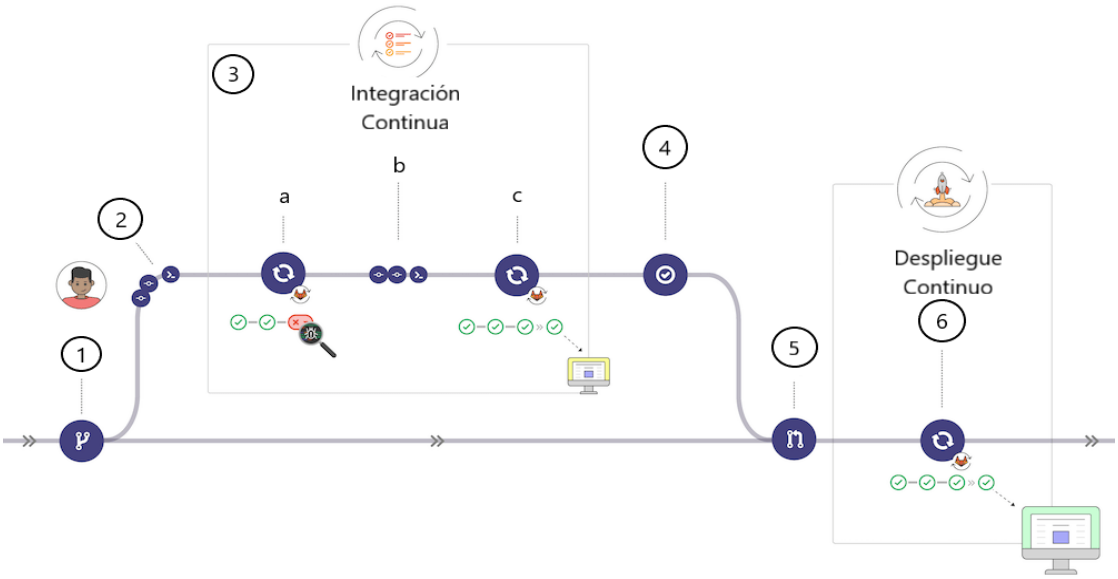


Figura 3.1: Esquema de un desarrollo de una nueva funcionalidad siguiendo CI/CD en GitLab.

1. Se crea una nueva rama del proyecto principal.

2. Una vez se ha desarrollado la nueva funcionalidad se suben los cambios al repositorio de la rama que se ha creado.

3. *a*) Al realizar el push uno de los tests nos devuelve un fallo

b) Arreglamos el fallo

c) Volvemos a realizar un push y ahora sí, los tests devuelven un resultado satisfactorio.

4. La rama que hemos creado tiene ahora el nuevo desarrollo y el sello de que todo está validado.

5. Se combina (merge) la rama creada con la rama del proyecto principal.

6. En cuanto la rama detecta un cambio se despliega automáticamente, quedando nuestra nueva funcionalidad en producción.

3.3. GitLab

GitLab es una aplicación web que nos permite controlar y desarrollar versiones del código basado en GIT. En este apartado vamos a analizar la aplicación de CI/CD que es en lo que nos vamos a centrar en todo este proyecto.

GitLab CI/CD [6] es una plataforma *opensource* de DevOps que ofrece todo un entorno preparado para la integración y despliegue continuo. Te permite cubrir todas las fases de la vida útil de un producto o aplicación. GitLab CI/CD brinda repositorios donde almacenar el código, una *wiki* donde tener toda la documentación del producto, un sistema de monitorizado a tiempo real, una capa de seguridad e incluso una capa de producción de paquetes o *releases* además de un despliegue continuo de la aplicación. GitLab CI/CD brinda una infraestructura con plataforma de *kubernetes*(parecido a *docker*) que nos permite alojar contenedores en la nube y una plataforma *serverless* que permite alojar paginas web estáticas sin el lado del servidor.

Dentro de la herramienta de CI/CD se ofrece un editor, donde podemos escribir código para el archivo de configuración de la herramienta, un apartado donde podríamos escribir test cases para por ejemplo pruebas de integración y por último Pipelines, Jobs y Schedule.

Los llamados Pipelines y Jobs, nos permiten programar tareas que se ejecutan en un determinado momento. No podemos continuar sin explicar que son estas Pipelines y Jobs:

- Job: Cada tarea que ejecutamos es un job, los tests o la revisión de estilo con el *pep8* son ejemplos de job. Veamos un ejemplo:

- Pipeline: La consecución de tareas o Jobs conforma un Pipeline. Esto permite ejecutar las tareas siguiendo un orden preconcebido.

A continuación en las figuras 3.2 y 3.3 , podemos ver unos ejemplo de cada caso:

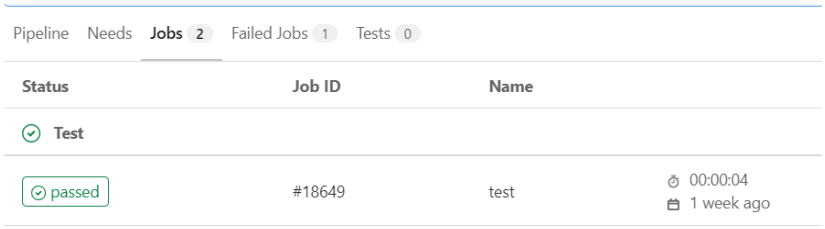


Figura 3.2: Job

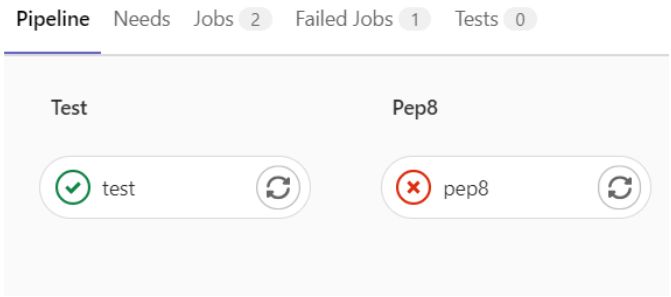


Figura 3.3: Pipeline

Como se aprecia en las imágenes el test en sí es un job que se ha ejecutado y se ha resulto satisfactoriamente, mientras que en la siguiente imagen se puede ver un Pipeline que lo compone los test y el PEP8, donde el PEP8 ha devuelto un resultado no satisfactorio. El hecho de que esa funcionalidad se integre o no, dependerá de lo que nosotros hayamos decidido con anterioridad.

Así pues podemos desarrollar tareas antes o después de un commit o de un push, evitando por ejemplo con una batería de tests pre-push que se integre un desarrollo con un error y notificando correctamente al responsable del desarrollo o del proyecto.

Por último la herramienta Schedule, del alemán calendario, nos permitiría programar de manera periódica o manera futura una serie de pipelines, por ejemplo nos podría interesar que todos los lunes se ejecutase una determinada tarea de eliminación de ramas que ya no están en uso.

3.4. Unittest

Unittest [12][11] es la herramienta que nos permite realizar los tests, inspirada en JUnit (herramienta equivalente para java) nos permite generar tests en Python tanto para clases y módulos enteros como para scripts. Creada por Kent Beck permite la automatización de los tests y la opción de crear colecciones, además no está ligado a ningún marco de reportes por lo que nos permite utilizar el más conveniente cómo puede ser el `coverage.py`, que es el utilizado en nuestro caso. Para todo lo mencionado Unittest se sirve de *tests fixtures, test cases, test suites y test runner*, propios de la programación orientada a objetos, conceptos explicados a continuación.

- Test fixture: lo definiríamos como la preparación del entorno pre-proceso de todas las características del test que queremos ejecutar, así pues, un servidor o una base de datos que requiera el programa original habrá que emularlo de alguna manera, aquí es donde este concepto entra en juego. En nuestro caso podemos ver varios casos en los que hay que instanciar la clase original de la práctica. Esto lo hacemos con el método `setUpClass`, donde inicializamos valores propios o el decorador `@classmethod` donde simulamos los métodos que usa la misma.

- Test case: es la unidad individual del test, nos permite generar un test unitario para testear cada una de las prácticas. En Unittest tenemos una clase base para esto, la clase TestCase, clase la cual heredan todos nuestros tests de modo que al realizar nuestros tests realmente lo que se generan son subclases de esta misma.

- Test suite: es la colección de los test cases que se ejecutan y agrupan juntos, también puede existir una colección de test suites. Ejecutar una suite es lo mismo que ejecutar una serie de test cases corriéndolos individualmente, en caso de este proyecto se puede apreciar su uso en el script que veremos más adelante.

- Test runner: es el componente encargado de ejecutar nuestros tests, en nuestro caso es la última línea del test `unittest.main()`, sin embargo cabe destacar que como ya veremos los tests son ejecutados por un script desarrollado especialmente para este proyecto.

El uso de la herramienta es simple ya que viene incluido en la distribución de Python 3.8 que es la utilizada en el proyecto, por lo tanto, únicamente hemos tenido que importarlo y trabajar con

ella. Para cada suite de pruebas tenemos que crear una clase que herede de unittest.TestCase, y añadir la serie de métodos que queremos testear. Es importante que todos estos métodos comiencen con la palabra `test` ya que el intérprete necesita esta particularidad para detectarlo como prueba unitaria, estos serán cada una de las pruebas que queremos ejecutar dentro de esa batería de pruebas. Una vez que se ejecuten los tests, se ejecutarán todos los métodos cuyo nombre comience con la palabra *test*, en orden alfanumérico. Una vez ejecutados podemos tener un resultado exitoso o un resultado de error, a continuación podemos ver los tres casos posibles:

1. Caso de Éxito.

```
........
----------------------------------------------------------
---------------
Ran 8 tests in 0.004s

OK
```

2. Assert Error.

```
.F......
==========================================================
==============
FAIL: test_plus (test.test_calc.TestCalc)
Should sum the operands. They have to be ints
----------------------------------------------------------
-------------
Traceback (most recent call last):
  File "C:\Users\Belen\Documents\projects\PTAVI_TFG\ptav
  i-p2\test\test
  _calc.py", line 13, in test_plus
    self.assertEqual(result, expected_result)
AssertionError: 6 != 5
```

```
----------------------------------------------------------
--------------
Ran 8 tests in 0.003s

FAILED (failures=1)
```

3. Excepción por error en el test.

```
....E
==========================================================
==============
ERROR: setUpClass (test.test_calcoohija.TestCalculadora)
----------------------------------------------------------
--------------
Traceback (most recent call last):
  File "C:\Users\Belen\Documents\projects\PTAVI_TFG\ptavi
  -p2\test\test_calcoohija.py", line 13, in setUpClass
    cls.test_calculadora = calcsfoohija.CalculadoraHija(c
    ls.test_value1, cls.test_value2)
NameError: name 'calcsfoohija' is not defined

----------------------------------------------------------
-------------
Ran 4 tests in 0.002s

FAILED (errors=1)
```

Podemos ver que dentro de los dos estados, acierto y error tenemos tres casos:

1. **OK**: El test no ha fallado y se ha obtenido el resultado que se esperaba.

2. **ASSERT ERROR:** El test devuelve que el resultado no es el esperado

3. **EXCEPTION ERROR:** Se ha elevado una excepción, aquí habría que distinguir entre dos casos, el test tiene un error en su codificación o es el programa que estamos testeando el que al ejecutar ha lanzado una excepción por algún motivo.

Tenemos que tener en cuenta que en el tratamiento de errores debemos ser muy cuidadosos, ya que en el último caso podríamos tener un error en nuestro fixture, que sea el que precisamente haga fallar el programa que estamos testeando al no servirle los datos o características que necesita. Para poder testear todo esto de lo que estamos analizando, la clase unittest.TestCase nos brinda de una gran variedad de métodos que podemos encontrar en su documentación `https://docs.python.org/3/library/unittest.html#test-cases`, sin embargo, a pesar de la cantidad de métodos que detalla su documentación se ha utilizado el `assert` como método básico para el banco de pruebas y el `setUp` o el decorador `@classmethod` como preparación del entorno. Para terminar debemos explicar la ejecución básica de un test con unittest, mediante línea de comandos el comando

```
$ python -m unittest tests/test_something.py
```

ejecutaríamos un test (siempre que tenga el método main() implementado). En nuestro caso esto no se da ya que tenemos un script que ejecutara todo nuestro test suite.

3.5. Python

Python [7] [11] es un lenguaje de programación multiparadigma creado por Guido van Rossum en 1989 e implementado en los anos 90. Es un lenguaje orientado a objetos pero su condición de multiparadigma hace que sea apto para programación imperativa o funcional.

Es un lenguaje interpretado, esto significa que no compila o traduce el lenguaje a código maquina, sino que hay un intérprete que a diferencia del compilador solo realiza la traducción a medida que es necesario, típicamente instrucción por instrucción y no guardan el resultado de la traducción. Los lenguajes interpretados suelen ser más lentos ya que necesitan traducir el programa mientras se está ejecutando esto además los puede hacer menos robustos a errores de ejecución pero a cambio los hace más flexibles y sencillos a la hora de programar y depurar; además se depende únicamente del intérprete y no de la maquina donde se ejecute lo que nos

permite generar máquinas virtuales. Python a su vez se podría concebir como un lenguaje semi interpretado esto significa que realiza una traducción a un pseudo código intermedio que se conoce como bytecode. En cuanto ejecutamos el comando Python en alguno de nuestros script podemos ver que se genera un archivo `.pyc` que es el que se ejecutará en futuras ocasiones.

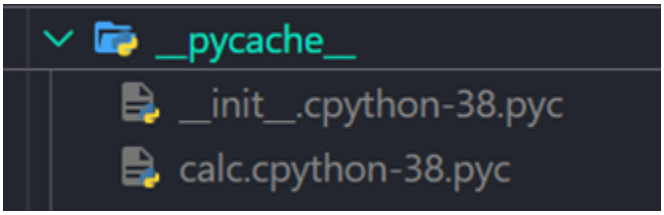


Figura 3.4: Ejemplo de archivos .pyc

Python a su vez es un lenguaje con un tipado dinámico, esto significa que no es necesario declarar el tipo de variable, sino que se declarará en tiempo de ejecución dependiendo el valor que se le asigne. Por el contrario una variable no podrá ser utilizada como un tipo diferente al asignado sin una conversión explícita, por lo tanto es un lenguaje fuertemente tipado.

Por último indicar que es un lenguaje multiplataforma y que se pensó como lenguaje amigable y sencillo con todo un manifiesto "zen"propio: `https://www.python.org/dev/peps/pep-0020/` del que destaco algunas de sus consignas:

- Hermoso es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Los casos especiales no son lo suficientemente especiales para romper las reglas.

3.5.1. Pip

Pip es un gestor de paquetes para el ecosistema de Python. Capaz de instalar y de administrar paquetes mediante línea de comandos es el gestor con el que se han instalado todos los módulos incluidos en este proyecto que no trae por defecto Python 3.8. Pip suele estar instalado siempre que tengamos instalado Python sin embargo podría no ser así si se utiliza una versión anterior

a 2.7 de Python. En ese caso se puede descargar siguiendo su documentación en: `https://pypi.org/project/pip/`.

3.6. Coverage

Coverage.py [3] es una herramienta que mide la cantidad de código que cubre un cierto test para programas en Python. La herramienta monitoriza que líneas del código están ejecutando nuestros tests y cuáles no. Esto va a permitir que ganemos efectividad cuando desarrollemos nuestros tests y además vamos a ganar en rapidez, ya que como veremos ahora coverage provee de una función por la que genera un informe detallado línea por línea en un *.html*. Coverage está preparado para trabajar con tres bibliotecas de tests en Python: *pytest, unittest y nosetest.*

Coverage es una herramienta no incluida en Python 3.8 por lo tanto el primer paso es instalarlo. Para instalarlo basta con ejecutar el siguiente comando en una terminal:

```
pip install coverage
```

Una vez instalado es conveniente hacer una pequeña configuración, a pesar que coverage trae una configuración previa, es difícil que vaya a ser la configuración necesaria para nosotros. Debemos crear, si no se ha creado ya, un archivo con el nombre *.coveragerc*, aquí definiremos ciertas reglas como qué archivos se deben monitorizar o que líneas deseamos saltar por cualquier motivo. Por ejemplo, un caso común es excluir la carpeta de tests o el *__init__.py* ya que no nos interesa la cobertura de los tests en estos archivos la cual va a ser nula. En cuanto tengamos todo preparado es el momento de ejecutar la herramienta. La forma más fácil es ejecutar el siguiente comando:

```
coverage run -m unittest discover
```

esto arrojará un resultado de la cobertura de los tests como se ve en la figura 3.5. Cabe destacar que en el caso de nuestro proyecto todo lo relacionado a el coverage lo ejecuta un script que gestiona los tests denominado *run_test.py*.

Como ya hemos mencionado se puede generar un informe en *.html* que detalla línea por línea la cobertura actual del test. Esto nos permite mucho más contexto porque además en caso de fallo podemos apreciar hasta que punto del archivo ha llegado el test. Para generar este informe basta con ejecutar el comando

```
$ coverage report -m
Name                      Stmts   Miss  Cover   Missing
-------------------------------------------------------
my_program.py                20      4    80%   33-35, 39
my_other_module.py           56      6    89%   17-23
-------------------------------------------------------
TOTAL                        76     10    87%
```

Figura 3.5: Ejemplo de cobertura en terminal

```
coverage html
```

y se creará una carpeta denominada *htmlcov* que contiene una serie de archivos *.html, .js, .css, .json,* y alguna imagen que se utilizaran para crear el informe detallado. Sí abrimos el *index.html* veremos un resumen de todos los archivos en los que se ha realizado la cobertura, pulsando en cualquiera de ellos veremos el informe línea por línea como el que se ve en la figura 3.6.

Coverage además se puede utilizar como módulo en Python que es como se ha utilizado en este proyecto. Provee de una clase denominada `Coverage` que contiene todos los métodos y la posibilidad de generar y mostrar todos los informes que se han explicado.

3.7. Guía de estilo PEP8

Una guía de estilo es una especie de manual de buenas prácticas que se sigue a la hora de escribir código. Es importante tener en cuenta que incluso una aplicación pequeña está escrita por un grupo de desarrolladores y que es fundamental que el código de todos sea legible y entendible por todos. Es por esto que existen las guías de estilo. Las indentaciones, el tipo de comillas que se utilizan, los comentarios, o la forma de nombrar a las variables o funciones, son ejemplos de problemas que suelen surgir al poner el código en común. Por supuesto a todo el mundo le gustaría que los demás programasen de la forma en que ellos programan, pero esto nos llevaría a un oxímoron técnico.

Para ello se pone en común unas ciertas reglas o convenciones que todos deben seguir, con el objetivo que al terminar de desarrollar la aplicación pareciese que lo hubiese escrito una única persona. Estas reglas o convenciones se agrupan en guías de estilo y suele haber unas cuantas por cada lenguaje de programación, será responsabilidad del equipo en cuestión decidir que

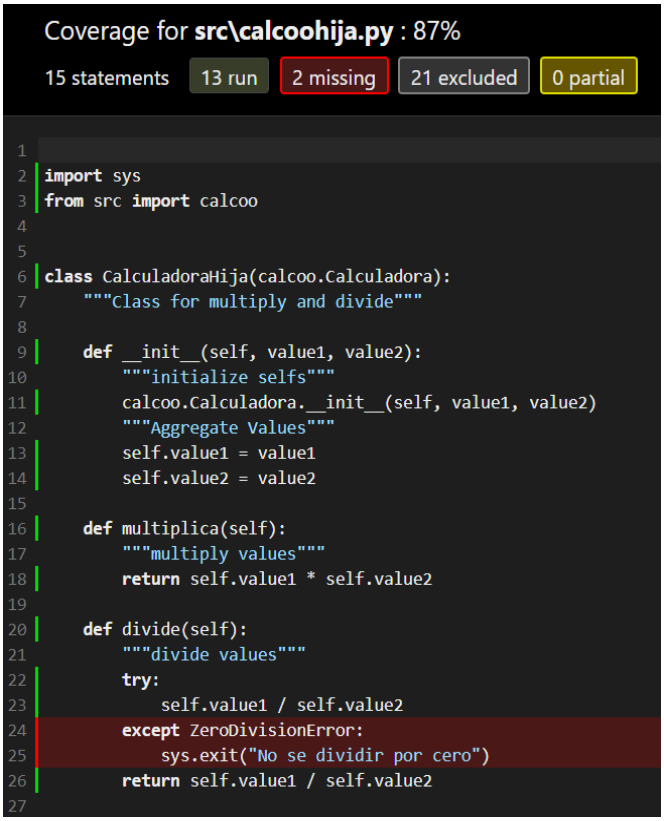


Figura 3.6: Ejemplo de informe de cobertura línea por línea

guía se sigue o generar una propia que siga todo el equipo.

PEP8 (Python Enhancement Proposal 8[1]) es una guía de estilo definida para Python. Es la guía de estilo elegida para la asignatura de PTAVI y en este proyecto va a ser una de las condiciones para la implementación continua. Para ello, como ya veremos, se han definido una serie de tareas que comprueban que se ha seguido en el desarrollo esta guía. Algunas de las reglas más importantes de esta guía son:

- Indentación en cuatro espacios.
- Alineación vertical de los argumentos en las funciones.
- Espacios antes que tabulaciones.
- Las líneas no pueden contener más de 79 caracteres.
- A la hora de saltar de línea si hubiera un operador matemático debe ir en la línea siguiente.

[1] `https://www.python.org/dev/peps/pep-0008/`

- Se debe escribir un *import* por cada importación de módulo.

Actualmente la herramienta ha sido renombrada como `pycodestyle` en su paquete de instalación y el comando de instalación es el siguiente:

```
pip install pycodestyle
```

3.8. Wireshark

Wireshark [13] [8] es un analizador de tráfico en tiempo real. Fue diseñado a finales de 1997 por Gerald Combs para comprobar posibles problemas en una red. Inicialmente bautizó como *Ethreal* y no fue hasta el año 2006 cuando empezaría a denominarse Wireshark.

Wireshark no solo permite capturar paquetes como podríamos hacer con `tcpdump` además podemos filtrar y analizar a un nivel realmente exhaustivo cada paquete que haya pasado por una interfaz. Cuenta con una interfaz gráfica muy intuitiva que dispone los paquetes capturados siguiendo el orden que nosotros elijamos

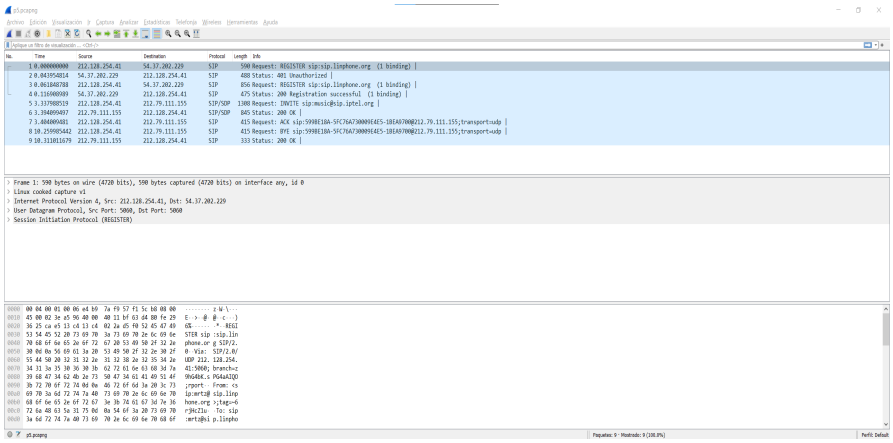


Figura 3.7: Ejemplo de captura en wireshark

En la pagina oficial https://www.wireshark.org/ se brinda un extenso abanico de características de las que procederemos a resaltar las que han sido más importantes en este proyecto:

- Capacidad de captura en tiempo real y análisis *offline*.

- Multiplataforma.

- Las capturas generadas pueden ser analizadas mediante la interfaz gráfica o mediante otro tipo de herramientas de terminal como Tshark.

- Capacidad de leer multitud de formatos diferentes como libcap, pcap (utilizados en el proyecto) y muchos otros detallados en la documentación.

- Ofrece la posibilidad de personalizar los colores en los que se muestran las capturas en su modo gráfico, esto puede sonar baladí, sin embargo, se torna fundamental cuando el número de paquetes aumentan. Esta característica combinada con un buen filtro permite analizar capturas de una manera cómoda y rápida.

- Permite exportar las capturas en variedad de formatos como puede ser el XML o el JSON, formato utilizado en este proyecto.

3.8.1. Tshark y Pyshark

Para este proyecto se ha utilizado el Wireshark para hacer una labor de análisis de las capturas de los alumnos y de las capturas que generábamos ejecutando ciertas prácticas. Sin embargo como veremos en el capítulo 4 hemos utilizado capturadores y analizadores de tráfico apoyándonos en un módulo de Python que es pyshark. Pyshark es lo que se conoce como un *wrapper* de Tshark. Un *wrapper* tiene varias acepciones similares dentro del lenguaje informático, en función es una forma encapsulada de un conjunto o de un subconjunto de funciones.

Por lo tanto, antes de hablar de pyshark, es de obligado cumplimiento hablar de Tshark. Tshark es un analizador de tráfico igual que Wireshark, con la diferencia que está orientado para un uso sin interfaz gráfica. Esto que puede parecer un hándicap permite a los programadores adaptar el funcionamiento a sus necesidades particulares y en nuestro caso poder trabajar con todas las funcionalidades de Wireshark pudiendo manejarlas en scripts o incluyéndolas en clases.

Como ya se ha explicado en este proyecto se ha trabajado con el módulo de pyshark, este módulo encapsula todas las funcionalidades de Tshark en funciones y clases programadas en Python. Esto ha permitido que se desarrolle un sistema de capturas y análisis de capturas utilizando únicamente Python como lenguaje de desarrollo. Este no es un módulo incluido en Python 3.8 por lo tanto habrá que instalarlo tanto para usarse en local como para cuando lo

integremos en GitLab CI. Para instalarlo habrá que ejecutar el siguiente comando:

```
pip install pyshark
```

3.9. Módulos y bibliotecas

Este proyecto se apoya en una serie de bibliotecas y módulos de Python [9] que nos permiten o ayudan a realizar todo el trabajo ejecutado. Hay dos casos que son el caso de unittest y el caso de coverage que ya han sido explicados de una manera más extensa en este mismo capítulo. A continuación explicaremos el resto de bibliotecas y módulos utilizadas para la realización de este proyecto.

3.9.1. Sys

Módulo que nos permite interactuar con los parámetros y funciones específicas del sistema. Utilizamos este módulo para poder pasar argumentos por consola o para salir del programa reproduciendo un mensaje deseado. Se puede consultar su documentación en `https://docs.python.org/es/3.10/library/sys.html`.

3.9.2. Os

Módulo que nos permite trabajar con las funcionalidades del sistema operativo. Utilizamos este módulo para ejecutar o matar procesos, trabajar con rutas e identificar colecciones de procesos ejecutados. Se puede consultar su documentación en: `https://docs.python.org/es/3.10/library/os.html`.

3.9.3. Signal

Módulo que nos permite definir *handlers* para eventos asíncronos. También nos permite ejecutar *signals,* una especie de señal de ejecución con ordenes concretas. Este módulo entre otras cosas es el encargado de poder emitir y simular un `KeyboardInterrupt` como veremos más adelante. Se puede consultar su documentación en: `https://docs.python.org/3/library/signal.html`.

3.9.4. Threading

Módulo que nos permite trabajar con hilos de ejecución creando entornos multihilo. Siempre que tengamos que utilizar un proceso multihilo como puede ser la ejecución de un cliente y servidor bajo el mismo programa necesitamos una interfaz con el que podamos controlar estos hilos. Podemos dividir el hilo creando así varios y finalmente juntarlos para poder terminar la ejecución. Se puede consultar su documentación en: `https://docs.python.org/3/library/threading.html`.

3.9.5. Time

Módulo que nos permite gestionar todas las operaciones relacionadas con el tiempo. Este módulo contempla operaciones de tiempo tales como devolver la hora actual o convertir entre formatos de tiempo. Además también contempla operaciones de gestión de tiempos de ejecución como la posibilidad de pausar una ejecución o *dormirla* durante un número de segundos. Esto puede servirnos si por ejemplo queremos controlar operaciones multihilo. Se puede consultar su documentación en: `https://docs.python.org/3/library/time.html`.

3.9.6. Subprocess

Módulo que nos permite generar y gestionar nuevas ejecuciones, procesos o hilos de ejecución. Es un módulo que pretende reemplazar a `os.spawn` y algunas funcionalidades de `os.system`. Para ello define un sistema de *pipes* que encapsulan cada proceso, define además una conexión a la entrada, salida y posible error en cada uno de estos pipes permitiendo un mayor control sobre ellos. Es relativamente sencillo de usar y resalto una gran facilidad a la hora de trabajar con scripts o con módulos Python diferentes en sistemas multihilo. Se puede consultar su documentación en: `https://docs.python.org/3/library/subprocess.html`.

3.9.7. JSON

biblioteca que nos brinda todas las operaciones con JSON en Python. Leer, escribir, codificar, decodificar crear una propia clase para la codificación de JSON, esta biblioteca te brinda una

interfaz sencilla para todos estos procesos. En este proyecto se ha utilizado en todas las operaciones con JSON. Se puede consultar su documentación en: `https://docs.python.org/3/library/json.html`.

Capítulo 4

Diseño e implementación

En este capítulo veremos qué y cómo se ha implantado el proyecto. Empezaremos por un vistazo general que nos ayude a entender la arquitectura general. Como es una implementación en varias prácticas este primer apartado nos explicará cómo funciona la implementación de forma global y que características son comunes a todas las prácticas.

Acto seguido tendremos una sección en la que se explicara la implementación en las prácticas con todo lujo de detalles. Se dividirá la sección en cada una de las prácticas, lo que nos permitirá un análisis pormenorizado de cada implementación.

4.1. Arquitectura general

Para la implementación de este proyecto lo primero que se ha hecho es reflexionar sobre el proceso actual de las prácticas en la asignatura. Actualmente el profesor entrega el enunciado de la práctica, el alumno se realiza un *fork*[1] de la práctica sin resolver del repositorio del profesor, descarga el repositorio, soluciona el ejercicio, sube su ejercicio a su repositorio-copia y finalmente el profesor lo corrige. Para mejorar su calificación el alumno suele comprobar el estilo manualmente ejecutando el programa pep8 o el programa codestyle a su práctica. La idea es no modificar este ciclo de vida de las prácticas por lo tanto nos limitaremos a trabajar sobre las fases del ciclo.

Consideraremos primeras fases al acto de entregar el enunciado al alumno y que este realice el *fork*. En la primera fase vamos a modificar ligeramente los enunciados para adecuarlos a las

[1] Copia exacta del repositorio objetivo

modificaciones que se van a realizar en las prácticas. Se pueden consultar los nuevos enunciados a partir del apéndice B que acompaña al proyecto.

Considerando la descarga del repositorio y la resolución del ejercicio como fases intermedias, se ha decidido no introducir ninguna modificación en esta parte del ciclo. El hecho de implementar integración continua nunca debería afectar a la resolución del ejercicio pues son dos temas diferentes que no guardan relación entre sí. Es en las fases finales del ciclo de vida donde se ha producido el grueso de la implementación. En la fase de subir los cambios a el repositorio para ser corregidos es donde se ha implantado el ecosistema de integración continua. Llegados a este punto es bueno volver a reflexionar sobre cómo tiene que realizar las prácticas el alumno.

Lo primero que se ha pensado es en el apartado del estilo del código, claramente no parece muy óptimo que el alumno deba ejecutar el paquete cada vez que quiera entregar una práctica. Por eso se ha decidido automatizar el proceso, ejecutando la tarea cada vez que el alumno decida subir su práctica.

El siguiente punto a tener en cuenta va a ser el de la calidad de la práctica en cuanto a funcionalidad. Es aquí donde van a entrar en juego los tests [2]. Desde que este proyecto empezó, los tests parecían un problema pues son archivos de código que el propio programador de una funcionalidad desarrolla para comprobar que es correcta. Pero en este caso no es el programador de la funcionalidad el que va a desarrollar esos tests, sino el creador de este proyecto, lo que de entrada choca frontalmente con el paradigma de los tests y dificulta la operación. Además, tests como los unitarios deben ir referenciados a métodos concretos lo que significa que el alumno tendría que programar siguiendo los métodos que se le dan en el enunciado. Esto no parece muy desafiante ni muy didáctico para el alumno por lo que se ha decidido tomar la siguiente medida. A medida que las prácticas aumenten el nivel también disminuirá la cantidad de métodos que se testean unitariamente, de hecho, solo habrá tests unitarios hasta la práctica 4. Esto permite que el alumno tenga que programar sus métodos sin una guía previa y no generar un ecosistema demasiado restrictivo. Sin embargo, se ha decidido testear de otras maneras (como generando tests *endtoend*) para que todas las prácticas tengan una cobertura suficiente.

Todas estas comprobaciones se realizan en el punto en que se realiza el push aprovechando los job y pipelines de la herramienta de GitLab CI/CD. Como veremos en este capítulo cada

[2]El proyecto grimoirelab-perceval [4] ha sido importante para el aprendizaje del desarrollo de test.

práctica tiene un determinado pipeline que se ejecutara y arrojara unos resultados en un archivo de texto denominado artefacto. Además cada vez que falle una de las tareas se enviara un correo al alumno recordándole que tiene la entrega incorrecta.

Es aquí donde llegamos a la fase de la corrección de prácticas. El objetivo es mejorar el proceso de la corrección y esto se consigue con la aparición de los artefactos. Los artefactos son archivos que genera GitLab CI/CD con los resultados que hayamos programado. Esto permite que estos artefactos funcionen como informe previo a la corrección de las prácticas.

En el caso de una práctica errónea el paradigma de la integración continua indica que esa práctica nunca podría volver al repositorio, sin embargo, se ha decidido simplemente indicarlo con un aspa roja como se pudo apreciar en la figura 3.3 del capítulo anterior para evitar que un alumno pudiera quedarse sin entregar su práctica por un fallo menor.

El esquema general de la implantación sería muy parecido entonces a la figura 3.1 que vimos en la explicación de GitLab CI/CD. La única diferencia es que el alumno realiza un *fork* del repositorio original para trabajar en paralelo al repositorio de la asignatura.

La implementación de integración continua en las asignaturas de PTAVI conlleva una implantación diferente por cada práctica, sin embargo, hay ciertos aspectos comunes que podemos dilucidar aquí. Todas las prácticas excepto la 5 tienen ciertos archivos comunes como podría ser el script de comprobación de estilo o la configuración de GitLab. Toda esta configuración puede consultarse en el siguiente repositorio de GitHub: https://GitLab.etsit.urjc.es/sfuente/ptavi-config.

Además se ha modificado la estructura de carpetas de algunas de las prácticas, puede consultarse las estructura nueva en el siguiente apéndice A

4.2. Implementación por prácticas

Todas las prácticas explicadas a continuación pueden encontrarse en:
`https://GitLab.etsit.urjc.es/sfuente/`

4.2.1. Práctica 2

La primera práctica en este proyecto consiste en dos simples calculadoras una que suma y resta y la otra que hereda de la primera para añadir la funcionalidad de multiplicar o dividir.

El alumno a su vez tiene que programar tres ejercicios,

1. Calc.py: script básico que recibe tres argumentos, el primer número el operador que queremos ejecutar y el segundo número, en este caso solo existe la suma y la resta. El programa debe ser capaz de operar y devolver su resultado por pantalla, a la vez estarán controlados los errores tales como no introducir un número o no introducir correctamente el nombre de la operación (suma o resta) mostrando en estos casos dos mensajes predefinidos por pantalla. A la vez se mostrará un mensaje con las instrucciones de como usar el programa si no se sigue la estructura correcta.

2. Calcoo.py: caso igual que el anterior pero esta vez programado como clase. Se crea la clase `Calculadora` y se instancia, luego se ejecutan sus funciones en el main(). El modo de uso es exactamente igual que en el anterior caso, seguirán controlándose los errores y mostrando los mensajes de error, este programa sirve para introducir al alumno el concepto de las clases y la programación orientada a objetos.

3. Calcoohija.py: en este caso el alumno deberá extender las funciones de su clase calculadora a una calculadora que sea también capaz de multiplicar y dividir. Para ello deberá generar una nueva clase llamada `CalculadoraHija` que heredará sus propiedades de la clase calculadora de `calcoo.py`. En este caso además de declarar la clase calculadora para que se herede e implementar dos nuevos métodos como son la multiplicación y la división el alumno tendrá que ser capaz de lidiar con la situación de no poder dividir por cero, por lo tanto, además de mostrar los mensajes de error anteriormente explicados el programa deberá ser capaz de mostrar un error controlado cuando en la división el divisor sea cero.

Para esta práctica se ha considerado que todos los métodos enunciados en el guión serán obligatorios por lo tanto se le aplicarán los tests unitarios a todos los siguientes métodos:

1. test_calc.py

   a) plus: el test comprobará que introduciéndole un 2 y un 4 respectivamente el resultado devuelto es un 6.

   b) minus: el test comprobará que introduciéndole un 2 y un 4 respectivamente el resultado devuelto es un 2.

2. test_calcoo.py

   a) suma: el test comprobará que introduciéndole un 1 y un 6 respectivamente el resultado devuelto es un 7.

   b) resta: el test comprobará que introduciéndole un 1 y un 6 respectivamente el resultado devuelto es un -5.

3. test_calcoohija.py

   a) multiplica: el test comprobará que introduciéndole un 5 y un 1 respectivamente el resultado devuelto es un 5.

   b) divide: el test comprobará que introduciéndole un 5 y un 1 respectivamente el resultado devuelto es un 5. El test también comprobará que introduciéndole un 5 y un 0 respectivamente, se controla correctamente el error de dividir por 0.

4.2.2. Práctica 3

La práctica 3 de la asignatura de PTAVI consiste en el uso de bibliotecas como SAX para comprender el concepto del `parsing` en archivos XML, SMIL y JSON.

El alumno a su vez tiene que programar dos ejercicios:

1. smallsmilhandler.py: archivo que crea una clase denominada `SmallSMILHandler` que permite parsear un archivo `*.smil` y devolver una serie de etiquetas predefinidas. Hereda de la clase `ContentHandler` de la biblioteca SAX que nos permite navegar entre las etiquetas. La clase que se pide consta de dos métodos el `startElement` que se llama cuando se abre una etiqueta y permite almacenar todos los campos de la misma en una propiedad de la clase denominada lista. El segundo `get_tags` simplemente devuelve la propiedad lista con un arreglo de todas las etiquetas encontradas en la lista.

2. karaoke.py: en este archivo se debe generar la clase `KaraokeLocal` que herede de la clase `SmallSMILHandler`. Esta clase debe ser capaz de recibir un archivo `*.smil` convertirlo en JSON y descargarse un número de archivos multimedia que están referenciados en las etiquetas del archivo que recibe por línea de comandos, se deben crear

dos métodos y redefinir el método propio de la biblioteca de Python `__str__`. El primer método `to_json` debe ser capaz de generar un archivo `*.json` siempre que no exista e introducir toda la información que le llega del archivo .smil. El segundo método `do_local` debe ser capaz de descargarse todos los archivos multimedia de las etiquetas `src` y renombrar los atributos correspondientes indicando su localización local. Por último se deberá modificar el método propio `__str__` para que al imprimir por pantalla muestre un listado ordenado de las etiquetas y de las tuplas atributo - valor cada uno en una línea, separados por tabuladores y sin espacios.

Para esta práctica se ha considerado que todos los métodos enunciados en el guión serán obligatorios por lo tanto se les aplicarán los tests unitarios a todos los siguientes métodos:

1. test_smallsmilhandler.py:

   a) test_startElement: el test comprobará dos casos, en el primero se le proporcionara un mock de la lista de atributos y un atributo que no existe en esta lista, la función debe ejecutarse sin arrojar ningún error, en el segundo caso se le proporcionará un atributo que si existe y una propiedades correctas, por supuesto la función no debe arrojar ningún error.

   b) test_get_tags: el test comprobará que la función es capaz de devolver correctamente un array

2. test_karaoke.py:

   a) test_str: al test se le proporcionará un mock con un ejemplo de la salida correcta que se quiere obtener, el test ejecutará la función con el mock de las etiquetas predefinidas anterior y comprobara que la respuesta es la misma.

   b) test_to_json: se ejecutaran dos comprobaciones, la primera debe ser capaz de transformar y crear un archivo nuevo JSON cuando este no exista, la segunda hará este mismo proceso en el caso en el que el archivo JSON ya exista.

   c) test_do_local: este es el caso en el que deberían descargarse los archivos multimedia, en este caso solo se comprobará que la función se ejecute sin errores.

4.2.3. Práctica 4

La práctica 4 de la asignatura de PTAVI consiste en el desarrollo de una aplicación cliente -servidor utilizando el protocolo SIP y la biblioteca UDPServer del módulo `socketserver`. El alumno debe de ser capaz de realizar un servidor de registro SIP y un cliente capaz de mandar paquetes de bytes con la información de registro como está descrito en la práctica.

El alumno a su vez tiene que programar dos ejercicios,

- client.py: script que funciona como cliente, envía el paquete con los datos de registro o un mensaje al servidor. Se le proporcionara el puerto, la IP y el mensaje por línea de comandos.

- server.py: archivo que establece la clase `SIPRegisterHandler` que permite levantar el servidor SIP que escuchará las peticiones del cliente y será capaz de registrar los usuarios en un JSON y borrarlos cuando estén expirados. Se le proporcionará un puerto de escucha que por supuesto debe coincidir con el puerto del cliente.

Para esta práctica se ha considerado que los métodos `register2json`, `check_expiration` y `json2registered` enunciados en el guión serán obligatorios por lo tanto se les aplicarán los tests unitarios a los métodos del archivo server.py siguientes:

1. test_server.py:

   a) test_register2json: El test debe comprobar que la función registra correctamente los usuarios, para ello le proporcionaremos un usuario y se comprobará que efectivamente se ha escrito el usuario en el JSON `registered.json` evaluando el archivo después de ejecutar la función.

   b) test_check_expiration: En este caso el test debe evaluar si la función es capaz de eliminar correctamente los usuarios expirados. Para ello lo primero le proporcionamos una lista de cuatro usuarios, dos con fecha del 2021 y uno con fecha del 2002. A la vez se establece que la fecha actual es *21-10-2015*, *07:28* horas. Seguidamente se le proporciona un mock con únicamente los datos de los dos primeros usuarios, los usuarios los cuales no vence su fecha de expiración. El test evaluará que efectivamente después de ejecutar la función solo queden estos dos en la lista `Users`.

c) test_json2registered: Parecido al primer caso, se debe que comprobar que el programa es capaz de leer de un JSON la lista de usuarios y registrarlos en la propiedad `Users` de la clase `SIPRegisterHandler`. Para ello se lee el archivo JSON y se almacena en un mock con una lista con los usuarios del archivo, acto seguido ejecutamos la función y comparamos la propiedad `Users` con la lista del mock comprobando que todo ha ido correctamente.

A esta práctica a su vez se le ha implementado un test end to end que procederemos a explicar, una vez que hemos comprobado que procedimientos tales como el registro de usuarios se han hecho correctamente necesitamos alguna manera de testar que efectivamente la conexión entre el cliente – servidor se produce y que además es exitosa. Para ello se ha diseñado un test en el archivo `test_p4.py` que es capaz de desarrollar un entorno con dos hilos de ejecución donde ejecutar la práctica tanto del lado del cliente como del servidor. Primero se ejecuta el servidor en un hilo , seguidamente se ejecuta el cliente con el método "register", el usuario luke@polismassa.com y un tiempo de expiración de una hora, el test registra todo el proceso y posteriormente simulamos un "KeyboardInterrupt" para poder terminar el proceso. Se puede ver en la figura 4.1 un esquema explicativo. Para la implementación de este método que concluye el proceso se han tenido en cuenta dos posibles casos, que se ejecute en un sistema Unix como es el caso de GitLab CI/CD o que se ejecute en Windows. Con ese fin se ha utilizado la biblioteca "signal" que contiene los métodos `CTRL_C_EVENT` y "SIGINT" que nos permiten simular el "KeyboardInterrupt" mencionado anteriormente. El método `CTRL_C_EVENT` está referido al sistema Windows mientras que el método "SIGINT" se asocia con el sistema Unix.

Además de la biblioteca *signal* ya mencionada se han utilizado las bibliotecas *threading*, *subprocess* y *time*. Con la biblioteca *treading* se han generado los hilos de ejecución donde se levantan el cliente y el servidor, mientras que la biblioteca *subprocess* nos permite gestionar los procesos en cada uno de los hilos mediante los *pipes* que facilitan el procedimiento de controlar los errores y terminar la ejecución. Por último la biblioteca *time* nos permite insertar intervalos de tiempo que hemos utilizado para controlar que procesos se ejecutaban antes y después.

Para esta práctica entonces el *pipeline*ejecutara tres *Jobs*: pep8, tests y test *e2e*(*end to end*).

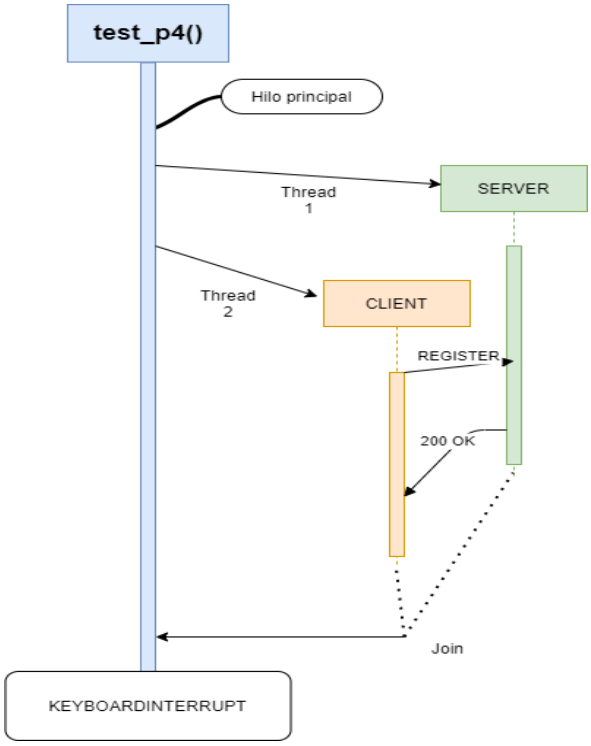


Figura 4.1: Esquema del test *e2e*

4.2.4. Práctica 5

La práctica 5 de la asignatura de PTAVI es diferente al resto de prácticas. Mientras que en el resto de prácticas se pide el desarrollo en código de ciertos ejercicios, en la práctica 5 se pide la resolución de una serie de cuestiones sobre el protocolo SIP y una posterior captura de una sesión SIP. Esto obviamente presenta un problema a la hora de seguir el objetivo de este proyecto ya que no hay desarrollo que testear, por lo tanto, debemos seguir otro camino para asegurar la correcta resolución de la práctica.

Dada esta cuestión se ha decidido que la mejor forma de testear la práctica es analizar que la captura SIP que el alumno tiene que generar es correcta. Esto a su vez presenta otro problema, y es cómo automatizar el análisis de una captura de *Wireshark*. Para la resolución de este problema hemos acudido a la biblioteca de `pyshark` que nos permite analizar una captura y comprobar que los paquetes son correctos. La estrategia entonces es la de conocer que paquetes debería contener la captura y compararlos con los paquetes de la captura del alumno. Para esto lo primero que necesitamos es una captura correcta que denominaremos modelo. Con este modelo generaremos un archivo JSON que contendrá la correcta estructura de paquetes,

incluyendo no solo que paquetes se deben generar sino también el orden de los mismos. Es importante tener en cuenta qué para evitar que el alumno posea la captura correcta antes de la realización de la práctica se debe haber generado el JSON con anterioridad incluyéndolo en la situación inicial de la práctica. Una vez con esto el alumno ya puede realizar su práctica como explica el enunciado, cuando la haya terminado al subir sus cambios se comprobará que existe la captura y que los paquetes son correctos.

Para la implementación en esta práctica se han desarrollado dos archivos:

- p5_json_capture_gen.py: El encargado de generar el objeto modelo que se utilizará para comparar y evaluar la práctica. El uso es sencillo, basta con correr el programa y pasarle como argumento la captura modelo de la que queremos extraer el objeto, el programa lo convertirá en un JSON y lo analizará. Se extraerán las peticiones teniendo en cuenta que se han podido generar desde máquinas diferentes y acto seguido se generará un archivo llamado `capture_test.json` que contendrá el modelo deseado.

- p5_test: El encargado de comprobar según el modelo la captura del alumno. Se crea una clase llamada `Test_Capture` que incorpora los siguientes métodos:

  - parse_capture: Parsea la captura en JSON y crea un array con los extractos de los paquetes que compararemos con el modelo.
  - check_len: Comprueba el numero de paquetes y devuelve un *OK* en caso de que sea correcto, y un *FAILED* en caso contrario.
  - check_general_packet: Comprueba que los paquetes son iguales según los criterios marcados. Devuelve un *OK* en caso de que sea correcto, y un *FAILED* en caso contrario.
  - test_p5: Gestiona los métodos anteriores realizando todos los pasos y generando un bucle para comprobar todos los paquetes.

Cuando el *runner* de GitLab empiece ejecutará el test_p5 y se recolectará el resultado en un artefacto, permitiendo así ser revisado en posteriori por el alumno o el profesor.

Para el *pipeline* ejecutara únicamente un *Job*: test_p5.

4.2.5. Práctica 6

La última práctica de este proyecto se basa en una aplicación cliente-servidor pero algo más compleja que la práctica 4. Mientras que en la práctica 4 el alumno tenía que desarrollar una aplicación capaz de comunicarse con el protocolo SIP en la práctica actual se debe extender esta funcionalidad generando una sesión SIP capaz de enviar audio vía RTP como la que se muestra en la figura 4.2.

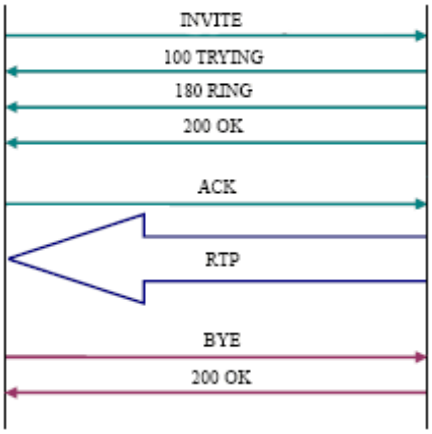


Figura 4.2: Esquema de sesión SIP

Esta es la última práctica antes de la práctica final de la asignatura y, como tal, se ha decidido no guiar al alumno con métodos obligatorios. Esto significa que al igual que en la práctica 5 no habrá tests unitarios. Para testear esta práctica se utilizará un test sobre la captura de `Wireshark` como pasaba en la práctica 5, y un test *endtoend* combinado con un test a una captura que se genera. Además en esta práctica hay ciertos ejercicios alternativos que podríamos controlar con capturas modelo de cada uno de los ejercicios adaptando así la forma de evaluar de los *jobs* de GitLab CI

La estrategia es entonces la siguiente: una vez que el alumno termine la práctica al subir sus cambios se comprobará que la captura es correcta, para ello se deberá suministrar el modelo dependiendo del ejercicio que tenga que realizar. Lo ideal sería que el alumno descargase el modelo más adecuado para el ejercicio que tiene que realizar. En caso de este proyecto, en cambio, se ha generado un modelo genérico que funciona con todos los ejercicios. El modelo contiene la lista de métodos y el orden correcto que como muestra la figura es común a todos los ejercicios.

Para la implementación en esta práctica se han reutilizado y adaptado cuatro archivos:

- p6_json_capture_gen.py: script con el mismo funcionamiento que p5_json_capture_gen.py de la práctica 5.
- p6_cap_test.py: script con el mismo funcionamiento que p5_test.py de la práctica 5.
- p6_e2e_test.py: script con el mismo funcionamiento que test_p4.py de la práctica 4. En este caso se ejecutará el servidor y el cliente correspondiente a la práctica 6.

Además, se ha generado el siguiente archivo:

- p6_test.py: programa encargado en desarrollar un entorno con dos hilos de ejecución. En el primero ejecutaremos un *sniffer* donde capturaremos los paquetes que se intercambien en el test *e2e*. En el segundo hilo ejecutaremos el p6_e2e_test.py donde se producirá la sesión SIP que capturaremos en el primer hilo. Se puede ver un esquema en la figura 4.3 A la vez se ha importado la clase `TestCapture` que será la encargada de generar el objeto y de compararlo con el modelo. Por último se generará un artefacto como en el resto de prácticas con las resoluciones de las comprobaciones.

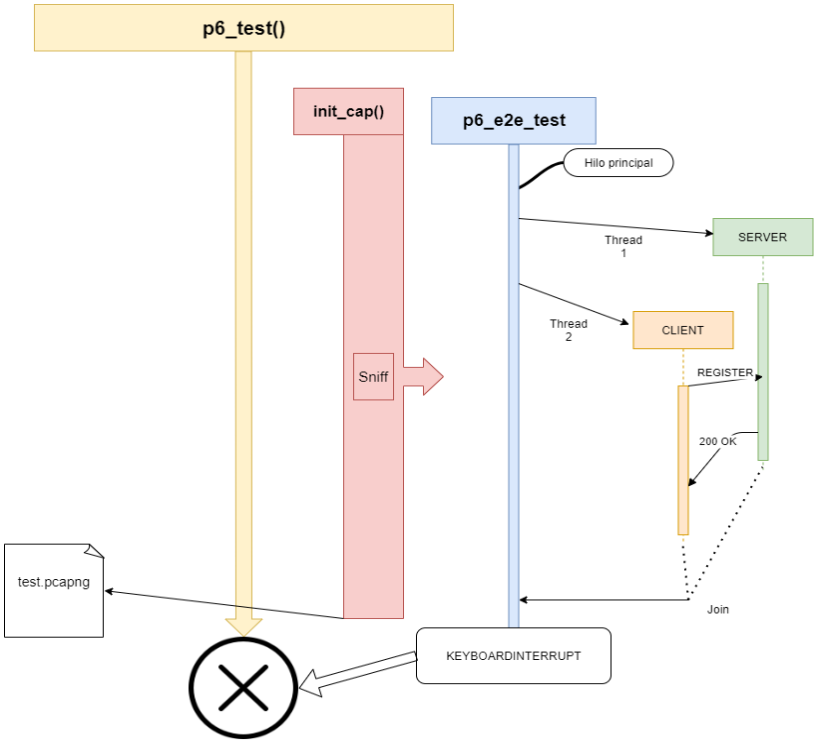


Figura 4.3: Esquema del p6_test

Nos puede asaltar la duda de si dada la posibilidad del análisis de la captura del test *e2e* es necesario el *job* de análisis sobre la captura del alumno. Esto se ha hecho así ya que inicialmente este ultimo test no estaba incluido en el proyecto, ya que se había ideado como un test a la captura del alumno y un test *e2e* como en la práctica 4, sin embargo, en la recta final de este proyecto se pensó en esta posibilidad de intentar combinar ambos tests y generar una herramienta que permitiese generar la captura obligatoria para el alumno automáticamente, y además poder analizarla como ya hemos explicado. Esto además de dar mucho más contexto al profesor puede prevenir las prácticas ilegales del alumno como la copia, al no poder utilizar una captura no generada por ellos a la hora de entregar la práctica.

El *pipeline* ejecutará cuatro *Jobs*: Pep8, test_capture, test_e2e, test_p6.

Capítulo 5

Experimentos, validaciones y resultados

Uno de los objetivos específicos que se marcaba este proyecto era el de evidenciar los resultados con alumnos al azar. Desgraciadamente esto no ha sido posible pues la asignatura de la que se realiza la herramienta solo se cursa en el primer cuatrimestre, y este proyecto se ha realizado en el segundo. Se pensó en el momento de definir los objetivos que se podría evaluar la herramienta con entregas de alumnos que quisieran reevaluar las prácticas, sin embargo, esto no se produjo. A pesar de ello si se ha podido comprobar la herramienta en el caso de la práctica 5 ya que se han extraído las capturas de los alumnos y se ha pasado el test descrito en el capítulo anterior. Esto también se ha podido hacer en menor medida con la práctica 6, de igual manera extrayendo las capturas que entregaron los alumnos se ha podido testear la validez de estas.

Por otro lado este proyecto se ha desarrollado en un modo de prueba y error lo que nos permite analizar resultados del histórico de los repositorios de GitLab. Esto es porque al tener que testear la integración continua se han ido subiendo muchos de los desarrollos y esto muestra una evolución de los resultados en los experimentos y su posterior validación.

En la figura 5.1 se puede ver un análisis de todos estos experimentos separados en prácticas.

Nombre	Casos	Stages	Succeed	Failed	Skipped
Práctica 2	22	27	11	14	2
Práctica 3	3	6	2	2	2
Práctica 4	18	49	33	12	4
Práctica 5	2	2	2	0	0
Práctica 6	6	13*	11	2	0

Figura 5.1: Resultados

Aclaración de figura 5.1

1. Casos: Cada vez que se realiza un push
2. Stages: Tareas que se ejecutan
3. Succed: Tareas completadas correctamente
4. Failed: Tareas que no han sido completadas por algún error
5. Skipped: Tareas que no han sido ejecutadas por un error en tareas anteriores

Como podemos ver tenemos no solo distintos resultados sino una gran varianza en los casos obtenidos. Esto es debido a que muchas funcionalidades se han aprendido a lo largo del proyecto. La primera práctica (Práctica 2) es por tanto la que más casos y fallos tiene, dado que no se tenía la suficiente experiencia. La segunda (Práctica 3) al ser muy parecida a la anterior en cuanto a arquitectura solo se ven reflejados 3 casos. No obstante, se realizaron muchas pruebas en el entorno local a la hora de desarrollar los tests. De hecho esta práctica es la que más tiempo requirió en materia de tests unitarios dada la curva de aprendizaje.

La práctica 4 vuelve a tener un elevado numero de casos ya que se introdujo el concepto de test *endtoend*, además se aumentaron el número de tareas que se ejecutaban por subidas por lo que también se ve un elevado número de stages.

Como ya conocemos en este punto la práctica 5 es diferente a las otras. Esta ha tenido muchos casos de ensayo-error en Wireshark, comprobando que lo que programábamos en pyshark era el resultado deseado. Es por esto que solo se subió la práctica un par de veces cuando el resultado esperado era el correcto. Previamente se ha explicado que en este caso se habían utilizado capturas de alumnos al azar, esto es así, pero las capturas se iban renombrando según se hacían las pruebas. Finalmente se ha optado por dejar la captura de uno de esos alumnos que se ha evaluado como correcta.

Por último esta el caso de la práctica 6. Como hemos comentado en el capítulo 4 la práctica 6 sufrió un cambio de alcance. Se marcaron unos objetivos que luego se ampliaron, por lo tanto el número de casos no es representativo ya que esta práctica debería haber tenido muchos más casos de experimentación. Llegados al momento de empezar a escribir esta memoria se realizo una versión estable de la implementación, es por ello que la mayoría de tareas están correctas.

Nombre	pep 8	test	test e2e	build
Práctica 2	3	10	-	1
Práctica 3	2	-	0	-
Práctica 4	1	6	5	0
Práctica 5	-	-	-	-
Práctica 6	0	-	0	2

Figura 5.2: Tipo de fallo según práctica

En la figura 5.2 se muestra una tabla con los casos de error pormenorizados en tareas. Esta tabla arroja luz a lo explicado en el párrafo anterior. Así pues podemos comprobar que en la práctica 2 tiene fallos de configuración con el entorno de GitLab, esto es porque al ser la primera no se tenía el suficiente contexto del funcionamiento. Los fallos en los tests son en algún caso errores reales y otros provocados para comprobar el funcionamiento de GitLab CI. Lo mismo pasa en el caso de pep8.

La práctica 3 solo tiene un par de fallos por pep8, estos venían arrastrados de mi práctica original. La práctica 4 tiene una variedad de fallos entre los tests y los tests *endtoend*, esto es debido a que fue la primera práctica que implementaba este tipo de test, y a pesar que solo se empezó a subir al repositorio cuando no provocaba fallos en local, la configuración del entorno en GitLab fue costosa. Algunos de los fallos de tests unitarios se provocaron por los cambios en la configuración para adecuarla al *e2e*. Por último la práctica 6 tiene dos errores de construcción, esto se debe al montar todo el entorno explicado el poder ejecutarlo correctamente.

Esto es un resumen de todas las tareas y pipelines del proyecto, para ver los casos detallados en la pagina de GitLab se puede acceder a: `https://GitLab.etsit.urjc.es/sfuente/(nombredelaprctica)/-/pipelines`.

Capítulo 6

Conclusiones

6.1. Consecución de objetivos

Este es el apartado donde vamos a elaborar todas las conclusiones de un proyecto que se marcó unos objetivos iniciales y que como norma general se han cumplido todos. El objetivo general del proyecto era el de generar un ecosistema de integración continua en la asignatura de PTAVI. Podemos concluir, que se ha elaborado un ecosistema que permite aumentar la calidad de las entregas de los alumnos y simular una integración continua al uso.

A su vez se ha mejorado mucho el ámbito de la corrección de prácticas pudiendo concluir que en las prácticas 2, 3, 4 y 5 se cumple la premisa que si el resultado de los tests es erróneo es seguro que la práctica no es correcta. Esto no significa que la práctica pueda pasar los tests con algún error en algún caso sobre todo con la práctica 4 en la que no se cubre el cien por cien de los métodos y que el test *e2e* debería ser probado en más casos diferentes. Diferente es el caso de la práctica 6 donde como hemos visto en el capítulo de validaciones la falta de pruebas con más alumnos y el cambio de alcance que se decidió en la misma la ha hecho ser una versión menos probada y en un estado más embrionario que el resto de prácticas.

Otro de los objetivos importantes que nos marcamos al principio de este proyecto era el de impactar en la menor medida de lo posible en las prácticas actuales de la asignatura. Este es un objetivo que se concluye como exitoso ya que además de los enunciados de las prácticas lo cual se había previsto, solo ha habido que cambiar ligeramente la estructura de carpetas en las tres primeras prácticas ya que el resto de funcionalidades se han conseguido añadiendo scripts, clases o archivos de configuración en los repositorios.

Respecto a la implementación de una comprobación de la guía de estilo automática se puede concluir que ha sido en todas las prácticas un éxito, aunque también debemos tener en cuenta que era el objetivo más asequible de todos los que nos habíamos propuesto.

La conclusión general es por lo tanto buena. Si nos remontamos al capítulo 2, podremos ver los objetivos específicos, y podremos comprobar que se han cumplido todos, excepto la posibilidad de implementarlo en modo de prueba en alumnos al azar. Por supuesto era uno de los objetivos más interesantes que se marcó este proyecto ya que nos daría una imagen real del estado del proyecto. Sin embargo, al ser este un proyecto que comenzó en febrero y ser la asignatura una materia del primer cuatrimestre ha sido totalmente imposible. Por otra parte, el estado actual de la implementación hace que pueda ser probada para todas las prácticas de una manera más completa en los alumnos del cuatrimestre que viene.

6.2. Aplicación de lo aprendido

En este apartado vamos a enumerar por asignaturas como lo aprendido en este grado me ha permitido realizar este proyecto:

- Informática 1: Esta fue la asignatura en la que aprendí los fundamentos de la programación, realicé mis primeros scripts y aprendí la base que me ha conducido hasta este proyecto.

- Arquitectura de Internet: La primera asignatura donde empecé a conocer algo sobre protocolos de Internet desde el modelo OSI hasta los protocolos TCP y UDP.

- Informática 2: Con la base aprendida tocaba aumentar el nivel. Esta fue la asignatura en la que programé mi primera aplicación cliente servidor. También fue en está asignatura en la que programé mi primer chat y empecé a descubrir los handler y a entender el concepto de la asincronía.

- Sistemas telemáticos: Asignatura en la que a pesar de no poder programar nada empecé a tener contacto con los paquetes en las redes. Fue en la primera asignatura donde tendría contacto con Wireshark.

- Protocolos para la transmisión de audio y vídeo en internet: Obviamente la asignatura que más me ha marcado de toda la carrera, este fue mi primer contacto con Python, lenguaje que desde el primer momento fue un amor a primera vista. Por supuesto es la asignatura que más ha influenciado este proyecto, partiendo de la base que si no hubiera realizado sus prácticas no me habría sido posible empezarlo.

- Gráficos y visualización 3D: A pesar de ser PTAVI donde aprendí el concepto de programación orientada a objetos fue en esta asignatura donde comencé a aplicarla de una manera más exhaustiva.

- Arquitectura de sistemas 2: En esta asignatura aprendí cómo funciona la programación a bajo nivel. Entendí que es un load inmediate y un load address y la diferencia entre ellos, aunque a alto nivel como pueda ser Python no lo tengamos en cuenta.

- Construcción de Servicios y Aplicaciones Audiovisuales en Internet: Asignaturas donde aprendí lo que es una aplicación web y cómo funciona. Pudimos construir una aplicación en Django potenciando así mis conocimientos sobre Python y sus frameworks.

Por último, debo acordarme del resto de asignaturas pues no tengo ninguna duda que absolutamente todas me han llevado al proyecto que tengo hoy entre manos. A pesar de no ser consciente el resto de asignaturas me han enseñado una forma de trabajar, de pensar, de razonar, de redactar y de otras muchas habilidades que me llevo de este grado.

6.3. Lecciones aprendidas

Este proyecto ha sido una fuente de aprendizaje en prácticamente todas sus fases. Llega el momento de recapitular que he aprendido con este proyecto.

- El concepto de implementación continua. La capacidad de saber elegir cuándo y cómo puedo implementarlo. El poder deducir si según el tamaño u otras características va a mejorar el sistema o por el contrario podría deteriorarlo.

- La capacidad de desarrollar tests. Esta sea seguramente la mayor lección aprendida, sin embargo, soy consciente que mi curva de aprendizaje no ha hecho más que empezar ya

que mis conocimientos sobre el desarrollo de tests son muy limitados viendo la complejidad que se puede llegar a dar en un test.

- El conocimiento sobre GitLab. Terminando este proyecto me doy cuenta de todo lo que he aprendido sobre esta plataforma. Al empezar el proyecto la operación más complicada que conocía era la de hacer un merge. Hoy podría montar una web con un sistema serverless con un despliegue continuo de la misma.

- A utilizar el sistema LaTeX y todas sus bondades a la hora de maquetar, un verdadero acierto del que te das cuenta según vas terminando el proyecto.

6.4. Trabajos futuros

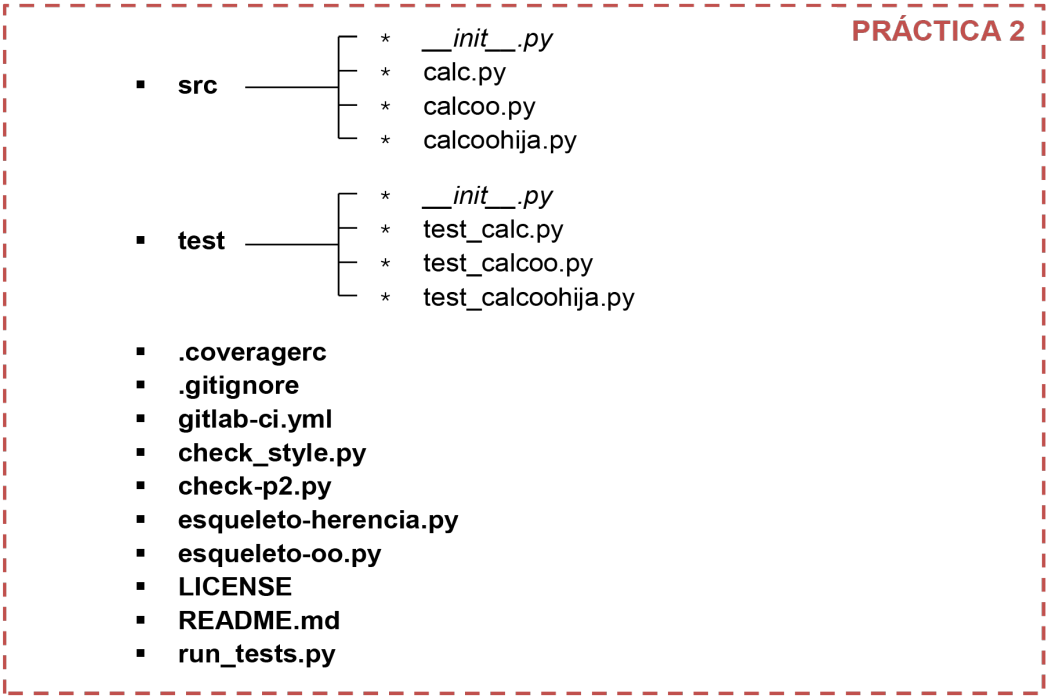
Como ya hemos comentado este proyecto no termina aquí, pues está pensado para ser utilizado en un entorno real. El siguiente paso es implementarlo en la asignatura de PTAVI, y probarlo en alumnos reales. Es probable que de estas pruebas surjan nuevas funcionalidades y problemas que no se han considerado.

El siguiente trabajo a futuro seria sacar parte de la funcionalidad de esta implementación y alojarla en repositorios separados. Este sería el caso por ejemplo de los tests unitarios, el alumno al subir sus cambios no tendría la carpeta de los tests en su repositorio, sino que esa carpeta se descargaría en la maquina remota de GitLab desde un repositorio diferente, se pasarían los tests y se subirían los cambios como actualmente.

Por último el trabajo más interesante a mi parecer es el de incluir la realización de tests unitarios a los propios alumnos y que el sistema de integración calcule según su cobertura de test, como hemos visto a lo largo de este proyecto, si la subida es correcta o no.

Apéndice A

Estructura de las prácticas



PRÁCTICA 3

- **src**
  - *__init__.py*
  - karaoke.py
  - smallsmilhandler.py
- **test**
  - *__init__.py*
  - test_karaoke.py
  - test_smallsmilhandler.py
- **chistes2.xmd**
- **karaoke.smil**
- **practica3-chistes.py**
- **.coveragerc**
- **.gitignore**
- **gitlab-ci.yml**
- **check_style.py**
- **check-p3.py**
- **LICENSE**
- **README.md**
- **run_tests.py**

PRÁCTICA 4

- **src**
  - *__init__.py*
  - client.py
  - server.py
- **test**
  - *__init__.py*
  - test_client.py
  - test_server.py
- **test_p4.py**
- **.coveragerc**
- **.gitignore**
- **gitlab-ci.yml**
- **check_style.py**
- **check-p4.py**
- **LICENSE**
- **README.md**
- **run_tests.py**

PRÁCTICA 5

- p5.txt
- sip.libpcap.gz
- capture_test.json
- p5_json_capture_gen.py
- p5_test.py
- .gitignore
- gitlab-ci.yml
- LICENSE
- README.md

PRÁCTICA 6

- bitstring.py
- cancion.mp3
- client.py
- server.py
- simplertp.py
- capture_test.json
- p6_cap_test.py
- p6_e2e_test.py
- p6_json_capture_gen.py
- p6_test.py
- .gitignore
- gitlab-ci.yml
- check_style.py
- check-p6.py
- LICENSE
- README.md

Apéndice B

Enunciado de la práctica 2

Nota: Esta práctica se puede entregar para su evaluación como parte de la nota de prácticas, pudiendo obtener el estudiante hasta un punto. Para las instrucciones de entrega, mira al final del documento. Para la evaluación de esta entrega se valorará el correcto funcionamiento de lo que se pide y el seguimiento de la guía de estilo de Python.

B.1. Introducción

La programación orientada a objetos en un paradigma de programación muy utilizado en la actualidad y que conviene conocer para la realización de las prácticas de la asignatura. En esta práctica se pretenden ver los conceptos más importantes de este paradigma, implementando funcionalidad en Python.

B.2. Objetivos de la práctica

- Conocer y practicar la programación orientada a objetos (con conceptos como clase, objeto, herencia, instanciación).
- Usar varios módulos Python, importando funcionalidad creada en otros módulos.
- Conocer y seguir la guía de estilo de programación recomendada para Python (ver PEP8).
- Utilizar el sistema de control de versiones git en GitLab.

B.3. Conocimientos previos necesarios

1. Nociones de Python3 (las de la primera práctica)

Tiempo estimado: 10 horas

B.4. Ejercicios

1. Comprueba que puedes entrar en tu cuenta del GitLab de la ETSIT (`https://GitLab.etsit.urjc.es`).

2. Con el navegador, dirígete al repositorio `ptavi-p2` en la cuenta del profesor en GitLab[1] y realiza un `fork`[2], de manera que consigas tener una copia del repositorio en tu cuenta de GitLab. Clona en tu ordenador local el repositorio que acabas de crear a local para poder editar los archivos. Trabaja a partir de ahora en ese repositorio, sincronizando (commit) los cambios que vayas realizando según los ejercicios que se comentan a continuación.

   Como tarde al final de la práctica, deberás realizar un `push` para subir tus cambios a tu repositorio en GitLab.

3. Investiga el archivo `src/calc.py` que se encuentra en la carpeta `src`. Comprueba que en él se implementa una calculadora sencilla que permite sumar y restar. **Es importante que los nombres de las funciones sean sumar y restar.** El programa tiene las siguientes características:

   - Ha de ser llamado de la siguiente manera por línea de instrucciones (*shell*):

     ```
     $ python3 src/calc.py operando1 operación operando2
     ```

     donde operación podrá ser `suma` o `resta`. Para tomar los parámetros del programa, se podrá hacer uso del módulo sys (`import sys`), en particular de la lista `sys.argv`. Se comprobará que los parámetros que el usuario pasa son numéricos

[1] `http://GitLab.etsit.urjc.es/ptavi/ptavi-p2`

[2] Tienes instrucciones de cómo realizar un `fork` en `https://GitLab.etsit.urjc.es/help/GitLab-basics/fork-project.md`.

(`integer` o `float`), imprimiendo `Error: Non numerical parameters` por pantalla en caso contrario.

- Tener dos funciones (métodos): `sumar` y `restar`.
- Imprimir el resultado por pantalla.

4. Crea en el archivo `src/calcoo.py` un programa Python que implemente la misma funcionalidad (y se ejecute de la misma manera) que `calc.py`, pero orientada a objetos. Para tal fin, crea una clase llamada `Calculadora` (las mayúsculas son importantes), que tenga los métodos suma y resta (suma y resta han de devolver el resultado, ¡pero no imprimirlo por pantalla!). A su vez, el programa principal deberá tomar los parámetros que el usuario ha dado en la línea de comandos (con `sys.argv`), instanciar un objeto de la clase Calculadora, llamar al método correspondiente e imprimir por pantalla el resultado.

5. Crea en el archivo `src/calcoohija.py` un programa Python que además de la misma funcionalidad de `calcoo.py`, pueda multiplicar y dividir. Para tal fin, crea una clase `CalculadoraHija` (las mayúsculas son importantes) que herede de `Calculadora`, y que además tenga los métodos multiplicar y dividir. En el caso de dividir, ha de capturar la excepción que saltará si `operando2` es cero, imprimiendo el siguiente mensaje de error por pantalla: `No se dividir por cero`. El programa será llamado de la siguiente manera desde la línea de instrucciones (shell):

```
$ python3 src/calcoohija.py operando1 operación operando2
```

donde operación podrá ser `suma`, `resta`, `multiplica` o `divide`.

6. Crea el archivo `src/calcplus.py` que será llamado de la siguiente manera desde la línea de instrucciones (shell):

```
$ python3 src/calcplus.py fichero
```

donde `fichero` es un fichero de texto posiblemente multilínea con formato `CSV` (`comma-separated`
esto es, cada línea del mismo tendrá la siguiente forma:

```
operación,operando1,operando2,operando3,...,operandoN
```

`calcplus.py` deberá tomar la operación línea a línea y realizarla de manera secuencial con todos los operandos (el primero con el segundo, el resultado de la operación con el tercero, y así sucesivamente) imprimiendo el resultado por pantalla. Así, para las siguientes líneas:

```
suma,1,2,3,4,5
resta,31,6,4,3,2,1
multiplica,1,3,5
divide,300,10,2
```

el resultado que se imprimirá por pantalla será, para el ejemplo que se da, 15 en todos los casos. En el caso de que la operación sea de división y uno de los operandos sea cero, se imprimirá por pantalla `No se dividir por cero`. El fichero `calcplus.py` deberá hacer uso de la funcionalidad implementada en `calcoohija.py`.

7. Crea el archivo `src/calcplusplus.py` que será llamado de la siguiente manera por línea de instrucciones (shell):

```
$ python3 src/calcplusplus.py fichero
```

`calcplusplus.py` ha de tener la misma funcionalidad que `calcplus.py` (ver ejercicio anterior), pero deberá hacer uso del módulo `csv` de Python3[3] y de la sentencia `with` de Python.

8. Aunque el intérprete de Python admite ciertas libertades a la hora de programar, los programadores de Python con la finalidad de mejorar principalmente la legibilidad del código han acordado seguir una guía de estilo. Esta guía de estilo se encuentra en el `Python Enhancement Proposal 8` (PEP 8), y contiene instrucciones sobre cómo situar los

[3] https://docs.python.org/3.8/library/csv.html

espacios en blanco, cómo nombrar las variables, etc. Puedes encontrar la guía (original, en inglés, y una versión parcial en castellano) en el Moodle de la asignatura.

Existe un programa de ayuda de línea de comandos llamado `pycodestyle`[4] que te permite comprobar si se siguen la mayoría de las indicaciones de PEP8. Puedes ejecutar el comando:

```
$ python3 check_style.py
```

para comprobar que tu código sigue efectivamente la guía de estilo.

B.5. ¿Qué deberías tener al finalizar la práctica?

La entrega de práctica se deberá hacer antes del miércoles 21 de octubre de 2020 a las 23:59. Para ello, se deberá contar a esa fecha con

1. Tener un repositorio git en GitLab con:
   - 5 módulos Python: `calc.py`, `calcoo.py`, `calcoohija.py`, `calcplus.py`, `calcplusplus.py`.
   - 2 clases: `Calculadora` y `CalculadoraHija`.
   - Todo ello siguiendo la guía de estilo PEP8.

Se han de tener en cuenta las siguientes consideraciones:

- Se valorará que al menos haya diez `commits` realizados.
- Se valorará que el código entregado siga la guía de estilo de Python (véase PEP8).
- Se valorará que los programas se invoquen correctamente y que muestren los errores correctamente, según se indica en el enunciado de la práctica.

[4] Antiguamente este programa se llamaba `pep8`, pero se cambió el nombre para no dar a entender que el programa es equivalente a la guía de estilo.

Se puede comprobar la correcta funcionalidad de la práctica utilizando el programa `run_tests.py` incluido en el repositorio de la práctica. Este programa se ejecuta desde la línea de comandos de la siguiente manera:

```
$ python3 run_tests.py
```

Podemos comprobar si la subida ha sido correcta en nuestro repositorio checkeando el tick verde.

Es imprescindible que todos los métodos y clases descritos se llamen igual que como se describe en la práctica en caso contrario no pasará los tests.

Apéndice C

Enunciado de la práctica 3

Nota: Esta práctica se puede entregar para su evaluación como parte de la nota de prácticas, pudiendo obtener el estudiante hasta un punto. Para las instrucciones de entrega, mira al final del documento. Para la evaluación de esta entrega se valorará el correcto funcionamiento de lo que se pide, el seguimiento de la guía de estilo de Python y el correcto uso (y entrega) con git en GitLab.

C.1. Introducción

Python ofrece una serie de bibliotecas para manipular ficheros en XML (como SMIL). En esta práctica, veremos cómo utilizar la biblioteca SAX.

C.2. Objetivos de la práctica

- Profundizar en el uso de SMIL, XML y JSON.
- Aprender a utilizar la biblioteca SAX para el manejo de XML, en particular con Python.
- Utilizar el sistema de control de versiones git en GitLab.

C.3. Conocimientos previos necesarios

1. Nociones de Python (las de la primera práctica) y de orientación a objetos (la segunda práctica)

2. Nociones de XML y SMIL (las presentadas en clase de teoría)

Tiempo estimado (para un alumno medio): 10 horas

C.4. Ejercicios

1. Con el navegador, dirígete al repositorio `ptavi-p3` en la cuenta del profesor en GitLab[1] y realiza un `fork`[2], de manera que consigas tener una copia del repositorio en tu cuenta de GitLab. Clona en tu ordenador local el repositorio que acabas de crear a local para poder editar los archivos.

   Trabaja a partir de ahora en la práctica, sincronizando (`commit`) los cambios que vayas realizando al hacer los próximos ejercicios. Recuerda que al final del todo tendrás que *empujar* estos cambios al repositorio en GitLab (con `push`).

2. Inspecciona el fichero `practica3-chistes.py`. Verás que el fichero consta de tres partes:

   - Importación de un método y una clase del módulo `xml.sax`. Nótese cómo la importación es ligeramente diferente a lo que hemos usado hasta ahora. Mediante esta forma incluimos el espacio de nombres de los módulos que importamos, por lo que no hace falta poner el nombre del módulo al llamar en nuestro programa a las clases, métodos y variables de esos módulos.
   - La clase `ChistesHandler`, que hereda de la clase ContentHandler. Los métodos de esta clase son eventos que el *parser* lanza cuando se encuentre una etiqueta de inicio (`startElement`), una etiqueta de final (`endElement`) o entre una etiqueta de inicio y final (`characters`). En este último caso, dependiendo de qué *flag*

[1] `https://GitLab.etsit.urjc.es/ptavi/ptavi-p3`

[2] Tienes instrucciones de cómo realizar un `fork` en `https://docs.GitLab.com/ee/workflow/forking_workflow.html#creating-a-fork`.

(`inPregunta` o `inRespuesta`) esté a uno, se irá almacenando el contenido en la variable correspondiente.

- Las instrucciones de ejecución: creación del `parser`, instanciación de la clase `ChistesHandler`, configuración del `parser` para que use `ChistesHandler` como manejador, y el *parsing* del fichero `chistes2.xml` (fichero que también encontrarás en el repositorio).

El *script* en Python guarda el contenido de los elementos en un atributo de la clase y, a continuación, lo borra para que no se concatene. Modifícalo para que, antes de borrarlos, imprima por pantalla las preguntas, las respuestas y la calificación de cada uno de los chistes.

[No hace falta hacer un *commit* de este ejercicio, ya que este ejercicio es simplemente para familiarizarse con el uso de SAX.]

3. En el fichero `src/smallsmilhandler.py`, crea una clase llamada `SmallSMILHandler` que herede de la clase `ContentHandler`. Las etiquetas SMIL que deberá reconocer nuestra clase son las siguientes (se enumeran, junto con las etiquetas, los atributos que se han de tenerse también en cuenta):

   - root-layout (width, height, background-color)
   - region (id, top, bottom, left, right)
   - img (src, region, begin, dur)
   - audio (src, begin, dur)
   - textstream (src, region)

   La clase `SmallSMILHandler` deberá tener, además, un método llamado `get_tags` que devolverá una lista con las etiquetas encontradas, sus atributos y el contenido de los atributos. Piensa bien cómo ha de ser esta lista - nótese que el orden de las etiquetas es importante y se ha de preservar, pero no es necesario que sea así con los atributos. Puedes utilizar el fichero `karaoke.smil` que encontrarás en el repositorio para probar, aunque tu código debería funcionar con cualquier SMIL con las etiquetas indicadas más arriba.

4. Crea un programa principal, en un archivo llamado `src/karaoke.py`, que haga uso de la clase `SmallSMILHandler`. Este programa deberá:

   - Leer un fichero SMIL que se pase por línea de *shell*. En caso de que no se especifique un fichero, mostrará por pantalla: `Usage: python3 src/karaoke.py file.smil`.

     Para realizar pruebas, se puede utilizar el fichero `karaoke.smil`. Nota que `file.smil` es un argumento que se pasa al programa, por lo que se podría pasar cualquier fichero, tenga o no extensión .smil.

   - Mostrar por pantalla un listado ordenado de las etiquetas y de los pares atributo-valor (para aquéllos que tengan un valor asignado), cada uno en una línea, separados por tabuladores y sin espacios antes y después del signo igual, tal y como se muestra a continuación[3]:

     ```
     Elemento1\tAtributo11="Valor11"\tAtributo12="Valor12"\t...\n
     Elemento2\tAtributo21="Valor21"\tAtributo22="Valor22"\t...\n
     ...
     ```

     Se valorará que la salida del programa siga al pie de la letra lo indicado. En la versión final a entregar, por tanto, no imprimas mensajes de trazas. Ejemplo de salida correcta:

     ```
     root-layout\twidth="248"\theight="300"\tbackground-color="blue"\n
     region\tid="a"\ttop="20"\tleft="64"\n
     ```

   - Crear un fichero de las etiquetas en formato JSON. Puedes utilizar para ello la biblioteca `json` de Python 3. El JSON creado deberá permitir regenerar fácilmente la estructura del fichero SMIL original. El nombre del fichero ha de ser el mismo que el del SMIL, simplemente modificando la extensión a `.json`. Así, si el fichero de entrada es `karaoke.smil`, el fichero JSON será `karaoke.json`.

[3] Fíjate en los tabuladores (\t) y los saltos de línea (\n). Puedes utilizar la función `format` de la biblioteca estándar de Python 3. Nota que, como se comentaba antes, los *elementos* han de seguir el orden del SMIL, mientras que los *atributos* dentro de los elementos no hace falta que guarden el orden.

5. Modifica el programa principal `karaoke.py` para que se descargue en local el contenido multimedia remoto referenciado en el SMIL. De esta manera, si el atributo `src` tiene como valor un elemento en remoto (o sea, algo que empiece por `http://`), se deberá descargar ese elemento en local. Para descargarnos por ejemplo `http://gsyc.es/logo.gif`, utilizaremos la función `urlretrieve` de la biblioteca de Python 3 `urllib.request`.

6. Si el recurso es remoto, modifica el valor del atributo correspondiente, indicando ahora su localización local. Así, si el remoto era `http://gsyc.es/logo.gif`, ahora será `logo.gif`.

7. Modifica el programa principal `karaoke.py` para que toda la funcionalidad descrita en los ejercicios 3, 4 y 5 sea orientada a objetos, específicamente, en una clase llamada `KaraokeLocal`. Esta clase deberá tener los siguientes métodos:

   - Inicializador: se le pasará como parámetro el fichero fuente SMIL que el usuario introduce por vía de comandos. El constructor parseará el fichero SMIL y obtendrá las etiquetas (mediante `get_tags` del objeto de tipo `SmallSMILHandler`).
   - `__str__`, que a partir de la lista de etiquetas y atributos, devolverá un string listo para ser imprimido como se hacía en el ejercicio 4.
   - `to_json`, que a partir de la lista de etiquetas y atributos, guardará un fichero en formato JSON tal y como se hacía en el ejercicio 4. Este método tendrá dos atributos: el nombre del fichero SMIL original y el nombre del fichero JSON resultante. Si el nombre del fichero JSON resultante se obviara en la llamada al método, el fichero JSON resultante ha de tener el mismo nombre que el SMIL original, pero con extensión `.json`.
   - `do_local`, que incluya la funcionalidad para descargar los recursos remotos (véase ejercicio 5).

   El programa principal, que vendrá al final del fichero con una sentencia `"__main__"`, constará de lo siguiente:

   a) Comprobará que no hay errores en la invocación por parte del usuario (véase ejercicio 4).

b) Se instanciará un objeto de la clase `KaraokeLocal`

c) Se imprimirá el objeto[4]

d) Se llamará a `to_json` pasándole sólo un parámetro (o sea, el fichero JSON resultante se deberá llamar como el fichero SMIL, pero con extensión diferente)

e) Se llamará a `do_local`

f) Se llamará a `to_json` (en este caso, el fichero JSON resultante se deberá llamar local.json)

g) Y se imprimirá otra vez el objeto.

Nótese que el programa sólo deberá imprimir por pantalla las salidas que se indican (básicamente, al hacer *print*s del objeto). El resto del programa, al entregar la práctica, no ha de contener trazas (o sea, no ha de tener más *prints*).

C.5. ¿Qué deberías tener al finalizar la práctica?

La entrega de práctica se deberá hacer antes del miércoles 4 de noviembre de 2020 a las 23:59. Para entonces, se debe:

1. Tener un repositorio git en GitLab con:
   - Los ficheros `.gitignore`, `LICENSE` y `README.md` que hay en (casi) todo repositorio GitLab.
   - 2 módulos Python (y únicamente estos dos ficheros):
     - `smallsmilhandler.py`
     - `karaoke.py`
   - 2 clases:
     - `SmallSMILHandler` (en `smallsmilhandler.py`)
     - `KaraokeLocal` (en `karaoke.py`)

Se ha de tener en cuenta las siguientes consideraciones:

[4] Se ha de tener en cuenta que cuando se imprime el objeto, se llama al método __str__ del objeto.

- Se valorará que al menos haya ocho commits realizados, en dos días diferentes.
- Se valorará que el código entregado siga la guía de estilo de Python (véase PEP8).
- Se valorará que los programas se invoquen correctamente y que muestren los errores correctamente, según se indica en el enunciado de la práctica.

Se puede comprobar la correcta funcionalidad de la práctica utilizando el programa `run_tests.py` incluido en el repositorio de la práctica. Este programa se ejecuta desde la línea de comandos de la siguiente manera:

```
$ python3 run_tests.py
```

Podemos comprobar si la subida ha sido correcta en nuestro repositorio checkeando el tick verde.

Es imprescindible que todos los métodos y clases descritos se llamen igual que como se describe en la práctica; en caso contrario, no pasará los tests.

Apéndice D

Enunciado de la práctica 4

Nota: Esta práctica se puede entregar para su evaluación como parte de la nota de prácticas. Para las instrucciones de entrega, mira al final del documento. Para la evaluación de esta entrega se valorará el correcto funcionamiento de lo que se pide, el seguimiento de la guía de estilo de Python y el correcto uso (y entrega) con git en GitLab.

D.1. Introducción

`socketserver` es un módulo en Python que simplifica la tarea de implementar servicios en Internet. Aunque hay cuatro tipos diferentes de servidores básicos, nosotros en esta práctica utilizaremos solamente `UDPServer`, que utiliza datagramas – paquetes discretos que pueden llegar desordenados, incluso se pueden perder por el camino (nótese que esto es en contraposición con `TCPServer` que trabaja con flujos TCP). `UDPServer` trabaja de manera secuencial, lo que significa que las peticiones serán atendidas de una en una, por lo que tendrán que esperar si hay una petición en proceso.

D.2. Objetivos de la práctica

- Manejar SIP de manera sencilla.
- Crear un esquema cliente-servidor en Python.

D.3. Conocimientos previos necesarios

1. Nociones de Python (las de las primeras prácticas) y de orientación a objetos.

2. Nociones de SIP (las vistas en clase de teoría)

Tiempo estimado (para un alumno medio): 10 horas

D.4. Ejercicios

1. Esta práctica requiere hacer capturas con `Wireshark`. Para que el administrador de sistemas te incluya en el grupo con permisos para poder hacerlo en las máquinas del laboratorio, abre una *shell* e introduce esta instrucción[1]:

```
$ touch $HOME/.ptavi2021
```

2. Con el navegador, dirígete al repositorio `ptavi-p4` en la cuenta de la asignatura en GitLab[2] y realiza un `fork`, de manera que consigas tener una copia del repositorio en tu cuenta de GitLab. Clona el repositorio que acabas de crear a local para poder editar los archivos. Trabaja a partir de ahora en ese repositorio, sincronizando (haciendo commit) los cambios que vayas realizando según los ejercicios que se comentan a continuación.

3. Estudia el código de un sencillo cliente UDP en `src/client.py` que encontrarás en tu repositorio local. Fíjate en que:

   a) Importa el módulo `socket`.

   b) Inicializa varias *variables* constantes (nota que al ser constantes, vienen en mayúsculas, como marca la convención correspondiente en PEP8).

   c) Crea un `socket`.

   d) Se conecta con un servidor.

[1] Esta instrucción crea un archivo vacío en tu cuenta. El administrador de sistema buscará por este archivo para incluir a los usuarios en el grupo con permisos para capturar con `Wireshark`.

[2] `http://GitLab.etsit.urjc.es/ptavi/ptavi-p4`

e) Envía una **secuencia de bytes** por el `socket` con el método `send()` y lee del `socket` con `recv()`.

f) Al recibir con `recv()`, indicamos el valor del *buffer* en bytes.

g) No hace falta cerrar explícitamente la conexión, ya que se hace automáticamente al salir del contexto del `with`.

4. Estudia el código de `server.py`, que implementa un servidor de respuesta basado en UDP. Fíjate en que:

   a) Importa el módulo `socketserver`[3].

   b) Tenemos una única clase que manejará las peticiones.

   c) Esta clase hereda de una clase `DatagramRequestHandler` que hay en el módulo `socketserver`.

   d) La clase no tiene constructor `__init__`; utiliza el de la clase padre.

   e) La clase sólo tiene un método, llamado `handle()`.

   f) El método `handle()` se ejecuta cada vez que recibimos una petición en el servidor.

   g) `self.wfile` y `self.rfile` son los atributos que abstraen el `socket` (como si fuera un fichero):

   - Podemos leer del mismo, iterando sobre `self.rfile` como si fuera un fichero de lectura.
   - Podemos escribir en el mismo con `self.wfile.write()`.

   h) Enviamos y recibimos **secuencias de bytes**.

   i) Un programa principal, donde se instancia la clase `EchoHandler`, indicando la IP y el puerto donde se deja al servidor escuchando en un bucle infinito (del que sólo se puede salir desde el terminal con `Ctrl+C`, que lanza una excepción `KeyboardInterrupt`.

5. Cambia el *script* del cliente para que:

   - Se pase como parámetro al *script* la IP y el puerto del servidor, así como a continuación el mensaje que se ha de enviar. Nota que la *shell* el mensaje a enviar nos lo

[3] `https://docs.python.org/3/library/socketserver.html`

dará como `string` y que tendremos que transformarlo en una secuencia de bytes. Para ejecutar el cliente, deberíamos hacer:

```
$ python3 src/client.py ip puerto linea
```

Una ejemplo de llamada sería:

```
python3 src/client.py 127.0.0.1 5060 eco eco, soy yo
```

6. Modifica el *script* del servidor para que:
   - Imprima por pantalla la IP y el puerto del cliente (esta información viene en el atributo `client_address` en forma de tupla[4]).
   - Se pase el puerto al que ha de escuchar como parámetro al *script*.
7. Modifica los *scripts* anteriores para tener un cliente que envíe mensajes de registro SIP y un servidor `Registrar` SIP. Cada vez que el cliente le mande una línea con el método `REGISTER` (en mayúsculas), el servidor guardará la dirección registrada y la IP en un diccionario[5] llamado `Users`(es imprescindible llamarlo así, en caso contrario no pasará los tests). Cambia el nombre de la clase del servidor de `EchoHandler` a `SIPRegisterHandler`. La petición SIP del cliente tendrá una pinta parecido a lo siguiente:

   ```
   REGISTER sip:luke@polismassa.com SIP/2.0\r\n\r\n
   ```

   El servidor deberá responder con un mensaje de este estilo:

   ```
   SIP/2.0 200 OK\r\n\r\n
   ```

   El cliente se deberá seguir ejecutándose desde línea de comando, ahora de la siguiente manera:

[4] Las tuplas son listas especiales, ya que son inmutables. Se definen con paréntesis, no con corchetes.

[5] Este diccionario ha de ser un atributo de clase no de la instancia, véase `http://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide`

```
$ python3 src/client.py ip puerto register luke@polismassa.com
```

8. Añade funcionalidad al cliente y al servidor para ofrecer la posibilidad de darse de baja a un usuario. Para ello, hay que añadir una cabecera `Expires` (cuyos valores vienen dados en segundos). En caso de que el valor de la cabecera `Expires` sea 0, el usuario será borrado del diccionario de registro; en cualquier otro caso, siempre que el tiempo sea positivo, el valor de `Expires` será el tiempo de expiración en el servidor. **En este ejercicio es indispensable la creación de un método denominado check_expiration al que se pasa como argumento un tiempo en formato *strftime* y que sea capaz de eliminar al usuario del diccionario Users.** La petición del cliente tendrá una pinta parecida a ésta:

   ```
   REGISTER sip:luke@polismassa.com SIP/2.0\r\n
   Expires: 0\r\n\r\n
   ```

   El servidor deberá borrar al usuario del diccionario y responder con un

   ```
   SIP/2.0 200 OK\r\n\r\n
   ```

   El cliente se deberá seguir ejecutando desde línea de comando, ahora de la siguiente manera:

   ```
   $ python src/client.py localhost 2500 register luke@polismassa.com 36
   ```

   donde 3600 es un ejemplo del tiempo de expiración y será el valor de la cabecera `Expires`. En caso de que no se introduzcan los valores necesarios, se imprimirá el siguiente mensaje:

   ```
   Usage: client.py ip puerto register sip_address expires_value
   ```

9. Modifica el servidor para añadir el método `register2json` en el que se implemente la siguiente funcionalidad: cada vez que un usuario se registre o se dé de baja, se imprimirá

en el fichero `registered.json` con información sobre el usuario, su dirección y la hora de expiración. A continuación, se puede ver un ejemplo[6]:

```
{
    [
        "luke@polismassa.com",
        {
            "address": "localhost",
            "expires": "2019-10-16 10:37:12 +0000"
        }
    ]
}
```

Para los formatos de tiempo, se recomienda utilizar el módulo time, en particular: `time()`, que devuelve los segundos desde el 1 de enero de 1970, `gmtime()`, que toma los segundos desde el 1 de enero de 1970 y te lo devuelve en una tupla, y `strptime()`, que representa el tiempo en un `string`. Así,

```
time.strftime('%Y-%m-%d %H:%M:%S', time.gmtime(time.time()))
```

devolverá un `string` con la hora GMT actual. Y

```
time.strftime('%Y-%m-%d %H:%M:%S', time.gmtime(1233213))
```

devolverá un `string` con la hora GMT del segundo 1.233.213 desde el 1 de enero de 1970. En este ejercicio, se ha de implementar funcionalidad para que el servidor `Registrar` gestione la caducidad de los usuarios registrados.

10. Modifica el servidor, añadiendo el método `json2registered`, para que cuando se lance, compruebe si hay un fichero llamado `registered.json`. Si existe, se leerá su contenido y se usará como diccionario de usuarios registrados. Si da *cualquier* error al leer el fichero, el servidor se ejecutará como si el fichero JSON no existiera.

[6]Tu fichero JSON no tiene que ser *igual* que lo mostrado; simplemente ha de ser un fichero JSON válido con los datos de usuarios registrados legible por humanos (i.e., la fecha ha de ser entendible).

11. Documenta tu código con `docstrings`[7]. Comprueba asimismo que los nombres de las variables siguen las indicaciones de PEP8. Ten en cuenta que el programa `pep8` no comprueba ninguna de estas dos circunstancias.

12. Realiza una captura con `wireshark` (interfaz `localhost`) con las siguientes interacciones:

    a) El cliente `marty.mcfly@sk8ing.com` se registra. Tiempo de expiración: 5.

    b) El cliente `doc@delorean.com` se registra. Tiempo de expiración: 3600.

    c) (Se dejan pasar unos segundos, más de cinco).

    d) El cliente `doc@delorean.com` se da de baja.

    e) El cliente `marty.mcfly@sk8ing.com` se da de baja.

    Investiga tu captura. Comprueba, en particular, que puedes ver el intercambio de mensajes y que todo va en texto claro por la red. Guarda la captura en un fichero de nombre `register.libpcap` y súbelo al repositorio.

D.5. ¿Qué deberías tener al finalizar la práctica?

La entrega de práctica se deberá hacer antes del miércoles 18 de noviembre de 2020 a las 23:59. Para entonces, se debe:

1. Tener un repositorio git en GitLab con:

   - 2 módulos Python y la captura realizada con `wireshark` (y únicamente estos tres ficheros):
     - `server.py`
     - `client.py`
     - `register.libpcap`
   - 1 clase `SIPRegisterHandler` (en `server.py`) con los métodos:

[7] La recomendación para `docstrings` en Python se puede encontrar en `http://legacy.python.org/dev/peps/pep-0257/`.

a) `handle`

b) `check_expiration`

c) `register2json`

d) `json2register`

- Los ficheros adicionales incluida la carpeta de tests que te descargaste en primera instancia.

Se han de tener en cuenta las siguientes consideraciones:

- Se valorará que al menos haya diez `commits` realizados, en dos días diferentes.
- Se valorará que el código entregado siga la guía de estilo de Python.
- Se valorará que los programas se invoquen correctamente y que muestren los errores correctamente, según se indica en el enunciado de la práctica.

Se puede comprobar la correcta funcionalidad de la práctica utilizando el programa `run_tests.py` incluido en el repositorio de la práctica. Este programa se ejecuta desde la línea de comandos de la siguiente manera:

```
$ python3 run_tests.py
```

Podemos comprobar si la subida ha sido correcta en nuestro repositorio checkeando el tick verde.

Es imprescindible que todos los métodos y clases descritos se llamen igual que como se describe en la práctica en caso contrario no pasará los tests.

Apéndice E

Enunciado de la práctica 5

Nota: Esta práctica se puede entregar para su evaluación como parte de la nota de prácticas. Para las instrucciones de entrega, mira al final del documento.

E.1. Introducción

El protocolo de iniciación de sesión (SIP) es un protocolo que se limita solamente al establecimiento y control de una sesión (RFC 3550). Los detalles del intercambio de datos, como por ejemplo la codificación o decodificación del audio/vídeo, no son controlados por SIP sino que se llevan a cabo por otros protocolos (por ejemplo, RTP). Los principales objetivos de SIP son:

- SIP permite el establecimiento de una localización de usuario (o sea, traducir de un nombre de usuario a su dirección de red actual)
- SIP provee funcionalidad para la negociación de las características de una sesión, de manera que los participantes en una sesión pueden consensuar características soportadas por todos ellos.
- SIP es un mecanismo para la gestión de llamadas, por ejemplo para añadir, eliminar o transferir participantes
- SIP permite la modificación de características de la sesión durante su transcurso.

E.2. Objetivos de la práctica

1. Conocer el protocolo SIP y otros protocolos utilizados en una sesión con clientes SIP (como `Ekiga` o `linphone`).

2. Profundizar en el uso de `Wireshark`: análisis, captura y filtrado.

E.3. Conocimientos previos necesarios

- Nociones de SIP y RTP (las de clase de teoría)

- Funcionamiento de `Wireshark`.

Tiempo estimado (para un alumno medio): 8 horas

E.4. Ejercicios

Creación de repositorio para la práctica

1. Con el navegador, dirígete al repositorio `ptavi-p5` en la cuenta de la asignatura en GitLab[1] y realiza un `fork`, de manera que consigas tener una copia del repositorio en tu cuenta de GitLab. Clona el repositorio que acabas de crear a local para poder editar los archivos. Trabaja a partir de ahora en ese repositorio, sincronizando los cambios que vayas realizando.

   Como tarde al final de la práctica, deberás realizar un `push` para subir tus cambios a tu repositorio en GitLab. En esta práctica, al contrario que con las demás, se recomienda hacer frecuentes `commits`, pero el `push` al final.

[1] `http://GitLab.etsit.urjc.es/ptavi/ptavi-p5`

Análisis de una sesión SIP

Se ha capturado una sesión SIP con el cliente SIP Ekiga (archivo `sip.cap.gz`), que se puede abrir con `Wireshark`[2]. Se pide rellenar las cuestiones que se plantean en este guión en el fichero `p5.txt` que encontrarás también en el repositorio.

2. Observa que las tramas capturadas corresponden a una sesión SIP con `Ekiga`, un cliente de VoIP para `GNOME`. Responde a las siguientes cuestiones:

   - ¿Cuántos paquetes componen la captura?
   - ¿Cuánto tiempo dura la captura?
   - ¿Qué IP tiene la máquina donde se ha efectuado la captura? ¿Se trata de una IP pública o de una IP privada? ¿Por qué lo sabes?

3. Antes de analizar las tramas, mira las estadísticas generales que aparecen en el menú de `Statistics`. En el apartado de jerarquía de protocolos (`Protocol Hierarchy`) se puede ver el porcentaje del tráfico correspondiente al protocolo TCP y UDP.

   - ¿Cuál de los dos es mayor? ¿Tiene esto sentido si estamos hablando de una aplicación que transmite en tiempo real?
   - ¿Qué otros protocolos podemos ver en la jerarquía de protocolos? ¿Cuales crees que son señal y cuales ruido?

4. Observa por encima el flujo de tramas en el menú de `Statistics` en `IO Graphs`. La captura que estamos viendo incluye desde la inicialización (registro) de la aplicación hasta su finalización, con una llamada entremedias.

   - Filtra por `sip` para conocer cuándo se envían paquetes SIP. ¿En qué segundos tienen lugar esos envíos?
   - Y los paquetes con RTP, ¿cuándo se envían?

5. Analiza las dos primeras tramas de la captura.

[2]Desde la shell: `$ wireshark sip.cap.gz` - recuerda que puedes utilizar el tabulador en la shell para autocompletar.

- ¿Qué servicio es el utilizado en estas tramas?
- ¿Cuál es la dirección IP del servidor de nombres del ordenador que ha lanzado Ekiga?
- ¿Qué dirección IP (de `ekiga.net`) devuelve el servicio de nombres?

6. A continuación, hay más de una docena de tramas TCP/HTTP.

   - ¿Podrías decir la URL que se está pidiendo?
   - ¿Qué `user agent` (UA) la está pidiendo?
   - ¿Qué devuelve el servidor?
   - Si lanzamos el navegador web, por ejemplo, `Mozilla Firefox`, y vamos a la misma URL, ¿qué recibimos? ¿Qué es, entonces, lo que está respondiendo el servidor?

7. Hasta la trama 45 se puede observar una secuencia de tramas del protocolo STUN.

   - ¿Por qué se hace uso de este protocolo?
   - ¿Podrías decir si estamos tras un `NAT` o no?

8. La trama 46 es la primera trama SIP. En un entorno como el de Internet, lo habitual es desconocer la dirección IP de la otra parte al realizar una llamada. Por eso, todo usuario registra su localización en un servidor `Registrar`. El `Registrar` guarda información sobre los usuarios en un servidor de localización que puede ser utilizado para localizar usuarios.

   - ¿Qué dirección IP tiene el servidor `Registrar`?
   - ¿A qué puerto (del servidor `Registrar`) se envían los paquetes SIP?
   - ¿Qué método SIP utiliza el UA para registrarse?
   - Además de `REGISTER`, ¿podrías decir qué instrucciones SIP entiende el UA?

9. Fijémonos en las tramas siguientes a la número 46:

   - ¿Se registra con éxito en el primer intento?

- ¿Cómo sabemos si el registro se ha realizado correctamente o no?
- ¿Podrías identificar las diferencias entre el primer intento y el segundo de registro? (fíjate en el tamaño de los paquetes y mira a qué se debe el cambio)
- ¿Cuánto es el valor del tiempo de expiración de la sesión? Indica las unidades.

10. Una vez registrados, podemos efectuar una llamada. Vamos a probar con el servicio de eco de Ekiga que nos permite comprobar si nos hemos conectado correctamente. El servicio de eco tiene la dirección `sip:500@ekiga.net`. Veamos el `INVITE` de cerca.
    - ¿Puede verse el nombre del que efectúa la llamada, así como su dirección SIP?
    - ¿Qué es lo que contiene el cuerpo de la trama? ¿En qué formato/protocolo está?
    - ¿Tiene éxito el primer intento? ¿Cómo lo sabes?
    - ¿En qué se diferencia el segundo `INVITE` más abajo del primero? ¿A qué crees que se debe esto?
11. Una vez conectado, estudia el intercambio de tramas.
    - ¿Qué protocolo(s) se utiliza(n)? ¿Para qué sirven estos protocolos?
    - ¿Cuál es el tamaño de paquete de los mismos?
    - ¿Se utilizan bits de `padding`?
    - ¿Cuál es la periodicidad de los paquetes (en origen; nota que la captura es en destino)?
    - ¿Cuántos bits/segundo se envían?
12. Vamos a ver más a fondo el intercambio RTP. En `Telephony` hay una opción RTP. Empecemos mirando los flujos RTP.
    - ¿Cuántos flujos hay? ¿por qué?
    - ¿Cuántos paquetes se pierden?
    - ¿Cuál es el valor máximo del delta? ¿Y qué es lo que significa el valor de `delta`?
    - ¿Cuáles son los valores de `jitter` (medio y máximo)? ¿Qué quiere decir eso? ¿Crees que estamos ante una conversación de calidad?

13. Elige un paquete RTP de audio. Analiza el flujo de audio en `Telephony ->RTP ->Stream Analysis`.

    - ¿Cuánto valen el `delta` y el `jitter` para el primer paquete que ha llegado?
    - ¿Podemos saber si éste es el primer paquete que nos han enviado?
    - Los valores de `jitter` son menores de 10ms hasta un paquete dado. ¿Cuál?
    - ¿A qué se debe el cambio tan brusco del `jitter`?
    - ¿Es comparable el cambio en el valor de `jitter` con el del `delta`? ¿Cual es más grande?

14. En `Telephony` selecciona el menú `VoIP calls`. Verás que se lista la llamada de voz IP capturada en una ventana emergente. Selecciona esa llamada y pulsa el botón `Play Streams`.

    - ¿Cuánto dura la conversación?
    - ¿Cuáles son sus SSRC? ¿Por qué hay varios SSRCs? ¿Hay CSRCs?

15. Identifica la trama donde se finaliza la conversación.

    - ¿Qué método SIP se utiliza?
    - ¿En qué trama(s)?
    - ¿Por qué crees que se envía varias veces?

16. Finalmente, se cierra la aplicación de VozIP.

    - ¿Por qué aparece una instrucción SIP del tipo `REGISTER`?
    - ¿En qué trama sucede esto?
    - ¿En qué se diferencia con la instrucción que se utilizó con anterioridad (al principio de la sesión)?

Captura de una sesión SIP

17. Dirígete a la web de Linphone[3] con el navegador y créate una cuenta SIP. Recibirás un correo electrónico de confirmación en la dirección que has indicado al registrarte (mira en tu carpeta de *spam* si no es así).

18. Lanza `linphone`, y configúralo con los datos de la cuenta que te acabas de crear. Para ello, puedes ir al menú "Opciones" y seleccionar "Preferencias" (o pulsando "Ctrl+P").

19. Verás una ventana similar a la que se muestra en la figura **??**. En la parte superior, verás tus credenciales del laboratorio. Necesitamos crear una *cuenta proxy*, para lo que pulsaremos sobre el botón de "+ Añadir" en la parte derecha.

20. Rellena con tus datos en el menú de la ventana emergente, similar a la figura. Si introduces tus datos de manera correcta, te pedirá la contraseña de tu cuenta SIP en linphone.org que hemos creado en un apartado anterior.

21. Cierra completamente linphone. Captura una sesión SIP de una conversación con el número SIP `sip:music@sip.iptel.org`. Recuerda que has de comenzar a capturar tramas antes de arrancar linphone para ver todo el proceso[4].

22. Observa las diferencias en el inicio de la conversación entre el entorno del laboratorio y el del ejercicio anterior[5]:

    - ¿Se utilizan DNS y STUN? ¿Por qué?
    - ¿Son diferentes el registro y la descripción de la sesión?

23. Identifica las diferencias existentes entre esta conversación y la conversación anterior:

    - ¿Cuántos flujos tenemos?
    - ¿Cuál es su periodicidad?

[3] `https://www.linphone.org/freesip/home`

[4] Linphone corre en *background* una vez abierto, por lo que simplemente "cerrar" su ventana no lo termina. Puedes *matar* todos los procesos de linphone desde la línea de `shell`, puedes hacer $ killall linphone.

[5] Si estás utilizando tu portátil en vez de una máquina del laboratorio, ten en cuenta que la captura puede ser diferente, porque con el portátil estarás conectado a la red *wifi*, mientras que los ordenadores de los laboratorios lo están a una LAN.

- ¿Cuánto es el valor máximo del delta y los valores medios y máximo del jitter?
- ¿Podrías reproducir la conversación desde `Wireshark`? ¿Cómo? Comprueba que poniendo un valor demasiado pequeño para el `buffer` de `jitter`, la conversación puede no tener la calidad necesaria.
- ¿Sabrías decir qué tipo de servicio ofrece `sip:music@iptel.org`?

24. Filtra por los paquetes SIP de la captura y guarda **únicamente** los paquetes SIP como `p5.pcapng`. Abre el fichero guardado para cerciorarte de que lo has hecho bien. Deberás añadirlo al repositorio.

E.5. Fecha y modo de entrega

La entrega de práctica se deberá hacer antes del miércoles 2 de diciembre de 2020 a las 23:59. Para entonces, se debe:

- Tener un repositorio git en GitLab con:
  1. Un archivo de texto con los comentarios a los ejercicios en `p5.txt`.
  2. Un archivo con una captura filtrada en `p5.pcapng`.
  3. Un test de `Moodle` rellenado sobre la práctica 5. Este test tendrá preguntas relacionadas con los ejercicios que se han realizado y tiene un tiempo de realización de 20 minutos.
  4. Los ficheros adicionales que te descargaste en primera instancia.

Se ha de tener en cuenta las siguientes consideraciones:

- Se valorará que al menos haya diez `commits` realizados en al menos dos días diferentes
- Se valorará que la captura esté bien filtrada y sólo contenga los paquetes que se indican en el guión.

Se puede comprobar la correcta funcionalidad de la práctica utilizando el programa `run_tests.py` incluido en el repositorio de la práctica. Este programa se ejecuta desde la línea de comandos de la siguiente manera:

```
$ python3 p5_test.py
```

Podemos comprobar si la subida ha sido correcta en nuestro repositorio checkeando el tick verde.

Es imprescindible que la captura a entregar se llame igual que como se describe en la práctica en caso contrario no pasará los tests.

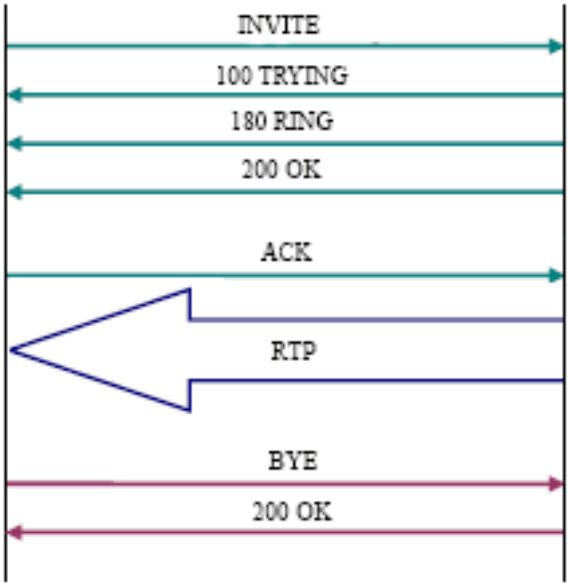
Apéndice F

Enunciado de la práctica 6

Nota: Esta práctica se puede entregar para su evaluación como parte de la nota de prácticas, pudiendo obtener el estudiante hasta 0,7 puntos. Para las instrucciones de entrega, mira al final del documento.

Introducción

Esta práctica tiene como objetivo implementar dos programas Python, de manera que se pueda realizar una sesión SIP como la que se muestra en la siguiente figura[1]:



[1]En la figura las respuestas SIP están en mayúsculas para mejorar su lectura; en tu implementación deberían seguir el estándar, en el que sólo la primera letra es mayúscula

En esta sesión:

- No habrá ni servidor de registro ni *proxy*. La comunicación será *peer-to-peer* (directa entre dos UAs[2]).
- El UA de la izquierda constará exclusivamente de la parte cliente. Esta parte será la que inicie la sesión con un `INVITE` y la cierre con un `BYE`.
- El UA de la derecha constará exclusivamente de la parte servidora. Esta parte será la que envíe el audio vía RTP al cliente.

Objetivos de la práctica

- Implementación de una conversación de audio iniciada con SIP.

Conocimientos previos necesarios

- Nociones de SIP (las de clase de teoría)
- Funcionamiento de `wireshark`.

Tiempo estimado: 10 horas

Creación de repositorio para la práctica

Con el navegador, dirígete al repositorio `ptavi-p6` en la cuenta de *ptavi* en GitLab[3] y realiza un `fork`, de manera que consigas tener una copia del repositorio en tu cuenta de GitLab. Clona el repositorio que acabas de crear a local para poder editar los archivos. Trabaja a partir de ahora en ese repositorio, sincronizando los cambios que vayas realizando.

Como tarde al final de la práctica, deberás realizar un `push` para subir tus cambios a tu repositorio en GitLab. En esta práctica, al contrario que con las demás, se recomienda hacer frecuentes `commits`, pero el `push` al final.

[2]UA: User Agent. Un UA se compone de un cliente y un servidor, aunque en esta práctica -por simplificación- cada UA tendrá sólo una de las dos partes.

[3]`http://GitLab.etsit.urjc.es/ptavi/ptavi-p6`

Parte cliente

El cliente ha de ejecutarse de la siguiente manera:

```
$ python3 client.py metodo receptor@IPreceptor:puertoSIP
```

donde *metodo* será un método SIP, `receptor@IPreceptor:puertoSIP` será el login, la IP y el puerto[4] al que se dirige el mensaje.

Por ejemplo:

```
$ python3 client.py INVITE batman@193.147.73.20:5555
$ python3 client.py BYE batman@193.147.73.20:5555
```

En caso de no introducir el número de parámetros correctos o de error en los mismos, el programa debería imprimir siempre por pantalla:

```
Usage: python3 client.py method receiver@IP:SIPport
```

Peticiones SIP

El cliente debería poder enviar las siguientes peticiones SIP:

- INVITE sip:receptor@IP SIP/2.0

  Mediante este método, el UA indica que quiere iniciar una conversación con el receptor con dirección *receptor* en la máquina dada por la *IP*[5].

- ACK sip:receptor@IP SIP/2.0

  Método de asentimiento. No se pasará al programa `client.py` como parámetro desde la `shell`, ya que se enviará de manera automática una vez se hayan recibido las respuestas `100 Trying`, `180 Ring` y `200 OK` de la parte servidora.

[4]En el mundo "real", el puerto sería el puerto de SIP por defecto, el 5060, ya que sólo puede haber un UA activo por máquina - si arrancas otro UA SIP cuando hay uno ya lanzado, no te dejará. Como en la práctica final tendremos dos UAs en la misma máquina, hemos de especificar un puerto para que no se "pisen".

[5]La IP puede ser 127.0.0.1.

- BYE sip:receptor@IP SIP/2.0

  Mediante este método se indica que queremos terminar una conversación con el receptor con dirección *receptor* en la máquina dada por la IP. Se deberá enviar una vez haya acabado de recibir *streaming* de audio (vía RTP) enviado desde el servidor.

Parte servidora

El servidor ha de ejecutarse de la siguiente manera:

```
$ python3 server.py IP puerto fichero_audio
```

Por ejemplo:

```
$ python3 server.py 127.0.0.1 5555 cancion.mp3
```

En caso de no introducir el número de parámetros correctos o de error en los mismos (se ha de comprobar si existe el fichero de audio), el programa debería imprimir:

```
Usage: python3 server.py IP port audio_file
```

En caso de no haber error al arrancar, el servidor imprimirá por pantalla:

```
Listening...
```

Códigos de respuesta

- `SIP/2.0 100 Trying`: al recibir un INVITE. Puede enviarse en un mismo paquete con las respuestas 180 y 200; en ese caso, no hace falta dejar una línea en blanco después del “100 Trying”.

- `SIP/2.0 180 Ringing`: al recibir un INVITE. Puede enviarse en un mismo paquete con las respuestas 100 y 200; en ese caso, no hace falta dejar una línea en blanco después del “180 Ringing”.

- `SIP/2.0 200 OK`: en caso de éxito.

- `SIP/2.0 400 Bad Request`: si la petición está mal formada.

Último número DNI/NIE	Ej. A	Ej. B	Ej. C	Ej. D	Ej. E	Ej. F
0	X			X		X
1	X			X	X	
2	X		X			X
3	X		X		X	
4	X			X		X
5		X	X			X
6		X	X		X	
7		X		X	X	
8		X		X		X
9		X		X	X	

Cuadro F.1: Tabla de ejercicios alternativos a realizar según el último número de DNI/NIE del estudiante.

- `SIP/2.0 405 Method Not Allowed`: si se manda en la petición cualquier otro método diferente de INVITE, BYE o ACK.

Para simplificar la práctica, se enviará **en un único mensaje** (i.e., un único paquete) la respuesta `SIP/2.0 100 Trying`, `SIP/2.0 180 Ringing` y `SIP/2.0 200 OK`.

Ejercicios alternativos

Dependiendo de tu último número de tu DNI / NIE, deberás realizar los siguientes ejercicios que se describen en la Tabla F.1.

Ejercicio A

El cuerpo del INVITE debe incluir la descripción de sesión en formato SDP (*Session Description Protocol*). Contará con los siguientes parámetros:

- v (versión, por defecto la “0”),
- o (origen e identificador de sesión: la dirección del origen y su IP),

- s (nombre de la sesión, que puede ser el que se desee),
- t (tiempo que la sesión lleva activa, en nuestro caso, siempre 0), y
- m (tipo de elemento multimedia y puerto de escucha y protocolo de transporte utilizados, en esta práctica "audio", el número de puerto pasado al programa principal como parámetro y "RTP").

Así, un ejemplo de descripción de sesión dentro de un INVITE podría ser:

```
INVITE sip:INVITE batman@193.147.73.20 SIP/2.0

v=0
o=robin@gotham.com 127.0.0.1
s=misesion
t=0
m=audio 34543 RTP
```

El servidor guardará la dirección de origen y la dirección IP recibida por SDP en un diccionario[6] para utilizarlo posteriormente en el envío de RTP.

Nótese que en el ejemplo anterior 34543 es el puerto donde esperamos que el otro participante en la conversación nos envíe los paquetes RTP con audio. También se ha de tener en cuenta que entre las cabeceras (en este caso la línea de INVITE) y el cuerpo (la descripción de la sesión) ha de haber obligatoriamente una línea en blanco.

Ejercicio B

El cuerpo de la respuesta 200 OK al INVITE debe incluir la descripción de sesión en formato SDP (*Session Description Protocol*). Contará con los siguientes parámetros:

- v (versión, por defecto la "0"),
- o (origen e identificador de sesión: la dirección del origen y su IP),

[6]Este diccionario ha de ser un atributo de clase no de la instancia, véase `http://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide`

- s (nombre de la sesión, que puede ser el que se desee),
- t (tiempo que la sesión lleva activa, en nuestro caso, siempre 0), y
- m (tipo de elemento multimedia y puerto de escucha y protocolo de transporte utilizados, en esta práctica "audio", el número de puerto pasado al programa principal como parámetro y "RTP").

Así, un ejemplo de descripción de sesión dentro de un INVITE podría ser:

```
SIP/2.0 200 OK

v=0
o=batman@gotham.com 127.0.0.1
s=misesion
t=0
m=audio 67876 RTP
```

Se ha de tener en cuenta que entre las cabeceras (en este caso la línea de INVITE) y el cuerpo (la descripción de la sesión) ha de haber obligatoriamente una línea en blanco. Nótese que como el UA que inicia la conversación no envía paquetes RTP, la información en el SDP no se utilizará.

Ejercicio C

El envío RTP se realizará mediante la biblioteca `simplertp`[7]. Un ejemplo del código Python de envío de paquetes RTP sería el siguiente:

```
import simplertp

RTP_header = simplertp.RtpHeader()
RTP_header.set_header(pad_flag=0, ext_flag=0, cc=0, marker=0, ssrc=ALEA
audio = simplertp.RtpPayloadMp3(audio_file)
simplertp.send_rtp_packet(RTP_header, audio, ip, port)
```

[7]En el repositorio se encuentra el módulo Python `simplertp.py`

Donde ALEAT será un número entero aleatorio obtenido a partir de la biblioteca `random` de Python3[8].

La IP y puerto a la que se ha de enviar el RTP vendrá dada en el SDP (Ejercicio A). En el caso de que el estudiante no tenga que realizar ese ejercicio, utilizará la IP 127.0.0.1 y el puerto 23032 como destino del envío.

Nótese que `audio_file` es uno de los parámetros que se le pasa al servidor al ejecutarse desde la línea de shell como parámetro. También ha de notarse que si sale el aviso **Warning: Connection refused. Probably there is nothing listening on the other end.** es porque no hay ningún servicio *escuchando* al otro lado; no es un error, indica que los paquetes han llegado al destino final, pero que no han sido gestionados.

Ejercicio D

El envío RTP se realizará mediante la biblioteca `simplertp`[9]. Un ejemplo del código Python de envío de paquetes RTP sería el siguiente:

```
import simplertp


RTP_header = simplertp.RtpHeader()
RTP_header.set_header(version=2, marker=BIT, payload_type=14, ssrc=2000
audio = simplertp.RtpPayloadMp3(audio_file)
simplertp.send_rtp_packet(RTP_header, audio, ip, port)
```

Donde BIT será un bit aleatorio obtenido a partir de la biblioteca `secrets` de Python3[10].

La IP y puerto a la que se ha de enviar el RTP vendrá dada en el SDP (Ejercicio A). En el caso de que el estudiante no tenga que realizar ese ejercicio, utilizará la IP 127.0.0.1 y el puerto 23032 como destino del envío.

Nótese que `audio_file` es uno de los parámetros que se le pasa al servidor al ejecutarse desde la línea de shell como parámetro. También ha de notarse que si sale el aviso **Warning: Connection refused. Probably there is nothing listening on the other end.** es porque no hay

[8] https://docs.python.org/3.8/library/random.html

[9] En el repositorio se encuentra el módulo Python `simplertp.py`

[10] https://docs.python.org/3/library/secrets.html

ningún servicio *escuchando* al otro lado; no es un error, indica que los paquetes han llegado al destino final, pero que no han sido gestionados.

Ejercicio E

Se ha de añadir la cabecera `Content-Type: application/sdp` a **todos** los paquetes SIP que contengan SDP.

Así, un ejemplo de descripción de sesión dentro de un INVITE podría ser:

```
INVITE sip:INVITE batman@193.147.73.20 SIP/2.0
Content-Type: application/sdp

v=0
o=robin@gotham.com 127.0.0.1
s=misesion
t=0
m=audio 34543 RTP
```

Ejercicio F

Se ha de añadir la cabecera `Content-Length`, cuyo valor será la longitud (en bytes) del cuerpo del paquete, a **todos** los paquetes SIP que contengan SDP.

Así, un ejemplo de descripción de sesión dentro de un INVITE podría ser:

```
INVITE sip:batman@193.147.73.20 SIP/2.0
Content-Length: 66

v=0
o=robin@gotham.com 127.0.0.1
s=misesion
t=0
m=audio 34543 RTP
```

Captura

Una vez terminada la práctica, se pide que se realice una captura de un establecimiento de llamada, el envío RTP de audio y la finalización de la llamada.

La captura original se filtrará para que sólo incluya los paquetes SIP y los tres primeros y tres últimos paquetes RTP con audio[11].

La captura filtrada resultante se guardará en el fichero `invite.libpcap` y se subirá al repositorio.

Fecha y modo de entrega

La entrega de práctica se deberá hacer antes del miércoles 16 de diciembre de 2020 a las 23:59. Para entonces, se debe tener un repositorio `git` en GitLab con:

- 2 módulos Python y la captura realizada con `wireshark`:
  - `server.py`
  - `client.py`
  - `invite.libpcap`

  Además del resto de archivos que se descargaron del repositorio.

Se han de tener en cuenta las siguientes consideraciones:

- Se valorará que al menos haya diez *commits* realizados en al menos dos días diferentes.
- Se valorará que la captura esté bien filtrada y sólo contenga los paquetes que se indican en el guión.
- Se valorará que el código entregado siga la guía de estilo de Python.
- Se valorará que los programas se invoquen correctamente y que muestren los errores correctamente, según se indica en el enunciado de la práctica.

[11] Los heurísticos de Wireshark puede que no identifiquen los paquetes RTP como tales, sino como paquetes UDP. La solución es la siguiente: en el menú de "Analyze", selecciona "Enabled protocols" y ahí busca por RTP para activar "rtp_udp".

Se puede comprobar la correcta funcionalidad de la práctica utilizando el programa `p6_e2e_test.py` incluido en el repositorio de la práctica. Este programa se ejecuta desde la línea de comandos de la siguiente manera:

```
$ python3 p6_e2e_test.py
```

Podemos comprobar si la subida ha sido correcta en nuestro repositorio checkeando el tick verde.

Es imprescindible que la captura se llame igual que como se describe en la práctica en caso contrario no pasará los tests.

Bibliografía

[1] Agile. https://www.iebschool.com/blog/que-son-metodologias-agiles-agile-

[2] M. S. Arefeen and M. Schiller. Continuous integration using gitlab. *Undergraduate Research in Natural and Clinical Science and Technology Journal*, 3:1–6, 2019.

[3] Coverage. https://coverage.readthedocs.io/en/coverage-5.5/.

[4] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona. Perceval: Software project data at your will. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 1–4, 2018.

[5] R. A. Española. https://dle.rae.es/test.

[6] Gitlab CI/CD. https://docs.gitlab.com/ee/ci.

[7] R. González Duque. Python para todos.

[8] A. Orebaugh, G. Ramirez, and J. Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[9] Python3.8 Documentation. https://docs.python.org/es/3.8/library/.

[10] R.A.E: Nueva Gramática de la Lengua Española. http://aplica.rae.es/grweb/cgi-bin/v.cgi?i=hngkOPRxQjvrPfhk.

[11] Real python. https://realpython.com/.

[12] Unittest. https://docs.python.org/3/library/unittest.html.

[13] Wireshark. https://www.wireshark.org/docs.