Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA DE VALLADOLID

GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Tablon3: Adaptación de una herramienta de gamificación al entorno de ejecución Python 3

Autor:
Daniel López Martínez

Tutor académico:
Arturo González Escribano
Francisco José Andújar Muñoz

Agradecimientos

Quiero agradecer a mis tutores, Arturo González Escribano y Francisco José Andújar Muñoz por su paciencia y dedicación a lo largo del desarrollo de este trabajo.

A mi familia y a Miriam por el apoyo constante que me han brindado a lo largo de toda la carrera, y a mis amigos por los ánimos y ayuda que me han proporcionado en todo momento.

Por último, agradecer el desarrollo de este trabajo a la Universidad de Valladolid, ya que este trabajo se ha desarrollado dentro del proyecto de innovación docente financiado por la misma universidad y titulado "Generalizando la integración de gamificación competitiva y colaborativa de forma ágil (III)", con identificador PID2021_065.

Resumen

En este trabajo se describe el proceso de adaptación del código de la herramienta de gamificación *Tablon* al lenguaje de programación Python 3. Como paso anterior a este proceso se analiza de manera exhaustiva el funcionamiento de la aplicación y se redacta una documentación que detalla la estructura, funcionamiento e implementación de la aplicación. En la documentación de esta herramienta se incluyen desde explicaciones detalladas del contenido de cada fichero hasta diagramas de diseño que representan la estructura de *Tablon*. También se recoge en este documento una explicación del proceso de conversión que se ha realizado para obtener la aplicación final *Tablon3*.

Abstract

This manuscript describes the process of adapting the code of the gamification tool *Tablon* to the Python 3 programming language. As a previous step to this process, the operation of the application is thoroughly analyzed and a documentation detailing the structure, operation and implementation of the application is written. This documentation includes detailed explanations of the content of each file, design diagrams that represent the structure of *Tablon*, etc. This document also includes an explanation of the conversion process that has been carried out to obtain the final *Tablon3* application.

Índice general

Resumen v

Índice de figuras xiv

Índice de tablas xv

1. Introducción 1

1.1. Contexto 1

1.2. Motivación 2

1.3. Objetivos 2

1.4. Estructura del documento 3

2. Plan de trabajo y seguimiento 5

2.1. Planificación del proyecto 5

2.2. Análisis de riesgos 6

2.3. Desviación de la planificación inicial 8

2.4. Software y hardware utilizado 9

2.5. Presupuesto y costes 9

3. Descripción del objeto de estudio 11

3.1. Conocimientos técnicos previos 11

3.1.1. De Python 2 a Python 3 11
3.1.2. Decorators 13
3.2. Estudio de la aplicación original 16
3.2.1. Puesta en marcha del servidor Tablon 16
3.2.2. La interfaz web 23
3.2.3. Envío de programas desde el cliente 28
3.3. Estructuración del código 32
3.3.1. Directorio `tablon/` 35
3.3.2. Fichero `client.py` 40
3.4. Diagramas de diseño 41
3.4.1. Diagrama de módulos y ficheros 41
3.4.2. Diagramas de submódulos y clases 42
3.4.3. Diagramas de secuencia 47
3.4.4. Diagrama de máquina de estados 48

4. Conversión del código **53**

4.1. 2to3 Python 53
4.2. Problemas tras la conversión 54
4.2.1. Localización de los fallos y soluciones aplicadas 54
4.3. Modificaciones y limpieza del código 58
4.4. Batería de pruebas 60
4.4.1. Casos de prueba 61
4.4.2. Resultados de las pruebas 63

5. Conclusiones y líneas futuras **65**

5.1. Conclusiones 65

5.2. Líneas futuras 65

Índice de figuras

2.1. Diagrama de Gantt del proyecto 6

2.2. Diagrama de Gantt final del proyecto 9

3.1. Página de inicio de la web de *Tablon* 24

3.2. Página de Users de la web de *Tablon* 25

3.3. Página de Queues de la web de *Tablon* 25

3.4. Página de Leaderboards de la web de *Tablon* 26

3.5. Página de FAQ de la web de *Tablon* 27

3.6. Página de Stats de la web de *Tablon* 27

3.7. Diagrama de módulos y ficheros de la aplicación 42

3.8. Diagrama de los submódulos y clases del módulo `tablon` 43

3.9. Diagrama detallado de los submódulos "execution" 44

3.10. Diagrama detallado del submódulo "executionqueue" 45

3.11. Diagrama detallado de los submódulos de conexión 46

3.12. Diagrama detallado del submódulo "leaderboard" 47

3.13. Diagrama de secuencia de la autenticación de un cliente 49

3.14. Diagrama de secuencia del envío de una *request* por el cliente 50

3.15. Diagrama de secuencia de la gestión de nuevas *requests* en la aplicación 51

3.16. Diagrama de máquina de estados para el "Status" de una *request* 52

4.1. Diagrama de módulos y ficheros de la aplicación modificada 58

4.2. Diagrama de los submódulos y clases del módulo `tablon` modificado 59

Índice de tablas

2.1. Riesgo de mala planificación 6

2.2. Riesgo de problema con el TFG de estadística 7

2.3. Riesgo de problemas con las máquinas 7

2.4. Riesgo de indisponibilidad de los tutores 7

2.5. Riesgo de mal planteamiento de soluciones 7

2.6. Tabla de software y hardware utilizado 9

2.7. Amortización de las máquinas 10

2.8. Coste estimado 10

4.1. Lista de programas para los casos de prueba 61

4.2. Casos de prueba para el Leaderboard `lb_practica1` 62

4.3. Casos de prueba para el Leaderboard `openmplb` 63

4.4. Resultados de los casos de prueba 64

Capítulo 1

Introducción

1.1. Contexto

La *gamificación* es una metodología que consiste en el uso de elementos propios de los juegos en otros ámbitos [1], con la finalidad de hacer más atractivas tareas que de otra forma serían tediosas y fomentar la participación. En la educación el uso de la gamificación mejora los resultados obtenidos por los estudiantes [2], su conducta [3], y su motivación o interés por los conocimientos enseñados [4].

Con el objetivo de fomentar el uso de técnicas de gamificación en la enseñanza universitaria, el Grupo Trasgo [5] desarrolló una herramienta de software en 2016 llamada *Tablon* [6]. Esta herramienta creada en Python 2 implementa un evaluador online, originalmente diseñado para concursos de programación paralela, añadiendo funcionalidades de gamificación.

Tablon recibe los programas enviados por los alumnos mediante un cliente que proporciona la herramienta, compila y ejecuta dichos códigos según las opciones especificadas y muestra el resultado en una página web. Estos programas pueden ser enviados a una cola de ejecución con los argumentos elegidos por el alumno o a un *Leaderboard*. En el caso de un *Leaderboard*, el programa será ejecutado con una serie de argumentos ocultos al alumno, para puntuar su envío a partir del resultado, el tiempo de ejecución o el número de líneas de código utilizadas, por ejemplo. Cada *Leaderboard* muestra en la página web un ranking público de los programas con mejor puntuación.

La acogida de la herramienta *Tablon* por parte del alumnado fue mayormente positiva [7]. En este artículo, los autores describen la experiencia del uso de la herramienta durante un curso de la asignatura de Computación Paralela del Grado en Ingeniería Informática de la Universidad de Valladolid. Dada la recepción positiva de *Tablon* por parte de los alumnos, se ha seguido utilizando en esta asignatura y su uso se ha extendido a otras asignaturas del grado de Ingeniería Informática de la Universidad de Valladolid, como Arquitectura y Organización de Computadoras, Fundamentos de Computadoras y Diseño de Software. Esto es una buena muestra de la versatilidad de la herramienta, ya que puede utilizarse en contextos muy diversos.

Gracias al éxito de la herramienta, se han realizado diversas publicaciones de innovación docente. En [8] se combina el uso de *Tablon* con otras estrategias para fomentar la colaboración entre los alumnos. En [9] se plantea el uso de esta herramienta para la evaluación de prácticas en asignaturas de arquitectura de computadoras. En este se presenta un estudio experimental realizado sobre un curso de la asignatura de Arquitectura y Organización de Computadoras, mostrando como los alumnos que realizaron las prácticas bajo la metodología gamificada obtuvieron mejores resultados que los alumnos del grupo de control, los cuales realizaron las prácticas con la metodología tradicional.

1.2. Motivación

A lo largo de los años *Tablon* ha recibido diversas modificaciones, de cara a adaptar sus funcionalidades a los distintos concursos de programación en los que se ha utilizado. Sin embargo, estas modificaciones se han ido añadiendo a la aplicación una sobre otra aumentando la complejidad de esta innecesariamente. Además, con el paso de los años Python 2 se ha ido convirtiendo en un lenguaje de programación anticuado y desactualizado, en comparación con Python 3. Esto implica que algunas de las funcionalidades de *Tablon* se encuentran desactualizadas e incluso podrían quedarse anticuadas en su implementación en un futuro. Es por este motivo que se plantea el proyecto *Tablon3*, una actualización de la herramienta *Tablon* al lenguaje de programación Python 3.

Además, otro de los problemas relativos a *Tablon* es la falta de documentación sobre este. Se dispone de poca información sobre su funcionamiento y mucho menos sobre su diseño e implementación, lo que dificulta enormemente su modificación y actualización. Por lo que gran parte de este proyecto se centrará en la creación de una documentación del diseño de la herramienta *Tablon*.

En general, la principal motivación de este proyecto es documentar y actualizar la herramienta *Tablon* para poder seguir utilizándola y ampliándola con mayor facilidad en un futuro. Por ello este trabajo se engloba dentro del proyecto de innovación docente financiado por la Universidad de Valladolid titulado "Generalizando la integración de gamificación competitiva y colaborativa de forma ágil (III)", con identificador PID2021_065.

1.3. Objetivos

El objetivo principal de este proyecto es la actualización de la herramienta *Tablon* al lenguaje de programación Python 3 y la documentación de su diseño. Sin embargo, se deben cumplir algunos objetivos antes de poder llevar acabo esto, como la redacción de una documentación que permita entender fácilmente el funcionamiento e implementación de la herramienta *Tablon*. A continuación se describen los distintos objetivos que se quieren alcanzar y en el orden en el que se deberían cumplir:

- Analizar y comprender el funcionamiento de la herramienta *Tablon*.

- Redactar una documentación lo más completa posible sobre el diseño de las partes más relevantes de la herramienta *Tablon*.
- Realizar la conversión del código de *Tablon*, obteniendo una versión funcional de esta herramienta en Python 3.
- Añadir algunos cambios y arreglos pequeños sobre el código convertido a Python 3, sin modificar el funcionamiento general de la aplicación.
- Depurar la aplicación resultante con una batería de pruebas, resolviendo los fallos que se vayan encontrando.

1.4. Estructura del documento

Este documento se ha divido en varios capítulos que desarrollan los siguientes contenidos:

- **Capítulo 1: Introducción**, el cual describe el contenido general del trabajo, contexto, motivación y los objetivos.
- **Capítulo 2: Plan de trabajo y seguimiento**, en el que se expone la planificación inicial del proyecto y el seguimiento de las distintas fases del proyecto, junto con los cambios sobre esa planificación inicial.
- **Capítulo 3: Descripción del objeto de estudio**, donde se presenta la documentación general que se ha ido redactando a lo largo del proyecto y la explicación de algunos conceptos técnicos necesarios.
- **Capítulo 4: Conversión del código**, en el que se incluye el proceso de conversión del código de *Tablon* a Python 3, las modificaciones realizadas y la batería de pruebas sobre el código final.
- **Capítulo 5: Conclusiones y líneas futuras**, donde se recogen las conclusiones obtenidas tras la realización del proyecto y las líneas futuras de trabajo.

Capítulo 2

Plan de trabajo y seguimiento

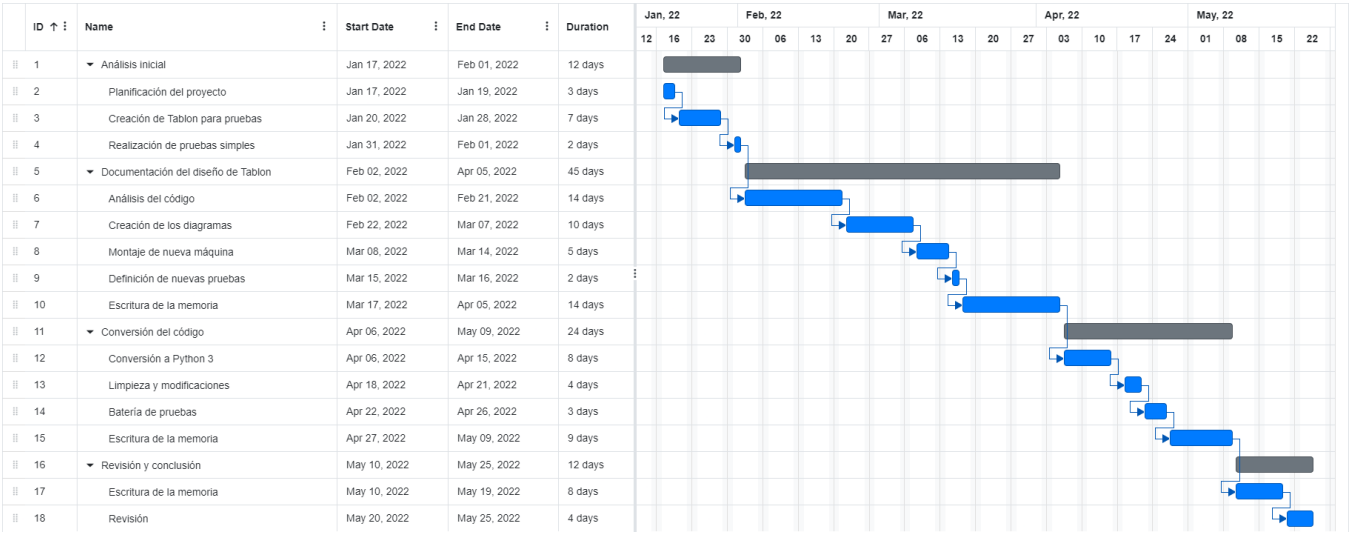
2.1. Planificación del proyecto

Si bien este proyecto no es el típico proyecto de desarrollo de software, se ha considerado realizar una planificación del mismo tratando de separarlo en fases. Para ello se ha tenido en cuenta los distintos productos resultado del proyecto, como son: una documentación del diseño de la aplicación, la conversión del código a Python 3 y la redacción de la memoria del proyecto.

Con todo esto las fases en las que se separa el proyecto son las siguientes:

- **Análisis inicial**: una primera fase para entender el funcionamiento más general de la aplicación y definir los objetivos de este proyecto.
- **Documentación del diseño de Tablon**: esta fase tiene como objetivo final la creación de una documentación lo más completa posible. Durante la misma se tratará de comprender el funcionamiento de todas las partes de *Tablon* y definir diagramas que expliquen la estructura y diseño de este.
- **Conversión del código**: incluye tanto la creación de una versión final de *Tablon* en Python 3, como la redacción de explicaciones que permitan entender los cambios realizados y como afectan estos cambios a la documentación ya escrita. Esto también incluye la realización de una batería de pruebas para la aplicación final.
- **Revisión y conclusión**: esta última fase tiene como objetivo la redacción de la memoria de trabajo y realizar una revisión final de toda ella.

El contenido más especifico de estas fases y la organización en el tiempo de las mismas se puede observar en el diagrama de Gantt representado en la Figura 2.1. Este diagrama ha sido creado con la herramienta gratuita *OnlineGantt* [10]. Para el seguimiento del proyecto se ha planteado realizar reuniones semanales todos los martes para consulta de dudas y revisión del trabajo de esa semana.



ID	Name	Start Date	End Date	Duration
1	Análisis inicial	Jan 17, 2022	Feb 01, 2022	12 days
2	Planificación del proyecto	Jan 17, 2022	Jan 19, 2022	3 days
3	Creación de Tablon para pruebas	Jan 20, 2022	Jan 28, 2022	7 days
4	Realización de pruebas simples	Jan 31, 2022	Feb 01, 2022	2 days
5	Documentación del diseño de Tablon	Feb 02, 2022	Apr 05, 2022	45 days
6	Análisis del código	Feb 02, 2022	Feb 21, 2022	14 days
7	Creación de los diagramas	Feb 22, 2022	Mar 07, 2022	10 days
8	Montaje de nueva máquina	Mar 08, 2022	Mar 14, 2022	5 days
9	Definición de nuevas pruebas	Mar 15, 2022	Mar 16, 2022	2 days
10	Escritura de la memoria	Mar 17, 2022	Apr 05, 2022	14 days
11	Conversión del código	Apr 06, 2022	May 09, 2022	24 days
12	Conversión a Python 3	Apr 06, 2022	Apr 15, 2022	8 days
13	Limpieza y modificaciones	Apr 18, 2022	Apr 21, 2022	4 days
14	Batería de pruebas	Apr 22, 2022	Apr 26, 2022	3 days
15	Escritura de la memoria	Apr 27, 2022	May 09, 2022	9 days
16	Revisión y conclusión	May 10, 2022	May 25, 2022	12 days
17	Escritura de la memoria	May 10, 2022	May 19, 2022	8 days
18	Revisión	May 20, 2022	May 25, 2022	4 days

Figura 2.1: Diagrama de Gantt del proyecto

2.2. Análisis de riesgos

A continuación se han definido una lista de posibles riesgos durante la realización del proyecto, junto con un plan de contingencia para cada uno de ellos. Cabe destacar que este proyecto se esta llevando a cabo al mismo tiempo que el trabajo de fin de carrera del grado en Estadística. De tal manera que un problema durante la realización este proyecto afectará a la planificación del otro trabajo y viceversa.

Identificador	R01 - Mala planificación del proyecto
Descripción	La duración de las tareas especificadas no se ajusta correctamente a la realidad.
Impacto	Alto
Probabilidad	Media
Plan de contingencia	Priorizar las tareas más importantes y reducir la cantidad de trabajo, replanificando el proyecto.

Tabla 2.1: Riesgo de mala planificación

Identificador	R02 - Problemas con el TFG de estadística
Descripción	Ocurre algún problema durante la realización del TFG del grado en estadística que reduce drásticamente la disponibilidad del alumno.
Impacto	Alto
Probabilidad	Alta
Plan de contingencia	Realizar una replanificación del TFG de estadística y de este proyecto reduciendo la carga de trabajo de ambos si es necesario.

Tabla 2.2: Riesgo de problema con el TFG de estadística

Identificador	R03 - Problemas con las máquinas virtuales
Descripción	Las máquinas virtuales no funcionan correctamente o el montaje de las mismas se retrasa, bloqueando las tareas que dependan de estas.
Impacto	Medio
Probabilidad	Baja
Plan de contingencia	Replanificación de las tareas para poder avanzar en aquellas que no requieran del uso de máquinas virtuales.

Tabla 2.3: Riesgo de problemas con las máquinas

Identificador	R04 - Indisponibilidad de los tutores
Descripción	Los tutores del proyecto no se encuentran disponibles por motivos de enfermedad, otras tareas o motivos personales, pudiendo llegar a bloquear el avance del proyecto.
Impacto	Alto
Probabilidad	Alta
Plan de contingencia	Replanificación de las tareas para poder avanzar en aquellas que no requieran consultar a los tutores.

Tabla 2.4: Riesgo de indisponibilidad de los tutores

Identificador	R05 - Mal planteamiento de soluciones
Descripción	Las soluciones planteadas inicialmente a los problemas detectados en la conversión del código no son suficientes, pudiendo requerir de cambios mucho más grandes que alargarían el tiempo de la tarea
Impacto	Muy alto
Probabilidad	Media
Plan de contingencia	Tratar de encontrar las soluciones más simples cuya implementación se adapte lo mejor posible a los tiempos estimados para la tarea.

Tabla 2.5: Riesgo de mal planteamiento de soluciones

2.3. Desviación de la planificación inicial

Sobre la planificación inicial descrita se han tenido que realizar diversas modificaciones, debido a la materialización de algunos riesgos. La mayoría de estos riesgos se encuentran entre los ya listados. Sin embargo, con la replanificación realizada como medida de contención para algunos riesgos, se han materializado algunos riesgos inesperados.

La primera fase de análisis inicial se llevó a cabo sin problemas, se dispuso de una máquina virtual a tiempo para poder realizar algunas pruebas y comprender el funcionamiento general de la aplicación. Sin embargo, durante la segunda fase se tuvo que alargar la duración del análisis del código, debido a los siguientes problemas:

- **Mala estimación de la duración de la tarea** (R01): el código era más complejo de lo esperado, con lo que se requirió de más tiempo para su comprensión y posterior explicación en la memoria.

- **Falta de tiempo para consultar a los tutores** (R04): debido al comienzo de curso y otro motivos, el tiempo de consulta a los tutores fue mucho menor de lo requerido para el correcto desarrollo del proyecto.

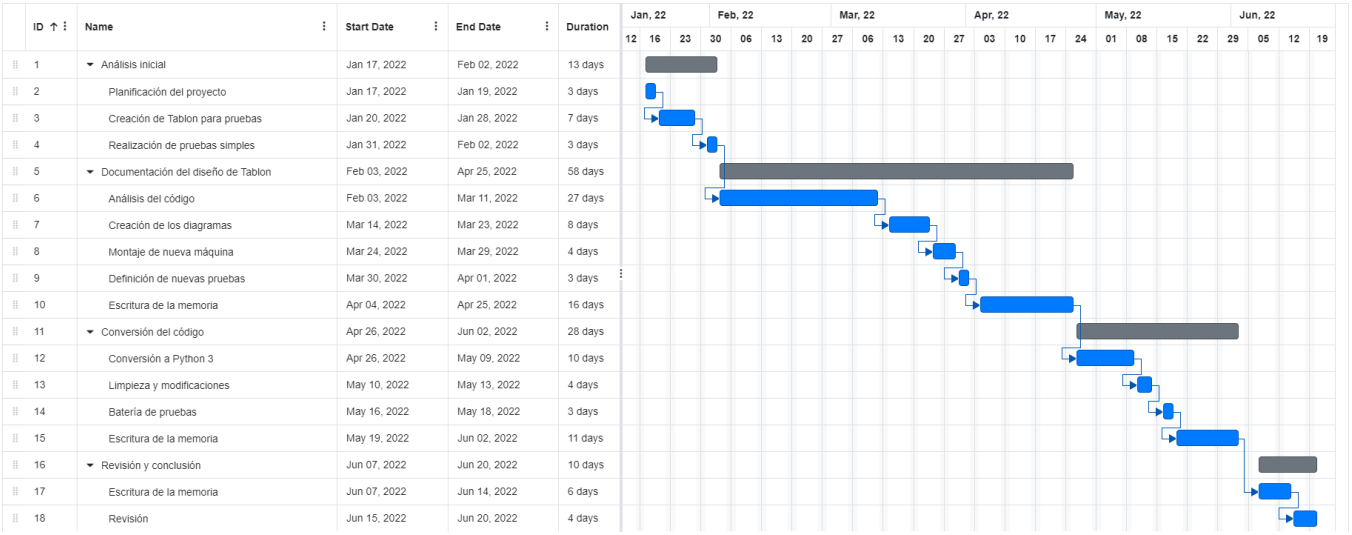
- **Falta de tiempo debido a una asignatura** (Riesgo no contemplado): se valoró erróneamente la carga de trabajo de la asignatura que se cursó durante ese cuatrimestre, reduciendo el tiempo para trabajar en el proyecto.

Con todo esto se tomó la decisión de alargar la duración de esta tarea, e intentar adelantar las tareas del TFG de estadística, como medida preventiva.

Una vez realizada la replanificación se continuó con el proyecto de forma normal, ajustando las fechas de algunas tareas cuya duración se había estimado erróneamente. Sin embargo, estos ajustes no fueron demasiado notorios. Durante la fase de Conversión del código se materializó el riesgo R03 al tener problemas instalando las bibliotecas necesarias de Python. Este problema se solucionó con suficiente rapidez como para no requerir de una replanificación demasiado grande.

Finalmente, debido al sucesivo retraso de las tareas, se materializó un riesgo no contemplado al coincidir las fechas de algunas tareas con el examen de convocatoria ordinaria de la asignatura que se cursó durante ese cuatrimestre. Debido a este riesgo se tuvo que retrasar las últimas tareas para tener tiempo para estudiar el examen.

Se puede observar el Diagrama de Gantt de la última replanificación realizada durante el proyecto en la Figura 2.2.



ID	Name	Start Date	End Date	Duration
1	Análisis inicial	Jan 17, 2022	Feb 02, 2022	13 days
2	Planificación del proyecto	Jan 17, 2022	Jan 19, 2022	3 days
3	Creación de Tablon para pruebas	Jan 20, 2022	Jan 28, 2022	7 days
4	Realización de pruebas simples	Jan 31, 2022	Feb 02, 2022	3 days
5	Documentación del diseño de Tablon	Feb 03, 2022	Apr 25, 2022	58 days
6	Análisis del código	Feb 03, 2022	Mar 11, 2022	27 days
7	Creación de los diagramas	Mar 14, 2022	Mar 23, 2022	8 days
8	Montaje de nueva máquina	Mar 24, 2022	Mar 29, 2022	4 days
9	Definición de nuevas pruebas	Mar 30, 2022	Apr 01, 2022	3 days
10	Escritura de la memoria	Apr 04, 2022	Apr 25, 2022	16 days
11	Conversión del código	Apr 26, 2022	Jun 02, 2022	28 days
12	Conversión a Python 3	Apr 26, 2022	May 09, 2022	10 days
13	Limpieza y modificaciones	May 10, 2022	May 13, 2022	4 days
14	Batería de pruebas	May 16, 2022	May 18, 2022	3 days
15	Escritura de la memoria	May 19, 2022	Jun 02, 2022	11 days
16	Revisión y conclusión	Jun 07, 2022	Jun 20, 2022	10 days
17	Escritura de la memoria	Jun 07, 2022	Jun 14, 2022	6 days
18	Revisión	Jun 15, 2022	Jun 20, 2022	4 days

Figura 2.2: Diagrama de Gantt final del proyecto

2.4. Software y hardware utilizado

A continuación se especifica el software y hardware utilizados durante la realización del proyecto:

Nombre	Descripción
Astah	Herramienta de modelado UML.
Visual Studio Code	Editor de texto y código fuente.
Overleaf	Editor online de ficheros en LaTeX.
OnlineGantt	Herramienta online de creación de diagramas de Gantt.
Ordenador portátil	Ordenador portátil de gamma media.
Ordenador de sobremesa	Ordenador de sobremesa de gamma media-alta.
VM de la UVa	Máquina virtual proporcionada por la UVa.
VM de Tablon de AOC	Máquina virtual que aloja el servidor *Tablon* de la asignatura de Arquitectura y Organización de Computadoras.
VM del Grupo Trasgo	Máquina virtual del Grupo Trasgo.
Trasgo Heterogenous Cluster	Cluster heterogéneo del Grupo Trasgo [11].

Tabla 2.6: Tabla de software y hardware utilizado

2.5. Presupuesto y costes

A continuación se muestran un análisis del presupuesto estimado de todo el proyecto. Este se ha calculado teniendo en cuenta el número de horas de trabajo y el valor de las licencias y la

amortización del hardware utilizado.

En primer lugar se ha calculado el coste por hora del uso de los dos ordenadores descritos en la Sección 2.4 teniendo en cuenta: el precio de compra inicial, el número de años de vida útil y un uso del mismo de 8 horas diarias, con 250 días laborales anuales. Los datos de ambos ordenadores se muestran en la siguiente tabla:

Máquina	Precio	Años de vida útil	Coste/hora
Ordenador de sobremesa	1.150€	2	0,29€
Ordenador portátil	750€	4	0,10€

Tabla 2.7: Amortización de las máquinas

A continuación se muestra la tabla de costes finales del proyecto:

Concepto	Cantidad	Coste por unidad	Coste total
Licencia Astah	1	88,00€	88,00€
Trabajo en horas del alumno	380	12,50€	4.750,00€
Ordenador de sobremesa (horas de uso)	300	0,29€	87,00€
Ordenador portátil (horas de uso)	80	0,10€	8,00€
			4.933,00€

Tabla 2.8: Coste estimado

Las horas de trabajo se han calculado en función de los días totales trabajados según el diagrama de Gantt, teniendo en cuenta que se trabajó una media de 3.5 horas al día, equivalente a una media jornada laboral, debido al desarrollo en paralelo de los dos TFGs necesarios para completar la titulación de doble Grado en Informática y Estadística. El presupuesto inicial partía de una estimación de 320 horas aproximadamente de trabajo, sin embargo, debido a la materialización de diversos riesgos, explicada en la Sección 2.3, se aumentó la duración del proyecto llegando a las 380 horas de trabajo totales. Se estima el sueldo anual bruto de un ingeniero informático junior entorno a los 25.000€, con lo que se obtiene un sueldo de 12,50€ la hora, teniendo en cuenta una jornada laboral completa de 8 horas diarias con 250 días laborables al año.

Capítulo 3

Descripción del objeto de estudio

Antes de poder convertir el código a Python 3, es necesario tener información sobre el funcionamiento de *Tablon*, el objeto de este estudio, y sobre tecnologías similares. Sin embargo, durante su creación no se llevaron a cabo las fases típicas de desarrollo de software, por lo que no se dispone de una documentación adecuada acerca del diseño, la arquitectura lógica y la implementación de *Tablon*. Únicamente se dispone del código fuente y de algunas explicaciones sobre cómo poner en marcha la aplicación.

Es por esto que, en este capítulo, se plasmarán los pasos que se han seguido para comprender el funcionamiento de este programa: desde cómo poner en marcha el servidor, hasta la descripción de la estructura mediante diagramas de diseño. Esta parte del trabajo es muy importante, no solo para la conversión del código a Python 3 que se ha realizado durante el desarrollo de este proyecto, sino también de cara a modificaciones que reciba la aplicación en un futuro.

3.1. Conocimientos técnicos previos

Para poder comprender el código de la aplicación y la estructura la misma, es importante revisar primero algunos conceptos y funcionalidades que usa la aplicación. En esta sección se tratan los conceptos más relevantes y menos conocidos, empezando por las diferencias entre Python 2 y Python 3, los decoradores de Python y algunas bibliotecas de Python.

3.1.1. De Python 2 a Python 3

El cambio de una aplicación en Python 2 a Python 3 no es un cambio tan grande como el requerido para pasar de un lenguaje de programación a otro distinto, sin embargo, algunas de las diferencias entre estas dos versiones no son triviales. Por ello es importante conocer estas diferencias y saber detectarlas en el código. A continuación, se comentan algunas de las diferencias más importantes entre estas dos versiones:

Print

Es de conocimiento general que la sentencia `print` en Python se utiliza para imprimir una cadena de caracteres por salida estándar.

En Python 2 esta sentencia es lo que se denomina un *statement*: una instrucción de código que el intérprete puede ejecutar (como por ejemplo, la asignación de una variable). Así pues, recibe uno o más argumentos y los imprime separados por espacios. Sin embargo, en Python 3, `print` es una función que realiza el mismo cometido, cambiando la sintaxis de una versión a otra. En Python 3, al ser una función, los parámetros deben ir separados por comas y dentro de paréntesis tras la palabra `print`. En Python 2, al ser un *statement*, los parámetros no van dentro de un paréntesis, ya que, en ese caso, el intérprete entendería que está recibiendo una tupla como parámetro en lugar de recibir múltiples parámetros. Por ejemplo, la siguiente línea:

```
print("Hola",1)
```

en Python 2 devolvería `('Hola',1)`, mientras que en Python 3 produciría `Hola 1`.

El mayor problema de esta diferencia no es que la salida se imprima diferente, sino que el programa puede fallar por una mala declaración de un `print` que se trasladó directamente de un programa en Python 2 a un programa de Python 3 sin una revisión previa. Una forma de solucionar esto es añadir paréntesis en todas las declaraciones de `print`.

División de dos enteros

En Python, el operador `/` devuelve la división entre dos números; si alguno de los dos números es decimal, es decir, de tipo `float`, el resultado será también de dicho tipo, independientemente de que dicho valor sea entero. Sin embargo, cuando los dos números son de tipo `int`, el tipo del resultado es distinto en Python 2 que el que proporciona Python 3.

En Python 2, cuando ambos números son de tipo `int`, realiza la división entera produciendo un número del mismo tipo. Sin embargo, en Python 3, esta división devuelve siempre un número de tipo `float`, independientemente de que el resultado sea entero o no. A primera vista esto puede parecer una diferencia trivial pero, si no se revisan estas operaciones al pasar un programa de Python 2 a Python 3, se pueden detectar errores muy graves ya que el resultado de esa división puede que ya no sea el mismo.

Una forma sencilla de solucionar este problema es usar el operador `//` que produce el mismo resultado que el operador `/` en Python 2.

Otras diferencias menores

Además de las mencionadas anteriormente, hay otras diferencias que merece la pena destacar:

- **Excepciones**: en Python 3 se deben mandar con el nombre de la excepción seguido de una cadena de caracteres entre paréntesis con el mensaje del error.
- **Variables globales**: en Python 2 las variables globales podían cambiar de valor dentro de un bucle `for`, mientras que en Python 3 esto ya no ocurre.
- **Bibliotecas**: la ruta e implementación de algunas bibliotecas básicas de Python han cambiado de una versión a otra, por lo que se debe revisar la forma en la que se importan esas bibliotecas.

Más información sobre estas diferencias y otras puede encontrarse en [12].

3.1.2. Decorators

En la aplicación *Tablon* se utiliza en múltiples ocasiones el patrón de diseño *Decorator* [13] para implementar una funcionalidad específica. El caso más destacable se puede encontrar en el fichero `tablondb.py`, el cual se describe en profundidad más adelante.

Este patrón tiene como objetivo añadir funcionalidad a un objeto de forma dinámica, de tal manera que no sea necesario usar herencia para ello. A menudo se implementa envolviendo al objeto inicial en una clase decoradora que añadirá funcionalidad. La clase decoradora reenvía las llamadas a la clase principal, pudiendo añadir acciones antes o después del reenvío.

En Python, esta clase decoradora se implementa como una función que recibe como argumento a otra función, siendo esta última la que se quiere decorar. La función decoradora modifica la función que recibe como argumento y la devuelve ya cambiada, es decir, que el decorador devuelve la función decorada y para obtener el resultado de esta modificación, se debe llamar a la función que devuelve el decorador. A continuación se muestra un ejemplo sencillo:

```
def decorador(f):
    print("Generando la función decorada")
    def decoracion():
        print("Invocando a imprime...")
        f()
    return decoracion

def imprime():
    print("Hola mundo")

print("---Función antes de ser decorada---")
imprime()
print("-------------------------------------")
imprime = decorador(imprime)
print("-------------------------------------")
print("---Función después de ser decorada---")
imprime()
```

Al ejecutar el programa se obtiene la siguiente salida:

```
---Función antes de ser decorada---
Hola mundo
------------------------------------
Generando la función decorada
------------------------------------
---Función después de ser decorada---
Invocando a imprime...
Hola mundo
```

En primer lugar, se ha invocado a la función `imprime` sin modificar para mostrar su salida original. En segundo lugar, se ha usado el decorador para modificar `imprime` y guardar esta modificación con el mismo nombre que tenía la función antes de ser modificada (es decir, `imprime`). Por último, se ha vuelto a llamar a `imprime`, solo que esta vez ya no es la función original y mostrará la salida de la función decorada.

Si bien esta forma de decorar es un poco complicada, hay una forma más rápida de obtener la función decorada sin necesidad de sobrescribir la función con la decorada: simplemente hay que escribir `@(nombre del decorador)` encima de la función que se quiere decorar. De esta forma, cada vez que se llame a esta función, devolverá el resultado de la función ya modificada. Sin embargo, con esta sintaxis ya no se puede llamar a la función antes de ser decorada. El ejemplo anterior usando esta sintaxis se vería así:

```
def decorador(f):
    print("Generando la función decorada")
    def decoracion():
        print("Invocando a imprime...")
        f()
    return decoracion

@decorador
def imprime():
    print("Hola mundo")

print("---Función después de ser decorada---")
imprime()
```

y el resultado de ejecutar este programa sería:

```
Generando la función decorada
---Función después de ser decorada---
Invocando a imprime...
Hola mundo
```

Como se observa, esta sintaxis realiza las mismas acciones que la anterior, por eso aparece el mensaje "Generando la función decorada".

En el caso de que la función a decorar necesite recibir parámetros, se debe indicar a la función dentro del decorador (`decoracion()` en el ejemplo anterior), escribiendo `*args, **kwargs` como los parámetros que recibe esta y, posteriormente, pasarle esos parámetros en la llamada a la función a decorar.

Aunque se ha explicado que el decorador es una función, eso no impide que se pueda usar una clase como decorador. Para hacer esto se utiliza la función `__call__` de la clase. Esta función se ejecuta cuando se llama a una instancia de la clase como si fuera una función. Para comprender mejor este concepto se proporciona un código de ejemplo, en el que se define una clase con su función `__call__` y se muestra como se ejecuta esta:

```
class decorador(object):
    def __init__(self):
                pass
        def __call__(self):
        print("Ejecutando __call__")

decor = decorador()
print("Llamando a la función __call__")
decor()
```

Al ejecutar este código se obtiene la siguiente salida:

```
Llamando a la función __call__
Ejecutando __call__
```

Para utilizar una clase como decorador hay que utilizar una de sus funciones como función decoradora. Habitualmente se utiliza la función `__call__`, ya explicada, como función decoradora. Para ello, en lugar de escribir `@(nombre del decorador)` encima de la función que se quiere decorar, se escribe `@(nombre de la clase decoradora)()`, para que entienda que el decorador es la función `__call__` de la clase. Los paréntesis después del nombre de la clase son muy importantes, porque si no, entenderá que la función decoradora es la función `__init__`. Se muestra a continuación un ejemplo similar al anterior y con el mismo resultado:

```
class decorador(object):
        def __init__(self):
                pass
        def __call__(self, f):
                print("Generando la función decorada")
                def decoracion():
                        print("Invocando a imprime")
                        f()
                return decoracion

@decorador()
def imprime():
    print("Hola mundo")

print("---Función después de ser decorada---")
imprime()
```

Aquí se han planteado ejemplos muy sencillos, pero esta funcionalidad de Python es muy flexible y se puede usar de formas mucho más complicadas. Sin embargo, en todos los casos, el decorador que se especifique después del símbolo `@` deber ser un objeto invocable de Python. Para más ejemplos y explicaciones más detalladas véase [14].

3.2. Estudio de la aplicación original

Para comprender mejor el funcionamiento de *Tablon* se dispone de una máquina virtual alojada en el servidor que usa el Grupo de Investigación Trasgo [5], y que también se usa para las prácticas de la asignatura Computación Paralela, del grado en Ingeniería Informática de la Universidad de Valladolid. En esta máquina se ha incorporado el código fuente de la aplicación, que se puede encontrar en un repositorio de *GitLab* del grupo Trasgo [15].

En este repositorio se ha ido actualizando el código de *Tablon* con los cambios realizados a lo largo del proyecto. Para mayor comodidad se han creado *Tags* para las distintas versiones de *Tablon*, con los cuales se puede acceder al código final de cada versión.

Antes de poner en marcha el servidor, hay que especificar algunas configuraciones y añadir algunos detalles en el código.

3.2.1. Puesta en marcha del servidor Tablon

A continuación se describen los pasos a seguir y las configuraciones necesarias para poner en marcha el servidor *Tablon* desde cero.

Base de datos en MySql

Tablon depende del gestor de base de datos *MySql* para guardar los datos de los programas de los alumnos y la puntuación obtenida por estos. Por lo tanto, el primer paso lógico es crear un usuario con contraseña en MySql. Una vez hecho esto, se deberá crear la base de datos y la estructura de tablas requerida, para lo cual se puede usar dos archivos `.sql` alojados en el directorio `sqlCreateDBStrcutre/`.

El primero, `db_createdb.sql`, crea la base de datos y da privilegios a un usuario sobre ella, por lo que se debe modificar este fichero escribiendo el usuario anteriormente creado, en lugar del que viene por defecto. El segundo fichero, `db_structure.sql`, crea toda la estructura de tablas de la base de datos automáticamente. No hay que modificar nada en este último. Para ejecutarlo desde MySql, se debe elegir primero la base de datos que se ha creado, y después ejecutar el fichero. Los comandos a ejecutar son los siguientes:

```
USE nombre-de-la-base-de-datos;
SOURCE db_structure.sql;
```

Creación de los usuarios o alumnos

Una vez generada la base de datos se necesita incorporar algunos nuevos usuarios para que puedan interactuar con *Tablon* y mandar sus programas, para lo cual se utiliza el script de Python `useradmin.py`. En el script `create_users.sh` se puede encontrar un ejemplo de cómo hacer esto, pero también se puede utilizar el flag de ejecución `--help` para obtener el manual de uso del script. Para las pruebas que se realizan se han creado 3 usuarios mediante los siguientes comandos:

```
python useradmin.py adduser u1 "u1pass"
python useradmin.py adduser u2 "u2pass"
python useradmin.py adduser u3 "u3pass"
```

Configuración de `server.py`

Teniendo creada la base de datos y algunos usuarios para poder mandar peticiones al servidor, solo falta cambiar la configuración del servidor, modificando manualmente el script `server.py`. En este fichero hay diversos parámetros que definen el servidor, algunos tan simples como el título de la web, y otros un poco más específicos como qué palabras prohibir en los programas que reciba el servidor. Por ahora, los únicos parámetros que importan para poder poner en marcha el servidor son:

- `host`: establece dónde estará alojada la página web de *Tablon*. Es importante que coincida con el que aparece en `client.py`.
- `port`: puerto para la conexión HTTP.

- `real_port`: puerto real en caso de que se redireccione la conexión.
- `bd_user`: nombre del usuario de MySql que se haya creado.
- `bd_pass`: contraseña del usuario de MySql.
- `bd_name`: nombre de la base de datos creada en los pasos anteriores.

Este fichero tiene más parámetros a configurar, pero para las pruebas que se realizarán estos parámetros serán mas que suficientes.

Colas de ejecución

Hay un último detalle que añadir al script `server.py` antes de poder iniciar el servidor: las colas de ejecución y las tablas de puntuación asociadas a ellas. Para comprender lo que son estos dos elementos hay que adentrarse en el código del módulo `tablon` alojado en el directorio `tablon/`.

Las colas de ejecución están pensadas para soportar distintos tipos de programas, con diversas tecnologías e incluso escritos en lenguajes de programación distintos. De esta manera, a cada cola se le puede especificar un número de procesos y un número máximo de hilos, dependiendo de las necesidades de la misma. Su implementación está descrita en el fichero `executionqueue.py`, en donde se pueden encontrar varias clases con relaciones de herencia:

- `BasicQueue`: es la clase padre; define los aspectos generales y comunes para cualquier cola de ejecución. Su constructor es común para todos los tipos de cola que heredan de esta, y algunas de sus funciones están vacías para que sus clases hijo las implementen según las necesidades de estas.
- `LocalSeqQueue`: es la clase hija de `BasicQueue`; define una cola local genérica e implementa las funciones de comprobación, compilación y ejecución, entre otras. Este tipo de cola sirve para la mayoría de lenguajes y tecnologías de uso habitual. Para otros programas más específicos, se definen 4 tipos de colas que heredan de esta y que se citan a continuación.
- `LocalOpenMPQueue`: es una clase hija de `LocalSeqQueue`; esta clase se encuentra obsoleta actualmente.
- `LocalMPIQueue`: es una clase hija de `LocalSeqQueue`; esta clase también se encuentra obsoleta actualmente.
- `LocalMPIOpenMPQueue`: es una clase hija de `LocalSeqQueue`; esta clase, como las dos anteriores, se encuentra obsoleta actualmente.
- `SlurmQueue`: es una clase hija de `LocalSeqQueue`; define una cola para programas que usan *MPI*, *OpenMP*, *CUDA*, o varias de estas tecnologías simultáneamente. Cambia el constructor de la clase padre añadiendo a este algunos atributos nuevos, y modifica la implementación de la función de ejecución.

En secciones posteriores se entrará más en detalle en el funcionamiento de estas clases, pero por el momento se explicará lo necesario para saber cómo crear las colas. Para ello, se crean dos colas: una de tipo `LocalSeqQueue` y otra de tipo `SlurmQueue`.

Algunos de los parámetros más importantes que se deben proporcionar al constructor de la clase `LocalSeqQueue` son:

- `name`: nombre de la cola.
- `extensiones`: lista con el formato de las extensiones de archivo que aceptará la cola. Si no se indica su valor, acepta solo programas con la extensión `.c` por defecto.
- `local_binpath`: camino relativo o absoluto al directorio donde se alojarán tanto los archivos binarios como los programas compilados de los alumnos.
- `makefile`: el nombre del *makefile* que usará la cola para compilar los programas, el cual debe encontrarse en el directorio `usercodes/src/`. Este parámetro es obligatorio.
- `threads`: número máximo de hilos que podrá utilizar un programa enviado por un alumno. Por defecto vale uno.
- `timewall`: límite de tiempo de ejecución en segundos para cada programa. Por defecto es de 60 segundos.
- `concurrentjobs`: número máximo de programas que pueden estar en ejecución al mismo tiempo. Si no se especifica su valor, este será uno.
- `interpreter`: nombre del intérprete de ejecución del programa, si es necesario, y que debe estar en el directorio `exec_scripts/`.
- `description`: breve descripción de la cola.

Además de los atributos de la clase padre ya explicados, una cola de tipo `SlurmQueue` tiene algunos atributos propios. Entre los más relevantes se encuentran:

- `slurm_queue`: el nombre de la cola en la máquina que usa *Slurm* para ejecutar los programas de los alumnos.
- `slurm_nodes`: lista de los nombres de los nodos de Slurm que se utilizarán.
- `remote_binpath`: camino relativo o absoluto al directorio remoto donde se alojarán los archivos binarios (como los programas compilados de los alumnos).

Para las pruebas que se realizarán solo se crean dos colas, ya que son los dos tipos distintos que funcionan actualmente. Estas colas, en particular, han sido reutilizadas de prácticas pasadas de Computación Paralela, y Arquitectura y Organización de Computadoras, ambas asignaturas del grado en Ingeniería Informática de la Universidad de Valladolid. Se puede encontrar su definición en el fichero `server.py` pero, por comodidad, también se muestra a continuación:

```
# Slurm OpenMP queue
q_omp = tablon.SlurmQueue('openmp',
        exclusive=True,
        extensions=['.c','.cpp'],
        forbidden_code='C',
        makefile='Makefile_openmp',
        threads=64,
        slurm_queue='lboard',
        slurm_nodes=['heracles'],
        remote_binpath="/home/tablon2/sandbox",
        local_binpath="usercodes/bin",
        timewall=50,
        concurrentjobs=1,
        description='Cola OpenMP que manda los trabajos a Heracles')

# Mars queue
q_mars = tablon.LocalSeqQueue('mars',
        isopen=True,
        extensions=['.asm'],
        forbidden_code='asm',
        local_binpath="usercodes/bin",
        makefile='Makefile_mars',
        extrafiles=['Mars4_5.jar','main-tablon.asm'],
        timewall=20,
        concurrentjobs=6,
        interpreter='mars.sh',
        description='Queue for the Mars 4.5 simulator')
```

LeaderBoards

Una vez creadas las colas, se debe asociar a cada una de ellas un objeto `LeaderBoard` para poder puntuar los programas de cada alumno y así, situarlos en una tabla de puntuaciones que será representada en el servidor web. Un *Leaderboard* ordena a los alumnos por puntuación, pero la puntuación puede ser simplemente un valor o puede utilizarse más de una métrica al mismo tiempo. Por ejemplo, puede puntuarse un programa simplemente por el número de casos que supera, especificados por el profesor; o bien por el número de casos que supera junto con el número de líneas de código que tiene. De esta manera, un Leaderboard puede mostrar múltiples métricas en la tabla de puntuaciones, e incluso diferentes formatos de visualización de la misma métrica (por ejemplo, mostrando el porcentaje de casos superados a mayores del número).

Si bien la clase `LeaderBoard` se encarga de detallar las métricas y el formato de la tabla de puntuaciones, existe otra clase relacionada con esta, llamada `LeaderBoardPhase` (que a partir de aquí se denominará "fase"), que especifica un test (o "caso", como se ha llamado en lo ejemplos anteriores) que deberá pasar el programa para ser evaluado.

Cabe destacar que una cola de ejecución puede tener uno o más Leaderboards asociados. Sin embargo, un Leaderboard está asociado únicamente con una cola de ejecución, por lo que para

crear uno de manera adecuada se debe usar la función `addLeaderboard()` de la clase `BasicQueue`. Los parámetros más relevantes que recibe esta función son:

- `name`: nombre del Leaderboard.
- `isopen`: parámetro de tipo *boolean* que indica si se pueden enviar programas para su evaluación o no. Por defecto, su valor es `True`.
- `shouldpass`: indica si el programa debe pasar todas las fases para ser puntuado.
- `description`: descripción que aparecerá encima de la tabla de puntuaciones en la web, en formato Unicode.
- `metric_capture`: lista de las cadenas de caracteres que se buscarán en la salida de los programas para dar valor a las puntuaciones (por ejemplo, `metric_capture = ['Instructions executed', 'Code lines']`).
- `metric_accumulate`: lista de booleans que registra si la métrica debe contarse de forma acumulativa a lo largo de cada fase (sobre el ejemplo anterior, tendría sentido acumular las instrucciones ejecutadas en cada prueba, pero no las líneas de código del programa, por lo que la configuración sería `metric_accumulate = [True, False]`).
- `score_formulae`: lista las fórmulas de las puntuaciones que se muestran en la tabla. Estas expresiones deben escribirse usando la cadena de caracteres `'scorei'`, junto con cualquier operación matemática, si así se requiere, donde i es un número de 0 a 9 que hace referencia a el índice de la lista `metric_capture`, y a las puntuaciones que se asignarán en las fases (volviendo al ejemplo anterior, `'score0'` es `'Instructions executed'` y `'score1'` es `'Code lines'`).
- `score_format`: lista con los formatos de la función `printf` que se quiere para cada tipo de puntuación determinada en `score_formulae`.
- `score_title`: lista de los nombres de cada tipo de puntuación que se mostrará en la tabla.
- `score_minimum`: relacionado con la posterior evaluación realizada por los profesores de la asignatura, indica los valores mínimos que deben obtenerse en los diferentes scores para que el programa pueda considerarse como aprobado. Si un determinado score no alcanza el valor mínimo, el score se mostrará en rojo en la clasificación.
- `score_classification`: lista con el criterio de ordenación de la tabla de puntuaciones. Este criterio depende completamente de las métricas especificadas en `score_formulae` seguido de `'desc'` o `'asc'` (es decir, descendiente o ascendente), y en caso de empate se tiene en cuenta el tiempo de llegada (por ejemplo, `score_classification = ['s3 desc, s0 asc, s1 asc, s2 asc']`).

Como en el caso de las colas de ejecución, el código para crear los `LeaderBoard` se ha reutilizado de prácticas pasadas y se encuentra en `server.py`, pero se muestran también a continuación:

```
lb_omp = q_omp.addLeaderboard('openmplb',
        isopen=True,
        on_error_keep_scores=True,
        shouldpass=True,
        removeblanks=False,
        description=u"Leaderboard OpenMP",
        metric_capture=['Time'],
        metric_accumulate=[True],
        metric_required=[True],
        score_formulae=['score1', 'score0'],
        score_format=['\%2d','\%.4f'],
        score_title=['Puntos', 'Tiempo'],
        score_minimum=[50, 0, 0],
        score_classification=['s0 desc', 's1 asc']
        )

lb_fc = q_mars.addLeaderboard('lb_practica1',
        isopen=True,
        classify="score_ic",
        shouldpass=False,
        removeblanks=True,
        description=u"Comprueba la función de la práctica 1 con una serie de"
            "entradas desconocidas para puntuar la función",
        metric_capture=['Instructions executed','Code lines'],
        metric_accumulate=[True, False],
        score_formulae=['score0 - (345-15)', 'score1', 'score2', 'score3',
            'score3*100/15'],
        score_format=['\%.0f','\%.0f','\%.0f','\%.0f','\%.2f'],
        score_title=[ 'Contador Instrucciones','Líneas Código', 'Excepc.',
            'Casos','\% superado'],
        score_minimum=[0,0,0,0,0],
        score_classification=['s3 desc, s0 asc, s1 asc, s2 asc']
        )
```

Con el Leaderboard creado se definen los `LeaderBoardPhases` que deben superar los programas para ser puntuados y, para ello, se utiliza la función de la clase `LeaderBoard addPhase()`. Esta función necesita dos parámetros: los argumentos, pasados como un diccionario de Python (por ejemplo, {`'arguments' : '0#1#10#2'`}); y *subphases* o casos, que representan las posibles salidas del programa para esos argumentos. En el caso de las fases para colas de Slurm se deberá especificar, junto con los argumentos, el número de procesos y el número de hilos para ese caso. Los subphases se escriben como listas de diccionarios de Python con los siguientes elementos relevantes en cada diccionario:

- `results`: cadena de caracteres con la salida esperada para ese caso.
- `score[i]` (siendo `i` un número de 0 a 9): valor de la puntuación `i` que se le da al programa si su salida coincide con el valor en `results`. Este parámetro hace referencia a las mismas

puntuaciones que en el parámetro `score_formulae` del Leaderboard (por ejemplo, si definimos `'score2' : 1`, y tenemos `score_formulae = ['score0', 'score1', 'score2', 'score2*100/15']`, cuando pase con éxito este caso se le sumará 1 a `score2`).

- `exception`: boolean que indica si uno de los resultados posibles de ese caso es que ocurra una excepción.

Se muestra a continuación algunas de las fases creadas para ambos `LeaderBoard`. El resto de las fases se pueden encontrar en el fichero `server.py`:

```
lb_omp.addPhase({'nprocs':1,'threads':48,
    'arguments':'2048 2048 -10 10 -11 7 1 48732 3221 12299'},
        [{'results':'19, 20501614, 3833485, 4192400', 'score1':20 } ] )
lb_omp.addPhase({'nprocs':1,'threads':60,
    'arguments':'20 12 -6.2 6.2 -6.1 6.0 3 57834 925 1028'},
        [{'results':'5, 223253, 397, 235', 'score1':5 } ] )
lb_omp.addPhase({'nprocs':1,'threads':60,
    'arguments':'20 12 -6.2 6.2 -6.1 6.0 3 57834 925 1028'},
        [{'results':'5, 223253, 397, 235', 'score1':5 } ] )
...

lb_fc.addPhase( {'arguments':'0#1#10#0.1'}, [
        {'results':'Runtime exception', 'score2':1, 'exception':True },
        {'results':'6.0273438', 'score3':1, }
        ])
lb_fc.addPhase( {'arguments':'1#-4#-1#0.1'}, [
        {'results':'Runtime exception', 'score2':1, 'exception':True },
        {'results':'-3.015625', 'score3':1, }
        ])
lb_fc.addPhase( {'arguments':'2#-3#-1.1#0.1'}, [
        {'results':'Runtime exception', 'score2':1, 'exception':True },
        {'results':'-1.4859376', 'score3':1, }
        ])
...
```

Finalmente, con todo preparado, se puede iniciar el servidor para realizar nuestras pruebas, que en este caso estará alojado en `http://frontendv.infor.uva.es/tablon2/`. Para ello, se ejecuta el fichero `run.sh`. Sin embargo, es importante destacar que si el servidor ya estaba ejecutándose con anterioridad, primero hay que pararlo con el fichero `stop.sh`.

3.2.2. La interfaz web

A continuación vamos a navegar por las distintas subpáginas de la visualización web, observando cómo los pasos realizados durante la puesta en marcha del servidor afectan a la información

que se va mostrando. Si se han realizado las etapas anteriores correctamente, al entrar en la página de *Tablon* se encuentra la página de inicio, que se muestra en la Figura 3.1.

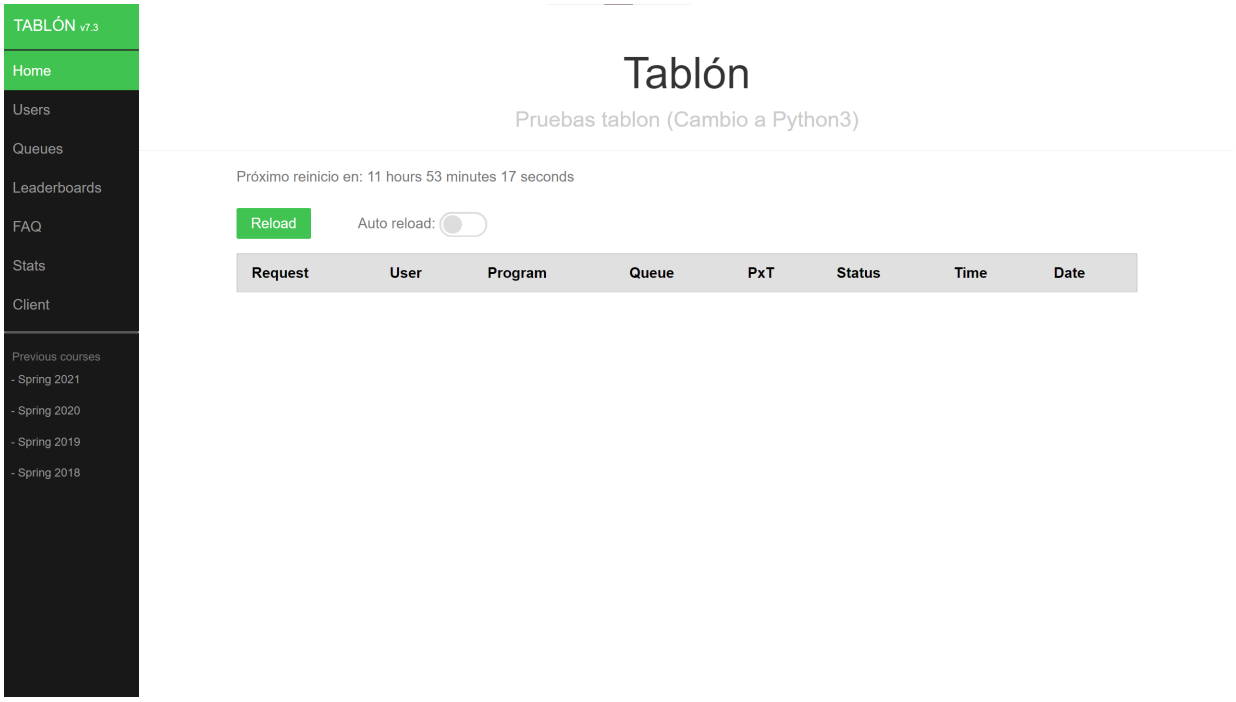


Figura 3.1: Página de inicio de la web de *Tablon*

En esta página se muestra una primera tabla resumen de las peticiones que se hayan ido mandando al servidor, con la información relevante sobre ellas. Dado que en esta prueba aún no se ha mandado ningún programa para ser puntuado, la tabla se encuentra vacía. En el borde izquierdo de la página hay un menú para acceder a los distintos apartados informativos de *Tablon*. Todos ellos se generan mediante la biblioteca Jinja2 [16] de Python , la cual usa plantillas html e información de la base de datos creada en MySql. Estas plantillas pueden editarse, hasta cierto punto, para cambiar algo de la información que presentan o también para cambiar el formato y visualización de la página. Sin embargo, a veces no es tan sencillo realizar cambios en la plantilla dado que hay muchas partes que dependen del código en Python.

Se puede acceder a más información sobre las peticiones realizadas seleccionándolas en la tabla principal. La página que se abre muestra más detalles sobre la petición, especialmente si se envió directamente a una cola de ejecución, ya que mostrará la salida de la ejecución o el tipo de error que haya ocurrido.

Si vamos ahora a la opción de *Users*, como se observa en la Figura 3.2, aparece una tabla con la información de las peticiones realizadas por cada usuario que se ha creado. Esta información se extrae de la base de datos, a la cual se le han añadido los usuarios durante el lanzamiento del servidor.

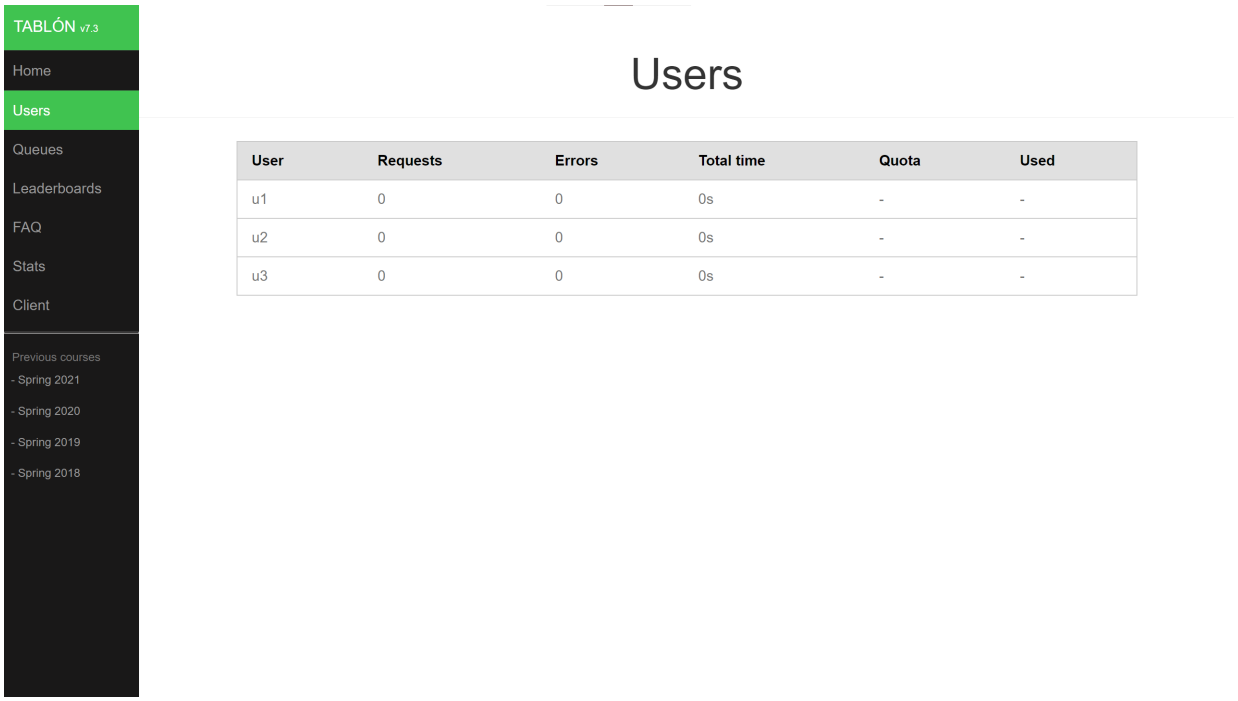


Figura 3.2: Página de Users de la web de *Tablon*

La opción de *Queues*, como se aprecia en la Figura 3.3, lista las colas que se habían creado junto con información sobre qué lenguaje soporta cada una, el nombre de los Leaderboards asociados a ella y otros detalles secundarios. La pequeña descripción que aparece de cada cola coincide con el atributo `description` que se rellenó al crear cada una de ellas.

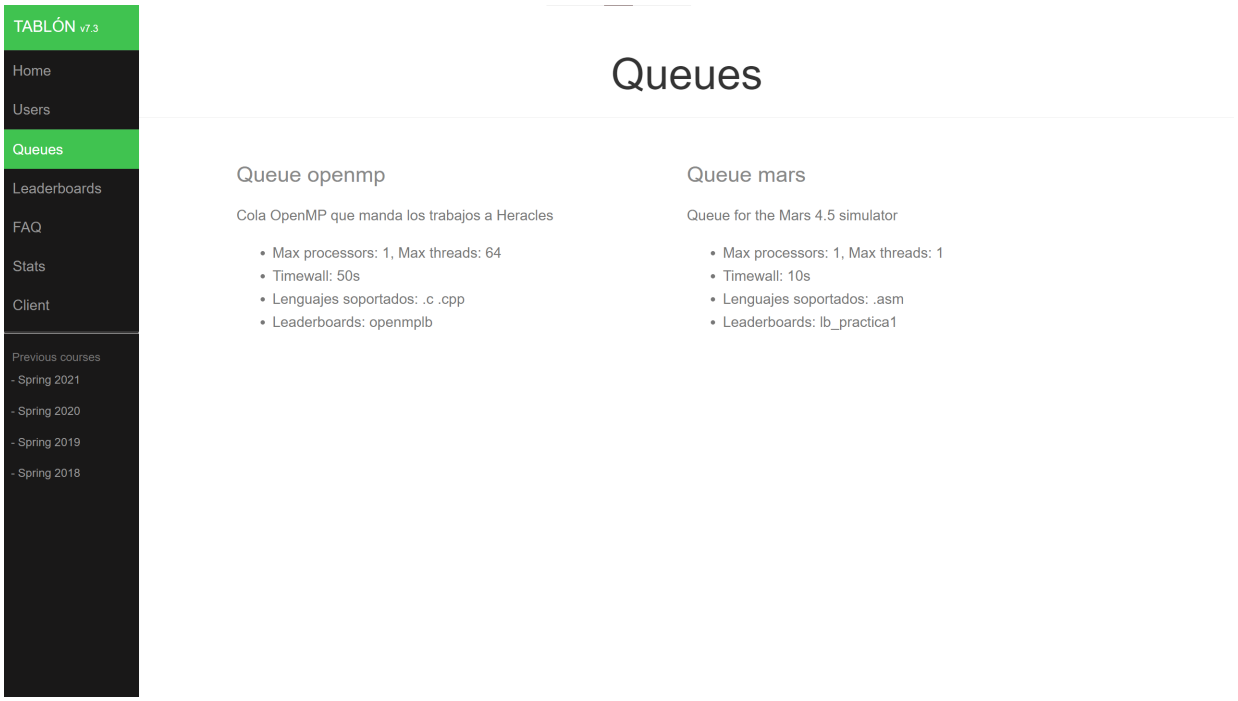


Figura 3.3: Página de Queues de la web de *Tablon*

En la pestaña de *Leaderboards* (ver Figura 3.4) se encuentra la información más relevante de esta aplicación: las tablas de puntuaciones de cada Leaderboard creado. En estas tablas se observa que las columnas indican las distintas medidas de puntuación que se han añadido en el atributo `score_formulae` con los títulos especificados. Cada tabla viene acompañada de la descripción que se especificó para el Leaderboard y una línea generada automáticamente que explica cómo enviar programas a este. Si algún Leaderboard ya no está disponible para enviar y evaluar programas como consecuencia del valor del atributo `isopen`, aparecerá [**CLOSED**] en color rojo.

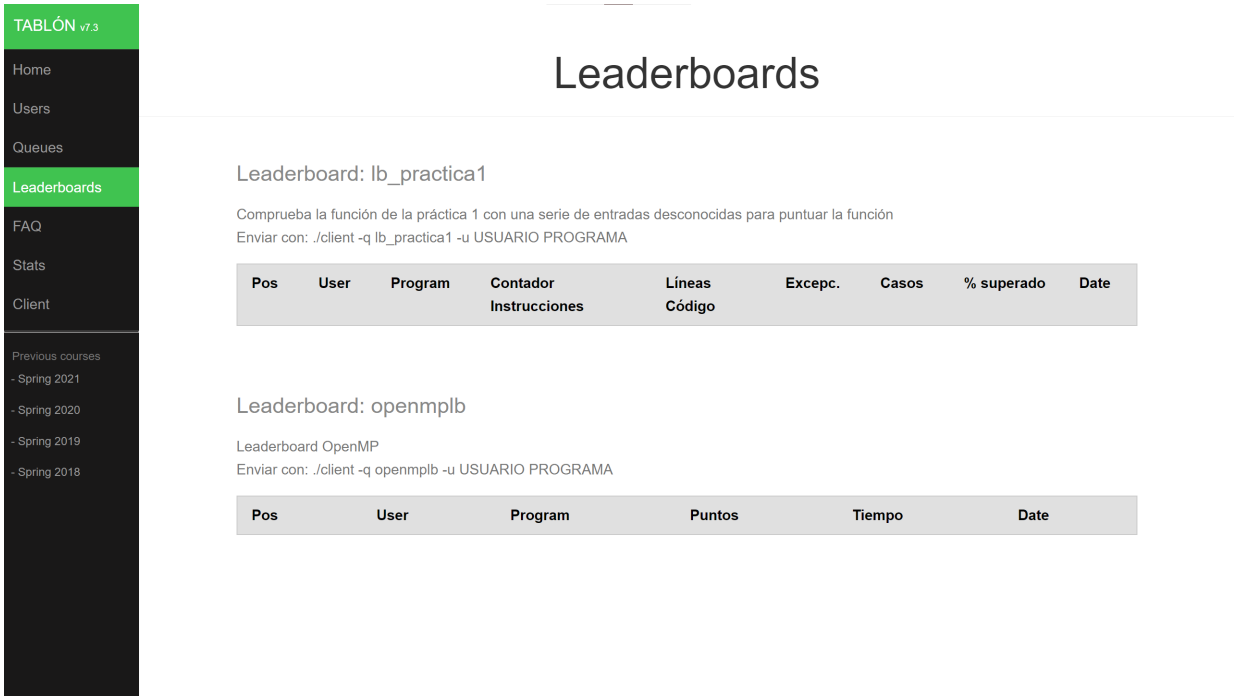


Figura 3.4: Página de Leaderboards de la web de *Tablon*

En la Figura 3.5, se observa cómo la opción *FAQ* presenta un listado de preguntas y respuestas frecuentes para aclarar a los alumnos el funcionamiento de *Tablon*. La plantilla html de esta página apenas depende del código en Python, por lo que es bastante cómoda de editar y ajustar a los contenidos del concurso o práctica que se esté realizando.

Figura 3.5: Página de FAQ de la web de *Tablon*

A continuación aparece la opción *Stats*(ver Figura 3.6), donde se muestra una página con enlaces a gráficos y estadísticas sobre las peticiones de los usuarios, y también sobre los distintos Leaderboards. Estos gráficos se generan automáticamente a partir de los datos guardados en la base de datos.

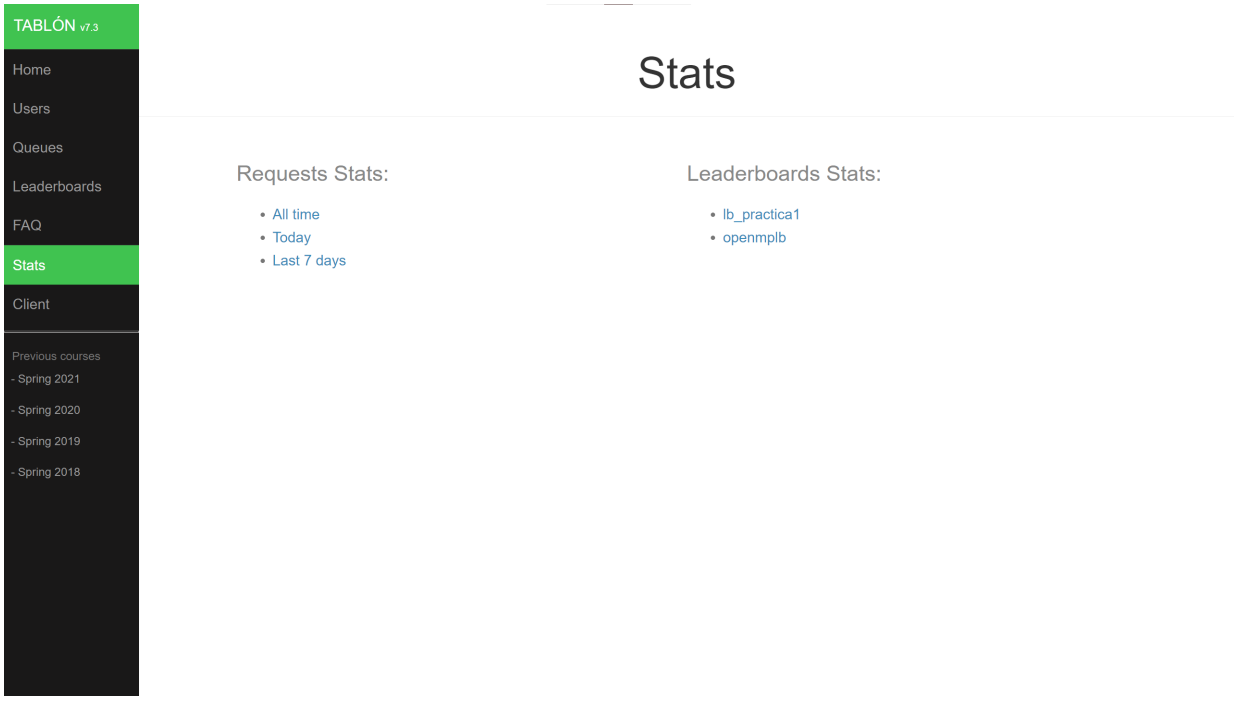


Figura 3.6: Página de Stats de la web de *Tablon*

Por último, la opción de *Client* descarga una versión comprimida del fichero `client.py`, a la cual se le añade todo el contenido del fichero `_pysrp.py` situado en el directorio `tablon/` junto con algunas modificaciones sobre el host y el puerto que utiliza. Esta versión del cliente se genera en ese mismo momento con el código de la función `client_page()`, del fichero `webpages.py` del directorio `tablon/`, y resulta imprescindible para enviar programas al servidor.

3.2.3. Envío de programas desde el cliente

Por último, como paso final para probar la aplicación, se llevará a cabo el envío de programas a los distintos Leaderboards que se han creado y así comprobar que el servidor responde correctamente. En la pestaña de FAQ de la página web se pueden encontrar explicaciones sobre el funcionamiento del servidor, incluyendo la información necesaria para mandar programas como cliente.

En este apartado se destaca el funcionamiento de los Leaderboards. Sin embargo, es posible enviar un programa a las colas de ejecución directamente para comprobar si opera correctamente, o si por el contrario tiene algún fallo. Esto sirve a los alumnos para probar sus programas con los parámetros que ellos elijan y comprobar, en la propia página, la salida o el tipo de error que ha ocurrido durante su ejecución o compilación. El comando y los requisitos son similares a los necesarios para enviar el programa a un Leaderboard.

En primer lugar, es importante destacar que como esta versión de *Tablon* está programada en Python 2, es necesario tener instalada una versión de Python que esté entre las versiones 2.7.10 y la 2.7.17. En las distribuciones más comunes de Linux suele estar instalado por defecto, pero si se está trabajando en Windows, muy probablemente habrá que instalar una versión de Python 2.

En el caso de Windows es probable que, cuando se instale Python 2, no se añada la ruta de este a las variables de entorno del sistema. Para hacerlo manualmente, primero hay que acceder a la configuración de Windows y buscar "Editar las variables de entorno del sistema". A continuación, seleccionamos la única opción que saldrá y se abrirá una ventana en la que seleccionamos "Variables de entorno". Por último, en esta ventana elegimos de entre las "Variable de usuario" la que se llama "Path", y la editamos añadiendo una nueva variable en donde escribiremos la ruta absoluta hasta la carpeta donde se ha instalado Python 2. Es importante no cerrar manualmente las ventanas, y elegir la opción de "Aceptar" en todas ellas para guardar los cambios.

Posteriormente, se debe descargar el ejecutable `client` de la página web (idealmente en el mismo directorio o carpeta donde están los programas a enviar) y ejecutar desde el terminal de comandos lo siguiente:

```
cd C:/ruta-hasta-la-carpeta-donde-esta-el-cliente-en-tu-maquina
python2 ./client -u USUARIO [-x PASSWORD] -q LEADERBOARD/QUEUE PROGRAMA [--ARGS]
```

En Windows, previamente, se debe abrir una terminal de comandos `cmd` y, si no se ha añadido la ruta de Python 2 a las variables de path del sistema, hay que escribir primero en la línea de comandos:

```
PATH=ruta-hasta-Python2;\%PATH\%
```

teniendo que repetir este paso cada vez que se quiera mandar un programa.

En ocasiones, si no se tiene instalado también Python 3, puede que el comando de ejecución del archivo `client` empiece con `py` en lugar de con `python2`.

El nombre de usuario y contraseña son los especificados cuando los usuarios fueron creados (tal y como se describió en la Sección 3.2.1). El nombre del Leaderboard o del de la cola de ejecución puede consultarse en la pestaña de Leaderboards y Queues de la página web, respectivamente. Si el usuario no incluye la contraseña en el comando (flag `-x`), se le pedirá después. Si el programa se envía a una cola de ejecución directamente, tras los dos guiones (`--`) se deben especificar los argumentos que se desean utilizar para su ejecución. Sin embargo, en el caso de envío a Leaderboard, los argumentos se escribirán en la ejecución de cada fase de evaluación.

A continuación se muestra el proceso de mandar programas a los dos Leaderboards que se están utilizando en esta prueba: `lb_practica1` y `openmplb`.

Pruebas para `lb_practica1`

Para esta prueba se dispone del código `prac1.asm`, el cual se ha programado para que supere todos los casos, es decir, todos los `LeaderBoardPhase`, tal y como se especifica en la Sección 3.2.1. Para enviar al servidor el programa como usuario "u1", se usa el siguiente comando desde el terminal:

```
python2 ./client -u u1 -x u1pass -q lb_practica1 prac1.asm
```

Si todo ha funcionado correctamente, debería tardar un poco en conectarse y, finalmente, enviar la petición informando del proceso por línea de comandos como se indica a continuación.

```
Conecting ...
Connected to u1@frontendv.infor.uva.es:80
OK
Authenticating user...
Successful authentication
Sending request
Request sent successfully
Request id 1
http://frontendv.infor.uva.es/tablon2/request?rid=1
```

En este caso, el programa enviado debería pasar todos las fases especificadas y obtener la máxima puntuación. En la página de inicio de *Tablon* se pueden observar los estados por los que pasa nuestra petición hasta llegar al estado "finished", que indica que todo ha salido bien y que

nuestro programa ha entrado a la tabla de puntuaciones (que se puede consultar en la página de Leaderboards).

Además del estado "finished", se observa que nuestra petición pasa por otros estados como "compiling", que indica que el programa se está compilando o "deploying", que indica que el programa compilado está esperando a que se acabe la ejecución de otros programas para empezar a ejecutarse. En los casos en los que ocurra algún problema el estado del programa será "error", lo cual indica que ha ocurrido algún fallo durante la compilación o durante la ejecución del programa. Estos errores, normalmente, son culpa del programa enviado, pero también pueden deberse a un fallo u olvido durante la puesta en marcha del servidor. Para poder consultar más información sobre los errores que surjan, habrá que cambiar un atributo de configuración del fichero `server.py` llamado `verbose`. Asignando el valor `True` a este atributo se puede consultar información extra de las acciones que realiza el servidor en el archivo `nohup.out`, que se genera al iniciar el servidor.

Como ya se ha comentado, el estado de error puede deberse a un fallo durante la compilación o debido a una excepción durante la ejecución no contemplada por la fase que se estuviera evaluando. Sin embargo, en ese último caso solo se mantiene el estado "error" cuando el atributo `shouldpass` del Leaderboard está en `True`, ya que esto significa que cualquier programa debe pasar todas las fases para ser puntuado. Para comprobar el buen funcionamiento del estado "error" se han creado dos archivos más, `comp_error.asm` y `exec_error.asm`, para comprobar el error durante la compilación y el error durante la ejecución, respectivamente.

En el caso de `comp_error.asm`, se ha optado por dejar vacío el archivo para comprobar el error de compilación. Tras enviar el programa, se observa que entra en estado "error" en el primer momento y que ha estado en ejecución cero segundos, lo cual indica que el error se debe a un fallo en compilación. Esto también se puede comprobar consultando el archivo `nohup.out`, en el cual se observa el siguiente mensaje:

```
INFO:tablon:Req2: user u1: compiling error. Error in
/home/tablon2/tablon_pruebas/usercodes/src/tmp.asm line 35 column 6:
Symbol "Bisec" not found in symbol table.
Processing terminated due to errors.
make: *** [p000002] Error 1
```

Claramente, se observa que el error ha ocurrido durante la compilación.

Por último, el archivo `exec_error.asm` es un caso especial, ya que el Leaderboard se ha definido con el atributo `shouldpass` en `False`. Esto implica que aunque haya errores de ejecución en alguna de las fases, el resto de fases se seguirán comprobando para puntuarlo. Solo en el caso de que supere el tiempo del atributo `timewall`, especificado durante la creación de las colas, no será evaluado; en cualquier otro caso, el programa se evaluará en cada fase aunque consiga una puntuación de cero en todas ellas. Por lo tanto, para poder comprobar esto, se ha creado un bucle infinito en el programa, añadiendo lo necesario para que no ocurra un fallo durante la compilación.

Al enviar el programa al servidor, se observa que la *request* asociada permanece en estado "running" durante 20 segundos (el tiempo especificado en `timewall`), y después pasa al estado "releasing". Este estado indica que se ha terminado la ejecución del programa por cualquier motivo

y que se van a liberar los recursos reservados por la *request* para que puedan ejecutarse otras *requests* en cola o enviadas posteriorment. Aquí realmente hay un fallo ya que el estado "releasing" debería durar milésimas de segundo y pasar al estado "error". Sin embargo, hay un error en el código de *Tablon* que impide que esto suceda correctamente, debido a que la base de datos no se actualiza tras cambiar del estado "releasing" al estado "error". Para comprobar que realmente se ha realizado este proceso, se pueden consultar los mensajes del archivo `nohup.out`:

```
INFO:tablon:Req 3: killed. User u2: timewall 20, time: 20.000180006
INFO:tablon:Control RELEASING: 1 of 6
```

Pruebas para `openmplb`

En esta segunda prueba se utiliza, en primer lugar, el programa `prac1.c`, el cual debe pasar todos las fases especificadas, obteniendo la puntuación máxima. Para enviar el programa se emplea el mismo comando que antes, pero cambiando el nombre del Leaderboard:

```
python2 ./client -u u2 -x u2pass -q openmplb prac1.c
```

Si todo ha funcionado correctamente, se recibirá un texto similar al obtenido en la prueba que superaba todos los casos para el Leaderboard `lb_practica1`.

Para este Leaderboard también se han creado dos archivos, `comp_error.c` y `exec_error.c`, que igual que antes servirán para comprobar los casos de error en compilación y error en ejecución, respectivamente.

El archivo `comp_error.c` es una archivo vacío que debería producir un error durante la compilación. Al mandar el programa al servidor se observa que, al instante, aparece la petición en estado "error" con tiempo de ejecución cero. Para acabar de comprobar que el error ha sido durante la compilación, se puede consultar el archivo `nohup.out` donde se observa que, efectivamente, el programa ha fallado durante la compilación con el siguiente mensaje:

```
INFO:tablon:Req5: user u1: compiling error. /lib/../lib64/crt1.o:
In function `_start': (.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
make: *** [p000005] Error 1
```

El error al que hace referencia es la ausencia de una función 'main' en el código.

Por último, el archivo `exec_error.c` contiene el código de un *Hola mundo*, de tal manera que pase la compilación pero falle durante la ejecución al no devolver la salida esperada por ninguna de las fases del Leaderboard. Dado que en este caso el valor del atributo `shouldpass` de este Leaderboard es `True`, al fallar durante la primera fase no se puntuará el programa y terminará en estado "error" sin evaluar más fases. Tras comprobar en la página web de *Tablon* que la petición ha finalizado con estado "error", se puede confirmar que el fallo ha sido durante la ejecución observando lo siguiente en el archivo `nohup.out`:

```
INFO:tablon:Request 6, user u3: executing exec_error.c
...
INFO:tablon:Req6: Output: Hello, World!salloc: Relinquishing job allocation 1181203
Request Error: None
INFO:tablon:Req6: user u3: execution finished exec_error.c
...
INFO:tablon:Passing to RELEASING state from: 9
INFO:tablon:Control RELEASING: 1 of 1
```

También se observa que la petición ha pasado del estado "error" al estado "releasing" para disminuir el número de procesos ejecutándose y no entrar en bloqueo.

3.3. Estructuración del código

Como siguiente paso en este proceso de documentación de *Tablon*, se explicarán los diferentes directorios y ficheros que componen la aplicación, recabando la información a cerca de las distintas clases y funciones que se definen, junto con la utilidad que tiene cada parte en la aplicación final. Primero se muestra la estructura de carpetas de la aplicación con una pequeña descripción de cada elemento, y posteriormente se detallarán los ficheros más relevantes o que requieran de una mayor explicación:

- ***exec_scripts***: en este directorio se encuentran los intérpretes para ejecutar los códigos que recibe cada cola de ejecución.
- ***httpdocs***: contiene los archivos estáticos para generar la página web.
- ***makefiles***: contiene los makefiles que se especifican para cada cola de ejecución creada. Posteriormente, se copiarán a `usercodes/src/` durante la creación de las colas.
- ***serverdata***: directorio en el que se almacenarán copias de los Leaderboards, usuarios y otra información de la base de datos.
- ***sqlCreateDBStructure***: contiene archivos para crear la base de datos:
  - `db_createdb.sql`: fichero sql para la creación de la base de datos.
  - `db_deletedb.sql`: script sql para eliminar la base de datos.
  - `db_structure.sql`: archivo sql que contiene la estructura de la base de datos. Crea las tablas y las dependencias entre ellas.
- ***sqlManteinance***: contiene ficheros sql que muestran información de la base datos y hacen modificaciones. Estos archivos están en desuso y no tienen una importancia especial para el funcionamiento de la aplicación.
- ***tablon***: contiene toda la estructura de programas que forman el módulo `tablon`. Es la parte clave de la aplicación donde se realizan toda la definición de clases y métodos de la misma.

- `__init__.py`: define los principales atributos del módulo `tablon` y la función `run()` de la misma, que se encarga de la puesta en marcha general del servidor.
- `_pysrp.py`: es una copia exacta del fichero `_pysrp.py` de la biblioteca srp de Python de la versión 1.0.5 [17]. Se utiliza para realizar la autenticación del cliente cuando envía una petición de ejecución de un programa al servidor. Este archivo se fusiona con el archivo `client.py` y se envía como ejecutable para que los alumnos puedan mandar sus programas con él.
- `executionmanager.py`: fichero que contiene la clase `ExecutionManager` encargada de gestionar las peticiones de los clientes y ejecutarlas usando las colas de ejecución. Al iniciar el servidor se inicializa y se ejecuta un bucle de recepción y gestión de peticiones hasta que se pare el servidor.
- `executionqueue.py`: como ya se ha explicado en la Sección 3.2.1, en él se definen las clases para representar los distintos tipos de colas de ejecución. Destacar la clase `BasicQueue`, donde se encuentran los métodos para gestionar los distintos estados por los que pasan las peticiones de los clientes para su ejecución y evaluación.
- `executionrequest.py`: define la enumeración `Status` con los distintos estados que pueden tener las peticiones y la clase `ExecutionRequest`, representación de las peticiones de los clientes.
- `executionthread.py`: fichero que define la clase `ExecutionThread`, que sirve como envoltorio para los procesos de ejecución y compilación de una petición y así poder extraer algunos datos interesantes como la salida de la ejecución o el tiempo que lleva ejecutándose.
- `frontend.py`: contiene la clase `Frontend` encargada de crear y lanzar el servidor HTTP, usando la clase `FrontendHandler`, también definida aquí, como gestor de las peticiones web. La clase `FrontendHandler` hereda de `WebServer`, `LaunchServer` y de `BaseHTTPRequestHandler` de la biblioteca BaseHTTPServer. Estas clases contienen distintas funciones que se utilizan para la correcta gestión de cada tipo de petición que reciba el servidor.
- `launchserver.py`: define la clase `LaunchServer`, la cual gestiona la conexión con el cliente, la autenticación del usuario y la gestión de la petición enviada que contiene el programa a evaluar.
- `launchserver_old.py`: esta es una versión antigua y en desuso de `launchserver.py`, se debería eliminar dado que puede entrar en conflicto con la definición de la clase `LaunchServer` del fichero más actual.
- `leaderboard.py`: como ya se ha explicado en la Sección 3.2.1, en este fichero se encuentra la definición de la clase `LeaderBoard`, el cual está formado por una lista de objetos de la clase `LeaderBoardPhase`. Estos últimos objetos comprueban los distintos test (previamente especificados) que debe superar un programa enviado por un alumno, asignando las puntuaciones obtenidas. También está definida aquí la clase `LeaderBoardItem` que sirve como clase envoltorio para la clase `ExecutionRequest`.
- `lizard.py`: código copiado del fichero `lizard.py` de una versión antigua de la biblioteca lizard de Python [18]. Esta biblioteca sirve para analizar la complejidad de un código y, en este caso, se utiliza también para obtener el número de líneas de código y tokens de un programa enviado por un alumno.
- `load_srp.py`: fichero que se usa para importar la biblioteca srp o en su defecto importar el fichero `_pysrp.py`.

- `mkpasswd.py`: contiene una función para crear contraseñas, que usa `useradmin.py`, fichero que se encuentra fuera de este directorio.
- `tablondb.py`: en este fichero se encuentran todas las funciones para gestionar la base de datos desde dentro del código de *Tablon*. La mayoría de estas funciones son envueltas en una clase decoradora llamada `dbcursor`. Además, dos funciones que no dependen de la clase decoradora están definidas aquí, una para crear la conexión con la base de datos y otra para cerrar la conexión.
- `threadpoolmixin.py`: contiene la clase `ThreadPoolMixIn` usada por la clase `Frontend` para poder manejar varias peticiones de clientes al servidor HTTP de manera simultánea.
- `users.py`: fichero que implementa una clase `User` similar a la implementada por la biblioteca srp, pero esta contiene los datos del usuario y algunas estadísticas sobre las peticiones que ha realizado. También contiene la clase `UserList` que implementa una lista de usuarios.
- `utils.py`: contiene algunas funciones varias que se utilizan en distintas partes del código, aunque algunas de ellas están en desuso. Además, contiene una clase decoradora llamada `cached`, que también está en desuso.
- `webpages.py`: contiene funciones que generan las distintas páginas de la web de *Tablon*. Estas funciones son envueltas por la clase decoradora `WebServerDispatcher`.
- `webpagesadmin.py`: en este fichero se encuentran más funciones como las del fichero `webpages.py`, solo que estas son para una parte de la web destinada a los administradores. Sin embargo, esa parte de la interfaz web está en desuso actualmente.
- `webpagesstats.py`: otro archivo más con funciones para el manejo de la web como en `webpages.py`, en este caso para la pestaña Stats de la página web.
- `webserver2.py`: fichero con varias clases para la gestión de la página web, como la clase `WebServerDispatcher` clase decoradora para las funciones de las funciones en los ficheros `webpages`. Su principal función es gestionar la peticiones web y enviar los html generados. Dado que no vamos a centrarnos en la parte de gestión y creación de la web, no se detallará más el contenido de estos últimos ficheros.

- ***templates***: la biblioteca Jinja2 de Python es la encargada de la creación de la estructura html de la web. Para ello, parte de plantillas html de cada página de la web. En este directorio se encuentran esas plantillas que corresponden a las distintas páginas a las que se puede acceder en la web y otros archivos necesarios para la creación de la página.
- ***test_codes***: en este directorio se encuentran diversos programas en distintos lenguajes que pueden usarse como pruebas para testear el servidor.
- ***usercodes***: carpeta en la que se guardan los códigos de los alumnos enviados al servidor.
- ***bestCodes.sh***: script de shell que extrae de la base de datos los programas de la cola especificada como argumento, por orden de puntuación.
- ***client.py***: este fichero es parte del ejecutable `client` que se puede descargar en la página web. En este fichero se encuentra la parte del cliente que se encarga de realizar la conexión con el servidor, pedir que se realice la autenticación y enviar el programa. Se puede encontrar aquí una implementación más simple de la clase `ExecutionRequest`, a la que más adelante el servidor añade información para completarla.

- ***client3.py***: una copia del cliente, pasada a Python 3.
- ***create_users.sh***: script de shell que crea usuarios para *Tablon*, es un buen ejemplo de como se usa el programa `useradmin.py`.
- ***kill_command_sandbox.sh***: script de shell que mata procesos *zombie* que a veces se generan durante la ejecución programas.
- ***reiniciador.py***: fichero al que se llama al iniciar el servidor para comprobar si se han realizado cambios en los demás archivos de *Tablon* y, si ya se está ejecutando, no ejecutarlo de nuevo. Llama a `server.py` para iniciar el servidor después de realizar las comprobaciones necesarias.
- ***run.sh***: script de shell para poner en marcha el servidor. El script detecta si el servidor ya está en marcha, y en caso contrario, llama a `reiniciador.py` para arrancarlo.
- ***server.py***: como ya se ha comentado en la Sección 3.2.1, en este fichero se crea la clase `CONFIG`, la cual contiene multitud de atributos de configuración del servidor. También contiene el código que crea las colas de ejecución y Leaderboards especificados, e inicia el servidor llamando a la función `run()` del módulo `tablon` al que le envía como argumento el objeto `CONFIG` creado.
- ***sqlite2mysql.py***: contiene algunas funciones para pasar la base de *sqlite* a MySql, sin embargo está en desuso, porque ahora se utiliza directamente MySql.
- ***stop.sh***: script de shell que se usa para detener el servidor, matando todos los procesos relacionados con el servidor. Si el servidor no esta en marcha no hace nada.
- ***test_send.sh***: script de shell que prueba el programa del cliente. Está desactualizado y en desuso.
- ***test.sh***: script de shell que imprime salidas. Esta desactualizado y en desuso.
- ***useradmin.py***: programa en Python que gestiona la lista de usuarios creados añadiendo otros, eliminando, cambiando contraseñas y otras funcionalidades más. Con el argumento `--help` se detallan sus diversas funciones.

3.3.1. Directorio `tablon/`

En esta sección se describirá más en profundidad algunos de los archivos más relevantes del módulo `tablon`. Estos archivos son clave para la comprensión de la aplicación, por lo que una explicación más detallada de los mismos es necesaria.

Antes de empezar a explicar los distintos archivos es importante hablar del módulo `tablon`. A lo largo del código de *Tablon* se utiliza `tablon` como una clase, con sus atributos y sus funciones, pero en realidad no es una clase, sino un módulo de Python. La principal diferencia es que un módulo no puede instanciarse igual que una clase. Habitualmente, para utilizar un módulo se escribe `import` y el nombre del módulo (por ejemplo, `import tablon`), mientras que para usarlo, se indica el nombre del modulo, seguido de un punto y el nombre de la variable, función o clase que se quiere utilizar (por ejemplo, `tablon.una_variable`). Sin embargo, cuando se carga y utiliza

un módulo de esa manera por primera vez se crea una instancia del mismo que se guarda en un diccionario estático del módulo `sys` [19] mientras el programa esté ejecutándose. De esta manera los cambios que se realicen en esa instancia se quedarán guardados y se podrá acceder a la instancia mediante el diccionario importando el módulo `sys` de la siguiente manera:

```
import sys
tablon = sys.modules['tablon']
```

Esta forma de trabajar con `tablon` recuerda al patrón de diseño *Singleton*, en el que se hacía que una clase solo tuviera una única instancia de si misma a la que se podía acceder desde cualquier código. Sin embargo, en este caso `tablon` es un módulo, por lo que no hay una definición como tal de clase y no se puede hacer que otra clase herede de él.

__init__.py

A la hora de definir un módulo en Python es necesario definir este archivo, ya que es el que indica al intérprete que todos los ficheros en ese directorio forman el módulo. Sin este archivo no se podrá importar el módulo desde fuera del directorio. Habitualmente se puede dejar vacío, pero en este caso se definen en él algunas variables para `tablon` y la función que pondrá en marcha el servidor. Es importante destacar que cuando se importa el módulo completo en algún programa, se ejecuta este archivo.

Se podría decir que este archivo se usa para inicializar el módulo, definiendo variables como las listas de las colas y Leaderboards y la función `run(config)`. Esta función se le llama por primera y única vez desde `server.py`, con la configuración especificada en este, para que ponga en marcha el servidor. En primer lugar creará y guardará en `tablon` una instancia de la clase `ExecutionManager` y ejecutará la instancia para que empiece a gestionar peticiones. A continuación pondrá en marcha el servidor http con la clase `Frontend`, que se quedará en bucle esperando hasta que se mate al proceso.

executionmanager.py

Este archivo define la clase `ExecutionManager`. Como ya se ha explicado, esta clase se encarga de gestionar los envíos de programas de los alumnos, de tal manera que cada vez que el servidor recibe una petición de un cliente se llama a esta clase para añadir la petición a la lista. Cabe destacar que todos los ficheros que utilizan esta clase utilizan la instancia guardada en el módulo `tablon` y creada en el fichero `__init__.py`.

Inicialmente, se crea la instancia de esta clase y se ejecuta mediante la función `run()` de la misma. En ella se definen los atributos de la clase y se crea el hilo de ejecución que añade y gestiona peticiones en bucle hasta que se pare el servidor.

Esta clase tiene dos atributos que contienen listas de peticiones `inrequests` y `requests`. La primera lista es en la que se guardan las peticiones cuando las recibe el servidor, por lo que además

del hilo principal de ejecución del servidor, también es manipulada por otros hilos concurrentes a este. Para evitar problemas de acceso concurrente a esta lista, se utiliza la clase `Lock` de la biblioteca `threading` [20], la cual se puede usar para bloquear a un hilo A el acceso a un trozo de código si otro hilo B está ejecutando ese mismo código, hasta que el hilo B termine de ejecutarlo. La segunda lista contiene también peticiones recibidas por el servidor, pero solo es usada por el hilo de ejecución del servidor, evitando así problemas de acceso concurrente. De esta manera en el bucle de esta clase primero se mueven peticiones de una lista a otra para poder gestionarlas.

La función `manage_requests()` se encarga de llamar a la función `update(request)` de la cola asociada a cada petición de la lista `requests`, para que se encargue de compilar y ejecutar el programa. Después eliminará todas las peticiones de la lista que hayan terminado de ejecutarse y evaluarse.

executionrequest.py

En este programa se definen dos clases: la clase `Status`, enumeración que contiene los distintos estados por los que puede pasar una petición durante la ejecución de su programa, y `ExecutionRequest` que define la estructura básica de una petición de ejecución de un programa. Más adelante se explicará en profundidad las acciones y transiciones de cada estado.

La clase `ExecutionRequest` contiene atributos sobre el usuario que envió la petición y sobre el programa que ha enviado. La petición inicialmente se crea como una instancia de la clase `ExecutionRequest` del fichero `client.py`. Esta definición contiene menos atributos y funciones que la clase definida en `executionrequest.py`. Cuando la petición entra a *Tablon* se convierte en una instancia de la clase definida en este último archivo mediante la función `sanitize()`, que añade a la instancia los atributos que faltan.

A parte de funciones para añadir información a la instancia, esta clase tiene otras funciones útiles para comprobar si ha finalizado, comprobar si hay algún elemento prohibido según el atributo `forbidden` de la clase `CONFIG`, o para eliminar los comentarios del programa enviado.

executionqueue.py

En la Sección 3.2.1 se ha hablado ya sobre las distintas clases de colas de ejecución y las relaciones de herencia entre ellas. En este apartado se describen únicamente las clases que no estén obsoletas actualmente.

A la hora de crear una cola de ejecución siempre se llama al mismo constructor, el de la clase `BasicQueue`. Esta clase define todos los atributos de una cola de ejecución. Sin embargo, la clase `SlurmQueue` añade algunos atributos más relativos al servidor de Slurm que se utiliza para ejecutar los programas. Además de definir los atributos, la clase `BasicQueue` contiene algunas funciones comunes a cualquier cola de ejecución y la definición de algunas que serán implementadas por las clases hijas. Una de las funciones más importante de esta clase, `update(request)`, se encarga de ejecutar las acciones necesarias para la ejecución de una petición según el estado de la misma y es a la que llama `ExecutionManager` en su bucle para gestionar las peticiones.

La función `update(request)` llama a una función distinta dependiendo del `Status` de la petición o `ExecutionRequest`. A continuación se explican brevemente cada estado y su función asociada:

- `QUEUED`, `check(request)`. Comprueba la petición y la añade a la cola.
- `COMPILING`, `compile(request)`. Compila el programa y comprueba si ha fallado en el proceso.
- `DEPLOYING`, `deploy(request)`. Espera hasta que termine de ejecutarse alguna otra petición.
- `WAITING`, `execute(request)`. Prepara el programa para su ejecución y llama a la función `execute_run(request)` que ejecuta el programa.
- `RUNNING`, `execute_end(request)`. Comprueba la salida de la ejecución y la puntúa si tiene un Leaderboard asociado con la función `execute_end_leaderboard(request)`.
- `RELEASING`, `release(request)`. Libera el proceso de ejecución de la petición.

En cada una de estas funciones, además de las acciones que se realizan para avanzar en la ejecución de la petición, se lleva a cabo la transición de la petición a un nuevo estado, hasta que esta llega a uno de los estados finales, `ERROR`, `FINISHED` o `CANCELED`. Algunas de estas funciones, como `release(request)` o `deploy(request)` son sencillas y comunes a cualquier tipo de cola, sin embargo, otras funciones se redefinen en las clases hija, como `compile(request)` o `execute(request)`. En el caso de la clase `SlurmQueue` se redefine `execute_run(request)`, función a la que llama la función `execute(request)` para crear el proceso de ejecución como una instancia de la clase `ExecutionThread`, e iniciar la ejecución.

Si la petición se realizó directamente para ejecución en una cola, *Tablon* se saltará toda la parte correspondiente a puntuar el programa con un Leaderboard y guardará la salida para mostrarla en la página. Sino, en la función `execute_end_leaderboard(request)` se comprobará el resultado de la primera fase del Leaderboard con la función `check(request)` de la clase `LeaderBoardPhase`. Si la función devuelve `True` se pasará a la siguiente fase del Leaderboard y se volverá al estado `WAITING` para que preparé la ejecución con los argumentos de la nueva fase, repitiendo todo el proceso de ejecución hasta que se hayan evaluado todas las fases o falle en alguna.

leaderboard.py

En este fichero se puede encontrar la definición de tres clases distintas que conforman un Leaderboard. En primer lugar la clase `LeaderBoardItem`, que sirve como envoltorio para la clase `ExecutionRequest`, de tal manera que cuando una petición ha sido evaluada y puntuada, se crea un `LeaderBoardItem` que encapsule la información relevante para añadir la petición a la tabla de puntuaciones. En segundo lugar, la clase `LeaderBoardPhase` define los tests que debe pasar un programa para ser puntuado, y además contiene la mayor parte de la funcionalidad de evaluación de los programas. Por último, la clase `LeaderBoard` define el formato y métrica de la tabla de puntuación, el cual está formado por uno o más `LeaderBoardPhase` y uno o más `LeaderBoardItem`.

Como ya se explicó previamente, la clase `LeaderBoardPhase` contiene la mayor parte de la funcionalidad, ya que se encarga de evaluar la salida y asignar una puntuación. Para ello, se llama a la función `check(request)`, que a su vez llama a la función `check_errors(request)` para comprobar si ha habido algún fallo durante la ejecución y obtener el mensaje de error correspondiente. Después itera sobre las subfases o posibles salidas de la fase que se está comprobando y filtra el resultado teniendo en cuenta los atributos del Leaderboard y de la fase definidos en la Sección 3.2.1, asignando una puntuación a la fase. Si el programa fallase en una fase y se ha definido que deben pasar todas las fases para ser puntuado, la función devuelve False y la petición termina en estado `ERROR`.

tablondb.py

`tablondb` no funciona tanto como una clase a la que se llama desde otras partes del código, sino como un conjunto de funciones de acceso a la base de datos. La base de datos es esencial en esta aplicación, ya que facilita la extracción de información para crear las distintas tablas que aparecen en la web.

Cada una de las funciones de acceso a la base de datos está envuelta por una clase decoradora `dbcursor`, que comprobará si existe una conexión con la base para crearla en caso contrario, y ejecutará la función deseada. En ese sentido funciona igual que el último ejemplo explicado en la Sección 3.1.2, ya que se indica a cada función que su objeto decorador es la función `dbcursor_and_call` creada dentro de la función `__call__` de la clase.

Por último, destacar que la declaración de `@dbcursor()` encima de cada función se ejecuta cada vez que se importa el fichero en cualquier programa, ya que cuando se importa un fichero se ejecuta todo el código que no esté encerrado en una función o clase. Al ejecutar esa línea se crea una instancia de la clase `dbcursor` y se ejecuta la función `__call__` del mismo, de tal manera que cada función tiene como clase decoradora una instancia distinta de la misma.

_pysrp.py

Como ya se ha comentado, este archivo es una copia de una versión antigua de la biblioteca srp [17]. En particular, es una copia del archivo con el mismo nombre, el cual contiene las definiciones más esenciales para el funcionamiento de la biblioteca. Este archivo contiene las clases `Verifier` y `User`, la primera sirve para verificar la identidad de un usuario remoto y la segunda para que el cliente pruebe su identidad a la clase `Verifier`. Por lo tanto, la clase `Verifier` será usada por la parte del servidor, y la clase `User` por la parte del cliente.

Estas clases contienen las funciones necesarias para crear un "salted verification", esto es, un conjunto de datos generados aleatoriamente y que se añaden a los mensajes entre cliente y servidor para verificar la conexión. Cuando se crea una sesión de verificación entre estas clases, la clase `Verifier` envía el "reto" de verificación al cliente para que usando la clase `User` puede verificarse. Además de estas clases, el fichero contiene otras variables y constantes para la generación de las distintas claves y tablas hash.

launchserver.py

La definición de la clase `LaunchServer` en este fichero es esencial para que los alumnos puedan enviar sus programas a *Tablon*. Esta clase implementa la función `do_CONNECT()` que se ejecuta cuando se envía una petición HTTP con la cabecera `'CONNECT ...'`. Estas peticiones, como ya se ha explicado, son manejadas por la clase `FrontendHandler` definida en el fichero `Frontend.py`.

La implementación de esta función se divide en dos partes, la autenticación del cliente y el manejo de la *request* que envía. A lo largo de esta función se utiliza un buffer de lectura y otro de escritura para poder comunicarse con el cliente. Inicialmente se envían mensajes de protocolo para asegurar que la conexión se está haciendo correctamente. A continuación se utiliza el fichero `_pysrp.py` para realizar la verificación del usuario. De esta forma, `LaunchServer` envia al cliente el reto de verificación y el cliente le manda de vuelta el reto completado para terminar así la autenticación.

Una vez se ha comprobado la identidad del cliente, se empieza a gestionar la *request* que ha enviado. Para ello, se crea un hash de los datos y se comprueba que el hash enviado por el cliente es igual que el que genera el servidor a partir de los datos enviados. Después de esto se trata la *request* como una instancia de la clase `ExecutionRequest` y se llama a la función `sanitize()` para añadir los atributos faltantes a esta. Tras diversas comprobaciones, como por ejemplo la existencia de la cola que especifica la *request*, se envía la instancia al `ExecutionManager` para que la gestione y mande su programa a ejecución.

3.3.2. Fichero `client.py`

Este archivo funciona de manera similar a lo explicado sobre `launchserver.py`, salvo que en este caso realiza la parte del cliente. Este programa tiene varios argumentos y opciones explicadas ya en la Sección 3.2.3. A partir de esos argumentos se crea la conexión con el servidor y se comunicará con él mediante los buffers de lectura y escritura.

El primer mensaje que se envía es la petición HTTP con la cabecera `'CONNECT ...'` para que el servidor sepa que se quiere comenzar una conexión de envío de programas ejecutando la función `do_CONNECT()` de la clase `LaunchServer`. En la primera etapa se autentica el cliente utilizando la clase `User` del fichero `_pysrp.py` como ya se ha explicado. Una vez realizada la verificación de la identidad del usuario, se crea una *request* a partir de la clase `ExecutionRequest`, pero la definida en este archivo, no la definición de `executionrequest.py`. Esta definición contiene los atributos más básicos y principales de una petición, y se completa su contenido cuando el servidor lo recibe.

Cuando la *request* ha sido creada, el cliente realiza el "marshalling" de esta instancia y lo envía al servidor junto con un hash de la misma. Después de esto espera a la respuesta del servidor para imprimir por pantalla lo que haya sucedido con su petición.

3.4. Diagramas de diseño

En esta sección se tratará de mostrar distintos diagramas que se deberían haber generado en una etapa de diseño de la aplicación, pero que se han generado a posteriori realizando ingeniería inversa de la aplicación. Estos diagramas muestran un comportamiento más específico de la aplicación, y dado su diseño desestructurado, y en ocasiones, enrevesado, algunos diagramas de diseño se han acompañado de explicaciones textuales para facilitar su correcta compresión.

En primer lugar se mostrarán diagramas de paquetes, módulos Python, submódulos y clases, junto con las relaciones de herencia y dependencia entre ellos. Posteriormente, se detallarán algunas de las acciones más importantes del servidor mediante diagramas de secuencia, junto con un diagrama de máquina de estados que explicará las fases por las que pasa una *request* durante su gestión.

3.4.1. Diagrama de módulos y ficheros

Para mostrar la estructura más superficial de la aplicación se ha creado un diagrama de clases, mostrado en la Figura 3.7. En este diagrama se representan también las relaciones entre los módulos y ficheros Python, junto con las relaciones de herencia con bibliotecas externas a la aplicación.

En este diagrama se utilizan distintos estereotipos para explicar las agrupaciones que se muestran. En primer lugar, el estereotipo «python module» se utiliza para representar una agrupación de programas que forman lo que el intérprete de Python considera un módulo, de tal manera que se pueda importar a cualquier programa con la sentencia habitual. Además del módulo `tablon` ya explicado, se representan `http.server` y `socketserver`, módulos internos de Python. Un módulo puede estar formado por uno o varios programas Python en los que se pueden definir clases, o también pueden contener únicamente funciones o código ejecutable al que se puede acceder desde fuera del mismo.

Para poder explicar mejor el contenido de un módulo se ha utilizado el estereotipo «python submodule», que representa los distintos archivos que contiene el módulo y que también puede importarse con la sentencia habitual. En este caso Python no distingue entre módulos y submódulos, pero hacemos esta distinción para representar la estructura de la aplicación más cómodamente.

Por último, usamos el estereotipo «python file» para representar ficheros de Python que no se encuentran dentro de ningún módulo, pero que definen clases, funciones o código ejecutable. Estos ficheros no puede importarse desde otro programa, pero si puede ejecutarse por línea de comandos como cualquier programa en Python. Destaca en ese sentido la dependencia entre `reiniciador.py` y `server.py` que no representa una llamada a métodos o clases de `server.py` de `reiniciador.py`, sino la ejecución como comando del programa `server.py` desde `reiniciador.py`.

Cabe destacar que `server.py` llama a la función `run()`, función del módulo `tablon` que no se encuentra en ningún submódulo y por ello no se representa dicha dependencia directa.

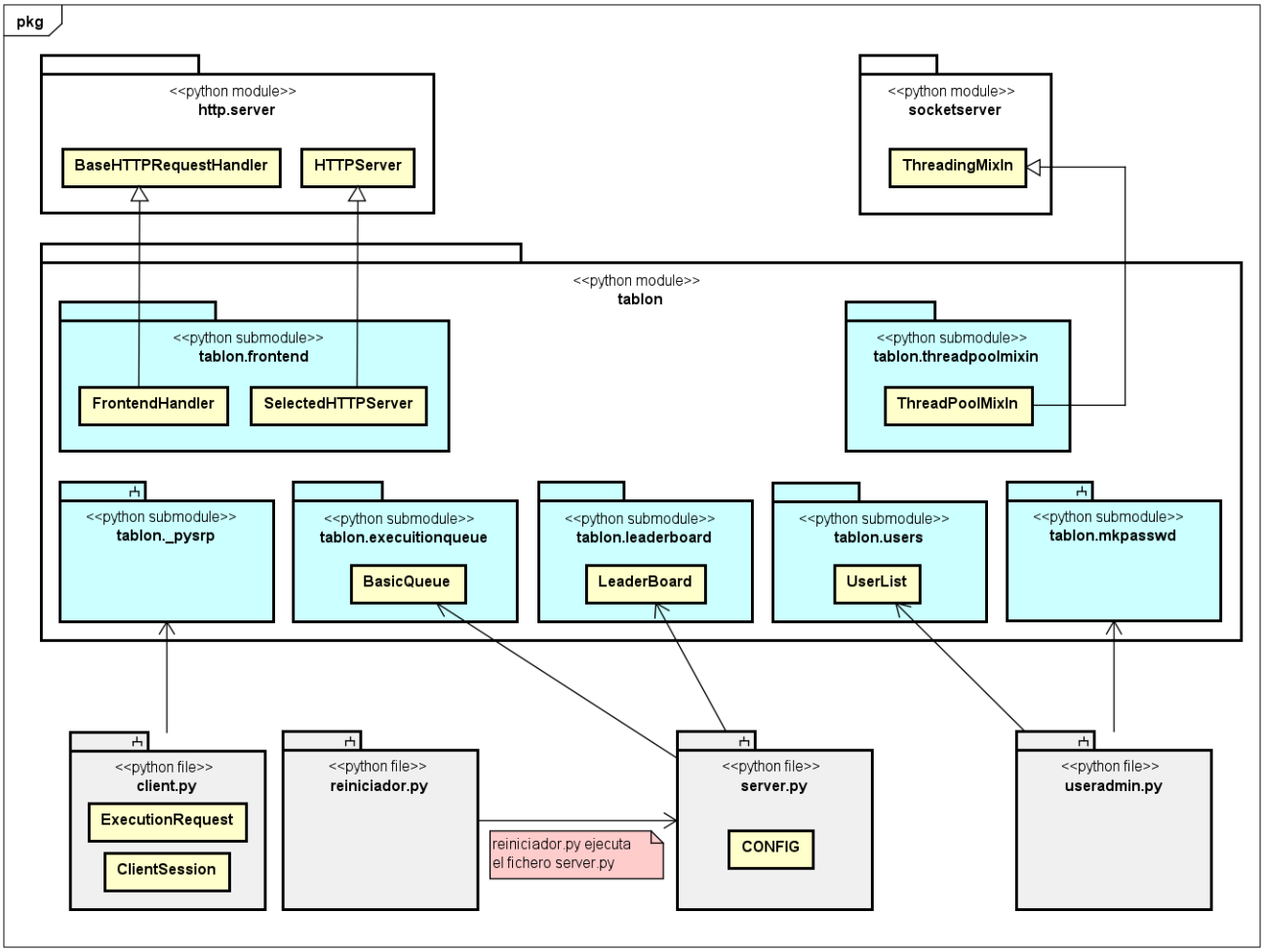


Figura 3.7: Diagrama de módulos y ficheros de la aplicación

3.4.2. Diagramas de submódulos y clases

A continuación se tratará de explicar la estructura interna del módulo `tablon` con distintos diagramas. La Figura 3.8 muestra un diagrama menos detallado sobre el contenido de las distintas clases, pero que ilustra las relaciones de dependencia, herencia y composición entre los submódulos y sus clases.

Este diagrama tiene la estructura habitual de un diagrama UML de clases salvo por la forma de agrupar las clases en submódulos, como ya se ha explicado anteriormente. Como se puede observar, algunas relaciones apuntan directamente a un submódulo como conjunto en lugar de a una clase. Esto se debe a que algunos submódulos contienen funciones o código ejecutable fuera de clases, por lo que la relación de dependencia es hacia esas funciones.

Como ya se ha comentado previamente, en este trabajo no se ahondará en la forma de generar los ficheros html que se muestran en la página web. Es por esto que en el submódulo `tablon.webserver2` faltan por representar clases con las que se relacionarían `tablon.webpages` y `tablon.webpagesstats`. Además se ha omitido del diagrama el archivo `webpagesadmin.py` dado que está actualmente en desuso.

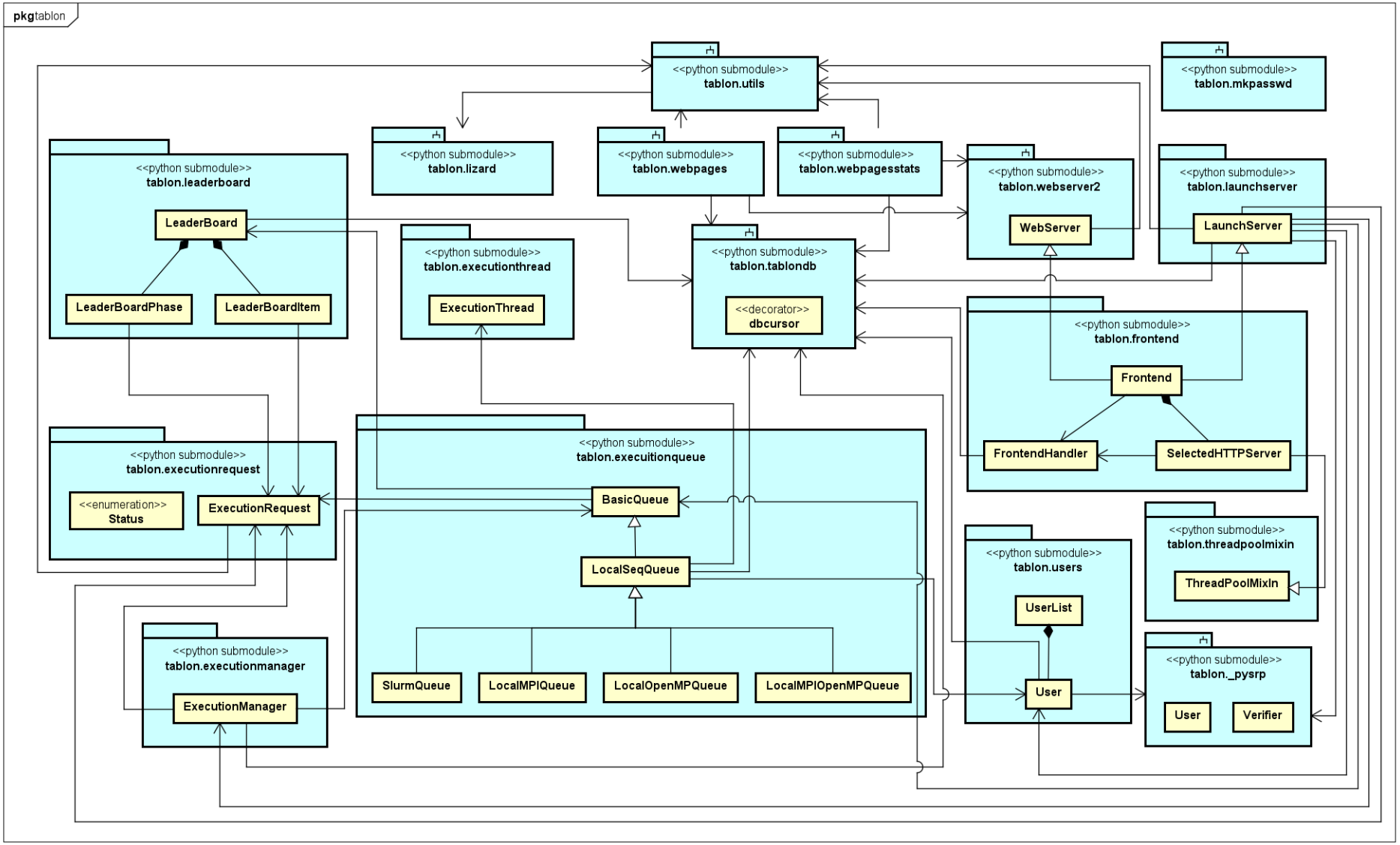


Figura 3.8: Diagrama de los submódulos y clases del módulo `tablon`

Dado que que el fichero `lizard.py` se ha copiado directamente de una biblioteca externa y se usa bastante poco a lo largo de la aplicación, no se detallará más la estructura interna de su submódulo.

A continuación, se muestran los diagramas que detallan los atributos y funciones de los submódulos más relevantes entre los ya presentados. La Figura 3.9 detalla el contenido de los submódulos "execution", entre los que se encuentran: `executionmanager`, `executionthread` y `executionrequest`. En la Figura 3.10 se muestra de forma detallada el contenido del submódulo `executionqueue`. Los submódulos relativos a la gestión de la conexión se han agrupado en la Figura 3.11, en la cual se detalla el contenido de los siguientes submódulos: `frontend`, `_pysrp`, `users`, `threadpoolmixin` y `launchserver`. Por último, la Figura 3.12 presenta de forma detallada el contenido del submódulo `leaderboard`. Se omite el submódulo `tablon.tablondb` dado que la mayoría de su contenido son funciones fuera de clases.

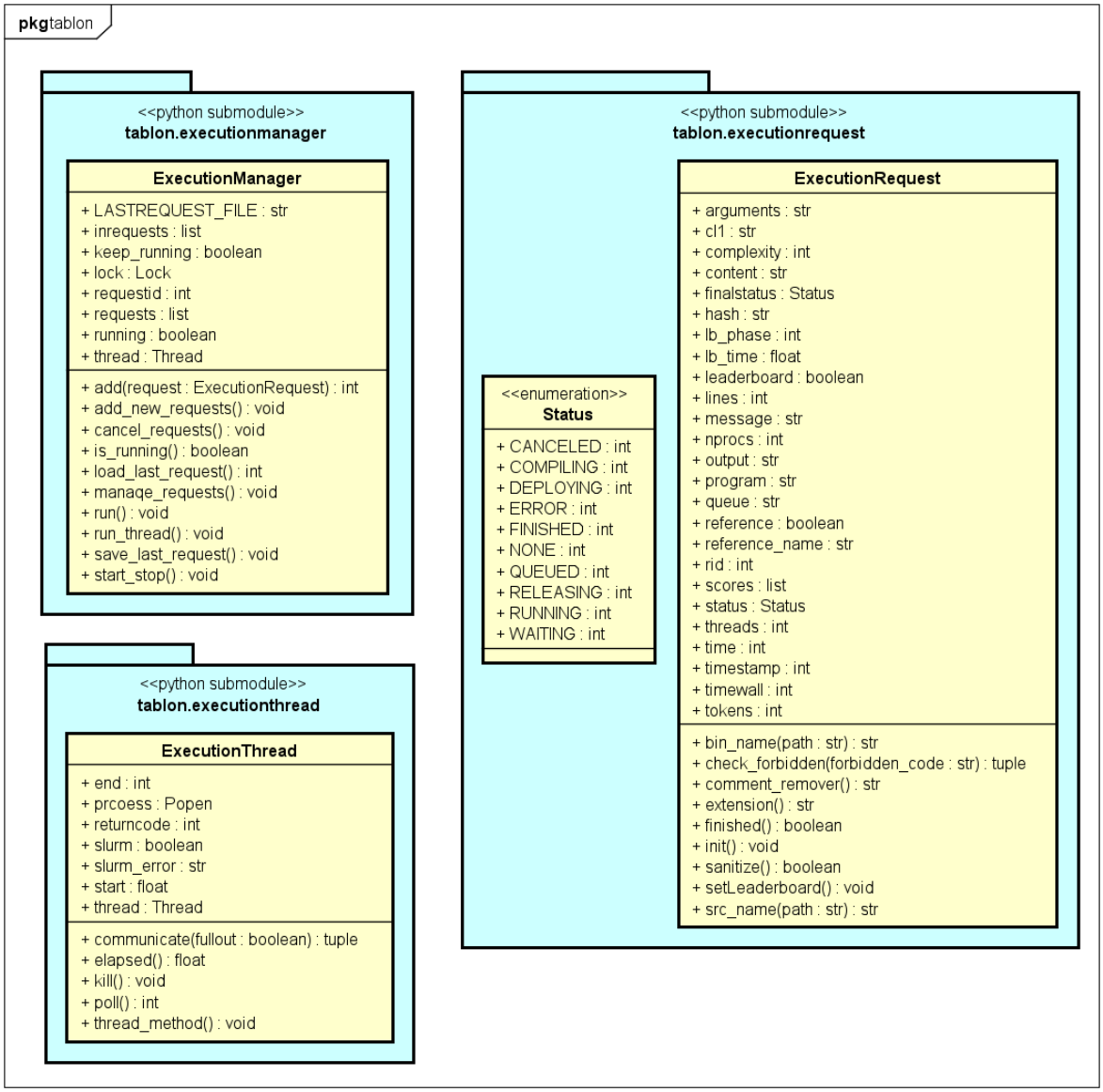


Figura 3.9: Diagrama detallado de los submódulos "execution"

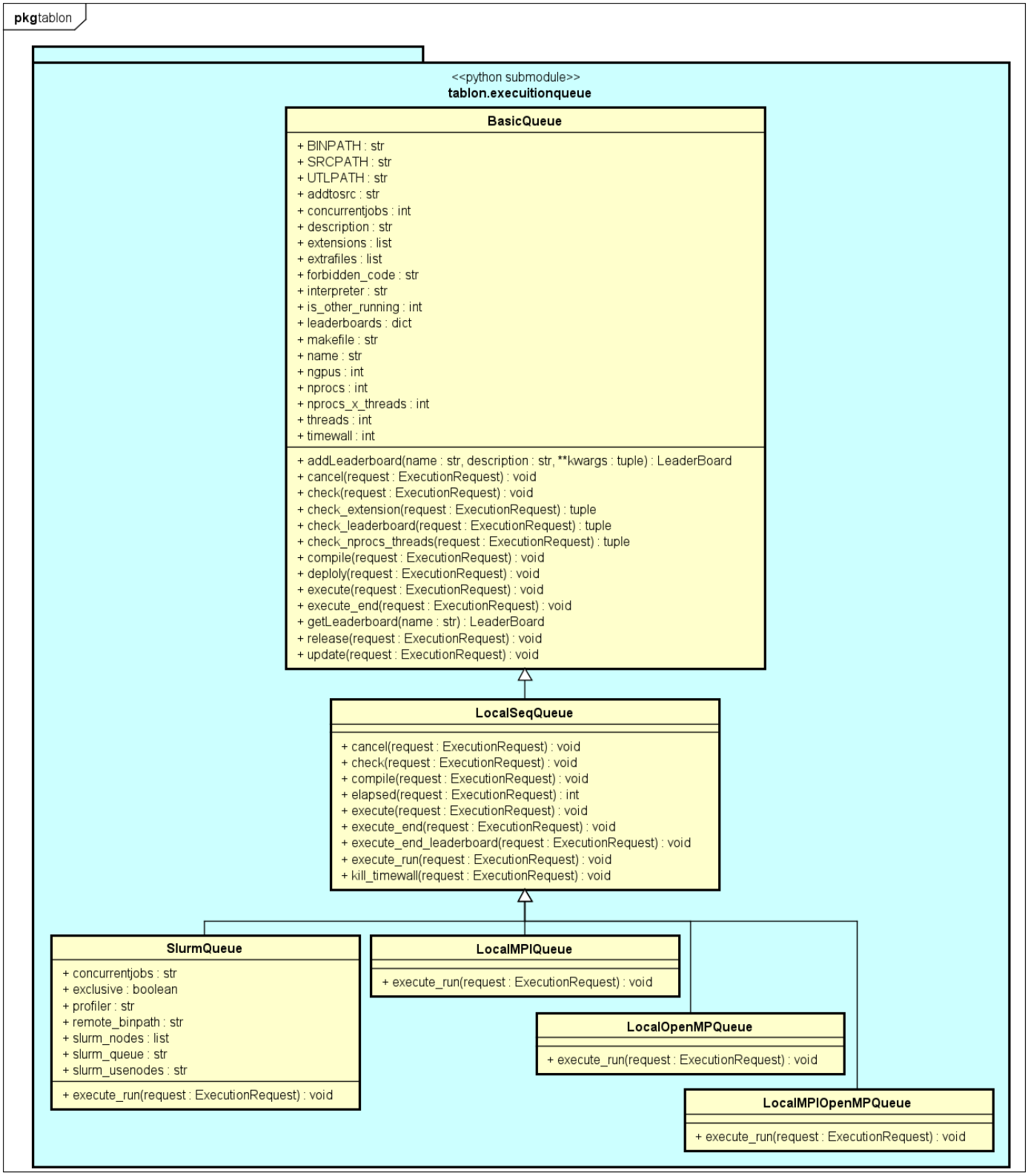


Figura 3.10: Diagrama detallado del submódulo "executionqueue"

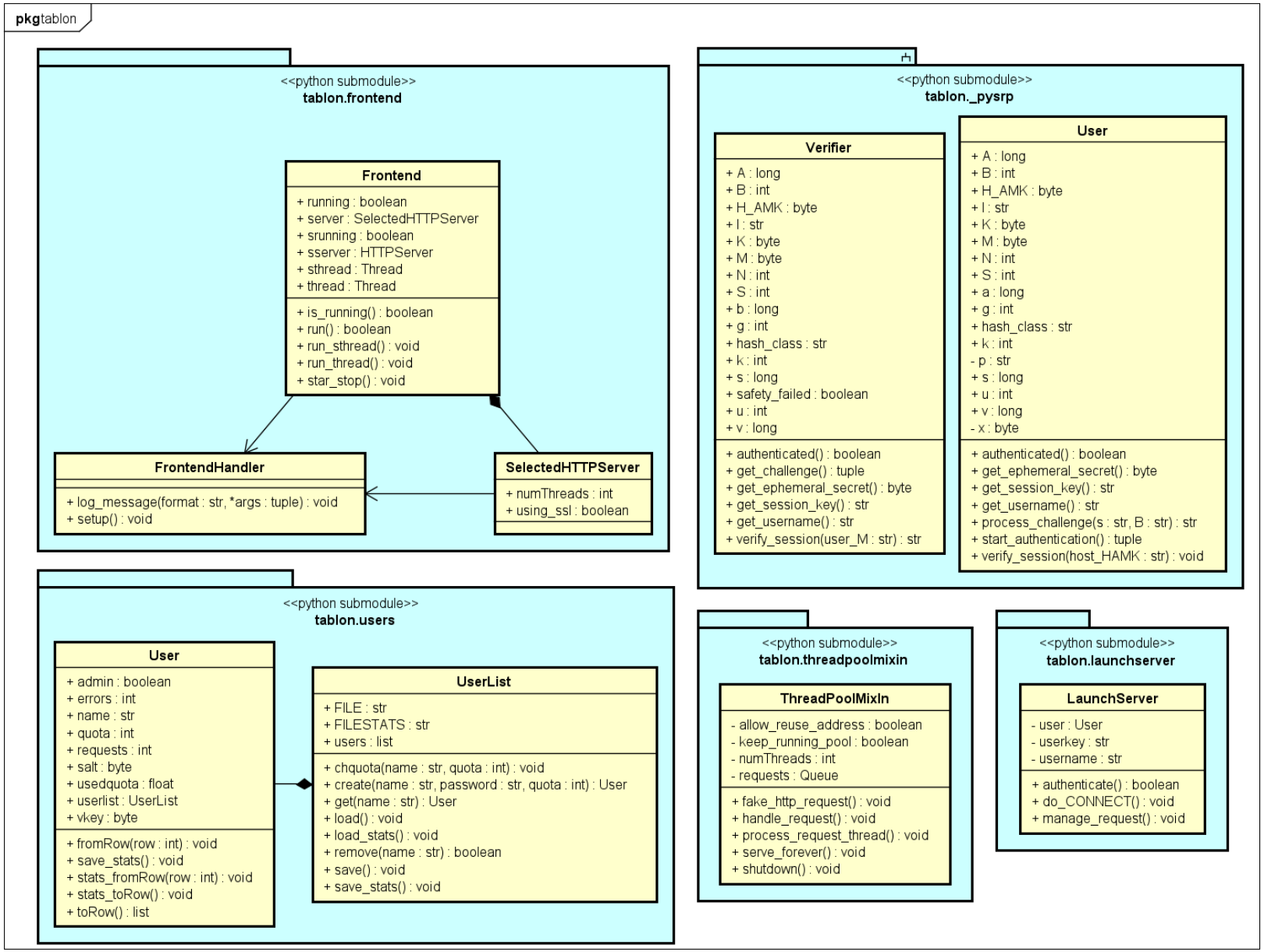


Figura 3.11: Diagrama detallado de los submódulos de conexión

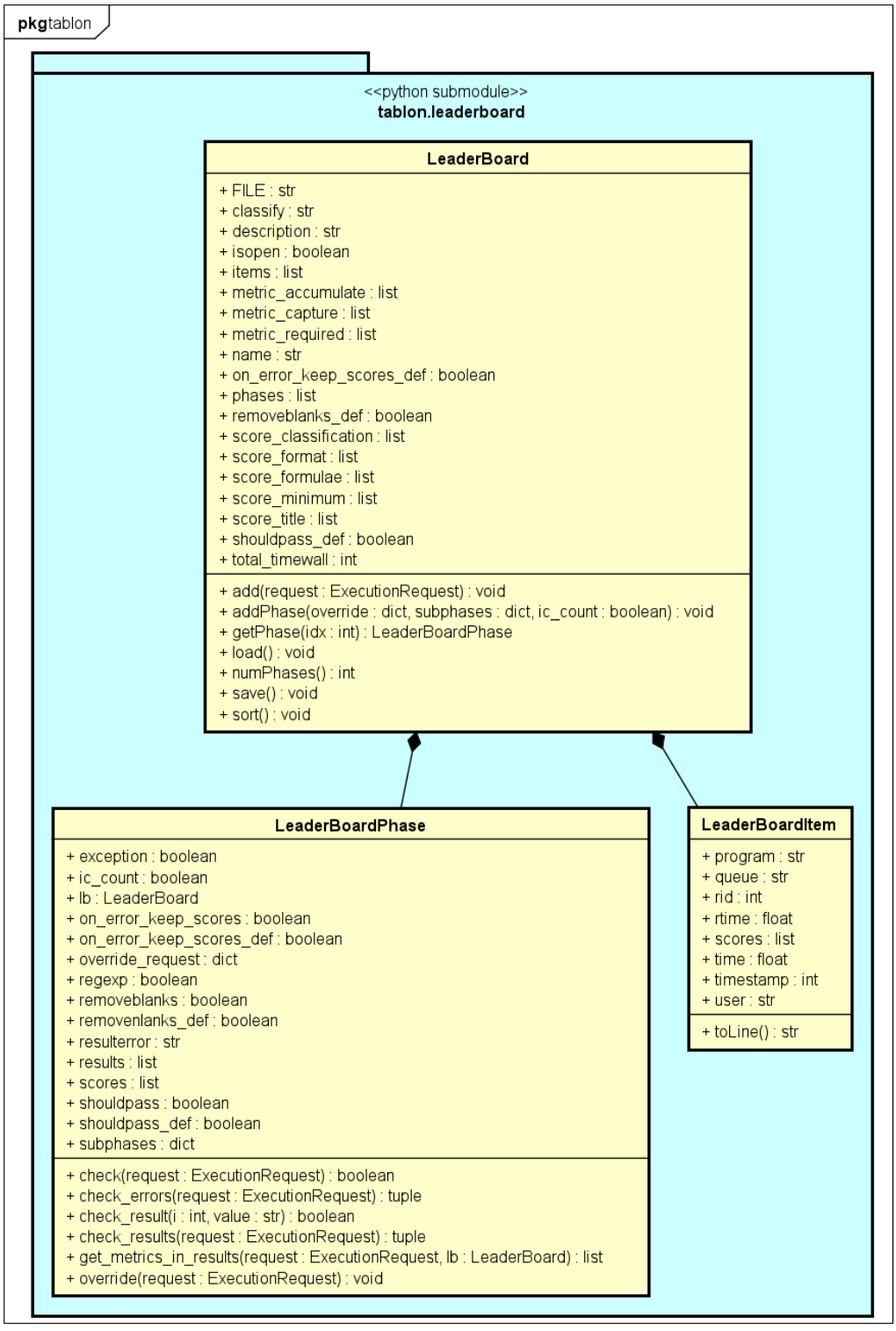


Figura 3.12: Diagrama detallado del submódulo "leaderboard"

3.4.3. Diagramas de secuencia

Seguidamente se ilustran diagramas de secuencia que explican el funcionamiento de las partes más relevantes de la aplicación, a partir de las clases que se han definido en los diagramas anterio-

res. Dado que la puesta en marcha del servidor se ha ido explicando a lo largo de este capítulo, se ha decidido no representarla en un diagrama de secuencia. Además el diagrama de secuencia que se generaría sería demasiado enrevesado y denso, ya que implicaría muchas clases. Por ello solo se explica el envío de programas por parte del cliente y como entran estas peticiones al servidor para ser gestionadas.

En primer lugar, la Figura 3.13 muestra el proceso de la autenticación del cliente durante el envío de un programa. En este diagrama se asume que el servidor está en marcha y, por tanto, la clase `BaseHTTPRequestHandler` se encuentra en un bucle de escucha de peticiones HTTP. Se representa entonces la comunicación entre cliente y servidor mediante buffers de lectura y escritura y toda la fase de verificación de la identidad del cliente mediante las clases del submódulo `tablon._pysrp`.

La Figura 3.14 muestra el diagrama de secuencia del envío de un *request* en sí, lo cual ocurre una vez finalizado el proceso de autenticación visto en el diagrama anterior. El servidor recibe la *request* creada por el cliente y realiza diversas comprobaciones sobre el mismo. Para facilitar la compresión del diagrama, la mayoría de estas comprobaciones se han omitido del diagrama. Sin embargo si se muestra la comprobación de que el hash enviado por el cliente coincide con el creado por el servidor a partir de los datos enviados, y el "saneamiento" de la *request* como instancia de la clase `ExecutionRequest`. Después de estas comprobaciones la petición es aceptada y se envía al `ExecutionManager`, que está ejecutándose en un hilo diferente, para que gestione y mande a ejecución el programa.

Por último, la Figura 3.15 muestra el proceso de gestión de nuevas *requests*. Aquí se observa como el `ExecutionManager` mueve la *request* de una lista a otra para evitar problemas de acceso concurrente, como ya se explicó anteriormente, y llama a la función `update(request)` de la cola adecuada, la cual, dependiendo del estado de la *request*, realizará una acción u otra. Esas acciones se muestran con más detalle en el diagrama de máquina de estados de la Figura 3.16.

3.4.4. Diagrama de máquina de estados

Dado que las acciones que se llevan a cabo durante la gestión de una *request* dependen del estado de la misma, la forma más adecuada de presentar estas acciones es mediante un diagrama de máquina de estados, mostrado en la Figura 3.16.

Este diagrama no solo ayuda a entender por qué etapas pasa una petición del cliente, también explica la transición entre esas etapas. Además este es uno de los diagramas donde más se percibe como la aplicación tiene un diseño poco intuitivo y enrevesado, con estados poco claros. Por ejemplo, el estado ERROR representa tanto los errores de compilación del programa como los errores de ejecución. Esto complica el diagrama, ya que un error de compilación causa la finalización de la gestión de la *request*. Sin embargo, un error de ejecución puede ser el resultado esperado de una fase del Leaderboard, por lo que no finalizaría ahí la gestión de la *request*.

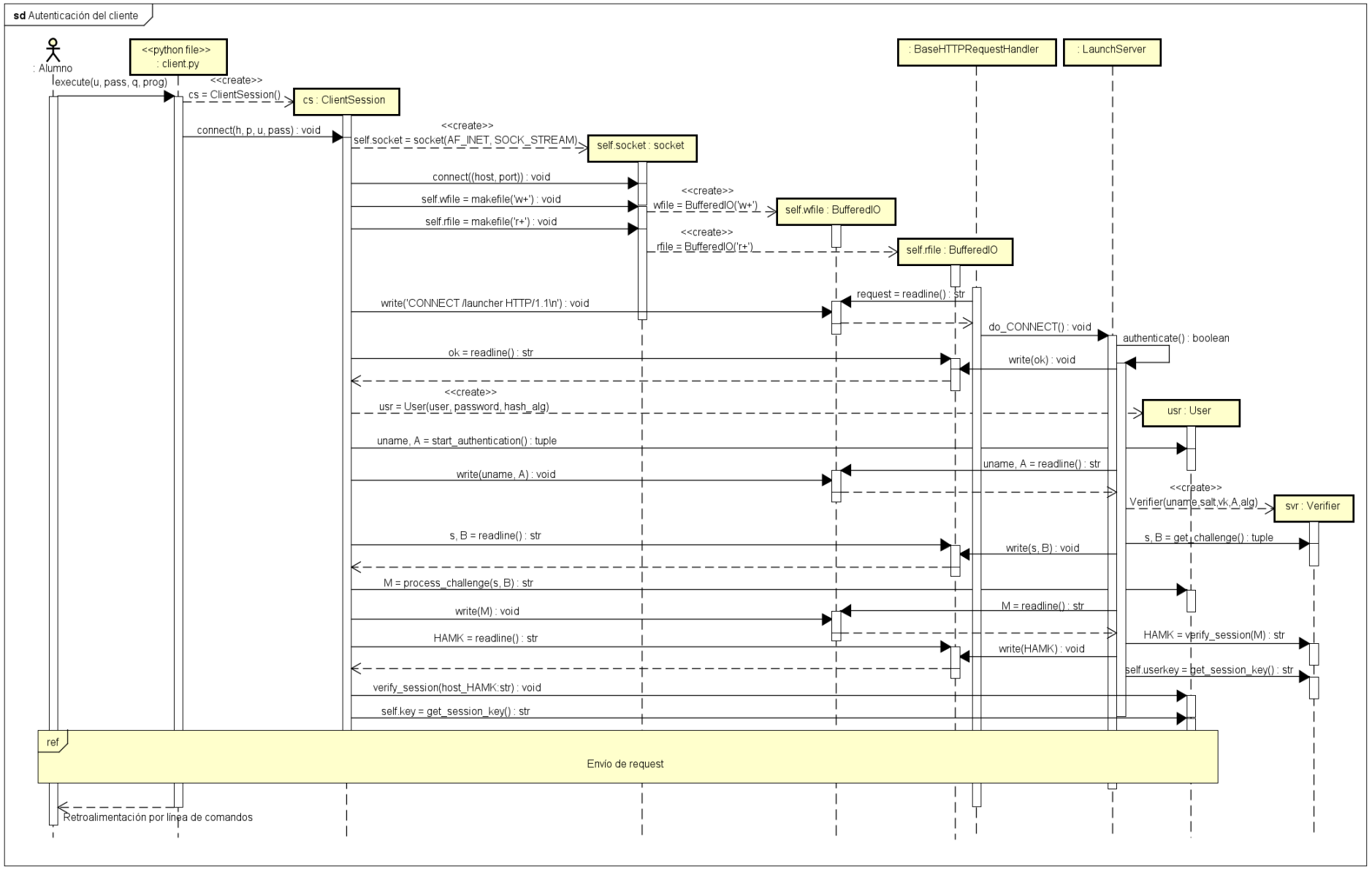


Figura 3.13: Diagrama de secuencia de la autenticación de un cliente

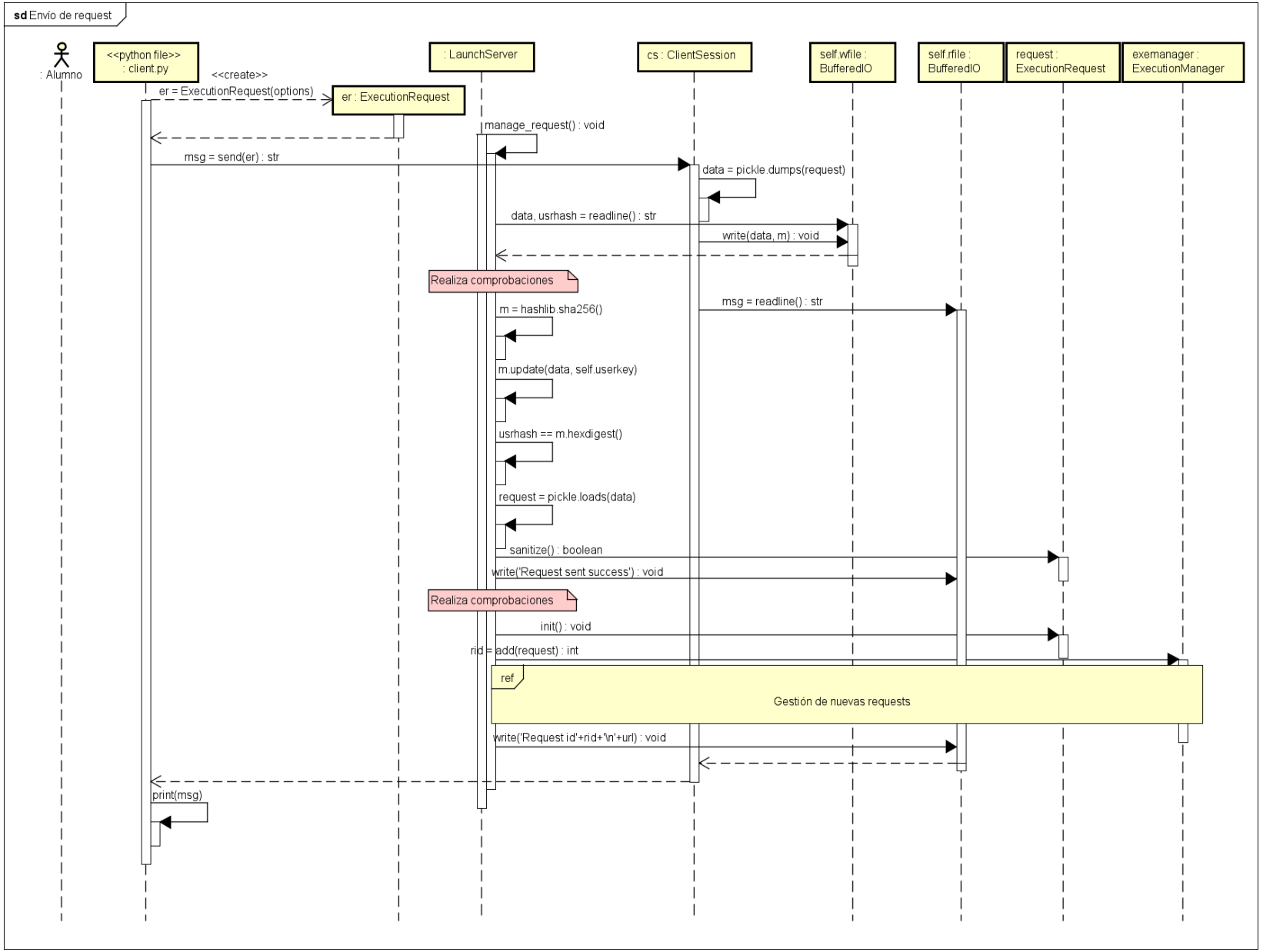


Figura 3.14: Diagrama de secuencia del envío de una *request* por el cliente

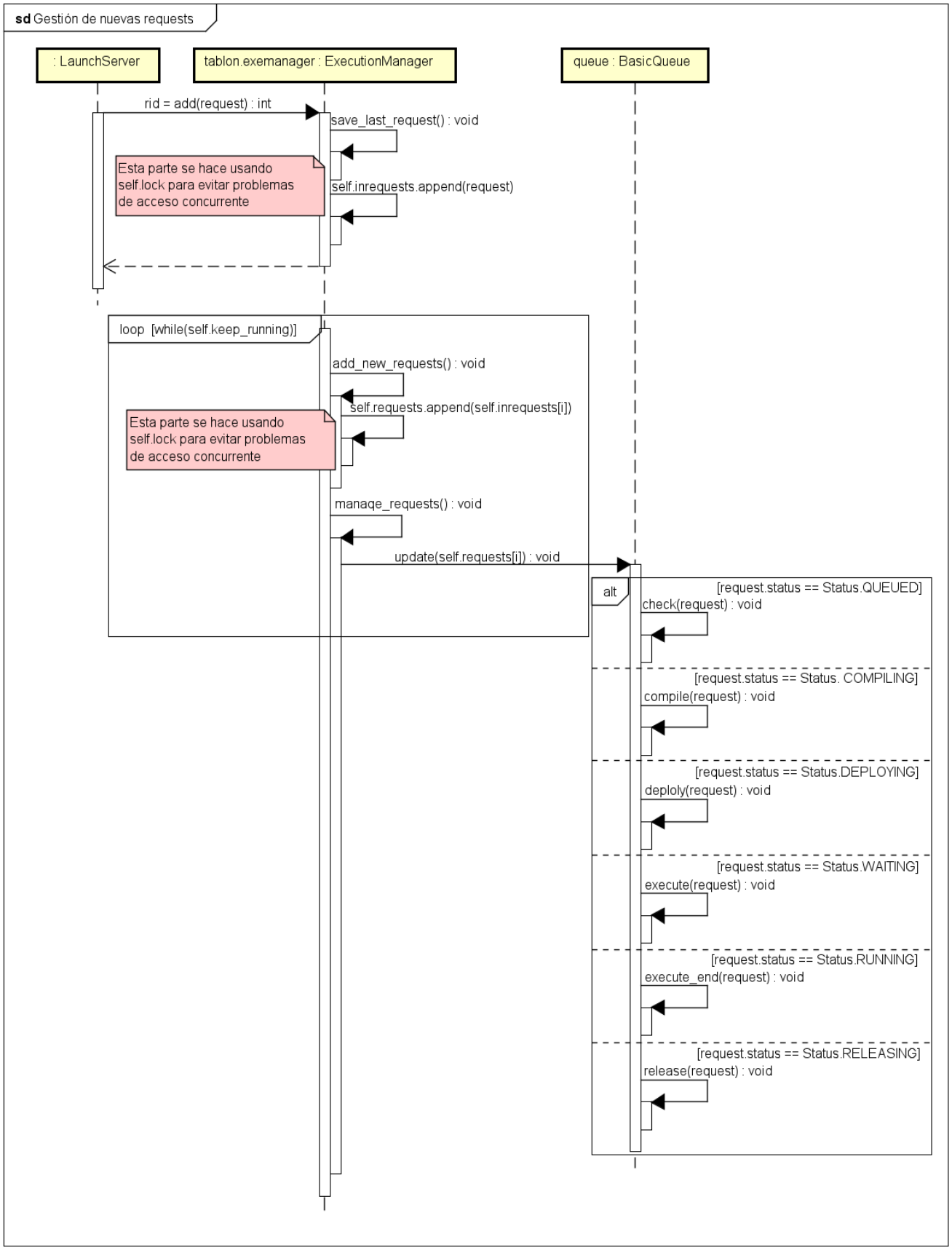


Figura 3.15: Diagrama de secuencia de la gestión de nuevas *requests* en la aplicación

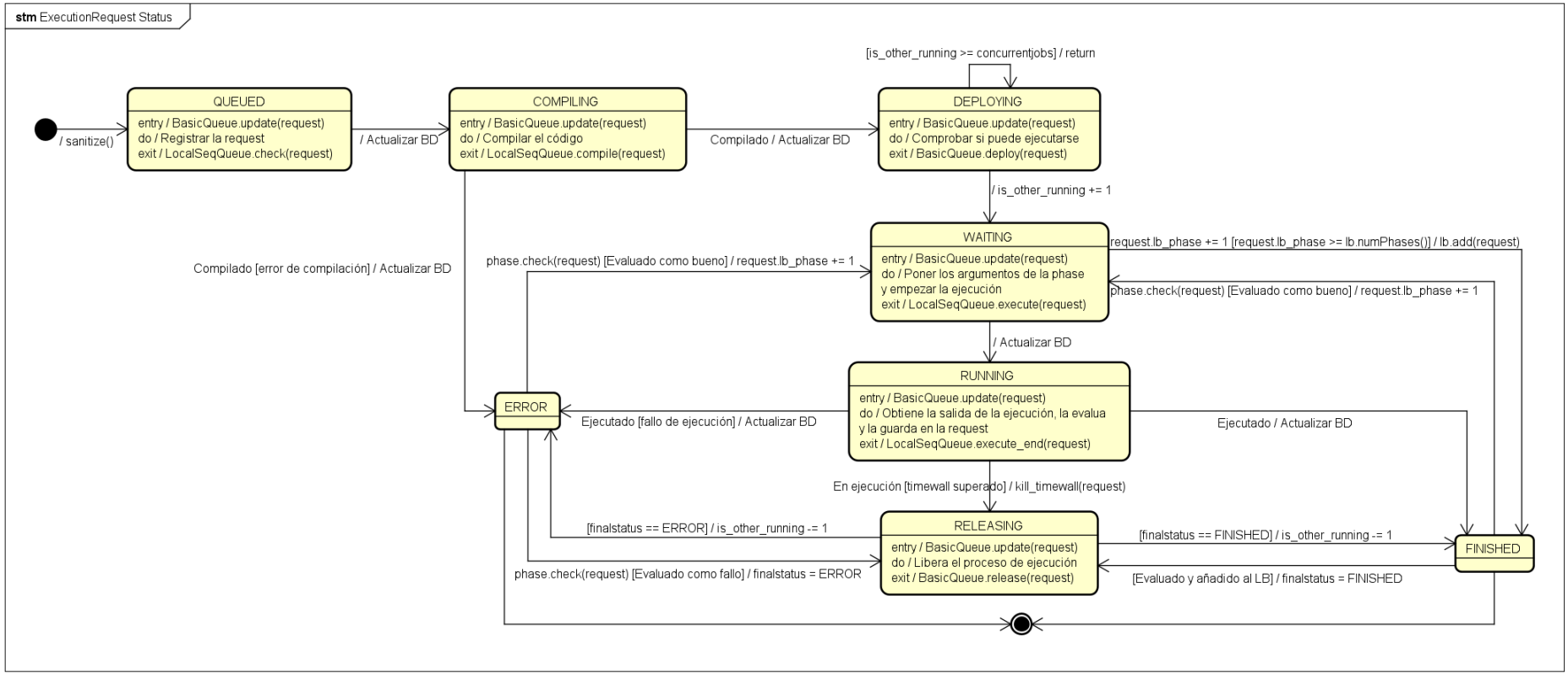


Figura 3.16: Diagrama de máquina de estados para el "Status" de una *request*

Capítulo 4

Conversión del código

Una vez analizado todo el código y redactada una documentación adecuada, se procederá a la conversión del código de Python 2 a Python 3. A priori, este paso no debería ser muy complejo, gracias a la funcionalidad *2to3* [21], que habitualmente viene por defecto con la instalación del intérprete de Python y transforma el código de Python 2 a Python 3. Sin embargo, dada la desorganización y la antigüedad de *Tablon*, es razonable esperar problemas y fallos a solucionar manualmente.

En primer lugar, en este capítulo se hablará de la funcionalidad *2to3* y como se ha utilizado durante la conversión. Posteriormente se tratará de sintetizar los problemas y errores que se han encontrado y las soluciones que se han tomado.

4.1. 2to3 Python

2to3 es un programa de Python que lee código en Python 2 y aplica una serie de soluciones para transformar el código a Python 3. Su ejecución se separa en *fixers* o soluciones de distinto tipo basadas en las principales diferencias entre Python 2 y Python 3. Cuando se ejecuta este programa se va pasando por cada fixer, buscando en el código una sentencia que pueda estar relacionada con dicho fixer, y se aplica una modificación acorde a este. Los distintos fixers se pueden consultar en la documentación del programa [21]. Entre ellos se encuentran algunas de las diferencias entre ambas versiones de Python, explicadas en la Sección 3.1.1, como por ejemplo la sentencia `print`.

Si bien esta herramienta puede ayudar con las diferencias básicas entre Python 2 y Python 3, no tiene compatibilidad con la mayoría de bibliotecas de Python, por lo que habrá que encargarse manualmente de las modificaciones relativas a las distintas versiones de las bibliotecas utilizadas.

A la hora de utilizar el programa, se puede ejecutar para comprobar cual sería la diferencia entre el código inicial y el modificado sin guardar esos cambios, o por el contrario, se pueden guardar esos cambios, junto con una copia del código inicial. Esta última opción se consigue ejecutando el programa con la opción `-w`. Este programa se puede ejecutar tanto con un único fichero de Python 2 como con todos los ficheros de un directorio, de tal manera que realizará las

modificaciones a cada fichero con extensión `.py` que encuentre en el mismo.

El programa *2to3* resuelve algunas de las diferencias mencionadas entre Python 2 y Python 3, pero hay otras diferencias básicas que no resuelve, como la división de dos enteros. Como ya se explicó, esto se puede solucionar cambiando el operador `/` por el operador `//`, que en Python 3 tiene el mismo comportamiento. Este fallo se encontró en varios fragmentos del código de *Tablon*.

Como era de esperar, tras utilizar *2to3* sobre toda la aplicación y ejecutar con Python 3, esta sigue dando fallos. A continuación se describen estos fallos y las soluciones que se han tomado para que funcione en Python 3.

4.2. Problemas tras la conversión

En esta sección se resumen los distintos errores encontrados tras usar el programa *2to3* e intentar ejecutar la aplicación en Python 3. Sin embargo, antes de realizar las pruebas con Python 3, es necesario volver a instalar las bibliotecas que necesita la aplicación, salvo que esta vez serán las versiones de Python 3. En concreto, la nueva versión de *Tablon* utiliza Python 3.6.8 y las siguientes versiones de las siguientes bibliotecas:

- Jinja2 3.0.3 [16]
- mysqlclient 2.0.1 [22]
- pygal 3.0.0 [23]
- six 1.15 [24]
- srp 1.0.19 [25]

Es importante destacar que estas bibliotecas deben instalarse con `pip3`, dado que en caso contrario podrían instalarse en directorio raíz de Python 2, con lo que al ejecutar la aplicación con Python 3 no tendría acceso a esas bibliotecas.

4.2.1. Localización de los fallos y soluciones aplicadas

La cantidad de errores que se han encontrado ha sido bastante alta, por lo que la conversión a Python 3 ha llevado mucho más tiempo del esperado. Sin embargo, en este apartado se trata de mencionar los errores más relevantes y complejos de detectar y solucionar.

Este proceso de localización y solución de fallos se ha llevado a cabo en dos etapas bien distinguibles: arranque del servidor y de la web, y envío de programas como cliente al servidor. Claramente se empezó por la arranque del servidor, ya que si la web no funciona no se puede comprobar si recibe peticiones o no.

Arranque de la web

Para comprobar el arranque del servidor se utilizó el fichero `run.sh`, mientras que para la parada del mismo se utilizó el fichero `stop.sh` (ambos ficheros ya fueron modificados para ejecutar los ficheros con Python 3). Para obtener información de los distintos fallos que iban ocurriendo, se puede consultar el fichero `nohup.out`.

Uno de los primeros errores bloqueantes que se encontraron no mostraba un texto explicativo del fallo en el fichero `nohup.out`. La página web no cargaba, devolviendo el mensaje `ERR_EMPTY_RESPONSE`, por lo que el fallo tenía que situarse en la parte relativa a la puesta en marcha del servidor HTTP. Tras varias pruebas apareció en el fichero `nohup.out` el siguiente texto de error:

```
Exception happened during processing of request from ('139.47.49.50', 59421)
Traceback (most recent call last):
    File "/usr/lib64/python3.6/socketserver.py", line 654, in process_request_thread
        self.finish_request(request, client_address)
    File "/usr/lib64/python3.6/socketserver.py", line 364, in finish_request
        self.RequestHandlerClass(request, client_address, self)
    File "/home/dani/tablonPy3/tablon/frontend.py", line 135, in __init__
        BaseHTTPRequestHandler.__init__(self, request, client_address, server)
    File "/usr/lib64/python3.6/socketserver.py", line 722, in __init__
        self.setup()
    File "/home/dani/tablonPy3/tablon/frontend.py", line 146, in setup
        self.rfile._sock.settimeout(CONFIG.http_timeout)
AttributeError: '_io.BufferedReader' object has no attribute '_sock'
```

Este error indicaba que había un fallo al tratar de especificar el timeout de cualquier petición HTTP, por lo que el timeout se encontraba en su valor por defecto (60 segundos). Es por esto que al principio no se encontró ningún mensaje de error, ya que el error solo aparecía pasado el timeout. En Python 3 el atributo `rfile` de la clase `BaseHTTPRequestHandler` no tiene un atributo `_sock`. La forma correcta de especificar el timeout de una petición HTTP para esa clase es la siguiente:

```
self.request.settimeout(CONFIG.http_timeout)
```

El atributo `request` de la clase `BaseHTTPRequestHandler` es un objeto de la clase `socket`, el cual tiene un atributo llamado `timeout`, y para modificar su valor se usa la función `settimeout` [26]. Tras realizar este cambio en el código del fichero `frontend.py` la página web ya cargaba, y se podían acceder a la mayoría de las opciones de esta.

A la hora de descargar el programa del cliente de la página web ocurría un fallo al tratar de juntar los ficheros `client.py` y `_pysrp.py`. Este fallo se debe a que la lectura de estos ficheros se estaba haciendo con la opción `rb` de la función `open`. En Python 2 esta opción devuelve una objeto de tipo `str`, pero en Python 3 esta opción devuelve un objeto de tipo `bytes`. La solución para esto era cambiar el modo de apertura a `r`, para que devuelva un objeto de tipo `str` [27, 28].

Con este último fallo resuelto la página web funcionaba al completo, y por lo tanto se puede pasar a la siguiente etapa de la corrección de errores.

Envío de programa como cliente

Para esta etapa se necesitará ejecutar el fichero client (descargable desde la página web de *Tablon*) para enviar programas. Sin embargo, este fichero esta comprimido y en formato binario, por lo que cualquier fallo que ocurra en él aparecerá como un texto ilegible. Por ello, esta fase se probará desde dentro de la máquina donde se está ejecutando el servidor, usando el fichero `client.py`. De esta manera se podrá observar los mensajes de error que vayan ocurriendo.

Inicialmente se probó a enviar programas al Leaderboard `q_mars`, ya que la implementación de su cola es más sencilla. Los principales errores de esta etapa se encontraron en los fichero `client.py` y `launchserver.py`, que son los principales encargados del envío y recepción de peticiones.

Como ya se ha explicado, en el fichero `client.py` se usan *buffers* de lectura y escritura para comunicarse con el servidor. Estos buffers se crean a partir de la función `makefile` de la clase `socket`. Esta función ha cambiado sus argumentos de Python 2 a Python 3, de tal manera que los modos se han reducido a: `'r'`, `'w'` y `'b'`; y el parámetro `bufsize` se ha cambiado por `buffering` (véase [29] y [30]). El parámetro `bufsize` determinaba la política del buffer, de tal manera que el valor 0 indicaba que la política a seguir era enviar lo que se escribiera al instante.

La solución lógica a esto sería cambiar el modelo del buffer de escritura a `'w'` e indicar al parámetro `buffering` la política 0. Sin embargo, en Python 3 esta política solo está disponible para el modo binario (`'b'`), el cual no tiene funciones de escritura. Por lo tanto esta no es una solución viable.

Por otra parte, existe una función llamada `flush` [31] para buffers de este tipo y objetos de tipo fichero (el tipo de objeto que devuelve la función `makefile`) que envía al instante todo lo que se haya escrito en el buffer. Por lo tanto, después de cada escritura en el buffer se añade una llamada a la función `flush`, para que así el servidor reciba correctamente los mensajes enviados por el cliente.

Si bien con esta solución gran parte de los problemas en `client.py` se solucionan, el fichero `launchserver.py` continua manipulando erróneamente los buffers de comunicación. La versión de la clase `BaseHTTPRequestHandler` de Python 2 es distinta a la de Python 3, y por ello el envío de mensajes desde `LaunchServer` al cliente no funcionaba correctamente. Por ello se trato de revisar la manera adecuada de enviar mensajes en la clase `BaseHTTPRequestHandler` [32]. En primer lugar, existen tres métodos para realizar la comunicación bajo el protocolo HTTP:

- `send_response`: envía una respuesta con un código del protocolo HTTP junto con una cabecera que contiene información de la conexión. Se usará para informar sobre el inicio de la conexión.
- `send_response_only`: envía una respuesta con un código del protocolo HTTP, pero esta vez sin incluir una cabecera. Se usará para enviar mensajes de retroalimentación sobre el correcto estado de la conexión.

- `send_error`: envía un mensaje de error con un código del protrocolo HTTP, junto con una descripción del error. Se usará para comunicar algunos errores al inicio de la conexión.

A parte de estos métodos, se puede seguir usando `wfile` para mandar mensajes. Sin embargo, la función `write` de este atributo solo acepta objetos de tipo `bytes` en Python 3. Por lo tanto en algunos casos se ha tenido que cambiar el mensaje enviado a formato de bytes.

Además de estos fallos, en `launchserver.py` también ha cambiado el tipo de dato que devuelve la función `readline` para el atributo `refile`. Dado que ahora esta función devuelve un objeto de tipo `bytes`, se ha usado la función `decode` [33] para pasar el objeto a tipo `str`. Se ha utilizado esta función en otras partes de `client.py` y `launchserver.py` en las que se esperaba un objeto de tipo `str` pero se recibía un objeto de tipo `bytes`. De la misma forma, cuando se esperaba un objeto de tipo `bytes` y se recibía un objeto de tipo `str`, se ha usado la función `encode` para pasar el objeto de un tipo a otro.

Como ya se ha explicado, el fichero `_pysrp.py` se encarga de la fase de autenticación del cliente durante el envío de un programa. Este programa es una copia de un fichero de una versión antigua de la biblioteca srp, y dado que causa problemas al usarlo en Python 3, se ha decidido actualizar esta copia con la última versión de la biblioteca srp [34].

El resto de fallos a destacar son relativos al formato de los objetos enviados durante la comunicación, por lo que en la mayoría de casos se han solucionado utilizando las funciones `encode` y `decode` ya mencionadas.

Tras solucionar estos errores, se ha comprobado que es posible enviar peticiones desde dentro de la máquina. Sin embargo, se ha comprobado también si es posible enviar peticiones desde fuera de la máquina, para asegurar que todo funciona correctamente. Al usar el archivo ejecutable `client` (descargado de la página web) se comprobó que no funcionaba. La razón se debía a que durante la creación del archivo se escribía un byte nulo, lo que imposibilita la ejecución del archivo. Para solucionarlo, se añadió una línea después de la creación del contenido del ejecutable en `webpages.py` en la que se remplaza el byte `b'\x00'` por `b''`. De esta manera se eliminaron los bytes nulos del archivo.

Una vez solucionados todos los fallos, se comprobó que la aplicación funcionaba en Python 3 sometiéndola a las mismas pruebas que se describen en la Sección 3.2.3. Tras obtener los mismos resultados, se concluye que la aplicación ya funciona en Python 3. Sin embargo, para comprobar que la aplicación funciona correctamente se realizó una batería de pruebas, explicada en la Sección 4.4.

Para consultar más específicamente los cambios realizados, se puede consultar el código en el repositorio de *GitLab* ya mencionado [15]. El *Tag* `Tablon3-v1.0` muestra el código de la primera versión funcional de *Tablon* en Python 3. Los cambios realizados de una versión a otra están marcados con un comentario explicativo con el formato:

```
# DANIEL py2 -> py3:---
```

4.3. Modificaciones y limpieza del código

Tras obtener una primera versión de *Tablon* en Python 3 funcional, se ha decidido realizar modificaciones mínimas sobre el código, junto con una limpieza del mismo. En este proceso se ha tratado de eliminar funcionalidades obsoletas y comentarios en el código que dificulten la comprensión del mismo. A continuación se enumeran algunas de las modificaciones realizadas:

- **Eliminación de** `mkpasswd.py`: dado que este archivo solo contenía una función, utilizada únicamente por el fichero `useradmin.py`, se ha decidido mover todo el contenido de este archivo al fichero `useradmin.py`. De esta manera se reducen las dependencias, y se elimina el fichero `mkpasswd.py` el cual tenía poco sentido como submódulo del módulo `tablon`. Esto repercute en lo representado en el Diagrama de la Figura 3.7, como se muestra en la Figura 4.1.

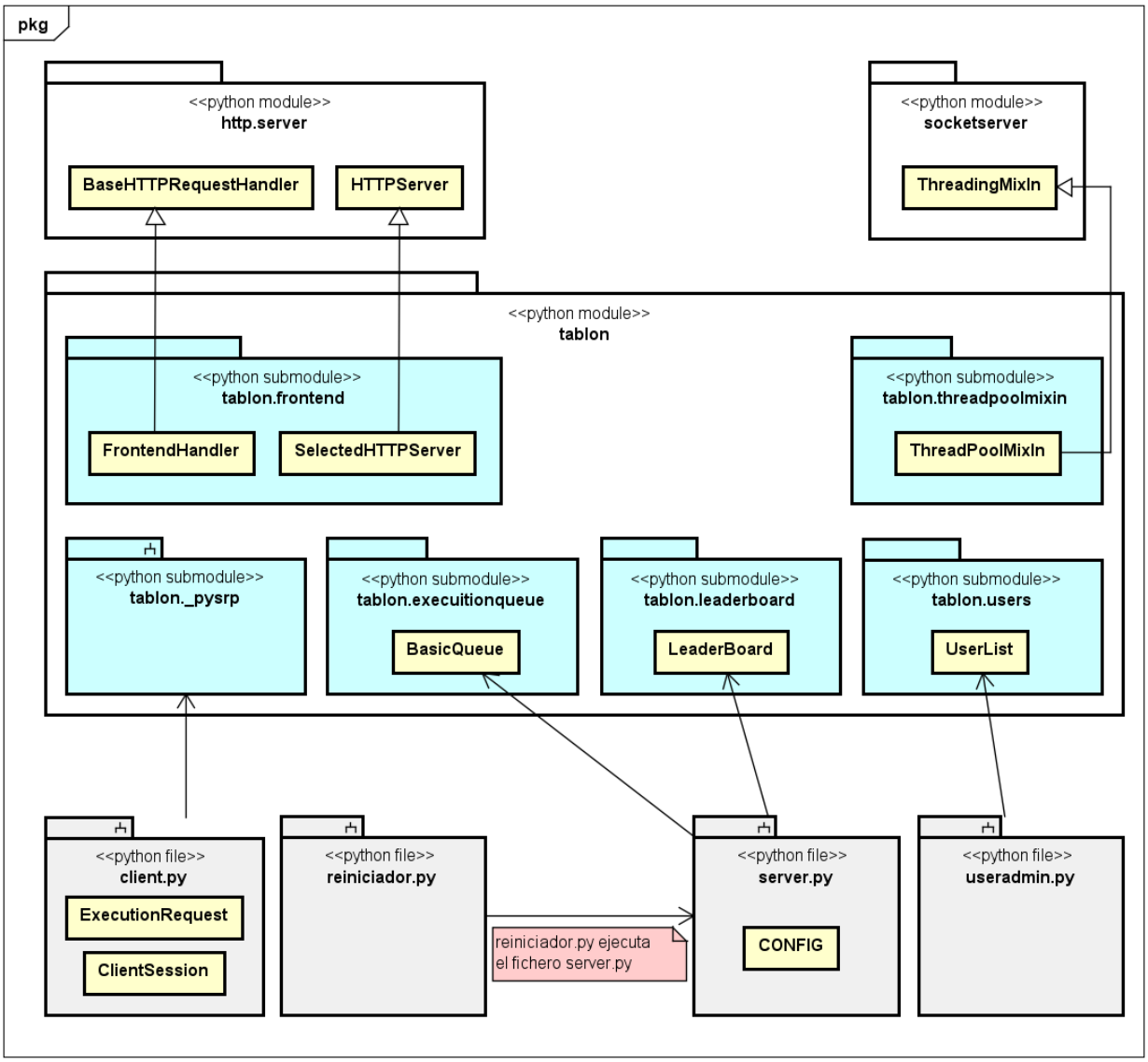


Figura 4.1: Diagrama de módulos y ficheros de la aplicación modificada

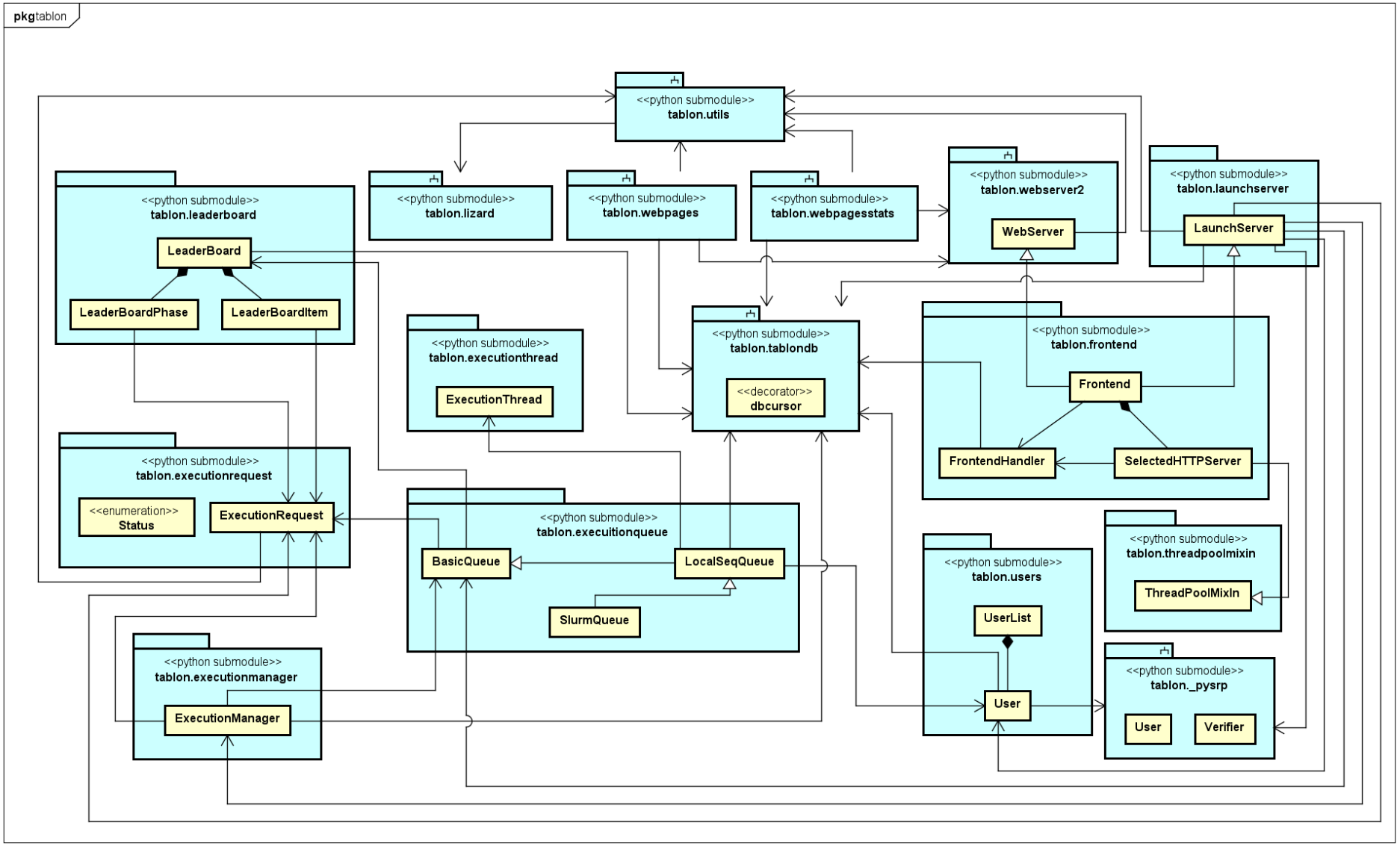


Figura 4.2: Diagrama de los submódulos y clases del módulo `tablon` modificado

- **Sustitución de la biblioteca srp por el fichero** `_pysrp.py`: ya que la aplicación solo usaba la biblioteca srp para la autenticación de un cliente, se puede prescindir de ella utilizando en su defecto el fichero `_pysrp.py`. En la versión de Python 2 ya existía esta opción, ya que la biblioteca srp se importaba mediante el fichero `load_srp.py`, el cual importaba el fichero `_pysrp.py` si la biblioteca srp no estaba instalada. Por ello se ha decidido eliminar el fichero `load_srp.py`, e importa el fichero `_pysrp.py` donde se necesite.

- **Eliminación de código obsoleto**: específicamente se han eliminado varias funciones del fichero `utils.py` y la clase `cached` del mismo. Además se han eliminado tres de los cuatro subtipos de colas de ejecución: `LocalMPIQueue`, `LocalOpenMPQueue` y `LocalMPIOpenMPQueue`.

- **Arreglo de *Bugs***: como ya se destacó en la Sección 3.2.3, cuando una petición entra en estado "releasing" aparece permanentemente en ese estado en la página web, en lugar de pasar a su estado final. Esto se ha solucionado añadiendo una llamada a la función `update_request(request)` de `tablondb`, en la función `release()` de `BasicQueue`, después de modificar el estado de la petición a su estado final.

Varias de estas modificaciones han provocado cambios en el diagrama que se muestra en la Figura 3.8, como se puede observar en la Figura 4.2. Con estos cambios el diagrama ahora es mucho menos enrevesado que en la versión de Python 2. Además de estas modificaciones, se han eliminado algunos trozos de código comentado que dificultaban la lectura del mismo. Se puede consultar el código de la aplicación con estas últimas modificaciones en el repositorio de *GitLab* ya mencionado [15], con el *Tag* `Tablon3-v2.0`.

4.4. Batería de pruebas

Como ya se ha explicado la aplicación *Tablon* se utiliza para la evaluación de prácticas de informática de alumnos, utilizando un sistema de puntuaciones que ordena los programas enviados según esta. Por lo tanto es de asumir que la aplicación se encontrará a menudo con situaciones en las que dos o más alumnos envíen un programa al mismo tiempo o con una diferencia de tiempo de segundos.

Es por esto que además del tipo de pruebas que se realizó en la Sección 3.2.3, se debe comprobar como responde la aplicación ante la recepción de varias peticiones concurrentes. Para este propósito se han creado tres usuarios, a mayores de los tres ya creados anteriormente: *u4*, *u5* y *u6*. Ya solo con esta acción se detectó un fallo en el fichero `users.py` que impedía completamente el uso del fichero `useradmin.py`. Tras analizar el error se concluyó que era un error de tipado acerca de la salida de una función cuya implementación había cambiado de Python 2 a Python 3.

Para estas pruebas se utilizarán las colas y Leaderboards definidos en la Sección 3.2.1. Sin embargo, en algunos casos se cambiará el número de peticiones concurrentes que puede manejar la cola (modificando el parámetro `concurrentjobs`) o el tiempo máximo de ejecución de un programa (modificando el parámetro `timewall`). Estos detalles y otros se definen en cada caso de prueba.

4.4.1. Casos de prueba

Para estas pruebas se utilizarán los archivos ya descritos en la Sección 3.2.3, exceptuando el archivo `exec_error.asm` que contenía un bucle infinito, de tal manera que la petición acabase en error por superar el `timewall` especificado. A continuación se muestra una tabla resumen de los programas de ejemplo que se usará:

Nombre	Cola destinataria	Descripción
`prac1.asm`	`mars`	Programa que supera todos los casos obteniendo la puntuación máxima.
`comp_error.asm`	`mars`	Programa vacío que falla durante la compilación.
`prac1.c`	`openmp`	Programa que supera todos los casos obteniendo la puntuación máxima.
`comp_error.c`	`openmp`	Programa vacío que falla durante la compilación.
`exec_error.c`	`openmp`	Programa que contiene un "Hola mundo" que falla durante la ejecución.

Tabla 4.1: Lista de programas para los casos de prueba

Se ha decidido prescindir del fichero `exec_error.asm`, ya que durante estas pruebas se detecto un error que producía procesos *zombie* al enviar programas con un bucle infinito a la cola `mars`. Estos procesos se pueden eliminar usando el fichero `kill_command_sandbox.sh` una vez el servidor esté apagado. Dado que no se ha encontrado el origen real del problema, se ha considerado más adecuado prescindir de este fichero.

Para simular errores por sobrepasar el `timewall` se utilizará los archivos `prac1.asm` y `prac1.c` y se reducirá el parámetro `timewall`, de tal manera que no de tiempo a terminar de ejecutar todos los casos a los programas. Como ya se ha explicado, cada caso de prueba simula el envío de uno, dos, tres o más programas de forma concurrente, de tal manera que en las siguientes tablas se especifica que usuario envía que programa en cada caso de prueba. Se utiliza `tw.` y `cj.` para referirse a los parámetros `timewall` y `concurrentjobs` respectivamente.

ID	Envío	`tw.`	`cj.`	Estado final esperado	Comentario
CP01	`u1:prac1`	20	1	finished - Program passed	
CP02	`u2:comp_error`	20	1	error - Compile error	
CP03	`u3:prac1`	10	1	error - Timewall reached	

CP04	`u1:prac1`
	`u2:comp_error`
	`u3:prac1`
		20	1	finished - Program passed
				error - Compile error
				finished - Program passed
					Se ejecuta uno y el resto se quedan en estado *deploying* hasta que termina.

CP05	`u1:prac1`
	`u2:prac1`
	`u3:prac1`
		10	1	error - Timewall reached
				error - Timewall reached
				error - Timewall reached
					Se ejecuta uno y el resto se quedan en estado *deploying* hasta que termina.

CP06	`u1:prac1`
	`u2:comp_error`
	`u3:prac1`
	`u4:comp_error`
	`u5:prac1`
	`u6:comp_error`
		20	3	finished - Program passed
				error - Compile error
				finished - Program passed
				error - Compile error
				finished - Program passed
				error - Compile error
					Se ejecutan tres y el resto se quedan en estado *deploying* hasta que termina alguno.

CP07	`u1:prac1`
	`u2:prac1`
	`u3:prac1`
	`u4:prac1`
	`u5:prac1`
	`u6:prac1`
		10	3	error - Timewall reached
				error - Timewall reached
				error - Timewall reached
				error - Timewall reached
				error - Timewall reached
				error - Timewall reached
					Se ejecutan tres y el resto se quedan en estado *deploying* hasta que termina alguno.

Tabla 4.2: Casos de prueba para el Leaderboard `lb_practica1`

ID	Envío	*tw.*	*cj.*	Estado final esperado	Comentario
CP08	*u1:prac1*	40	1	finished - Program passed	
CP09	*u2:comp_error*	40	1	error - Compile error	
CP10	*u3:exec_error*	40	1	error - Error in result	
CP11	*u4:prac1*	10	1	error - Timewall reached	
CP12	*u1:prac1* *u2:comp_error* *u3:exec_error*	40	1	finished - Program passed error - Compile error error - Error in result	Se ejecuta uno y el resto se quedan en estado deploying hasta que termina.
CP13	*u1:prac1* *u2:prac1* *u3:prac1*	10	1	error - Timewall reached error - Timewall reached error - Timewall reached	Se ejecuta uno y el resto se quedan en estado deploying hasta que termina.
CP14	*u1:prac1* *u2:comp_error* *u3:exec_error* *u4:prac1* *u5:comp_error* *u6:exec_error*	40	3	finished - Program passed error - Compile error error - Error in result finished - Program passed error - Compile error error - Error in result	Se ejecutan tres y el resto se quedan en estado deploying hasta que termina alguno.
CP15	*u1:prac1* *u2:prac1* *u3:prac1* *u4:prac1* *u5:prac1* *u6:prac1*	10	3	error - Timewall reached error - Timewall reached error - Timewall reached error - Timewall reached error - Timewall reached error - Timewall reached	Se ejecutan tres y el resto se quedan en estado deploying hasta que termina alguno.

Tabla 4.3: Casos de prueba para el Leaderboard `openmplb`

4.4.2. Resultados de las pruebas

Tras ejecutar cada caso de pruebas se obtuvieron los resultados expuestos en la Tabla 4.4, de tal manera que aquellos casos de prueba que terminaron en fallo han sido analizados y arreglados como se explica en la columna de Solución.

Prueba	Resultado	Solución
CP01	OK	
CP02	OK	
CP03	Fallo: se queda en estado releasing y no pasa a estado error.	Actualizar correctamente el estado de la base de datos tras hacer el release.
CP04	OK	
CP05	Fallo: no se liberan correctamente los recursos, un programa acaba y el resto permanecen en estado deploying.	Hacer que se llame correctamente a la función release cuando se supera el timewall.
CP06	OK	
CP07	OK	
CP08	Fallo: el valor de la columna Time se calcula erróneamente y el programa no termina de ejecutarse.	Arreglar una lectura de la salida del proceso en executionthread.py.
CP09	OK	
CP11	OK	
CP12	OK	
CP13	OK	
CP14	OK	
CP15	OK	

Tabla 4.4: Resultados de los casos de prueba

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

Como resultado del proyecto aquí presentado se ha obtenido una versión completamente funcional de la herramienta *Tablon* en el lenguaje de programación Python 3, lo cual facilitará su utilización y actualización en futuras ocasiones. Además, se ha redactado una documentación extensa y bastante sencilla para comprender el funcionamiento de la aplicación, para que cualquier persona pueda utilizar *Tablon* o realizar modificaciones en su código de manera más sencilla. Esta documentación se ha complementado con diagramas que detallan la estructura interna, facilitando así la comprensión de la estructura de la aplicación.

También se ha tratado de simular el tránsito de peticiones propio de un concurso de programación a la herramienta mediante los casos de prueba planteados, sacando a la luz algunos fallos que se han podido corregir, por lo que gran parte de la depuración de *Tablon* se ha completado en este proyecto.

Con todo el trabajo aquí realizado, se podría presentar al público una primera versión de *Tablon3* (versión de *Tablon* para Python 3) junto con las documentación aquí desarrollada.

5.2. Líneas futuras

Si bien en este trabajo se ha conseguido completar la conversión del código de *Tablon* a Python 3 y la implementación de algunas modificaciones simples, durante la redacción de la documentación se ha podido comprobar algunas partes de la aplicación que se pueden mejorar. Por lo tanto, sería adecuado realizar un análisis posterior de *Tablon3*, en el que se detecten las partes más desorganizadas y las mejoras que podrían realizarse ahora que está programado en Python 3.

Como ejemplo, se enumeran a continuación algunas de las mejoras que podrían realizarse en la aplicación:

- Reestructurar la transición entre estados de una *request*, ya que la interpretación del estado de una petición y de la transición entre estados no resulta intuitiva.
- Mejorar la forma de crear, acceder y manejar a la base de datos, automatizando en mayor medida estos procesos.
- Modificar la manera en la que se realiza la puesta en marcha del servidor, ya que actualmente requiere de mucha configuración manual. De cara al uso de esta herramienta por parte de personas poco familiarizadas con su funcionamiento, puede ser interesante hacer un interfaz de usuario para configurar y poner en marcha el servidor.
- Cambiar la forma en la que se crea y se gestiona la página web. Se puede utilizar para esto el framework de desarrollo web django [35] que es bastante popular actualmente.

Junto con las modificaciones y mejoras que se apliquen se debería realizar el añadido de explicaciones sobre estos cambios en la documentación aquí recogida.

Bibliografía

[1] S. Deterding, D. Dixon, R. Khaled, and L. Nacke. From game design elements to gamefulness: defining gamification. In *15th International Academic MindTrek Conference: Envisioning Future Media Environments*, pages 9–15. ACM, 2011.

[2] S. Arnab, T. Lim, M. B. Carvalho, F. Bellotti, S. Freitas, S. Louchart, and et al. Mapping learning and game mechanics for serious games analysis. *British Journal of Educational Technology*, 46(2):391–411, 2014.

[3] L. Hakulinen, T. Auvinen, and A. Korhonen. Empirical study on the effect of achievement badges in trakla2 online learning environment. In *Learning and Teaching in Computing and Engineering (LaTiCE)*, pages 47–54. IEEE, 2013.

[4] C. I. Muntean. Raising engagement in e-learning through gamification. In *6th International Conference on Virtual Learning ICVL*, pages 323–329, 2011.

[5] Grupo Trasgo. Trasgo research group (universidad de valladolid). Página de acceso: `https://trasgo.infor.uva.es/`, último acceso: Junio 2022.

[6] Grupo Trasgo. Tablon: A tool to introduce gamification methodologies in computer science courses, 2016/2017. Página de acceso: `https://trasgo.infor.uva.es/tablon/`, último acceso: Junio 2022.

[7] Fresno Javier, Hector Ortega-Arranz, Alejandro Ortega-Arranz, Arturo Gonzalez-Escribano, and Diego R. Llanos. Applying gamification in a parallel programming course. In *Gamification-Based E-Learning Strategies for Computer Programming Education. edited by Alexandre Peixoto de Queirós, Ricardo, and Mário Teixeira Pinto, 106-130. Hershey, PA: IGI Global*, 2017. Página de acceso: `https://doi.org/10.4018/978-1-5225-1034-5.ch006`, último acceso: Junio 2022.

[8] A. Gonzalez-Escribano, V. Lara-Mongil, E. Rodriguez-Gutiez, and Y. Torres. Toward improving collaborative behaviour during competitive programming assignments. In *2019 ACM/IEEE Workshop on Education for High Performance Computing (EduHPC)*, pages 68–74. IEEE Press, 2019.

[9] Francisco J. Andújar, Arturo González-Escribano, Javier Bastida, and Yuri Torres de la Sierra. Aplicación de gamificación competitiva y colaborativa en asignaturas básicas de arquitectura de computadoras. In *Actas de las XX Jornadas sobre Enseñanza Universitaria de la Informática (JENUI 2020)*, volume 5, pages 85–92, Junio 2020.

[10] Onlinegantt: Free online gantt chart software, 2016/2017. Página de acceso: `https://www.onlinegantt.com/#/gantt`, último acceso: Junio 2022.

[11] Grupo Trasgo. Computational resources - grupo trasgo, 2016/2017. Página de acceso: `https://trasgo.infor.uva.es/computational-resources/`, último acceso: Junio 2022.

[12] Sebastian Raschka. The key differences between python 2.7.x and python 3.x with examples. Junio 2014. Página de acceso: `https://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html`, último acceso: Junio 2022.

[13] R. Johnson E. Gamma, R. Helm and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, Marzo 2009.

[14] Bernd Klein. Python-course.eu, decorators and decoration, Mayo 2022. Página de acceso: `https://python-course.eu/advanced-python/decorators-decoration.php`, último acceso: Junio 2022.

[15] Tablon3: repositorio de gitlab para el proyecto tablon3. Página de acceso: `https://repo.infor.uva.es/fandujarm/Tablon3`, último acceso: Junio 2022.

[16] The Pallets Projects. Jinja2 python página web oficial. Página de acceso: `https://palletsprojects.com/p/jinja/`, último acceso: Junio 2022.

[17] Tom Cocagne. Código de la biblioteca srp de python en github. Versión 1.0.4. Página de acceso: `https://github.com/cocagne/pysrp/tree/1.0.4`, último acceso: Junio 2022.

[18] Terry Yin. Código de la biblioteca lizard de python en github. Página de acceso: `https://github.com/terryyin/lizard`, último acceso: Junio 2022.

[19] The Python Software Foundation. Python sys.modules. documentación del atributo modules de la biblioteca sys de python 3. Página de acceso: `https://docs.python.org/3/library/sys.html#sys.modules`, último acceso: Junio 2022.

[20] The Python Software Foundation. Python threading.lock. documentación de la clase lock de la biblioteca threading de python 3. Página de acceso: `https://docs.python.org/3/library/threading.html#lock-objects`, último acceso: Junio 2022.

[21] The Python Software Foundation. Python 2to3. documentación del programa de python 2to3. Página de acceso: `https://docs.python.org/es/3.8/library/2to3.html`, último acceso: Junio 2022.

[22] Inada Naoki. Read the docs mysqlclient. documentación de la biblioteca mysqlclient de python. Página de acceso: `https://mysqlclient.readthedocs.io/`, último acceso: Junio 2022.

[23] Kozea. pygal library. documentación de la biblioteca pygal de python. Página de acceso: `https://www.pygal.org/en/stable/`, último acceso: Junio 2022.

[24] Benjamin Peterson. Read the docs six. documentación de la biblioteca six de python. Página de acceso: `https://six.readthedocs.io/`, último acceso: Junio 2022.

[25] Tom Cocagne. srp library. documentación de la biblioteca srp de python. Página de acceso: `https://pythonhosted.org/srp/srp.html`, último acceso: Junio 2022.

[26] The Python Software Foundation. Python 3 socket. documentación de la función settimeout de la clase socket en python 3. Página de acceso: `https://docs.python.org/3.6/library/socket.html#socket.socket.settimeout`, último acceso: Junio 2022.

[27] The Python Software Foundation. Python 2 built-in-functions. documentación de la función open en python 2. Página de acceso: `https://cpython-test-docs.readthedocs.io/en/latest/library/functions.html#open`, último acceso: Junio 2022.

[28] The Python Software Foundation. Python 3 built-in-functions. documentación de la función open en python 3. Página de acceso: `https://docs.python.org/3.6/library/functions.html#open`, último acceso: Junio 2022.

[29] The Python Software Foundation. Python 2 socket. documentación de la función makefile de la clase socket en python 2. Página de acceso: `https://docs.python.org/2.7/library/socket.html#socket.socket.makefile`, último acceso: Junio 2022.

[30] The Python Software Foundation. Python 3 socket. documentación de la función makefile de la clase socket en python 3. Página de acceso: `https://docs.python.org/3.6/library/socket.html#socket.socket.makefile`, último acceso: Junio 2022.

[31] The Python Software Foundation. Python 3 flush. documentación de la función flush en python 3. Página de acceso: `https://docs.python.org/3.6/library/io.html#io.IOBase.flush`, último acceso: Junio 2022.

[32] The Python Software Foundation. Python 3 http.server. documentación de la clase basehttprequesthandler en python 3. Página de acceso: `https://docs.python.org/3.6/library/http.server.html#http.server.BaseHTTPRequestHandler`, último acceso: Junio 2022.

[33] The Python Software Foundation. Python 3 codecs. documentación de la función decode en python 3. Página de acceso: `https://docs.python.org/3.6/library/codecs.html#codecs.decode`, último acceso: Junio 2022.

[34] Tom Cocagne. Python 3 codecs. código de la biblioteca srp de python en github. Versión 1.0.19. Página de acceso: `https://github.com/cocagne/pysrp/tree/1.0.19`, último acceso: Junio 2022.

[35] Django Software Foundation. Django: The web framework for perfectionists with deadlines. Página de acceso: `https://www.djangoproject.com/`, último acceso: Junio 2022.