Trabajo de Final de Grado

Grado en Ingeniería en Tecnologías Industriales

Herramienta de simulación de colas para dimensionar puntos de servicio

MEMORIA

Autor: David Caballero Flores
Director: Pere Grima Cintas

Convocatoria: Septiembre, 2016

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Lo importante es no dejar de hacerse preguntas.

Albert Einstein

Resumen

Este proyecto tiene por objetivo el desarrollo de un programa informático de código libre, con interfaz de usuario y capaz de modelar y simular de forma flexible un sistema de colas. El móvil del programa es poder valorar el comportamiento de un punto de servicio frente a la llegada de usuarios y, estudiar mediante simulaciones y extracciones de datos, si es más o menos óptimo que otras configuraciones del mismo sistema.
Para su desarrollo se ha utilizado el lenguaje de programación *Python* y, entre otros, el módulo *SimPy* para construir el gestor de la simulación y *QtDesigner* para el diseño de la interfaz de usuario.
SimuQ 1.0 se convierte, al final del proyecto, en una herramienta versátil a la hora de definir sistemas de colas que modelan puntos de servicio. Es capaz de simular el comportamiento del sistema e incluso mostrar la simulación por pantalla para casos poco complejos. Tras la simulación se generan archivos de datos que permiten tomar decisiones y valorar la eficiencia del sistema.

Por ser una primera versión y formar parte de un proyecto de final de carrera, con calendario de entrega y presentación, se han limitado las líneas de desarrollo. Sin embargo, el hecho de ser de código libre, permite que el programa se siga desarrollando o se pueda modificar para un uso específico de sus funciones.
En este texto y en el propio código se puede encontrar la información necesaria para comprender el funcionamiento de *SimuQ 1.0*.

Índice general

Resumen III

Lista de figuras VI

1. Prefacio 1

1.1. Contexto del proyecto 1

1.2. Antecedentes 2

1.3. Motivación y justificación 3

2. Introducción 5

2.1. Objetivos 5

2.2. Alcance y limitaciones 6

3. Elementos y características de un sistema de colas 7

3.1. Elementos de un sistema de colas 7

3.1.1. Usuarios 7

3.1.2. Ventanas de servicio 8

3.1.3. Cola 8

3.1.4. Reloj 8

3.2. Características de un sistema de colas 8

3.2.1. Patrón de llegada de los usuarios 9

3.2.2. Patrón de servicio de las ventanas 10

3.2.3. Disciplina de cola 11

3.2.4. Capacidad del sistema 12

3.2.5. Número de puntos de servicio por ventana o servidores en paralelo 12

3.2.6. Número de etapas de servicio 12

4. Herramientas para el desarrollo del programa **15**

4.1. Lenguaje de programación y entorno de desarrollo 15

4.2. Módulos y librerías 16

4.3. Constructor de interfaz 18

4.4. Compilador 18

5. Desarrollo de *SimuQ 1.0* **19**

5.1. Definición del sistema 19

5.1.1. Nombrar ítems 19

5.1.2. Ventanas de servicio 20

5.1.3. Usuarios, definición de las llegadas 23

5.1.4. Configuración de la simulación 25

5.1.5. Funciones y variables esenciales 27

5.2. Ejecución de la simulación 28

5.2.1. Elementos del sistema 28

5.2.2. Funciones y métodos principales 31

5.2.3. Síntesis de la simulación gráfica 36

5.3. Visualización de datos 38

6. Resultados **41**

6.1. Alcance de objetivos 41

6.2. Diagramas de flujo 42

6.3. Relación de errores 42

7. Planificación temporal y costes **47**

Conclusiones y líneas futuras **49**

Agradecimientos **51**

Bibliografía **53**

Bibliografía complementaria **55**

Índice de figuras

3.1. Ejemplo de función de densidad de una distribución Normal. Fuente: *Wikipedia*. 11
3.2. Esquema de cola monocanal y multicanal. Extraída de [2]. 12

4.1. Logo de *Python*. 16
4.2. Logo de *Spyder*. 16

5.1. Esquema de diferentes tipos de ventana definibles con *SimuQ 1.0*. 21
5.2. Esquema de definición de patrón de llegada de usuarios con *SimuQ 1.0*. 24
5.3. Períodos de llegadas de usuarios de 200 cada 500 con un tiempo de simulación de 1800. 26
5.4. Código para crear el entorno de simulación con *SimPy*. Extraído de *intensive_simulation.py*. 29
5.5. Código de ejemplo para lanzar un evento de petición de una ventana de servicio (*windows*) con prioridad *prio*. 30
5.6. Expresión para poner a la cola un evento de avanzar el reloj un tiempo de (*t_atention_A+dead_time_A*). Código extraído de *intensive_simulation.py*. 31
5.7. Código para lanzar la simulación mediante la función *user_generator*. Extraído de *intensive_simulation.py*. 32
5.8. Diagrama de flujo de la función *choose_window(user)*. 33
5.9. Código para comparar parámetros entre opción de ir a dos ventanas o a una que ofrezca dos servicios. Extraído de *intensive_simulation.py*. 35
5.10. Diagrama de flujo del método *param* de la clase *Windows*. 35
5.11. Código para recalcular el parámetro *param* de una ventana con prioridad frente a un usuario no prioritario. 36
5.12. Ejemplo simulación gráfica. 37
5.13. Ejemplo de gráfico interactivo de evolución temporal de colas. 39

6.1. Diagrama de flujo de la interfaz de SimuQ 1.0. 43

ETSEIB

6.2. Diagrama de flujo del proceso de simulación intensiva de SimuQ 1.0. 44
6.3. Árbol de fallos de *SimuQ 1.0*. 45

7.1. Diagrama de Gantt del desarrollo del proyecto. 47

Capítulo 1

Prefacio

1.1. Contexto del proyecto

Seamos más o menos impacientes, todos tenemos asimilado que, aunque no nos guste, en muchas ocasiones se ha de esperar haciendo cola. Esperamos si tenemos que ir a urgencias médicas, en un peaje, en el banco, en la caja de un supermercado, al ser atendidos por un operador telefónico o, incluso al ir al baño en una discoteca.

Si no nos gusta esperar y no nos interesa que nuestros clientes o usuarios esperen, ¿por qué existen las colas? La respuesta es muy sencilla; existe lo que llamamos capacidad de servicio y capacidad demandada. Resulta fácil entender que, cuando la capacidad de servicio es superada por la demanda, se forma una cola de los clientes que están dispuestos a esperar. Dado el compromiso existente entre la capacidad de servicio y el coste de aumentarla, existe un punto óptimo que sitúa la capacidad de servicio por debajo de los picos de capacidad demandada.

El presente proyecto se desarrolla 107 años después de que se publicara el primer artículo acerca de la Teoría de Colas[1]. Hoy en día, esta herramienta nos aporta soluciones que nos permiten dimensionar puntos de servicio evitando aglomeraciones masivas en las colas. Sin embargo, las fórmulas analíticas en las que se basa la Teoría, se encuentran limitadas por suposiciones matemáticas que típicamente no se ajustan a la realidad que se quiere estudiar. Por este motivo, para estudiar casos complejos con muchas variables, recurrimos a una herramienta muy potente; la simulación computacional.
La elevada velocidad de cálculo que ofrecen a día de hoy los ordenadores permite desarrollar programas de simulación para sistemas muy complejos. Tanto es así, que existen simuladores

[1] Fue Agner Krarup Erlang quien, en 1909, publicó el primer artículo sobre la Teoría de Colas. Extraído de [3].

de sistemas físicos de transferencia de calor, resistencia a esfuerzos, comportamiento de fluidos y, en definitiva, sistemas cuyo estado cambia de forma continua a lo largo del tiempo.
Por otro lado, existen sistemas que se pueden simular por eventos discretos, es decir, sistemas cuyas variables de interés cambian en instantes concretos de tiempo. Así es como, por ejemplo, el número de personas que está esperando en una cola se mantiene constante entre eventos (la llegada o salida de un elemento de la cola).

1.2. Antecedentes

Ya se ha comentado con anterioridad el extenso trabajo que se ha hecho en el campo de la simulación por ordenador. Si nos centramos en analizar el estado del arte de la simulación de sistemas por eventos discretos y, concretamente, en procesos con formación de colas frente a un servicio limitado (en tiempo), encontramos igualmente mucho trabajo desarrollado.
Existen en el mercado programas informáticos que nos permiten la simulación de sistemas de colas. Se citan y se comentan a continuación los más importantes:

- **Arena®**, actualmente de *Rockwell Software Inc.*, es uno de los software más difundido por su precio respecto a los demás (entre 800 y 1000 dólares). Se presenta como una herramienta muy versátil y 'orientada al proceso' en la que el usuario define el sistema mediante diagramas de flujo. Pese a que el usuario define el sistema orientándolo a un proceso, *Arena*® ejecuta la simulación mediante eventos discretos y permite ver animaciones dinámicas de la simulación, incluso en 3D.

- **Promodel**, propiedad de *Promodel Corporation.* Es un software muy amplio y genérico que ofrece la posibilidad de simular una gran cantidad de sistemas, entre ellos de colas. Su precio es bastante más elevado que el de *Arena*®.

- **FlexSim**, **AutoMod**, **Analytica**, **AnyLogic** son un tipo de software más específico y, pese a que podrían ser igualmente válidos para simular sistemas de formación de colas, se trata de programas muy caros enfocados a grandes empresas especializadas.

- **WinQSB** Es un programa gratuito, de código libre. Dispone de diferentes módulos que permiten abordar distintos tipos de problemas de simulación. Pese a que su interfaz es relativamente sencilla, en algunas situaciones resulta poco intuitiva.
  Además, WinQSB se trata de un programa antiguo que presenta problemas de compatibilidad con los nuevos sistemas operativos de Windows.

A modo de síntesis, el uso de todos los programas citados anteriormente, a excepción de WinQSB, está restringido o acotado a menos que se compre la licencia que, en cualquiera de los casos supera los 800 o 1000 dólares.

Si se opta, como se ha hecho en este proyecto, por el *free software* (programas de código libre), resulta obligado citar *SimPy*. Se trata de un módulo basado en el lenguaje de programación *Python* y desarrollado inicialmente por Klaus Müller y Tony Vignaux, según su web [4], que posteriormente ha sido mejorado por sus colaboradores Stefan Scherfke y Ontje Lünsdorf y, actualmente, se trata de una herramienta casi imprescindible para gestionar eventos en la simulación con *Python*.
Con el uso del módulo *SimPy*, por ejemplo, Pedro Rafael Bohórquez [1] ha realizado una librería para ayudar al modelado y simulación de procesos productivos y de prestación de servicios.

1.3. Motivación y justificación

Visto el gran abanico de programas existentes en el mercado, parece difícil pensar que existe la necesidad de desarrollar una nueva herramienta de simulación. Sin embargo, si resaltamos los precios de los software anteriormente citados, a excepción de WinQSB, no existen herramientas que permitan llevar a cabo simulaciones de manera sencilla y gratuita.

Nótese pues, la falta de una herramienta de simulación adaptada a los nuevos sistemas operativos de Windows, que permita registrar y extraer los datos que paso a paso genera la simulación; datos vinculados a cada uno de los usuarios del sistema, y que, además, pueda servir como base o programa BETA para que otras personas la puedan mejorar tanto resolviendo posibles errores, aumentando su rendimiento o incluso añadiendo nuevas funcionalidades. Esto, junto al evidente empuje que tiene el software libre, en gran parte gracias al enorme potencial de poder ser modificado y mejorado por cualquier usuario, es la motivación que empuja el proyecto a desarrollar un programa informático que ayude a tomar decisiones en la etapa de dimensionamiento de puntos de servicio.
El proyecto se alienta, además, del querer desarrollar un programa con una interfaz sencilla e intuitiva, capaz de ser usado por usuarios mínimamente introducidos en el mundo de la simulación y dotado de la posibilidad de ver gráficamente y a tiempo real lo que ocurre mientras se ejecuta la simulación en el ordenador.

Capítulo 2

Introducción

En esta memoria se detallan las justificaciones, el proceso, las herramientas, la toma de decisiones y el conjunto de explicaciones para entender el desarrollo de *SimuQ 1.0*. Además, en los apéndices A y B se añaden el manual de utilización y casos ejemplificados, respectivamente.
La primera de las justificaciones, se da en la propia introducción y hace referencia al porqué del nombre del programa. Asumido está que la simulación de colas es la esencia de la herramienta que se está desarrollando. Dada la semántica de la palabra inglesa *queue* que se traduce como cola y que, además, tiene la misma fonética que la letra *Q*, en el mismo idioma; se juega con la unión de *Simulation* y *queue*, derivando en *SimuQ 1.0*.
Cabe resaltar que, aunque a lo largo del texto se ejemplifiquen sistemas más bien cotidianos, los mismos métodos y técnicas, así como la herramienta desarrollada, son válidos para dimensionar máquinas que procesan piezas, dispositivos que reciben paquetes de datos y cualquier sistema que se pueda modelar como elementos que esperan un servicio y elementos con capacidad limitada para ofrecerlo.

2.1. Objetivos

El objetivo primario del proyecto es desarrollar una herramienta de simulación asistida por ordenador con la que poder llevar a cabo simulaciones versátiles de sistemas de colas. Con ello, poder evaluar el rendimiento de la configuración simulada y determinar así, si las dimensiones del sistema son, o no, óptimas respecto a otra configuración.
Se trata de un programa informático que permita simular y extraer los datos de la simulación de un sistema cualquiera definido a partir de los tipos de servicio que ofrece, los tipos de usuario que acuden al sistema y la frecuencia con la que lo hacen, número de puntos de servicio, tiempos de atención, etc.

Un objetivo a añadir junto al desarrollo del programa es que este ofrezca la opción de, además de lo expuesto anteriormente, visualizar por pantalla la simulación de forma gráfica al mismo tiempo que el programa la ejecuta. De esta manera dar un valor añadido al programa y facilitar la comprensión de lo que ocurre internamente cuando el programa lleva a cabo la simulación.

Por último, con el fin de poder ser editado y mejorado por otros usuarios en caso de ser menester, se escribirá y se comentará el código del programa en lengua inglesa por ser la más extendida en el lenguaje de programación.

2.2. Alcance y limitaciones

Siempre que se habla de una simulación computacional, se debe pensar que antes de introducir el modelo en el ordenador, ha habido un trabajo previo en el que se han determinado los modelos matemáticos y los parámetros más significativos para definir el sistema simulado con la mayor exactitud posible. Pese a que existen tipos de sistemas que se modelan casi a la perfección con parámetros fácilmente calculables, es evidente que están sujetos a aleatoriedad y a una gran cantidad de factores que los pueden alterar. Sin embargo, parte de la esencia de un modelo es obviar y simplificar todos esos factores poco significativos en el comportamiento del sistema.

A la hora de desarrollar un programa que reciba estos factores, parámetros y variables que definen el sistema modelado para posteriormente ejecutar la simulación, se deben definir tanto el alcance como las limitaciones para evitar abordar situaciones que se escapen de la línea de trabajo o que puedan traducirse en un tiempo de programación inadmisible.

En el caso de *SimuQ 1.0* se ha optado por desarrollar el programa atendiendo a satisfacer el alcance que se propone en los objetivos y cerrar líneas de desarrollo a medida que su complicación sea tal que existan limitaciones, bien temporales o conceptuales. Todas ellas quedan reflejadas en apartados posteriores, en los que se detalla el funcionamiento de *SimuQ 1.0*.

Capítulo 3

Elementos y características de un sistema de colas

En este capítulo, a modo de marco teórico, se identificarán los elementos que forman un sistema de colas genérico y, a continuación, se estudiarán las características que definen el sistema. De esta manera, una vez conocidos los elementos y sus características se podrá pasar a valorar y definir la manera en la se implementarán en el programa.

3.1. Elementos de un sistema de colas

A continuación se mencionan y se explican los elementos, físicos o no, que forman un sistema de colas.

3.1.1. Usuarios

El usuario es el elemento que llega al sistema, dispuesto a esperar una cola, para recibir uno o más servicios. Es un elemento dinámico. Esto quiere decir que su estado, definido por sus variables, cambia en función del tiempo. Pese a que usuario tiene una connotación de persona, cabe decir que se pueden definir usuarios que sean, por ejemplo, piezas que esperen a ser procesadas en una máquina. En ese caso, la máquina sería la ventana de servicio que, no tiene porqué ser tampoco una taquilla convencional.

Para definir un usuario que llega al sistema, se puede hacer a partir de las siguientes variables:

- Identificador.
- Tipo de usuario.

- Tipo(s) de servicio(s) que requiere.
- Tipo(s) de servicio(s) que ya ha recibido.
- Tiempo de llegada.
- Número máximo de personas en la cola o tiempo que está dispuesto a esperar.

3.1.2. Ventanas de servicio

Las ventanas de servicio son los puntos del sistema que, con una capacidad de atención determinada, satisfacen las necesidades de servicios de los usuarios.

Para definir una ventana del sistema, se pueden utilizar las siguientes variables:

- Identificador.
- Tipos de usuario a los que da servicio.
- Tipos de servicio que ofrece.
- Número de servidores en paralelo.
- Tiempo de atención.
- Tiempo muerto entre servicios.

3.1.3. Cola

La cola también es un elemento dinámico del sistema. La fila de usuarios que se encuentran a la espera de una ventana de servicio forma la cola de esa ventana. Dicha fila está ordenada según una prioridad establecida.

Los parámetros que definen una cola genérica son los siguientes:

- Número de usuarios en cola.
- Disciplina de la cola.
- Número máximo de usuarios en cola.

3.1.4. Reloj

El reloj del sistema es un elemento intangible pero importante a la hora de lanzar y coordinar eventos en diferentes instantes de tiempo. El reloj marca el instante en el que se encuentra el sistema respecto a un instante inicial que se toma de referencia.
El reloj queda definido al determinarse el instante inicial o tiempo cero.

3.2. Características de un sistema de colas

Las características que definen un sistema de colas y que se mencionan en esta sección se basan en la recopilación de José Pedro García Sabater en [2].

3.2.1. Patrón de llegada de los usuarios

Si se observa la llegada de usuarios a un sistema cualquiera, seguramente parezca que no existe ningún patrón entre llegadas consecutivas. Sin embargo, normalmente existen una gran cantidad de factores que determinan que un usuario decida ir al sistema en un momento u otro y que permiten modelar con precisión la llegada de usuarios mediante variables estadísticas. Entonces, se dice que la llegada es estocástica, pues depende de una variable aleatoria. Para el caso de procesos, por ejemplo, la llegada de piezas a la máquina sí que está más determinada por el propio proceso y no está sometida a tanta aleatoriedad.
La determinación de las variables aleatorias que definen las llegadas de usuarios se escapa del objetivo del proyecto. Sin embargo, su uso está fuertemente anclado a la definición del modelo de simulación y, es por ello, que se hará una revisión de los principales modelos utilizados para definir llegadas de usuarios.

Distribución exponencial y de Poisson Las distribuciones más utilizadas a la hora de definir llegadas consecutivas de usuarios son la distribución exponencial y la distribución de Poisson. Entre ambas distribuciones existen características que las relacionan. Sin embargo, mientras la distribución exponencial es continua y, dada una media λ, contiene las probabilidades para cada tiempo entre llegadas consecutivas, la distribución de Poisson, a partir de una media λ, contiene las probabilidades de que, en un tiempo determinado, lleguen k usuarios al sistema. La relación ambas distribuciones indica que, si el número de llegadas sigue una distribución de Poisson de media λ, el tiempo entre llegadas sigue una distribución exponencial de media $(1/\lambda)$ y viceversa.

Pese a ser las más conocidas o extendidas, las distribuciones que se han explicado no son, ni mucho menos, las que mejor se adaptan a todos los procesos. Es por ello que, a la hora de modelar un sistema, se debe ser conocedor de las diferentes distribuciones y como determinar, a partir del sistema real, las variables que las definen. A continuación se enumeran otras conocidas distribuciones estadísticas, continuas (útiles para representar tiempo) o discretas (útiles para representar número de usuarios):

- **Continuas**
- Continua uniforme.
- Erlang.
- Normal.

- **Discretas**
- Uniforme discreta.
- Bernoulli.
- Binomial.

Las llegadas consecutivas de usuarios pueden ser individuales o en lotes. Esto supone que el patrón de llegadas debe contener información estadística sobre la probabilidad de que los usuarios lleguen al sistema en grupo (y dimensiones del grupo) o solos.

Por último, se debe definir si el patrón de llegadas es estacionario (si se mantiene constante) o no-estacionario (si varia a lo largo del tiempo). Es muy habitual que el patrón de llegadas sea no-estacionario. Ocurre, por ejemplo, en un bar-restaurante donde, en horas de comida, el número de clientes que llega aumenta respecto al resto del día. Igualmente ocurre momentos antes del inicio de cada película en las taquillas de un cine o en horas de cierre de oficinas en los peajes de las salidas de Barcelona.

3.2.2. Patrón de servicio de las ventanas

El tiempo de atención en una ventana no es, tampoco, una constante. Puede seguir una distribución de probabilidad concreta para cada tipo de usuario y servicio o incluso para cada ventana de atención. Además, el patrón de tiempo de atención puede ser estacionario y no-estacionario e incluso depender de otras variables. Ejemplos de estas variables pueden ser el número de personas en la cola, la hora del día o las horas acumuladas de trabajo de las personas que trabajan en las ventanas. De modo que el rendimiento de las ventanas a la hora de dar el servicio puede ser dependiente.

Es importante en muchas ocasiones tener en cuenta un tiempo muerto entre servicios. En cualquier sistema, hay un tiempo en el que el usuario está recibiendo el servicio y, una vez finalizado el servicio y el usuario abandona la ventana, transcurre un tiempo hasta que el siguiente usuario, que estaba esperando en la cola, comienza a recibir su servicio. A esto se le llama tiempo muerto y, puede ser más o menos amplio en función del motivo que lo origine. Por ejemplo, el tiempo muerto será bajo si simplemente contempla el cambio entre un usuario y otro. Sin embargo, puede ser elevado si la ventana de servicio ha de ser acondicionada entre servicio y servicio (como puede ser el caso de un consultorio médico o una sala de dar masajes).

Distribución Normal Ya se han mencionado con anterioridad algunas de las más conocidas distribuciones de probabilidad. Además, se han explicado brevemente la distribución exponencial y de Poisson, válidas también para definir el tiempo de atención en una ventana de servicio. Por su utilidad a la hora de definir el tiempo de atención, en este apartado se hará una introducción a la distribución Normal.
La distribución Normal, también conocida como distribución de Gauss, depende de dos paráme-

tros estadísticos; μ y σ^2 que corresponden respectivamente a la media y a la varianza (el cuadrado de la desviación estándar) de su función de probabilidad. Esta función tiene como peculiaridad el ser de forma acampanada y simétrica, centrada en el parámetro μ.

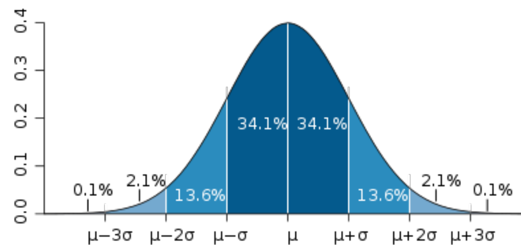

Figura 3.1: Ejemplo de función de densidad de una distribución Normal. Fuente: *Wikipedia*.

3.2.3. Disciplina de cola

La disciplina de la cola hace referencia a la forma en que los usuarios se ordenan en la cola para ser atendidos. La más conocida y extendida es la disciplina *FIFO*, del inglés *First in, First out*, donde el orden se establece a medida que llegan los usuarios. Así, el que llega antes tiene prioridad ante el que llega más tarde. Existe también la disciplina de cola opuesta, donde el último en llegar es el primero en ser atendido. Se trata de la disciplina *LIFO*, del inglés *Last in, First out*.

Puede haber disciplinas de cola en las que existe otro tipo de prioridad de atención. Por ejemplo, colas en que los clientes del tipo A son prioritarios respecto a los del tipo B. Entonces se distingue entre dos tipos; *preemptive*, si al llegar el usuario prioritario es atendido inmediatamente sin importar si algún otro usuario está utilizando la ventana de servicio en esos momentos o, *no-preemptive* si, pese a ponerse delante en la cola, el usuario prioritario debe esperar a que se libere la ventana para ser atendido.

Se puede hablar en esta sección del comportamiento de los usuarios frente a la cola. Las decisiones que puede tomar un usuario que no está dispuesto a esperar un largo tiempo a ser atendido pueden ser muy impredecibles y, son situaciones complejas de modelar. Existe la posibilidad de que el usuario, al llegar y ver una cola larga o prever que el tiempo de espera va ser grande, abandone directamente el sistema. También es posible que lo haga una vez ya ha esperado un tiempo determinado en el sistema o que incluso llegado ese tiempo, decida cambiar de cola. Estas situaciones se pueden modelar simplificando el comportamiento de los usuarios y limitando la capacidad del sistema.

3.2.4. Capacidad del sistema

Se le llama capacidad del sistema a la cantidad máxima de usuarios que pueden estar esperando en las colas del sistema. En el caso de que exista tal limitación, se trata de un problema de colas finitas. Se ha comentado en el apartado anterior que son muy comunes a la hora de simplificar el hecho de que usuarios decidan esperar una cola o marcharse de ella.

El otro caso es el de problemas de colas infinitas. En estos, la capacidad del sistema es ilimitada y, por lo tanto, la cantidad de usuarios esperando a ser atendidos no está acotada por ningún valor.

3.2.5. Número de puntos de servicio por ventana o servidores en paralelo

Para acceder a un tipo servicio se puede establecer una cola por cada punto de servicio o, se puede establecer una sola cola que alimente a una serie de servidores en paralelo. Véase en la Figura 3.2 los dos tipos de formaciones de cola; monocanal y multicanal.

Figura 3.2: Esquema de cola monocanal y multicanal. Extraída de [2].

Para ejemplificar el caso de una formación de cola monocanal, imagínese una serie de cajas de atención de una gran tienda de ropa. Existe una única cola y, cuando una caja queda libre, se avisa al primer cliente de la cola que ha de ir a la caja liberada. Igualmente sirve de ejemplo si los clientes obtienen un número al llegar al sistema y son llamados por orden numérico a la ventanilla que ha quedado libre.
En un supermercado, sin embargo, los clientes de colocan en una cola independiente delante de la caja a la que eligen ir.

3.2.6. Número de etapas de servicio

Por último, el sistema de colas puede ser unietapa o multietapa. En los sistemas unietapa el usuario requiere un servicio del sistema y, tras cubrirlo, lo abandona. Por su lado, en los sistemas multietapa el usuario puede requerir más de un servicio en el sistema y, de ser menester, pasar por más de una ventana de servicio antes de abandonar el sistema.

A modo de ejemplo, para comprar un bocadillo o un refresco en una feria, es muy típico primero tener que comprar el tique correspondiente en una ventana y luego ir a otra ventana a intercambiarlo por el producto. Este es un claro ejemplo de sistema multietapa.
También puede darse el caso de una combinación de sistema multietapa y unietapa. Si pensamos en unos lavabos, habrá usuarios que únicamente han de acceder a lavarse las manos. Sin embargo, existirá un grupo de usuarios que ha de utilizar el servicio y después lavarse las manos.

Capítulo 4

Herramientas para el desarrollo del programa

En el presente capítulo se justificarán las herramientas utilizadas para el desarrollo de *SimuQ 1.0.* A través de la propia experiencia y la bibliografía consultada se justificarán las elecciones del lenguaje de programación, editor, librerías utilizadas, compilador, etc.

4.1. Lenguaje de programación y entorno de desarrollo

Son muchos los lenguajes de programación que existen actualmente. Según el Índice TIOBE, elaborado por una empresa holandesa especializada en evaluar la calidad de programas informáticos, y consultado en [5], los cinco lenguajes más populares a fecha de consulta son, por orden: *Java, C, C++, C# y Python.*

El lenguaje adoptado para llevar a cabo el proyecto ha sido *Python.* El principal motivo es la experiencia y familiaridad que ya se tenía con este lenguaje al haber trabajado en el desarrollo de anteriores programas menos complejos. Además, de entre las ventajas que ofrece *Python*, se destacan para el desarrollo del proyecto la gran cantidad de librerías que se encuentran al alcance de cualquier usuario, la simplicidad y legibilidad que permite ser comprendido por otros usuarios, su formato multiplataforma y multiparadigma (con opción de crear programas para cualquier Sistema Operativo, orientando a objetos o con paradigma imperativo) y su gran vinculación con el entorno científico. Esto último facilita el acceso a librerías matemáticas y estadísticas e incluso la posibilidad de llamar a lenguajes más específicos para la estadística, como el lenguaje *R.* Por último, otra ventaja de *Python* es que puede ser compilado en otros lenguajes.

Evidentemente no todo son ventajas. Al asumir *Python* como lenguaje de programación, se renuncia a la potencia de, por ejemplo, lenguajes como *C y C++*. Sin embargo, el sacrificio de potencia no es grave al ser *Python* un lenguaje suficientemente rápido para el propósito del proyecto.

Figura 4.1: Logo de *Python*.

Una vez escogido el lenguaje, se debe escoger el entorno de trabajo. *Python* es un lenguaje que puede ser escrito en cualquier editor de texto como, por ejemplo, el bloc de notas de Windows. Existen, además, editores de texto más específicos que facilitan la escritura y lectura de código gracias a la diferenciación de fuentes y colores que automáticamente utilizan para variables, funciones, palabras clave del lenguaje, etc. Ejemplos de estos editores de texto son *Emacs* y *SciTE*. Por otro lado, existen los Entornos de Desarrollo Integrado (IDE). Estos programas disponen de herramientas que facilitan mucho la tarea de escribir el código.
Se escogió la IDE de *Spyder* por, además de tener las ventajas de un editor de texto avanzado, permite ejecutar el código en un terminal, acceder a la documentación de objetos y cuenta con un explorador de variables y unas herramientas muy interesantes para depurar el código. Además, es muy fácil de instalar y tiene una interfaz amigable que resalta errores y advertencias a tiempo en el que se escribe el código del programa.

Figura 4.2: Logo de *Spyder*.

4.2. Módulos y librerías

A la hora de desarrollar un programa *Python* se definen clases y funciones que desempeñan, en conjunto, la función del programa. En muchos casos existen módulos o librerías que ya

contienen funciones y clases definidas previamente por otros usuarios y que pueden resultar útiles al ser implementadas en el programa que se está desarrollando. De esta manera se evita volver a escribir código que ya ha sido escrito con anterioridad.
A continuación se introducen tanto los módulos como las librerías que se han utilizado para agilizar o hacer posible el desarrollo de *SimuQ 1.0*:

SimPy Ya se habló en el apartado de *Antecedentes* del módulo de simulación más conocido de *Python*; *SimPy*. Este módulo ha sido elegido como base de la gestión de eventos de *SimuQ 1.0*. *SimPy*, que se basa en generadores de *Python*, permite, entre otras opciones, crear ventanillas de servicio y gestionar la ocurrencia de eventos (entradas y salidas de usuarios a la ventanilla de servicio) de forma muy sencilla e integrada en el código del programa. Además, permite combinar el lanzamiento de eventos con funciones propias o de otras librerías de manera que se convierte en una herramienta muy potente y versátil. La versión utilizada ha sido *SimPy 3.08*.

Pygame Es un conjunto de módulos escritos en lenguaje *Python* con herramientas utilizadas esencialmente para el desarrollo de videojuegos. *Pygame* permite construir, de forma sencilla, elementos gráficos dinámicos de poca complejidad. Aún así, también permite desarrollar juegos más elaborados y obtener unos muy buenos resultados. Ha sido utilizado en el proyecto para mostrar gráficamente la simulación durante su ejecución, en la opción de *Simulación gráfica*. Ya se había trabajado anteriormente con *Pygame* y, ese fue uno de los móviles a la hora de escogerla como herramienta de trabajo.

Matplotlib Es una de las librerías más conocidas de *Python*. Permite la generación de una gran cantidad diferente de tipos de gráficos. En el proyecto, ha sido utilizada para graficar resultados de la simulación.

PyQt En realidad, *PyQt* es un *binding* [1] muy popular de la plataforma *Qt* para *Python*. Para el proyecto se ha utilizado *PyQt4*, que ha permitido, junto al constructor de interfaz de usuario que se ve a continuación, *QtDesigner*, implementar la GUI[2] de *SimuQ 1.0*.

Otros Para acabar de listar todas las librerías utilizadas, faltan por añadir; *xlsxwriter*, que es un módulo con el cual se exportan los datos de la simulación a un fichero Excel y *NumPy*, que es una conocida y muy potente extensión de *Python* pero que, en el proyecto ha sido utilizada para la generación de variables aleatorias.

[1] En el campo de la programación, un *binding* es una adaptación de una biblioteca para ser usada en un lenguaje de programación distinto de aquel en el que ha sido escrita. *Wikipedia, Julio 2016*

[2] GUI: Graphical User Interface. Interfaz gráfica de usuario.

4.3. Constructor de interfaz

Las clases de *PyQt* permiten crear una GUI para el programa. Sin embargo, se debe escribir manualmente el código que la define. Esto es una tarea laboriosa y poco visual. Existen herramientas que permiten diseñar la interfaz de forma visual creando la ventana, añadiendo botones, *checkbox*, cuadros de diálogo y, en definitiva, elementos de una interfaz de usuarios y, *a posteriori*, generan el código de programa que las crea.

QtDesigner Ha sido la herramienta utilizada para diseñar y construir la GUI de *SimuQ 1.0* a partir de componentes de *Qt*. Se decidió utilizarla por su operatividad y simplicidad, además de tener un mínimo de experiencia al trabajar con ella.

4.4. Compilador

Un compilador es un programa informático cuya función es, dado un código escrito en un lenguaje determinado, traducir dicho código a otro lenguaje de programación. Generalmente se utiliza para que el sistema operativo del ordenador sea capaz de ejecutar el programa sin tener que depender de otros programas para hacerlo. Ya se comentó con anterioridad que una de las ventajas de *Python* es que se puede ejecutar en cualquier sistema operativo, por ser multiplataforma. Sin embargo, cuando se compila el código, se ha de tener en cuenta para que sistema operativo se quiere hacer. Se ha decidido que la primera versión de *SimuQ* sea compilada para Windows OS y, por ello, se presentan a continuación dos importantes compiladores de *Python* para Windows.

Py2exe Es quizás el compilador más conocido para pasar de extensión *.py* (archivo *Python*) a *.exe* (archivo ejecutable de Windows). Permite crear ejecutables que dependan de otros archivos incluidos en una sola carpeta o crear archivos ejecutables que son independientes, aunque tarden más tiempo en cargarse. Sin embargo, el compilar *SimuQ 1.0* con esta herramienta originaba algunos problemas debidos a que no se importaban archivos necesarios de librerías como *Matplotlib* o *Numpy* y, es por ello, que se optó por un compilador alternativo.

Pyinstaller Este ha sido el programa que se ha elegido para compilar el código de *SimuQ 1.0* a un archivo *.exe*. *Pyinstaller* también ofrece la opción de crear un único archivo pero se ha preferido optimizar el arranque del programa y compilar en una carpeta donde se guarda tanto el ejecutable como los archivos necesarios para el arranque y ejecución.

Capítulo 5

Desarrollo de *SimuQ 1.0*

En este capitulo se definirá la estructura del programa. Se verá paso a paso el desarrollo de las partes que componen *SimuQ 1.0* sin profundizar en el código del programa en exceso, ya que este está comentado entre las líneas del código. Sin embargo, se hablará de las funciones principales, su rol y se relacionarán con las herramientas presentadas en el capítulo anterior. Se seguirá la línea de diseño y, a medida que sea necesario, se justificarán las decisiones que se han tomado para llegar al resultado final.

Tras arrancar *SimuQ 1.0* y escoger uno de los dos tipos de simulación, se define el sistema, se ejecuta la simulación y se extraen los datos. Para hacer el texto más claro, se dividirá en estos tres apartados. En cada uno de ellos se harán las distinciones necesarias entre los dos modos de simulación; intensiva o gráfica.

5.1. Definición del sistema

Esta sección está dividida en los mismos pasos en los que lo está la definición del sistema en *SimuQ 1.0*. Es decir, se empieza por nombrar los ítems (tipos de usuarios y tipos de servicios) involucrados en el sistema y se termina definiendo el tiempo que durará la simulación. En cada sección se explicitará lo que se puede y lo que no se puede definir en el sistema con *SimuQ 1.0*, así como de que manera es obviado por el programa si no se define.

5.1.1. Nombrar ítems

La definición de cualquier sistema con *SimuQ 1.0* empieza con todas las opciones de edición e introducción de datos bloqueadas a excepción de los recuadros en los que el usuario introduce, vía teclado, el nombre de los diferentes tipos de servicios y usuarios que intervienen en el sistema.

De esta manera, el programa guarda la correlación entre sus variables internas y el nombre dado, de modo que en todas las siguientes fases de la simulación, cada tipo de usuario y de servicio del sistema tendrá, a vistas del usuario del programa, el nombre que se haya definido en este apartado.
Desde un principio se decidió que la simulación intensiva podría trabajar, como máximo, con tres tipos de usuario y tres tipos de servicio diferentes. Por ello que en la pantalla de *Nombrar ítems* tiene seis recuadros divididos en dos columnas de tres, para usuarios y servicios. Se limitó la cifra a tres por considerarse suficiente para modelar una gran cantidad de sistemas y por evitar complicar en exceso el código, alargando el tiempo de dedicación.
Por lo que a la simulación gráfica respecta, se decidió acotar el número de variables a dos tipos de servicio y dos tipos de usuario distintos. En este caso, no tanto por las cargas temporales que podría suponer el desarrollar la simulación gráfica, sino porque en si, el mostrar la simulación por pantalla es un valor añadido que sirve para entender el funcionamiento de la simulación, no tanto para ejecutar simulaciones de gran envergadura.

En ambos casos se puede definir el nombre de cualquier combinación de tipos de usuario y de servicio. Evidentemente, siempre que no sean más del número máximo en cada caso. Se ha de definir, al menos, un tipo de usuario y un tipo de servicio en cualquiera de los casos. El programa guarda en el diccionario los recuadros que tengan alguna cosa escrita, obviando que los que se dejen en blanco no son necesarios.
Resaltar que, como se detalla en el *Manual de utilización* (Apéndice A), *SimuQ 1.0* no acepta caracteres especiales como *ñ* o *ç*. Estos deberán ser evitados.

Al hacer clic sobre el botón *Validar y guardar* de la interfaz, el programa crea el diccionario con las variables introducidas y completa los campos de las siguientes fases de diseño en los que aparecen escritos los nombres definidos. Además, desbloquea las casillas correspondientes en la fase de *Ventanas de servicio*. Si todo este proceso ocurre sin errores, el programa muestra que todo está correcto con una ventana emergente. Si existe alguna advertencia (que falte escribir el nombre de al menos un usuario o un servicio) el programa también lo muestra a través de una ventana emergente.

5.1.2. Ventanas de servicio

En la segunda fase de la modelación del sistema, desbloqueada y operativa una vez se han validado los cambios de la primera fase, consiste en crear las ventanas de servicio. Para esta fase, el objetivo es hacer que la creación de ventanas sea lo más versátil posible, pudiendo

modelar la mayor cantidad de sistemas diferentes; desde unas taquillas de cine o las cajas de un supermercado, a un aparcamiento de coches o un peaje. Para ello, se da la posibilidad de, al crear la ventana, poder otorgarle cualquier combinación de tipos de usuario y servicio definidos en la primera fase.

La GUI de *SimuQ 1.0* ha sido diseñada para ser lo más simple posible. Sin embargo, su simpleza puede suponer un problema si el usuario no tiene claro como definir su sistema o de que maneras puede hacerlo. En la Figura 5.1 se muestran esquemas genéricos de diferentes tipos de ventanas que se pueden crear con *SimuQ 1.0*, si se han definido en la primera fase los usuarios P y Q y los tipos de servicio A y B:

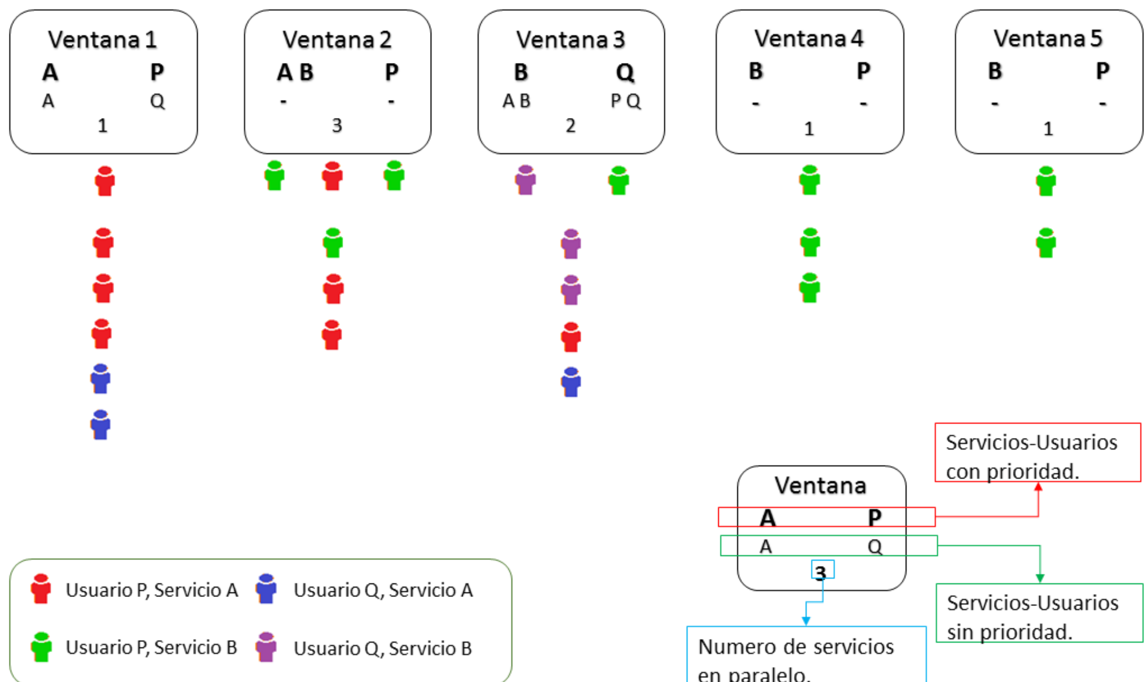

Figura 5.1: Esquema de diferentes tipos de ventana definibles con *SimuQ 1.0*.

Número de colas por servicio

SimuQ 1.0 permite definir el número de colas por servicio. Esto quiere decir que se puede hacer que una misma cola alimente a diversos puntos de servicio (servidores en paralelo en una misma ventana) o que cada punto de servicio tenga su cola de espera (creando distintas ventanas iguales). Véase Figura 3.2. Con esto conseguimos poder definir sistemas como, por ejemplo, un peaje (donde los usuarios forman una cola delante de cada ventanilla, pese a que haya dos

ventanillas consecutivas que ofrezcan el mismo servicio). Las ventanas 4 y 5 de la Figura 5.1 ejemplifican este caso. Además, podemos definir sistemas como una batería de urinarios (donde los usuarios esperan en una única cola). La ventana 2, si únicamente ofreciera un tipo de servicio (el servicio *Urinario*) podría ser ideal para ejemplificar una batería de tres urinarios.

Definición de la disciplina de cola

La disciplina de las colas se puede considerar poco flexible al definir el sistema con *SimuQ 1.0*. Sin embargo, las posibilidades que ofrece son suficientes para la mayoría de los casos reales. Generalmente, los sistemas cotidianos siguen una disciplina de cola FIFO[1] y, esta es la disciplina que el programa aplica a todas las colas del sistema. A pesar de ello, permite definir para cada ventana, usuarios y servicios que tienen prioridad sobre otros. La prioridad en *SimuQ 1.0* es siempre de tipo *no-preemptive*[2]. De modo que los usuarios prioritarios se ordenarán por orden de llegada, delante de todos los usuarios no prioritarios (también ordenados por orden de llegada). Podemos verlo ejemplificado en ventana 1 de la Figura 5.1, que da servicio A a usuarios de tipo P y Q, siendo P prioritarios. La ventana 3, por su lado, refleja la disciplina *no-preemptive*, habiendo dos usuarios Q que requieren servicio B (combinación prioritaria) esperando mientras un usuario P acaba de ser atendido con un servicio B (combinación no prioritaria).

Ya se ha visto en el apartado 3.2.3 que comportamiento de los usuarios en las colas es bastante impredecible y difícil de modelar con variables estadísticas. Existen usuarios que si la cola es muy larga no se disponen a esperar, otros que a medias de la espera se marchan o se cambian de cola y así una infinidad de comportamientos distintos. *SimuQ 1.0*, por ser una primera versión, obvia estos comportamientos y asume que, una vez el usuario se coloca en una cola, espera el tiempo que necesario para ser atendido sin cambiar de cola.

Capacidad del sistema

Una buena y sencilla manera de modelar el comportamiento impaciente de usuarios que no se colocan en la cola si esta es muy larga, es acotar la capacidad máxima de la cola o del sistema. Esta podría ser la próxima línea de desarrollo de *SimuQ* ya que de momento, *SimuQ 1.0* asume colas y sistemas de capacidad infinita. A pesar de poder ser una mejora, no se trata de un elemento esencial ya que, en si, la utilidad de *SimuQ 1.0* es validar dimensionamientos de puntos de servicio reales, donde el interés global es que las colas no sean tan extensas como para tener que ser acotadas.

[1]Explicada en el apartado 3.2.3.
[2]Ídem [1].

Diferencias entre simulación intensiva y gráfica

Por lo que a ventanas de servicio se refiere, existen algunas diferencias entre la opción de simulación intensiva y la opción gráfica. Cuando se escoge la simulación intensiva, se pueden definir tantas ventanas de servicio como se quiera y, cada una de ellas, puede atender hasta 10000 usuarios en paralelo. Sin embargo, para evitar colapsos en la interfaz en el momento de mostrar sistemas muy complejos, se optó por limitar la cantidad de ventanas a cinco y, a dos, el número de atenciones en paralelo de cada una. De esta manera, *SimuQ 1.0* puede gestionar y mostrar durante la simulación gráfica un máximo de diez puntos de servicio y cinco colas.

5.1.3. Usuarios, definición de las llegadas

Al *Validar y guardar* las ventanas creadas en la fase anterior, el programa desbloquea las casillas necesarias de la siguiente fase, la definición de llegadas de usuarios. Esto quiere decir que, aunque se hayan definido en la primera fase tres tipos de usuario y tres tipos de servicio, si únicamente se ha creado una ventana que da servicio A a usuarios tipo P; solamente se podrán definir la llegada de usuarios P que requieran servicio A. De esta manera se evita que se generen usuarios que no pueden ser atendidos en el sistema.

Patrón de llegadas

En esta fase pues, toca definir el patrón de llegadas de usuarios. *SimuQ 1.0* ofrece una única forma de hacerlo. El tiempo entre llegadas consecutivas de usuarios se define con el parámetro estadístico λ, media de una distribución exponencial. Antes de continuar, cabe resaltar que si se trata de una simulación gráfica, el parámetro λ se ha de dar en segundos. Esto es debido a que las funciones internas de simulación a tiempo real trabajan con segundos.
Por ser las distribuciones exponencial y de Poisson las más utilizadas para modelar patrones de llegadas a un sistema y, siendo estadísticamente equivalentes si se relacionan correctamente sus medias, se optó por implementar únicamente la distribución exponencial. Incluir otras distribuciones podría ser también el móvil de mejoras de *SimuQ*.
Ya se ha insistido con anterioridad en el afán por la simpleza e intuitividad de la GUI. Es por ello que se optó por definir el patrón de llegadas mediante un único parámetro λ que defina el tiempo de llegada de un nuevo usuario al sistema (y no un parámetro para cada tipo de usuario). La forma de introducir datos acerca de la probabilidad de que el usuario sea de un tipo u otro o el servicio que requiere, es a través de porcentajes. Así pues, se deberán definir los porcentajes de probabilidad de que el usuario sea de cada tipo. Luego, a cada tipo de usuario le corresponden porcentajes acerca de la probabilidad de requerir un servicio u otro. Véase la

Figura 5.2. El esquema completo corresponde al de simulación intensiva, mientras que los recuadros mostrados en verde corresponden a la parte del esquema que coincide con la definición del patrón en simulación gráfica.

En el mismo esquema, el grupo de porcentajes de tipos de usuario (en azul) deberá sumar 100 %. Igualmente, cada uno de los grupos de porcentajes púrpuras, si el porcentaje de tipo de usuario correspondiente es diferente de cero, deberá sumar también 100 %. En caso de no ser así, tras hacer clic en el botón de *Validar y guardar* aparecerá una ventana emergente indicando el error.

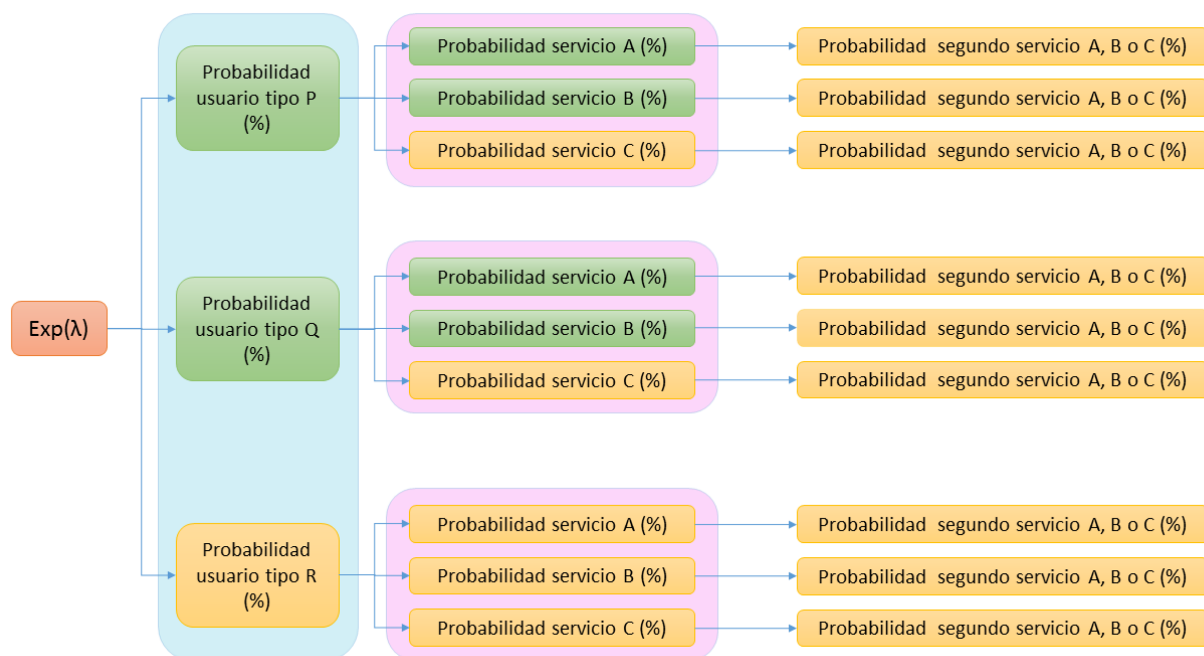

Figura 5.2: Esquema de definición de patrón de llegada de usuarios con *SimuQ 1.0*.

El patrón de llegadas queda definido casi totalmente en *SimuQ 1.0* con lo expuesto anteriormente. Se vio en la sección 3.2.1, que el patrón de llegadas puede ser estacionario o no estacionario y puede contemplar llegadas de usuarios individuales o en grupo.
Respecto a las llegadas individuales o en grupo, el programa desarrollado en este proyecto asume que los parámetros introducidos hacen referencia a llegadas individuales. Bien es cierto que estadísticamente pueden existir llegadas casi simultáneas al sistema, pero *SimuQ 1.0* no contempla la opción de definirlas como tal. Sin embargo, en la siguiente y última fase de definición del sistema, el programa ofrece la opción de asignar períodos de llegadas a usuarios. Se verá más adelante, pues, que se puede definir un patrón de llegadas no estacionario con periodicidad.

Número de etapas de servicio

En el apartado 3.2.6 se vio que un sistema puede ser unietapa o multietapa. *SimuQ 1.0*, en la opción de simulación intensiva, permite definir un sistema al que llegan usuarios que requieren uno o dos servicios. En el esquema de la Figura 5.2 se ve como a cada tipo de usuario-servicio se le puede añadir un porcentaje correspondiente a la probabilidad de que un usuario de esas características, tras el servicio 1, requiera el servicio 2 (que se define con un *comboBox* al lado del porcentaje). Esta opción amplía mucho el abanico de tipos de sistemas definibles con el programa. A pesar de ello, una próxima mejora podría ser añadir la posibilidad de contemplar más de dos etapas (dos servicios) y la posibilidad de definir diferentes segundos servicios para un mismo tipo de usuario y servicio.

5.1.4. Configuración de la simulación

En la última fase de definición del sistema, antes de poder ejecutar la simulación, se introducen los parámetros que gobiernan el tiempo de atención en las ventanas de servicio, los períodos de llegada de usuarios y el tiempo total que se desea ejecutar la simulación.

Patrón de servicio de las ventanas

En *SimuQ 1.0*, el patrón de servicio de las ventanas, visto en el apartado 3.2.2, se define en su totalidad a partir de los parámetros estadísticos de las distribuciones normales (μ, σ^2) que gobiernan los tiempos de atención y tiempos muertos entre servicios para cada pareja servicio-usuario.

Al *Validar y guardar* los cambios tras crear las ventanas en la fase dos, *SimuQ 1.0* determina cuales son las posibles parejas usuario-servicio que se pueden dar durante la simulación en función de las ventanas creadas y, en esta última fase, pide que se introduzcan todos los parámetros para definir los tiempos de atención. La definición de los tiempos muertos es opcional para cada pareja usuario-servicio.
El tiempo de simulación y los tiempos de períodos de llegada de usuarios que se definirán a continuación, han de concordar en unidades temporales todos ellos con el parámetro λ definido en la fase dos (*Ventanas de servicio*). En la opción de simulación gráfica, no solamente han de concordar, sino que todos ellos deben estar expresados en segundos (s).

SimuQ 1.0 admite solamente la opción de definir un patrón de servicio estacionario a partir de parámetros de distribuciones normales. Sin embargo, pueden existir patrones no estacionarios,

dependientes de variables externas o incluso ser estacionarios y depender de otras distribuciones de probabilidad. Pese a que, como ya se ha comentado en anteriores ocasiones, se han escogido los casos más generales y comunes para desarrollar la primera versión del programa, el añadir cualquier función adicional que admita la definición de otros patrones de servicio sería un valor añadido al programa que, por acotaciones temporales no entran dentro los objetivos de este proyecto.

Períodos de llegada de usuarios

Por último, a parte de introducir el tiempo que se quiere hacer durar la simulación, existe la opción de modificar el patrón de llegadas y hacerlo no-estacionario. Si se marca la casilla del *checkBox* y se introduce el período de llegadas de forma válida, el patrón de llegadas será periódico. Tendrá espacios de tiempo en que los usuarios llegan de forma continua siguiendo la ley exponencial y el patrón definido en la tercera fase y, existirán espacios de tiempo donde no llegarán usuarios. El período se define con la longitud temporal total del mismo y el tiempo durante el cual llegan usuarios. Véase ejemplo en la Figura 5.3. Para que sea válido, la longitud total del período ha de ser más pequeña o igual que el tiempo total de simulación. Igualmente el tiempo de llegadas ha de ser más pequeño que el del período entero y ambos diferentes de cero.

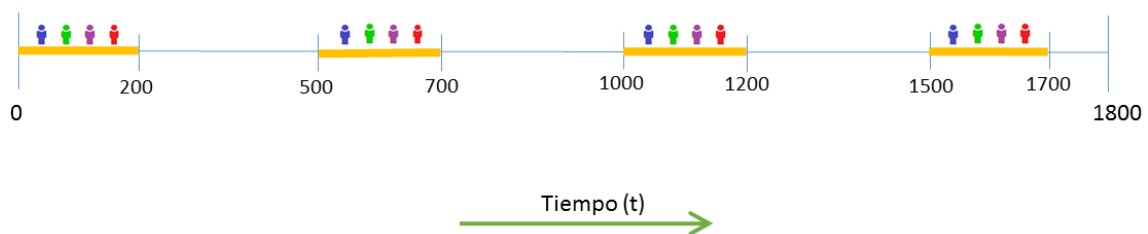

Figura 5.3: Períodos de llegadas de usuarios de 200 cada 500 con un tiempo de simulación de 1800.

Diferencias entre simulación intensiva y gráfica

En el cierre de esta sección, antes de pasar a ver la ejecución de la simulación en *SimuQ 1.0*, se comentan las diferencias entre la simulación intensiva y gráfica en la última fase de definición. A parte de la obligatoriedad de definir todos los tiempos en segundos (s), la simulación gráfica presenta una última diferencia. Se trata de un factor de velocidad de simulación. En otras palabras, se utiliza para regular la velocidad en que la simulación va a ser ejecutada y mostrada en pantalla. Si el factor es unitario, cada segundo será un segundo de simulación. Si el factor es

diez, la simulación correrá diez veces más rápida.

5.1.5. Funciones y variables esenciales

En este apartado se hará una síntesis de las principales funciones y variables definidas en el archivo de programa con el nombre de *SimuQ.py*, que gobierna el funcionamiento de la GUI y el almacenamiento de los *inputs* [3] del usuario. El archivo contiene distintas clases de *Python* para gestionar las diferentes ventanas de la interfaz. Las funciones de las que vamos a hablar en este apartado son comunes (con pequeñas variaciones) en la clase *Intensive_Simulation* y *Graphical_Simulation*, que gobiernan la ventana de simulación intensiva y gráfica respectivamente. Se seguirá el mismo orden de definición del sistema para comentar las funciones.

En primer lugar, tras escribir el nombre de los tipos de usuario y servicio en la primera fase, el programa genera un diccionario de *Python* con la información introducida que será consultado por el programa para traducir sus variables internas a variables del usuario de *SimuQ 1.0*. La función que lo genera y completa siguientes fases de la interfaz con los nombres introducidos, es *validate_users_services(self)*. Esta se ejecuta cuando se pulsa el botón de *Validar y guardar*.

La segunda fase está gobernada por dos funciones principales. Cada vez que se hace clic en el botón *Añadir*, se llama a la función *add_windows(self)* que guarda la información de la ventana creada en el diccionario de *Python* con el nombre de *windows_in*. Este diccionario contiene la siguiente información acerca de cada ventana: identificador de la ventana, si es o no de tipo prioritario, tipos de usuarios que atiende y servicios que ofrece y cuales de ellos son prioritarios (si los hay) y, por último, el numero de servidores en paralelo.
La otra función, *validate_windows*, gobierna acciones sobre la interfaz. Desbloquea o bloquea opciones de la GUI para continuar la definición del sistema de forma correcta.

Una vez definido el patrón de llegadas de usuarios en la tercera fase y hacer clic al botón de *Validar y guardar* de la GUI, la función *validate_users(self)* se encarga de guardar los datos las variables siguientes: *users_type_in*, *users_service_in*, *users_after_service_in* y *lamda* (no es *lambda* por ser un nombre reservado de *Python*).

Las últimas dos funciones a comentar en esta sección pertenecen a la fase de definición del patrón de atención y configuración general de la simulación. La primera de ellas es la función *add_param_normal(self)* que es llamada cada vez que se pulsa el botón *Añadir* para definir un

[3] Del inglés, cualquier entrada de información al programa.

tiempo de atención. Esta función almacena en el diccionario *t_atention_in*, para cada pareja servicio-usuario, los parámetros estadísticos de la distribución normal del tiempo de atención y el tiempo muerto entre servicios (si se define).
Por su lado, la función *validate_configuration_simulation(self)*, lanzada al pulsar el botón *Validar y guardar*, guarda el tiempo de simulación introducido y los períodos de llegadas de usuarios (en caso de estar definidos). En simulación gráfica, guarda también el factor de velocidad de simulación. Además, valida que todo esté correctamente definido y, de ser así, desbloquea el botón para iniciar la simulación.

5.2. Ejecución de la simulación

En esta sección se explicarán las clases y funciones principales involucradas en la ejecución y gestión de la simulación. Además se explicará la manera en la que se definen los elementos del sistema vistos en el apartado 3.1. Sobre todo, esta parte del texto es importante para comprender como funciona la simulación y de que manera se han solventado problemas importantes; por ejemplo, el algoritmo que selecciona a que ventana decide ir el usuario que llega al sistema. Saber como funciona este algoritmo puede permitir a usuarios de *SimuQ 1.0* hacer modificaciones en él para adaptarlo a sistemas más específicos.

Una vez el sistema está completamente definido, se desbloquea el botón para empezar la simulación. Al pulsarlo, el programa llama a la función *Execute_Simulation(datos)* que tiene como variables de entrada las definidas en el apartado 5.1.5 y que, en el texto, han sido substituidas por la variable *datos*. La función con el mismo nombre, pero correspondiente a cada tipo de simulación se encuentra en los archivos *graphical_simulation.py* y *intensive_simulation.py*. El programa accederá a una u otra en función de la modalidad de simulación escogida.

Ambos archivos, *graphical_simulation.py* y *intensive_simulation.py*, importan los archivos *User_Class.py* y *Windows_Class.py* que contienen las clases *User*, *Graphic_User*, *Windows* y *Graphic_Windows*. Además importan el archivo *distributions.py*, que contiene funciones para generar variables aleatorias de distintas distribuciones estadísticas.

5.2.1. Elementos del sistema

En este apartado se plantea la forma en la que se implementan los diferentes elementos teóricos del sistema en la simulación.

Entorno de simulación

Para gestionar los eventos de la simulación, *SimPy* necesita crear el entorno que contendrá los elementos del sistema gestionados por el módulo.

SimPy ofrece la posibilidad de crear dos tipos de entornos diferentes. Uno de ellos, utilizado en la simulación intensiva, ejecuta la simulación a la velocidad de computación del ordenador. El otro, utilizado en la simulación gráfica, ejecuta la simulación a tiempo real. Es decir, un segundo real equivale a un segundo en la simulación. Pese a ejecutarse en tiempo real, este entorno permite acelerar la velocidad de simulación mediante un factor que se introduce como variable al instanciarse el entorno.

```
44
45      env=simpy.Environment()
46
```

Figura 5.4: Código para crear el entorno de simulación con *SimPy*. Extraído de *intensive_simulation.py*.

Usuarios

Los usuarios del sistema no son elementos que puedan crearse en forma de objetos de alguna clase de *SimPy*. Por este motivo se han definido las clases *User* y *Graphic_User*.

La clase *User* es utilizada en ambos modos de simulación. Los usuarios, objetos de la clase, tienen por atributos el identificador con el número de usuario y el tipo, el servicio o servicios que requiere. Además, durante la simulación, se definen atributos de cada usuario como el tiempo de llegada, tiempo de espera y ventana de atención. De esta manera se almacenan datos de la simulación en cada usuario. Al iniciar cualquier simulación se crea un diccionario *Python* llamado *Users* que contiene todos los usuarios que van llegando al sistema, de modo que quedan accesibles en cualquier momento que se requiera.

Por su lado, la clase *Graphic_User* que hereda de la clase *pygame.sprite.Sprite* se utiliza únicamente en la simulación gráfica para gestionar la representación en pantalla. Al heredar de la clase *pygame.sprite.Sprite*, permite tener asociado el usuario a una imagen y asignarle una posición en la pantalla. Además permite crear grupos de *sprites*. Se trabaja de forma que las colas de usuarios frente a las ventanas de servicio sean grupos de *sprites*, muy útiles para ser modificados o representados en bloque.

Ventanas de servicio

Para la gestión de ventanas de servicio se ha definido la clase *Windows* que hereda de la clase *simpy.resources.resource.PriorityResource*. Las ventanas del sistema, objetos de la clase *Windows* tienen, además de los atributos y funciones de la clase de *SimPy*, otros atributos como el identificador (número de ventana), tipo de usuario que atiende y tipo de servicio que ofrece, prioridades de atención y otros atributos como listas que permiten almacenar información durante la simulación. Al inicio de cada simulación se crea un diccionario de *Python* que contiene todas las ventanas (objetos de la clase *Windows*) del sistema. La clase *Windows* tiene definidos los métodos *param* y *param_B* a los que se dedicará un apartado más adelante por su importancia en el código.

Una de las ventajas de heredar de la clase de *SimPy* para *PriorityResources* es que la gestión de los eventos de salida y entrada de usuarios a la ventana está implementada en la clase mediante iteradores de *Python*. Además, ofrece la posibilidad de poner a la cola eventos con mayor o menor prioridad. En la Figura 5.5 se muestra el código para poner a la cola (o no, si la ventana está libre) un evento de petición de ventana con prioridad. Cuando se libera la ventana y se lanza este evento (por ser el siguiente en la cola), el código continúa ejecutando lo que siga a los puntos suspensivos.

```
with windows.request(priority=prio) as req:

    yield req
    ...
```

Figura 5.5: Código de ejemplo para lanzar un evento de petición de una ventana de servicio (*windows*) con prioridad *prio*.

Pese a ser elementos estáticos durante la simulación, también se ha definido una clase *Graphic_Windows* que hereda de la clase *pygame.sprite.Sprite*. Esta clase, igual que la clase *Graphic_User* se utiliza durante la simulación gráfica para gestionar la representación del grupo de ventanas en pantalla.

Colas

La gestión de colas en *SimuQ 1.0*, en realidad, no se lleva a cabo creando objetos cola de cada unas de ventanas de servicio. El caso es que los atributos y métodos de la clase *Windows* permiten acceder a listas de usuarios en la cola de la ventana, capacidad y usuarios utilizando

el recurso al mismo tiempo. Mediante estos atributos y métodos se lleva a cabo la gestión de las colas.

Reloj

El reloj de cualquier simulación en *SimuQ 1.0* se inicializa en cero y finaliza en el valor de la variable *simulation_time*. Durante la simulación, el reloj avanza (se actualiza) con la línea de código de la Figura 5.6. Este método de la clase *simpy.Environment* nos permite avanzar el tiempo del reloj el valor que se introduce como variable entre paréntesis.

```
191                 yield env.timeout(t_atention_A+dead_time_A)
192
```

Figura 5.6: Expresión para poner a la cola un evento de avanzar el reloj un tiempo de (*t_atention_A+dead_time_A*). Código extraído de *intensive_simulation.py*.

5.2.2. Funciones y métodos principales

Vista la manera en que *SimuQ 1.0* implementa los elementos del sistema, en este apartado se verán las funciones y métodos principales que se utilizan para gestionar los estados de estos elementos durante el proceso de simulación.

Execute_Simulation

Ya se ha mencionado con anterioridad la función *Execute_Simulation(datos)*. Es la que recibe todos los datos recogidos por la GUI y gobierna la preparación e inicio de la simulación.

Cuando se llama a la función *Execute_Simulation(datos)* se inician procesos de generación del entorno, se crean las ventanas de servicio, se definen las subfunciones *simulation, choose_windows* y *user_generator* y se crean diccionarios y variables que se usarán durante la simulación.

Función *user_generator(env, lamda)*

El primer proceso que *Execute_Simulation(datos)* lanza es la llamada a la subfunción *user_generator(env, lamda)*. Esta es una función que contiene un bucle infinito y que, por lo tanto, por ella sola no pararía de ejecutarse. Sin embargo, las líneas de código de la Figura 5.7 que lanzan la función como un proceso del entorno de *SimPy*, permiten ejecutar el proceso hasta que el reloj interno de la simulación alcance el valor de *simulation_time*.

Una vez lanzada por primera vez, *user_generator(env, lamda)* genera tiempos de llegadas de usuarios (a través de las funciones de *distributions.py* y *lamda*) y, si se encuentran dentro de

```
604
605     env.process(user_generator(env,lamda))
606     env.run(until=simulation_time)
607
```

Figura 5.7: Código para lanzar la simulación mediante la función *user_generator*. Extraído de *intensive_simulation.py*.

períodos de llegada de usuarios, genera objetos de la clase *User* y *Graphic_User*(si la simulación se muestra por pantalla), actualiza el reloj interno con cada llegada, registra información del nuevo usuario y llama a la función *simulation(user)* o *simulation(user, graphic_user)*, según proceda, para continuar la simulación.

Funciones *simulation(user) y simulation(user, graphic_user)*

El esquema básico de ambas funciones es exactamente el mismo. La única diferencia es que, la función usada para la simulación gráfica requiere el objeto de la clase *Graphic_User* para mostrar el usuario en la pantalla.

Las dos funciones reciben al objeto de la clase *User* y lanzan el algoritmo que determina a que ventana decidirá ir el usuario (función *choose_window(user)*). Una vez determinado, ejecutan el *request* de la Figura 5.5 para colocar al usuario en la cola de la ventana seleccionada. Por último gestionan la actualización del reloj de simulación una vez el usuario ha salido de la ventana y gestionan el almacenamiento de datos de la simulación.

Función *choose_window(user)*

Esta función retorna, dado un usuario, la ventana en la que se pondrá a esperar la cola para ser atendido. Existen situaciones en las que es evidente el comportamiento que tendrá el usuario. Por ejemplo, si hay dos ventanas que pueden dar servicio en igualdad de condiciones y, en una de ellas hay cola y en la otra no, el usuario irá directamente a la ventana en la que no ha de esperar. Sin embargo, existen muchas otras situaciones en las que la decisión del usuario es más compleja y tiene en cuenta muchos factores. Por este motivo es importante explicar la manera en que *SimuQ 1.0* toma dicha decisión. De esta manera se podrá comprender la simulación que se ejecute y, en caso de ser menester, se podría modificar el código para adaptar las decisiones a sistemas específicos.

Se explicará esta parte del código únicamente para la función que pertenece a *intensive_simulation.py*. La perteneciente a *graphical_simulation.py* tiene la misma estructura pero con un desarrollo más simple. Por este motivo, basta con entender la más compleja, para saber

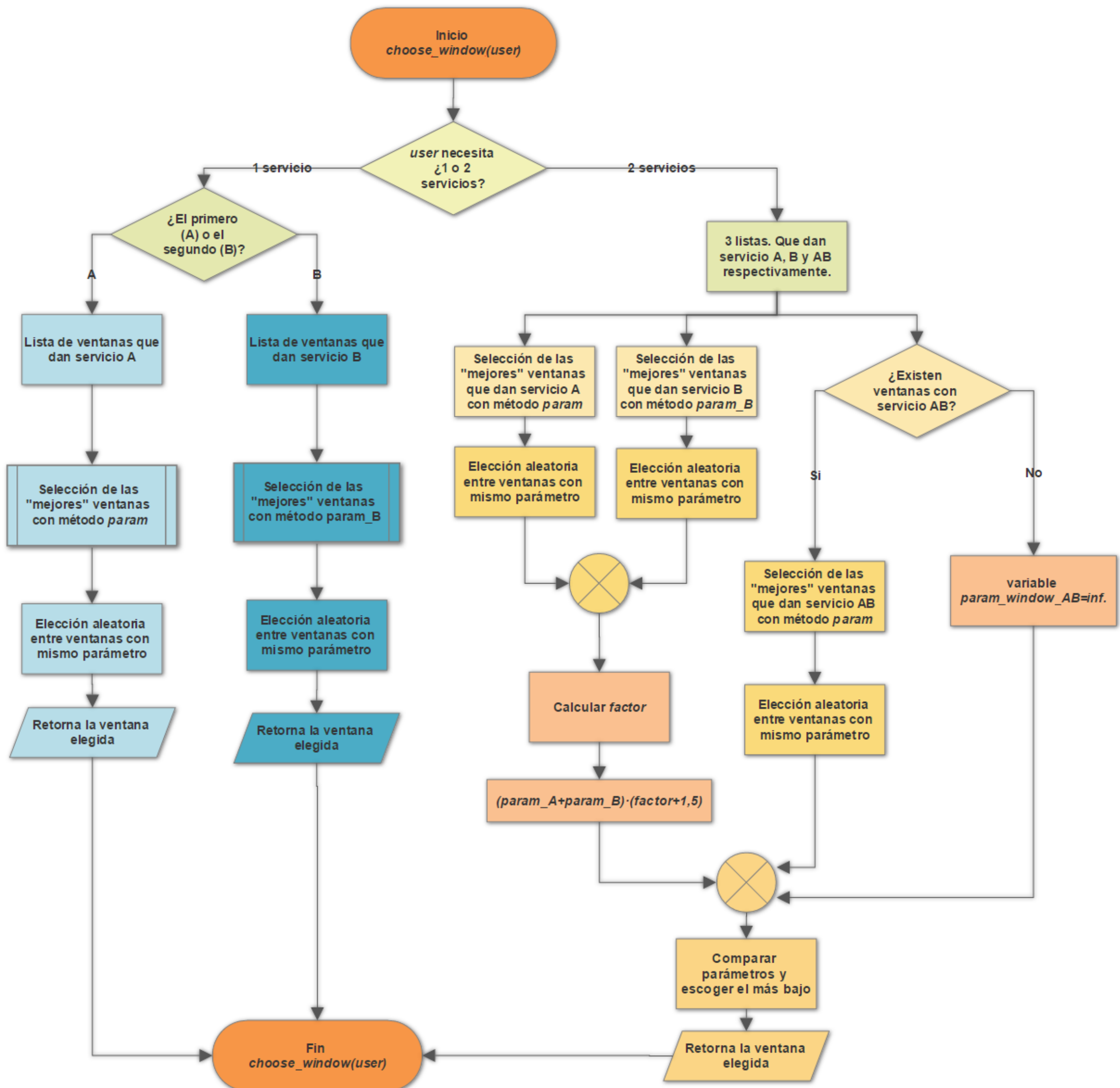

Figura 5.8: Diagrama de flujo de la función *choose_window(user)*.

como funcionan ambas. En la Figura 5.8 se puede ver el diagrama de flujo de la función *choose_window(user)*.

Existen tres situaciones distintas que se pueden dar cuando se llama a la función *choose_window(user)*. El usuario *user* puede llegar nuevo al sistema y requerir un servicio o también puede ser que requiera dos. La tercera situación es un usuario que requería dos servicios en su llegada al sistema pero, acaba de ser atendido del primer servicio y ahora ha de ponerse en la cola de una ventana que le ofrezca el segundo servicio. En esta sección y en la siguiente, se denotará como A el primer servicio y, como B, al segundo servicio (si lo hay), sin importar el tipo de servicio que sea.

Antes de pasar a ver como se resuelven las tres posibles situaciones, se deben presentar los métodos de la clase *Windows* que son esenciales para determinar la ventana elegida por el usuario y que se explicarán con más detalle en el siguiente apartado. Los métodos *param(user,users_type_in)* y *param_B(user,users_type_in)* retornan un parámetro de valor entre 1 y ∞ que permite comparar lo "buena opción"que es una ventana frente a otra. Las "mejores opciones"tienen un valor de parámetro más bajo que las demás.

Nuevo usuario que requiere A y B La función *choose_window(user)* empieza, en este caso, creando tres listas. Cada una de ellas contiene, respectivamente, todas las ventanas del sistema que ofrecen el servicio A, el servicio B y el servicio A y B al usuario determinado como variable de la función.
El segundo paso es, con el método *param* selecciona la mejor (o mejores en caso de haber más de una ventana con el mismo parámetro) ventana de cada lista. En cualquier caso, debe quedar como máximo una ventana en cada lista. Por lo tanto, en caso de empate, se aplica una selección aleatoria (como la que podría hacer un usuario que no tiene claro a que ventana ir).
El último paso es comparar los parámetros entre las dos opciones: ir a la ventana que ofrece A y luego a la que ofrece B o ir directamente a la ventana que ofrece A y B.

Para hacer la comparación se define un *factor* que tiene en cuenta el porcentaje de usuarios que llegan al sistema y pueden ser atendidos en la ventana que da servicio B. Con esto se pretende tomar la decisión, teniendo en cuenta que mientras se espera a ser atendido en la ventana que da servicio A, pueden llegar nuevos usuarios que se pongan en la ventana (escogida como mejor) que da servicio B. El *factor*, inferior a la unidad, se suma al valor arbitrario de 1,5 y se multiplica al parámetro de la opción de ir a dos ventanas distintas (*param_window_A* + *param_window_B*). Se puede ver el fragmento de código que toma la decisión en la Figura 5.9.
Tanto el valor arbitrario de 1,5 como el factor son susceptibles a ser modificados por usuarios de *SimuQ 1.0* que tengan información específica de los patrones de toma de decisión de los usuarios del sistema.

Si el sistema se ha definido siguiendo los pasos y fases de *SimuQ 1.0*, se asegura que existirá una ventana que da servicio A y otra B. No se asegura que exista una ventana que da los dos servicios. Si no existe esta última, su parámetro tenderá a ∞.

Usuario que requiere A o B Si el caso es el de un usuario que requiere solamente un servicio, se hará únicamente una lista con las ventanas que lo pueden atender. Luego se recortará la lista, dejando las ventanas que tengan el parámetro mas bajo. Para calcular el parámetro se llamará

```
313
314             factor=0
315             for i in window_B.user_tipe:
316                 factor=factor+users_tipe_in[i]
317             factor= factor/100.0
318
319             if (param_window_A+param_window_B)*(1.5+factor) < param_window_AB:
320                 choosed_window=window_A
321             else:
322                 choosed_window=window_AB
323                 both_services=True
324
```

Figura 5.9: Código para comparar parámetros entre opción de ir a dos ventanas o a una que ofrezca dos servicios. Extraído de *intensive_simulation.py*.

a los métodos *param(user,users_type_in)* si se trata de servicio A o *param_B(user,users_type_in)* si se trata de servicio B. En caso de empate, la elección será aleatoria.

Métodos *param(user,users_type_in)* y *param_B(user,users_type_in)*

Ambos métodos tienen la misma función en el programa y su código es idéntico a excepción de que el método *param* tiene en cuenta que el usuario requiere su primer servicio (servicio A en esta sección) y *param_B* tiene en cuenta que el servicio es el segundo (tras haber recibido A), servicio B.

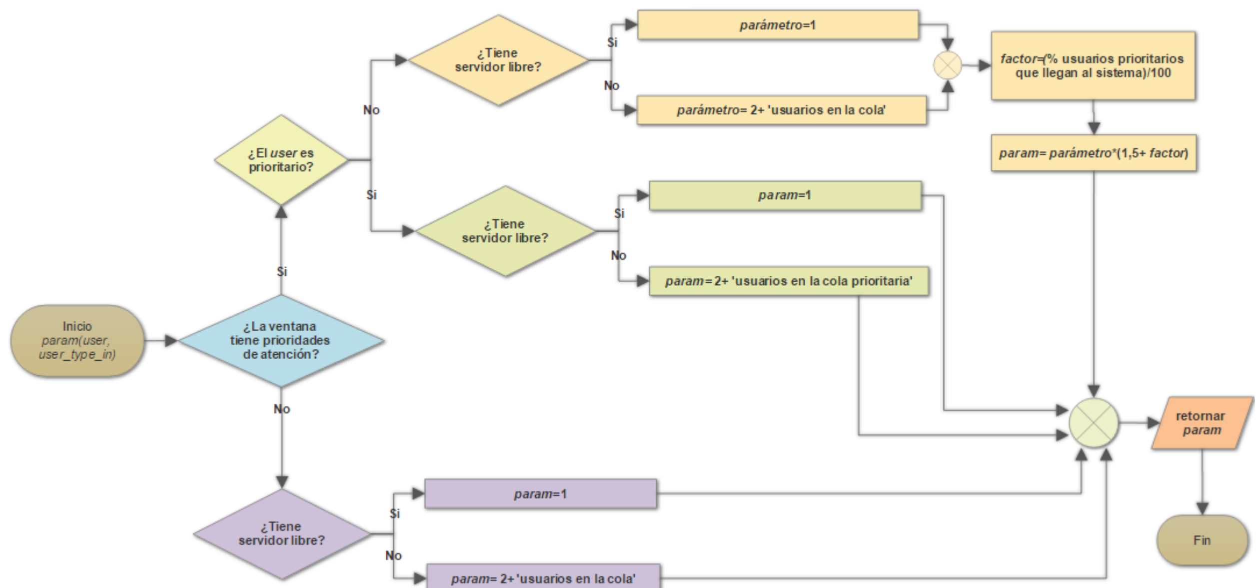

Figura 5.10: Diagrama de flujo del método *param* de la clase *Windows*.

La Figura 5.10 muestra el diagrama de flujo de los métodos *param* y *param_B*. Se puede ver que también se define un *factor* para el caso de que el usuario decida ir a una ventana con prioridades de atención donde él no es prioritario. El *factor* y el valor arbitrario de 1,5 sirven

para compensar el riesgo de ponerse en esa cola y que lleguen usuarios con prioridad. De nuevo, estos son valores susceptibles a ser modificados en el caso de ser menester. En la Figura 5.11 se muestra el código correspondiente al cálculo del factor y redefinición del parámetro *param*.

```
61
62                  factor=0
63                  for i in self.priority_user:
64                      factor=factor+users_tipe_in[i]
65                  factor= factor/100.0
66                  param=param*(1.5+factor)
67
```

Figura 5.11: Código para recalcular el parámetro *param* de una ventana con prioridad frente a un usuario no prioritario.

Notar en el diagrama de flujo que, como ya se dijo con anterioridad, el parámetro *param* existe de 1 a ∞. Siendo 1 la mejor opción (la ventana tiene algún servidor libre).

5.2.3. Síntesis de la simulación gráfica

A lo largo de la sección no se ha hecho mucho hincapié en las diferencias entre los dos tipos de simulaciones. Este apartado ha se servir para hacer una síntesis del funcionamiento de la simulación gráfica.

El código que ejecuta la simulación en modo gráfico, *graphical_simulation.py*, ya se ha comentado que utiliza las mismas funciones que la simulación intensiva con algunas diferencias. Las diferencias más destacadas son la mayor simplicidad por parte de la simulación gráfica y la inclusión de lineas de código que representan lo que está sucediendo a tiempo real en la simulación. A continuación veremos como se representan los diferentes elementos que aparecen en pantalla.

Inicio de la simulación. Ventanas y leyenda.

Al pulsar el botón *Empezar simulación* y ejecutarse el código *graphical_simulation.py*, se crea la pantalla de *Pygame* por la que se va a mostrar la simulación. Antes de mostrar la pantalla, se dibujan las ventanas de atención con un *Sprite* cuadrado de color gris con el identificador de la ventana escrito en su centro. Se dibuja la leyenda que indica la relación entre el color del usuario en la pantalla y la pareja tipo de usuario y servicio. Y por último se dibuja el reloj de la simulación inicializado a tiempo cero. Una vez mostrada la pantalla con todos los componentes iniciales y se inicia automáticamente la simulación, esta se debe dejar acabar antes de tocar (mover, minimizar o cerrar) la pantalla que muestra la simulación. Al final de la simulación se cerrará automáticamente y se podrá volver a interactuar con el programa y extraer los datos de

la simulación. El hecho de tocar la pantalla mientras se ejecuta la simulación, produce un error desconocido que bloquea el proceso de mostrar la simulación pese a que esta se sigue ejecutando. Dado este caso se puede esperar a que finalice de simulación y la interfaz vuelva a la normalidad o, cerrar todo el programa.
Durante la simulación, los elementos dinámicos, reloj y usuarios son los que se han de actualizar a medida que avanza el tiempo. Se puede ver en la Figura 5.12 un ejemplo del aspecto que tiene una simulación mostrada por pantalla.

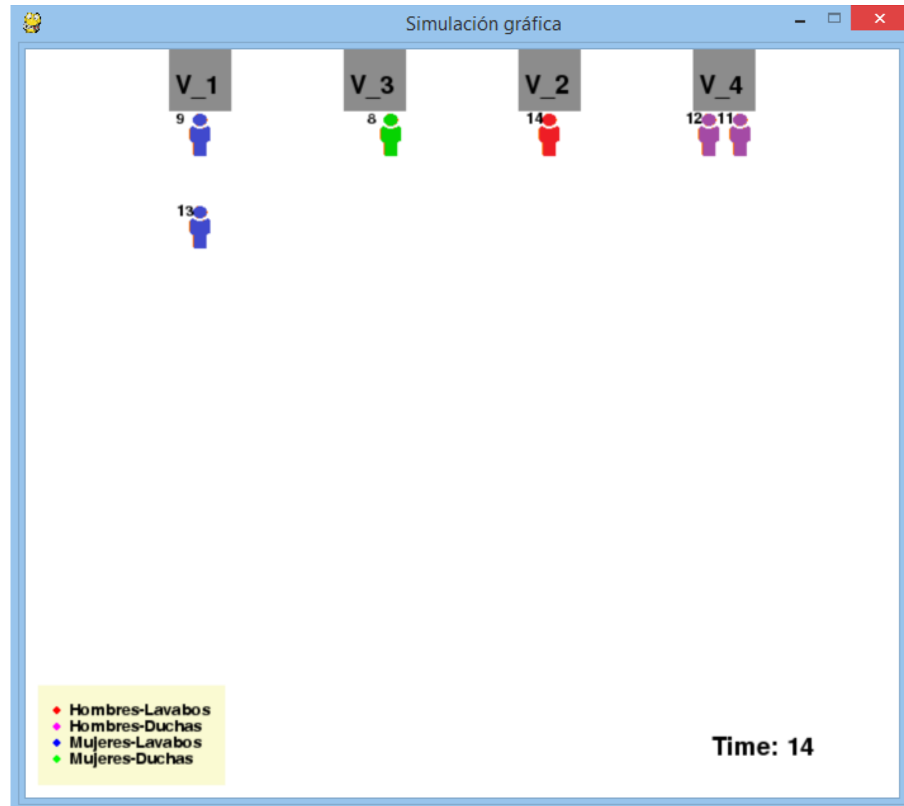

Figura 5.12: Ejemplo simulación gráfica.

Reloj

El tiempo de simulación, en realidad es una variable continua que va de cero a *simulation_time*. Sin embargo, al tratarse de simulaciones por eventos discretos, el tiempo avanza también de forma discreta con el lanzamiento de cada evento. Por este motivo, el tiempo se actualiza en la pantalla cada vez que el reloj interno de la simulación cambia.

Usuarios

Igual que el reloj, el movimiento de usuarios sucede en instantes discretos del tiempo. Es por ello que se actualizan los usuarios mostrados en pantalla, solamente cuando llega un usuario

nuevo, se cambia de ventana para recibir un segundo servicio o sale del sistema.

La representación de los usuarios se hace con cuatro iconos de persona idénticos entre ellos, pero cada uno de un color. Estos iconos se asignan a un objeto de la clase *Graphic_User* que, como ya se ha visto, hereda de la clase *pygame.sprite.Sprite* que le otorga métodos muy interesantes para mostrarlos por pantalla. Además, se muestra el identificador al lado izquierdo de la parte superior de cada usuario, es decir, el número del usuario. Este número va del 0 al ∞ y se asigna a cada usuario por orden de llegada al sistema.

5.3. Visualización de datos

Cuando la simulación finaliza, la GUI de *SimuQ 1.0* muestra directamente la pantalla de *Visualización de datos* y una ventanilla emergente informando que la simulación ha finalizado con éxito.
La pantalla de *Visualización de datos* contiene un resumen de los elementos del sistema definidos para la última simulación que se ha ejecutado. Se puede ver el patrón de llegadas, el patrón de atención, ventanas de servicio del sistema, tiempo de simulación y períodos de llegada de usuarios. Además, la pantalla contiene cuatro botones que permiten acceder a información de la simulación, recogida y procesada por *SimuQ 1.0*.

Entre los objetivos de *SimuQ 1.0* está el ayudar a tomar decisiones de dimensionamiento de puntos de servicio en base a datos estadísticos que proporciona la simulación pero, la explotación de estos datos, no entra como parte de desarrollo del programa. Sin embargo, *SimuQ 1.0* ofrece las siguientes cuatro opciones para tener una primera impresión de los datos que se extraerán de la simulación.

Graficar evolución temporal de colas

Esta opción muestra un gráfico interactivo donde se muestra un *time series plot*[4] para cada una de las ventanas del sistema simulado. En él, se muestra la longitud de la cola de cada ventana a lo largo del tiempo de simulación. La interactividad del gráfico permite seleccionar desde la leyenda, que ventanas deben mostrarse en el gráfico y cuales no, para facilitar su lectura.

[4]Del inglés, gráfico que muestra una serie de datos en función del tiempo.

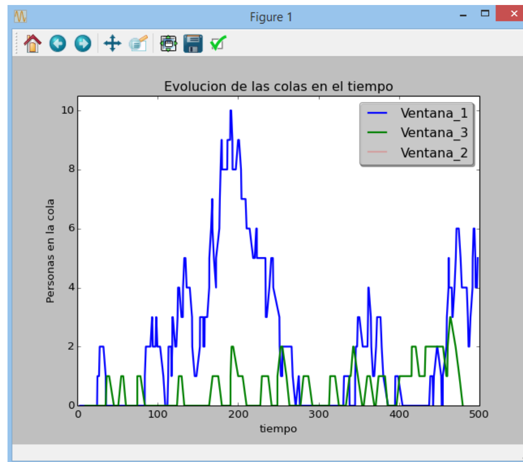

Figura 5.13: Ejemplo de gráfico interactivo de evolución temporal de colas.

Graficar Boxplot de los tiempos de espera

La segunda opción muestra otra ventana emergente con gráficos. En este caso los gráficos son de tipo boxplot en los que se muestran los tiempos de espera de todos los usuarios del sistema. Por un lado se muestra un boxplot de los tiempos de espera generales (de todo el sistema) y, por otro lado, se muestra un boxplot para cada ventana de servicio.

Extraer documento de texto con datos

El documento de texto con datos es un resumen de algunos datos estadísticos post-simulación. Se detalla el número de usuarios que han llegado al sistema y los que han sido atendidos durante el tiempo de simulación, en total y por cada ventana. Además, se muestra información de los tiempos de espera como el máximo, el medio y el percentil del 95 %.

Extraer documento de simulación

La última opción muestra un texto en el que se han escrito de forma ordenada cronológicamente los eventos que han cambiado el estado del sistema. Llegadas y salidas de usuarios así como información acerca del tiempo de llegada, de atención, ventana de servicio, etc.

Se utilicen o no las opciones de visualización de datos de la interfaz de *SimuQ 1.0*, el programa guarda en la carpeta los dos documentos de texto descritos anteriormente; el de datos y el de la simulación. Además, gracias al módulo *xlsxwriter* se exportan datos a un documento Excel que se guarda en la misma carpeta, bajo el nombre de *simulation_excel_file*. Este archivo contiene datos de los usuarios; identificador, tipo, servicio, segundo servicio (si lo hay), tiempo de llegada, ventana de servicio, ventana de segundo servicio (si lo hay), tiempo de servicio y

tiempo de espera. También contiene datos de cada una de las ventanas en forma de parejas tiempo-número de personas en la cola.

El documento Excel permite a los usuarios del programa explotar los datos según sus intereses.

Capítulo 6

Resultados

Este capítulo puede servir de síntesis del capitulo anterior por mostrarse los resultados tras el desarrollo del programa. Cabe pues, hacer una valoración de los resultados obtenidos en base al alcance de los objetivos planteados al inicio del proyecto. Además, esta sección se valdrá de herramientas gráficas como diagramas de flujo para representar el esquema del funcionamiento del programa final y un árbol de fallos para sintetizar las fuentes que pueden hacer que el programa presente errores.

6.1. Alcance de objetivos

Llegados al punto en que se han cerrado, para este proyecto, las líneas de desarrollo de *SimuQ 1.0*, si se valora el alcance de objetivos, el resultado es muy positivo.

Se ha conseguido desarrollar un programa que, pese a presentar limitaciones de diseño más o menos significativas, puede resolver simulaciones de una gran cantidad de sistemas de colas. Gracias a la versatilidad que se ha dado al programa en todas las funciones de definición implementadas, se consigue poder definir sistemas tan diversos como; un sistema de peajes, unos lavabos públicos, las taquillas de un cine, un aparcamiento de coches y motos, una gasolinera, etc. (se pueden consultar ejemplos resueltos con *SimuQ 1.0* en el Apéndice B).
Como se ha dicho, existen limitaciones de diseño que impiden, por ejemplo, limitar la capacidad de las colas o del sistema, definir patrones de llegada o de atención de usuarios que sigan distribuciones diferentes a la exponencial y normal o definir sistemas multietapa con más de dos etapas. Sin embargo, gracias al haber escrito y comentado el código del programa en lengua inglesa, se da accesibilidad a más usuarios que requieran, por motivos individuales o para mejorar el programa, entender y modificar o extender partes del código e implementar nuevas

características.

Las herramientas que ofrece *SimuQ 1.0* para visualizar los datos permiten tomar decisiones sencillas, como por ejemplo descartar un sistema en que el gráfico de colas en función del tiempo contenga ventanas donde la curva es monótonamente creciente.
Para tomar decisiones más complejas sobre el dimensionamiento, se pueden extraer datos del archivo de Excel que el programa crea al final de cada simulación.

Por último, se ha logrado cumplir con el objetivo de poder mostrar la ejecución de una simulación por pantalla y controlar la velocidad del proceso. Pese a ser menos potente que la intensiva, esta opción también es bastante versátil y permite igualmente visualizar y extraer datos de simulación.

6.2. Diagramas de flujo

En esta sección se presentan los diagramas de flujo del programa-resultado del proyecto; SimuQ 1.0. Se muestran los diagramas de flujo del funcionamiento de la interfaz, en Figura 6.1 y del proceso de simulación, representativo tanto de la opción intensiva como gráfica, en Figura 6.2.

6.3. Relación de errores

SimuQ 1.0 como cualquier proceso informático y más, siendo una primera versión, puede presentar errores o fallos de ejecución. En el árbol de fallos de la Figura 6.3 se muestran todas las acciones conocidas que acaban haciendo fallar al programa de un modo u otro.

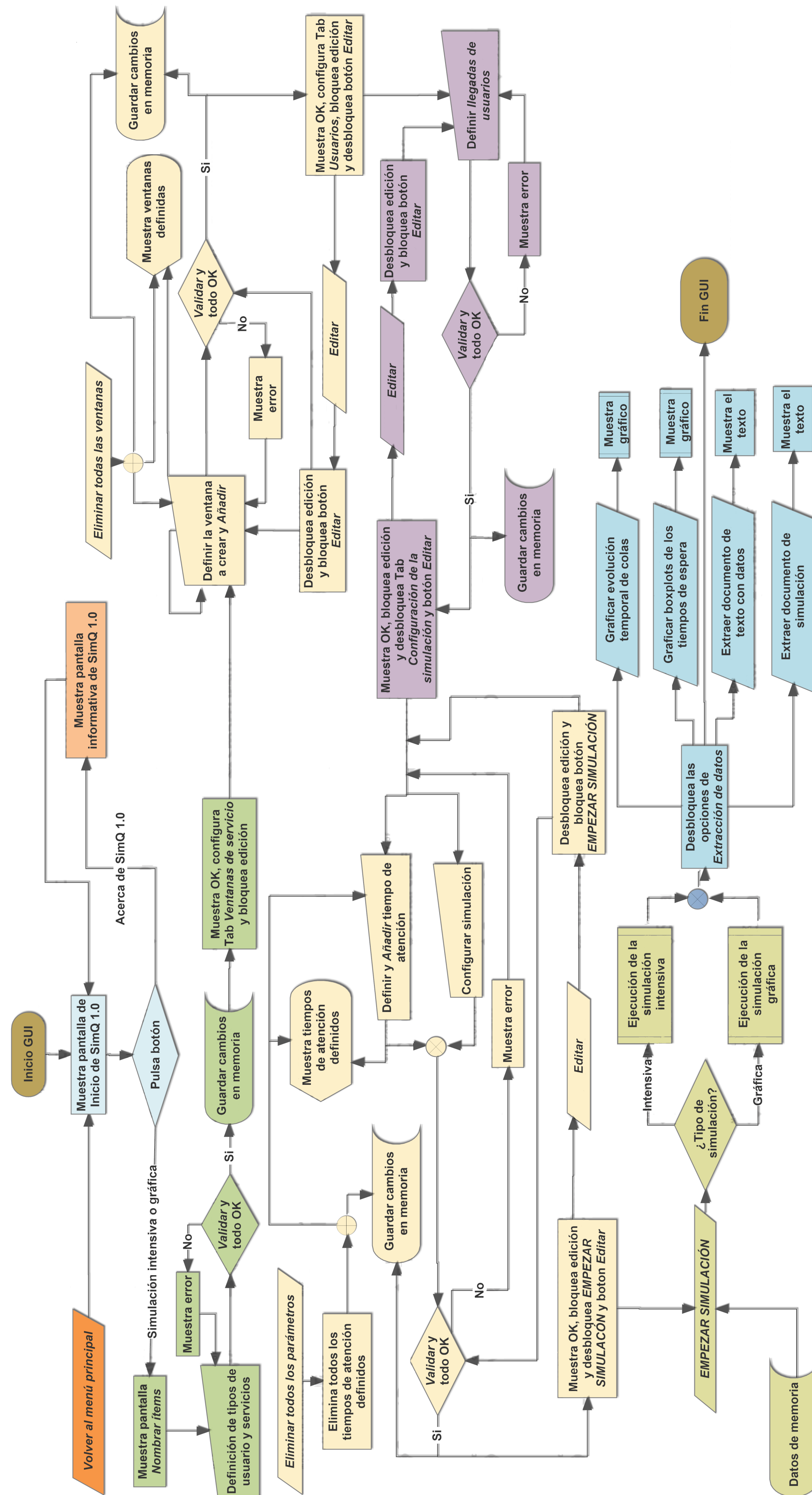

Figura 6.1: Diagrama de flujo de la interfaz de SimuQ 1.0.

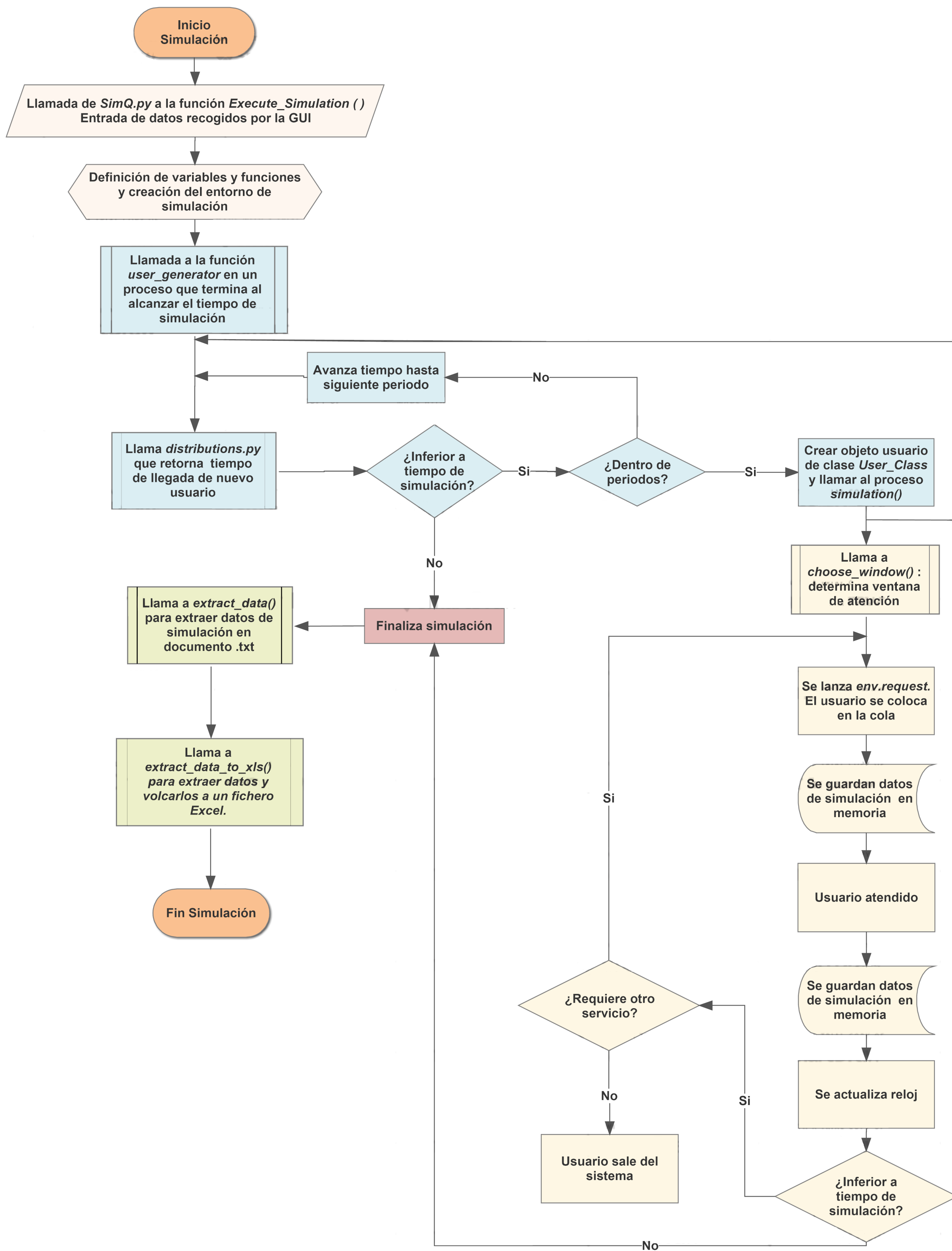

Figura 6.2: Diagrama de flujo del proceso de simulación intensiva de SimuQ 1.0.

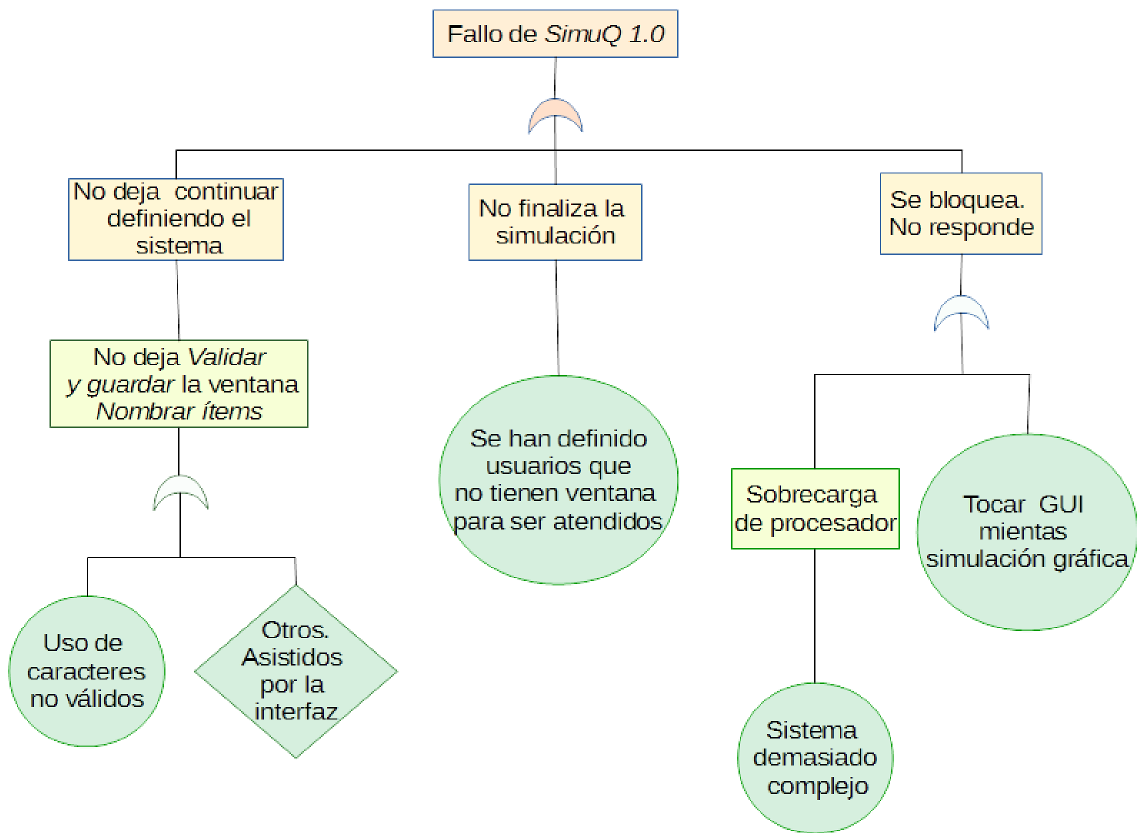

Figura 6.3: Árbol de fallos de *SimuQ 1.0*.

Capítulo 7

Planificación temporal y costes

Este capítulo se dedica a presentar, mediante un gráfico de Gantt (ver Figura 7.1), las fases que se han seguido para desarrollar el proyecto y como se han dividido temporalmente en los casi siete meses que ha durado el trabajo. Además, se hará un cálculo estimado de los costes que ha supuesto *SimuQ 1.0*.

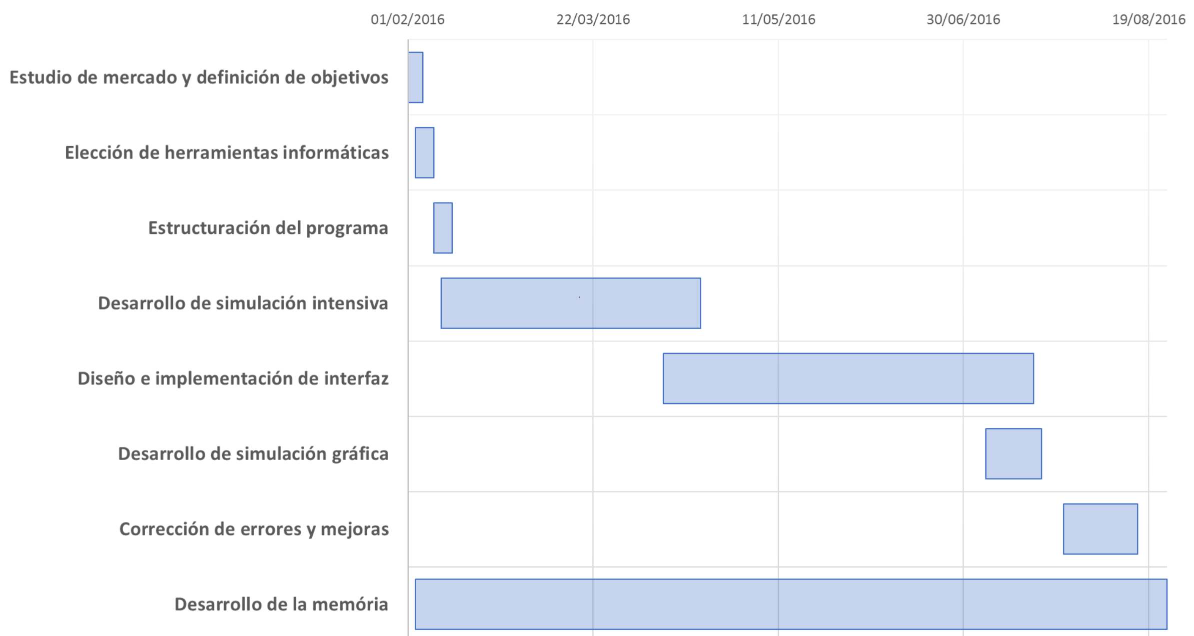

Figura 7.1: Diagrama de Gantt del desarrollo del proyecto.

Puesto que para el desarrollo de *SimuQ 1.0* y la memoria no se ha requerido ninguna herramienta de pago y ya se disponía del hardware, los costes del proyecto se calcularán a partir de las horas que se han empleado para llevarlo a cabo.

Para hacer el cálculo de las horas, se ha tenido en cuenta una media de dos horas diarias durante los días laborales de los siete meses. El precio de la hora para un graduado en ingeniería industrial se ha estimado en 35 €.

$$2\text{h} \cdot 5 \text{ días} \cdot 4 \text{ semanas} \cdot 7 \text{ meses} = 280\text{h}$$

$$280\text{h} \cdot 35\text{€} = 9800 \text{ €}$$

El cálculo del coste se puede redondear sin problemas a 10000€si se tienen en cuenta las horas que el tutor ha empleado en la guía y supervisión del trabajo y otros gastos derivados de desplazamientos y material de oficina.

Conclusiones y líneas futuras

La simulación de sistemas permite prever el comportamiento que tendrán dichos sistemas implantados en el entorno de trabajo. Por su importancia, se tomó la iniciativa de desarrollar *SimuQ 1.0*, con el que se ha probado la potencia del código libre y el gran número de personas que trabajan en compartir conocimientos acerca de las posibilidades que ofrece. Además, se ha brindado la opción de modificar o continuar la construcción del programa en otras líneas de desarrollo.

Se plantean durante el texto diferentes líneas de desarrollo que no se han podido extender bajo justificaciones temporales. Desde añadir nuevas distribuciones de probabilidad para definir los patrones de llegada de usuario o de atención, a permitir definir capacidades del sistema o incluso augmentar el número de variables para definir un sistema más complejo. Todas ellas pueden formar parte de la linea de desarrollo de una futura versión de *SimuQ 1.0*.

Agradecimientos

Intentaré ser escueto en el cierre del proyecto, a modo de un primer agradecimiento al lector. Me gustaría, sin embargo, dejar escrito en esta sección la enorme satisfacción que me deja este trabajo que me ha enseñado, sobre todo, que las cimas que parecen inalcanzables, se consiguen con pequeños y constantes pasos. Por supuesto, en este caso, no sin la ayuda del tutor del proyecto, Pere Grima Cintas, que propuso la idea y ha guiado este proyecto durante su desarrollo. Sin olvidar el inestimable apoyo de mi padre y mi madre que, una vez más, han estado en todo momento para hacerme mirar hacia la cima y, por último, y no por ello menos importantes, quiero agradecer de verdad a todas esas personas que comparten su conocimiento y experiencia, libre y desinteresadamente, ya sea a través de foros, librerías o manuales, permitiendo que el mundo del código libre siga tomando cada vez más importancia.

Bibliografía

[1] BOHÓRQUEZ, P.R. y MEDINA, S.. 2009. *Simulación de lineas de producción y servicios mediante el uso de Python-SimPy*. Tesis Grad. Ing. Sist. Univ. de Los Andes. 132p. Págs. 1-31.

[2] GARCÍA, J.P. 2010/2011. *Teoría de Colas*. Valencia, Grupo ROGLE. 66p.

[3] PLUS, 2016 *Biografía de Agner Krarup Erlang*. (Disponible en: https://plus.maths.org/content/os/issue2/erlang/index. Consultado en 2016)

[4] TEAM SIMPY, 2002-2016 *Documentación del módulo SimPy*. (Disponible en: https://simpy.readthedocs.io/en/latest/. Consultado entre 2015-2016)

[5] 2016 TIOBE SOFTWARE BV, 2016. *Índice TIOBE*. (Disponible en: http://www.tiobe.com/tiobe-index/. Consultado el 17 de agosto de 2016)

Bibliografía complementaria

[1] CAO, RICARDO. 2002. *Introducción a la simulación y a la teoría de colas*. A Coruña, NET-BIBLO, S.L. 224p.

[2] JORBA, A. y MASDEMONT, J. 1995. *Introducció a la simulació*. Barcelona, Edicions UPC. 198p. Págs. 1-65

[3] MATPLOTLIB DEVELOPMENT TEAM, 18 de junio de 2016 *Documentación del módulo Matplotlib*. (Disponible en: http://matplotlib.org/. Consultado en 2016)

[4] PYGAME COMMUNITY, Agososto 2009 *Documentación del módulo Pygame*. (Disponible en: http://www.pygame.org/docs//. Consultado en 2016)

[5] 1994-2016 SMARTDRAW, LLC, 2016. *Programa e información sobre el diagrama de flujo*. (Disponible en: https://www.smartdraw.com/flowchart/. Consultado el 18 de agosto de 2016)

ETSEIB