Modelado y ejecución de pruebas de performance

Gabriel Barbatto
Nicolás Díaz

Universidad de la república
Facultad de ingeniería
Montevideo, Uruguay
Año 2014

Modelado y ejecución de pruebas de performance

Gabriel Barbatto
Nicolás Díaz

Trabajo de presentado en el marco del proyecto de grado del año 2014

Docentes:
Ing. Gustavo Vazquez
Ing. Gustavo Guimerans
Dr. Alvaro Martín

Universidad de la República
Facultad de Ingeniería
Montevideo, Uruguay
Año 2014

*Let's face the truth, performance testing *IS* rocket science.*

Dawn Haynes

*… but even rocket science involves *SOME* easy stuff.*

Scott Barber

Agradecimientos

En primer lugar a nuestras parejas, familiares y amigos, por su infinita paciencia, y por apoyarnos en nuestras largas jornadas frente a nuestros monitores, los cuáles miran con cierta preocupación convencidos (con razón) de que somos seres extraños.

Además agradecemos a la Facultad de Ingeniería de la Universidad de la República Oriental del Uruguay por brindarnos toda la formación necesaria, y que con sus limitados recursos nos ha sabido brindar una educación de excelencia. Muchas veces durante nuestras carreras nos vemos agobiados (particularmente cuando en varios momentos miramos por la ventana y nos percatamos que ya está saliendo el sol y pasamos la noche entera estudiando o programando) y no entendemos el porqué de tanta exigencia, y es al terminar la carrera que logramos entenderlo y por eso hoy lo valoramos.

Resumen

En la actualidad el diseño de las pruebas de performance se realiza de forma artesanal, y no existe ninguna herramienta que asista dicho trabajo. Nuestra propuesta es automatizar la generación de los escenarios para ejecutar pruebas de performance, a través de una aplicación que procesará los registros de usuarios y generará por sí solo escenarios de prueba de manera automatizada.

Se presenta el modelado y desarrollo completo de la herramienta para la generación de pruebas de rendimiento de aplicaciones web, utilizando como origen de registros de usuario en formato W3C, obtenidos del servidor web que aloja la aplicación objetivo o SUT (System Under Test). Para hacer coincidir los registros con alguna entidad lógica, se utiliza el modelo de RUM, donde se buscan ocurrencias de sesiones específicas, y con la ayuda de un modelo de árbol basado en procesos de Markov, se crean escenarios de prueba similares a los detectados en los archivos de registro pero con una componente estocástica. El escenario de salida con los usuarios virtuales también contiene los tiempos de espera (think times) calculados mediante un análisis estadístico realizado también sobre los registros de aplicación.

Matemáticas: 1) Procesos de Markov, 2) Histogramas y 3) Modelo árbol

Ciencia y tecnología: 1) Pruebas de rendimiento. 2) RUM, 3) Usuarios virtuales, 4) Registros web y 5) SUT

Desarrollo de software: 1) Java, 2) JMeter, 3) ASP.net MVC y 4) IIS

Abstract

At the present the design for performance tests is performed in an artisanal way, and it doesn't exist a tool to assist this work. Our proposal is to automatize the generation of performance test scenarios with an application which processes the application logs and generates the test scenarios automatically.

It is presented the complete modeling and development of a web application stress testing tool, that uses application logs retrieved from the web server hosting the SUT (System Under Test) in W3C format as source. A RUM model is used in order to match the logs with some logical entity, it searches for specific session occurrences, and with the help of a tree model based on a Markov process, it creates specific test scenarios similar to the ones detected on the log files but with a stochastic component. The output scenario containing the virtual users will also have the think times calculated using statistical analysis over the data retrieved from application logs.

Maths: 1) Markov Processes, 2) Histograms and 3) Tree Model

Science and Technology: 1) Stress tests, 2) RUM, 3) Virtual Users, 4) Web Logs and 5) SUT

Software Development: 1) Java, 2) JMeter, 3) ASP.net MVC y 4) IIS

Contenido

Pág.

Resumen ..... **IX**

Lista de figuras ..... **XIII**

Lista de tablas ..... **XIV**

Lista de símbolos y abreviaturas ..... **XV**

1 Introducción **1**

2 Estado del arte **7**
2.1 Servload 7
2.1.1 Método de multiplicar (*multiply Method*) 8
2.1.2 Método pico (peak Method) 9
2.1.3 Método de puntuación (score Method) 9
2.2 Modelo orientado a formularios 9
2.3 Modelo de Uso Realista (RUM) 11
2.3.1 Modelo Simple de Carga 14
2.4 Cadenas de Markov de tiempo continuo 16
2.5 Tiempos de espera 17
2.6 Conclusiones 18

3 Modelado del problema y descripción de la solución **19**
3.1 Modelado del comportamiento de los usuarios 19
3.2 Estimación de parámetros del modelo 23
3.2.1 Construcción del modelo árbol 24
3.2.2 Implementación para los tiempos de espera 26
3.3 Resumen del proceso completo 28

4 Aplicación **30**
4.1 Arquitectura 31
4.2 Módulos 32
4.2.1 Diagrama de clases 32
4.2.2 Subsistemas 37
4.3 Modelo de datos 44
4.4 Otros aspectos de implementación 46
4.5 Uso de JMeter 48

5 Caso de prueba **51**

5.1 Contoso University 51
5.1.1 Modelo RUM....52
5.1.2 Filtro HTTP 54
5.1.3 Ejemplo de uso....54

6 Conclusiones y trabajos a futuro....62
6.1 Conclusiones....62
6.2 Trabajos a futuro 62
6.2.1 Agregado de información semántica....63
6.2.2 Modelado de tiempos de procesamiento....63
6.2.3 Agregar características de JMeter 63
6.2.4 Realizar pruebas exhaustivas....66

7 Anexos....67
7.1 Protocolos, lenguajes y tecnologías 67

8 Referencias 69

Lista de figuras

Pág.

Figura 1-1: Esquema de las etapas de implementación de pruebas de carga 2
Figura 2-1: Ejemplo de un grafo de formularios para una aplicación bancaria [4] 10
Figura 2-2: Grafo de formularios estocástico para una aplicación bancaria [4] 11
Figura 2-3: Modelo RUM para una aplicación web simple [5] 13
Figura 2-4: Diagrama relacional para el modelo RUM de ejemplo [5] 15
Figura 3-1: Árbol ejemplo para $A = \{0,1\}$ y 2 niveles..... 22
Figura 3-2: Flujo completo para simular escenarios..... 28
Figura 4-1: Subsistemas..... 31
Figura 4-2: Aplicación Web..... 32
Figura 4-3: Diagrama de clases..... 33
Figura 4-4: Diagrama de clases de los parámetros..... 36
Figura 4-5: Capa de procesamiento..... 38
Figura 4-6: Capa de simulación. 41
Figura 4-7: Diagrama ER..... 45
Figura 4-8: Diagrama ER de la estructura utilizada para almacenar parámetros. 46
Figura 4-9: Configuración. 47
Figura 4-10: Tabla de configuración. 47
Figura 5-1: Modelo entidad relación de Contoso University 52
Figura 5-2: RUM para Estudiantes e Instructores 53
Figura 5-3: RUM para Departamentos y Cursos 53
Figura 5-4: Modelo entidad relación para la recolección de datos del filtro HTTP 54
Figura 6-1: Modelo maestro esclavo de JMeter. 64
Figura 6-2: Modelo maestro esclavo en acción..... 65
Figura 7-1: Protocolos, lenguajes, frameworks y herramientas utilizados. 67

Lista de tablas

Pág.

Tabla 1-1: 5 Herramientas más utilizadas para desarrollar pruebas de carga. 3

Tabla 2-1: Ejemplo de pedidos detectados en los registros de usuario 14

Tabla 5-1: Sesiones reconocidas por la herramienta en el log de ejemplo 55

Tabla 5-2: Datos estadísticos del modelo árbol 56

Tabla 5-3: Escenarios simulados 60

Lista de símbolos y abreviaturas

A continuación se presentan los símbolos y abreviaturas utilizadas en el presente documento.

Abreviaturas

Abreviatura	Término
SUT	System Under Test
RUM	Realistic Usage Model
SLM	Simple Load Model
CMTC	Cadena de Markov de Tiempo Continuo
EMC	Embedded Markov Chain
HTTP	Hypertext Transfer Protocol
IIS	Internet Information Services
W3C	World Wide Web Consortium
RDF	Resource Description Framework
Vuser	Usuario virtual

1 Introducción

Nos ubicamos en el mundo de las aplicaciones web, donde se ha convertido en una práctica habitual el desarrollo de pruebas de carga para evaluar el desempeño de la aplicación en situaciones de alta exigencia. Existen varias razones por las cuales realizar este tipo de pruebas, entre las que se destacan:

- Identificar los componentes que consumen muchos recursos
- Localizar errores que solamente surgen en situaciones de alta carga
- Obtener indicadores en requerimientos no funcionales de la aplicación
- Definir los límites de stress máximo que puede tolerar la aplicación bajo determinada configuración
- Probar distintas configuraciones para obtener la que presenta el mejor desempeño

Frente al auge de este tipo de aplicaciones, las cuales han ido paulatinamente remplazando a los sistemas clásicos de ventanas o consola, los estudios en el área han proliferado, apareciendo en el mercado diversas herramientas que facilitan el modelado y la ejecución de las pruebas de carga. El mayor inconveniente que presenta este tipo de pruebas es que el uso de cada aplicación debe ser estudiada concretamente, y se debe elaborar un modelo de carga que sea consistente y representativo con el uso real de la aplicación, para que de esa manera los resultados obtenidos a posteriori provean información útil. Esta no es una tarea fácil y a menudo es la que lleva mayor tiempo del plan de pruebas.

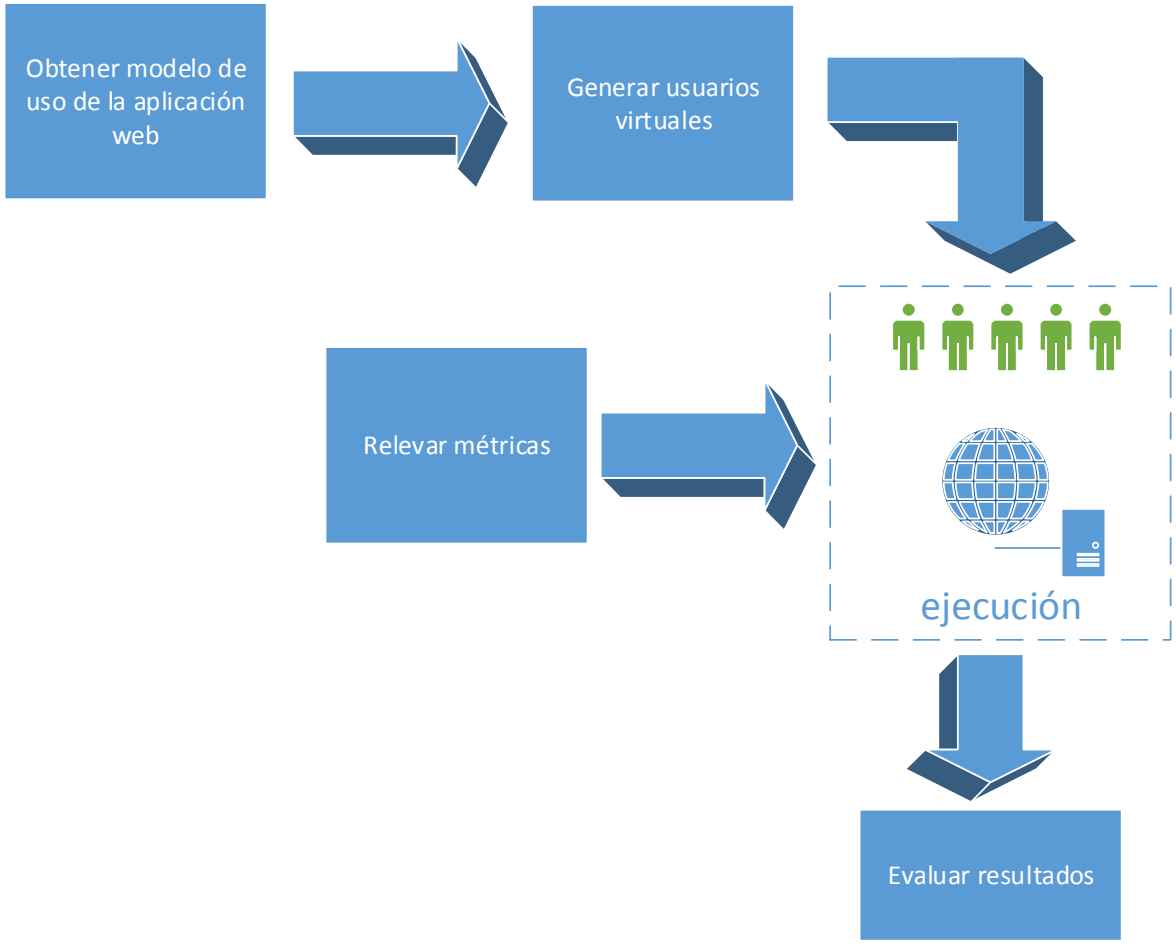
Una vez han sido diseñados dichos modelos de carga, se recrean los escenarios como usuarios virtuales, los cuales serán ejecutados posteriormente contra el SUT.

Dichos usuarios virtuales se pueden implementar manualmente mediante scripts, o lo que es más común, utilizar una herramienta que permita recrear paso a paso los escenarios en un explorador web real, en donde se capturan todos los pedidos realizados y se generan los scripts para los usuarios virtuales de manera automática.

La etapa de ejecución consta de correr dichos usuarios virtuales contra el SUT, mientras se monitorizan en tiempo real las métricas que son consideradas relevantes. Dichos datos serán los que finalmente se analizan.

En la **Figura 1-1** presentamos el flujo completo para el modelado y ejecución de pruebas de carga.

Figura 1-1: Esquema de las etapas de implementación de pruebas de carga



Se destacan en el mercado diversas herramientas, tanto libres como de como de código abierto, que brindan facilidades para describir y ejecutar los scripts para los usuarios virtuales, recolectar métricas en tiempo real, y exponer diversos reportes para su posterior análisis. Sin embargo no existen aplicaciones que ayuden al analista a obtener un modelo de uso real de la aplicación web, tarea de vital importancia para que el resultado del estudio sea representativo de la realidad.

Tabla 1-1: 5 Herramientas más utilizadas para desarrollar pruebas de carga.

Nombre	Modalidad
Apache JMeter	Open Source
LoadRunner	Comercial
WebLoad	Comercial
Appvance	Comercial
NeoLoad	Comercial

La **Tabla 1-1** enumera las 5 herramientas más utilizadas para ejecutar pruebas de carga [1]. Dichas herramientas, tienen una o varias funcionalidades que las hace destacar una sobre la otra, sin embargo no existe ninguna que provea facilidades para el análisis del uso real de la aplicación y generar en base a ello los usuarios virtuales. Éste nicho dejado por las herramientas existentes es una de las principales motivaciones que dieron a luz al presente proyecto.

Los tipos de SUT que considera nuestra herramienta son los que disponen de un servidor de aplicación configurado para guardar registros en formato W3C, y que se encuentren en producción por un tiempo necesario como para recolectar datos de una cantidad de usuarios suficiente para generar escenarios realistas. No tiene sentido utilizar la herramienta en aplicaciones que no han sido puestas en producción ya que los datos de registro de actividad contendrán información generada artificialmente (por ejemplo por los desarrolladores o beta testers) y no es fiel reflejo del uso real que tendrá dicha aplicación, por lo que para estos casos se impone el uso del método tradicional de análisis de funcionalidades y generación manual de escenarios de prueba.

Proponemos generar carga representativa del uso real de la aplicación web de una manera automatizada, liberando de esa manera al analista del trabajo de deducir los patrones de uso reales de la aplicación web, concentrando su esfuerzo en el análisis de los resultados y en la resolución de potenciales problemas o cuellos de botella. Para ello se toman como entrada los registros de uso real de la aplicación (logs en inglés), y a través de un modelo (que describiremos en el capítulo 3) generamos los usuarios virtuales que tienen un comportamiento semejante al de los usuarios reales de la aplicación desde un punto de vista estadístico. Si bien la herramienta brinda una solución integral para la ejecución de las pruebas de performance (descripta de forma completa en el capítulo 4), la misma entrega un resultado intermedio. Se trata de los resultados de la simulación luego de aplicado el modelo árbol de nuestro modelo (ver capítulo 3). Estos resultados podrán ser tomados por el analista para ser utilizados con otras herramientas foráneas, aprovechando así las características específicas de éstas en la ejecución de los scripts y la posterior recolección de datos. Por ejemplo se puede utilizar nuestra herramienta para procesar los registros de uso del SUT, y simular una determinada cantidad de usuarios para exportar los resultados en formato CSV. Luego con estos datos se generan scripts de manera manual, los cuales imitarán a los escenarios exportados como si se trataran de un guion, y con ayuda de otra herramienta (por ejemplo LoadRunner) se ejecutan los scripts y efectúa el posterior análisis de los datos.

En el capítulo 2, que está destinado al estudio del estado del arte, describimos en profundidad los artefactos y mecanismos existentes que nos fueron de utilidad para modelar partes de nuestra herramienta. También presentamos definiciones y/o mecanismos teóricos que nos fueron de utilidad, los cuales sin embargo en general no son aplicables en el mundo real tal cual están planteados. La conclusión de nuestro estudio es que si bien existen diversos trabajos en el área, ninguno presenta un caso de aplicación completo y realista. Además existen diversas áreas inexploradas que serán las que en definitiva son cubiertas por nuestro trabajo. Nuestro modelo ha tomado del estado del arte principalmente el denominado RUM (Modelo de Uso Realista) para fijar relaciones entre los registros de uso (que presentan pedidos de bajo nivel) y conceptos de alto nivel (denominados sesiones) que son equivalentes a los casos de uso del SUT. También tomamos conceptos sobre el modelado de los tiempos de espera para generar un modelo de estimación basado en histogramas, los cuales son calculados tomando los tiempos reales desde los registros de uso. Finalmente se analiza un mecanismo para el

modelado de la carga basado en cadenas de Markov de tiempo continuo, que de alguna manera sienta precedentes del uso de éste tipo de simulaciones para la generación de carga real, y como se muestra en el capítulo 3 son el tipo de simulaciones que utilizamos para la generación de escenarios.

En el capítulo 3 abarcamos el estudio del modelo desarrollado para nuestra solución, y analizamos los aspectos teóricos, explicando el uso de cada parte que lo compone. Se plantean además las limitaciones del modelado propuesto, como ser que el analista deberá proveer una fuente de datos para ejecutar las simulaciones, es decir, los datos que irán desde el usuario virtual hacia el servidor, o que los tiempos de espera se simularán basados solamente en el tipo de caso de uso, o sea que no se considerarán otras variables en la estimación de dichos tiempos. Veremos que variables como la historia del escenario (los pasos previos hasta llegar a un punto dado del escenario) y el volumen de datos que interviene en cada caso de uso resultan importantes a la hora de modelar los tiempos de espera que consumen los usuarios para responder a cada pedido. Hacemos mención de estos aspectos para dejar sentado que los mismos fueron analizados, aunque finalmente hayan quedado por fuera del alcance de nuestro proyecto.

En el capítulo 4 analizamos los aspectos arquitectónicos de la aplicación desde un punto de vista de ingeniería de software, mostrando el diseño de los diversos componentes, para llegar al modelado conceptual y diagrama de clases de cada módulo. Allí se exponen las decisiones de implementación que se han tomado y las posibilidades de extensión a futuro soportadas por la arquitectura.

En el capítulo 5 presentaremos un caso de prueba (Contoso University [2]) el cual nos provee un modo de evaluación para nuestra solución a través de una aplicación web real. Además mostramos la implementación de un filtro HTTP en ASP.net que captura el flujo de datos completo entre cliente y servidor, volcando dichos datos a una base de datos SQL Server. Esta base de datos nos servirá luego como entrada para ejecutar las simulaciones y obtener los datos de rendimiento. De esta manera mostramos un mecanismo que puede implementar el analista para obtener una base de datos que luego sirva para ejecutar las simulaciones, que como mencionamos más arriba es un prerrequisito de nuestra herramienta.

Finalmente, en el capítulo 6 realizamos las conclusiones finales y sugerimos posibles trabajos a futuro.

2 Estado del arte

En el presente capítulo describimos los trabajos de investigación relacionados con nuestro proyecto, los cuales fueron relevados durante la etapa de investigación.

Las aplicaciones web tienen características que las hacen especiales en comparación con otro tipo de sistemas interactivos, en particular en lo que tiene que ver con los flujos de información devenidos de la aplicación del paradigma cliente-servidor.

Durante la fase de estudio del estado del arte encontramos algunos trabajos preexistentes sobre la temática, sin embargo éstos presentan diversas limitaciones u objetos de estudio abiertos. La restricción más común que encontramos es que en general todos se basan en sitios web estáticos y no consideran las aplicaciones web dinámicas, las cuales son el objetivo de nuestro trabajo.

Describimos entonces a continuación las construcciones del estado del arte existentes para el modelado de los flujos, así como la simulación de los mismos.

2.1 Servload

Para comenzar nuestro estudio, describimos el sistema *servload*, presentado en [3], cuyos objetivos son similares a los de nuestro trabajo.

Resulta interesante estudiar los artefactos planteados por *servload* para la obtención de carga representativa a partir de los registros de uso, y si bien no son los mismos que utilizamos en nuestro trabajo, éstos presentan un esquema general para la resolución de este tipo de problemas (es decir, de obtener carga representativa a partir de los registros de uso). La principal carencia de este trabajo es que se basa en reproducir pedidos a nivel de red con un enfoque genérico sin considerar ningún aspecto específico de la SUT a nivel de software. De hecho su principal

objetivo es el de medir el desempeño de un servidor de aplicaciones, y no medir el desempeño de una aplicación específica.

En una primera instancia el artículo presenta una serie de métricas para analizar los registros de uso del SUT e identificar y cuantificar los diferentes usuarios y su comportamiento para poder generar carga representativa y poder reproducirla de forma apropiada.

Luego se plantean una serie de campos enumerados como una clave para los pedidos de bajo nivel, de manera de poder identificar unívocamente las distintas ocurrencias en los registros de uso. Los campos que forman dicha clave son (en orden de prioridad): *usuario remoto, id de sesión, host remoto y user agent, o host remoto.*

A continuación, para identificar el comportamiento de los usuarios se discuten dos aspectos, que son la cantidad de pedidos realizados en una sesión y el cálculo de los tiempos de espera (en segundos). Las opciones planteadas son tomar el tiempo transcurrido entre la llegada de dos pedidos consecutivos, y tomar el tiempo entre dos acciones concretas de un usuario, teniendo en cuenta que una acción se puede componer de varios pedidos. Finalmente el modelo planteado opta por usar la primera definición porque resulta sencillo de extraer desde los registros de uso. En nuestro caso como veremos más adelante, también optaremos por esta forma por el mismo razonamiento.

En resumen las métricas registradas por el método son el *número de pedidos en una sesión, el largo de una sesión, y el tiempo de espera entre pedidos.*

Para finalizar se plantean tres métodos para la generación de carga representativa, los primeros dos añaden pedidos adicionales, mientras que el tercero estima sesiones y añade casos típicos. Como veremos más adelante, nuestro trabajo plantea una metodología radicalmente distinta para la generación de carga representativa, no obstante resulta interesante resumir estos métodos como posibles opciones, además de ser una introducción a la temática.

2.1.1 Método de multiplicar (*multiply Method*)

Este método se basa en repetir una cantidad fija de veces cada pedido (lo que se denomina factor de inserción). El resultado es de al menos el doble del total de pedidos.

Este método resulta en un aumento en la diferencia en la cantidad de pedidos entre los momentos en que la cantidad de pedidos es baja y los momentos en los que hay muchos pedidos. En otras palabras los momentos en que existe más saturación se saturarán aún más en proporción con aquellos momentos en los que se encuentran menos pedidos.

2.1.2 Método pico (peak Method)

Se toman 2 valores, denominados *factor de inserción*, que al igual que en el método de multiplicar es la cantidad de veces que se repetirán los pedidos seleccionados, y un *umbral* que define la cantidad de apariciones que debe tener un pedido para ser repetido.

Se toman entonces intervalos de tiempo fijos, y solo aquellos pedidos cuya cantidad de apariciones en dicho intervalo superen el valor del *umbral* serán multiplicados por el *factor de inserción*.

2.1.3 Método de puntuación (score Method)

Este método identifica sesiones y las califica. Las sesiones con mayor calificación son las típicas (de mayor ocurrencia) y serán las utilizadas para aumentar la carga.

La calificación está basada en cuatro métricas: *número de pedidos, largo de sesión, tiempo de think time medio, y cantidad media de bytes.*

A partir de las métricas se define un algoritmo que calcula la calificación para las sesiones fuente (sesiones que fueron ejecutadas por usuarios reales y capturadas por los registros de uso del SUT) y devuelve el conjunto de sesiones a ser simuladas.

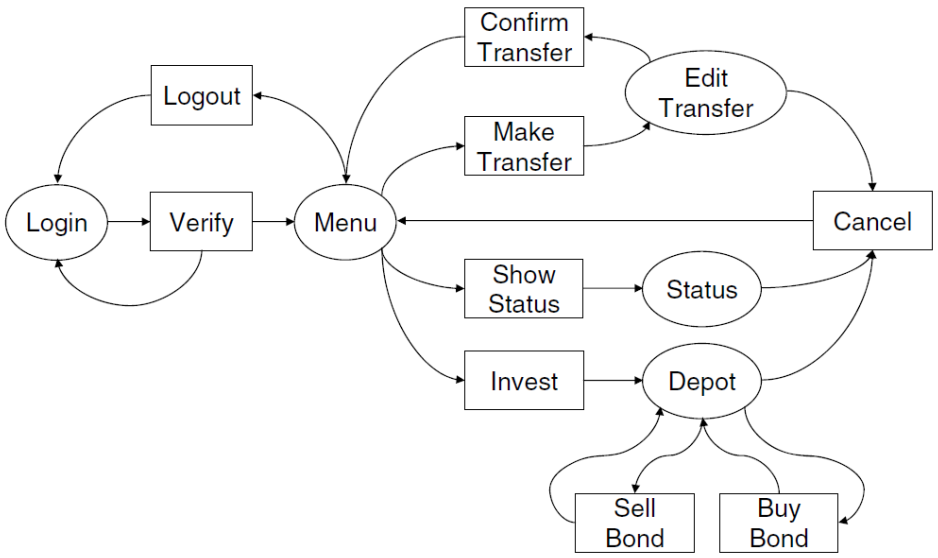
2.2 Modelo orientado a formularios

El modelo orientado a formularios (form oriented) [4] se trata de un artefacto de utilidad para modelar los flujos cliente/servidor existentes en una aplicación web a nivel de funcionamiento, es decir a nivel de las páginas que forman parte de la aplicación. Esto nos permite representar la interacción existente entre las distintas páginas de manera esquemática en forma de una máquina de estados, que consta de páginas, acciones e interacciones entre ellas. El artículo [4] nos explica detalladamente el modelo y nos presenta sus propiedades a través de ejemplos, analizando además su aplicación

mediante diversas técnicas para generar carga realista sobre aplicaciones web basada en los resultados.

Estas máquinas de estado se visualizan a través de grafos de formularios (form charts), que son una representación muy similar a la representación tradicional de una máquina de estados. Las páginas son representadas con óvalos, las acciones con rectángulos, y las transiciones por flechas. Estos elementos forman un grafo dirigido en el cuál las páginas y las acciones se suceden unas a otras en las transiciones, es decir, de una página solo se puede ir hacia una acción y de una acción solo se puede ir hacia una página como se muestra en el ejemplo de la **Figura 2-1**.

Figura 2-1: Ejemplo de un grafo de formularios para una aplicación bancaria [4]

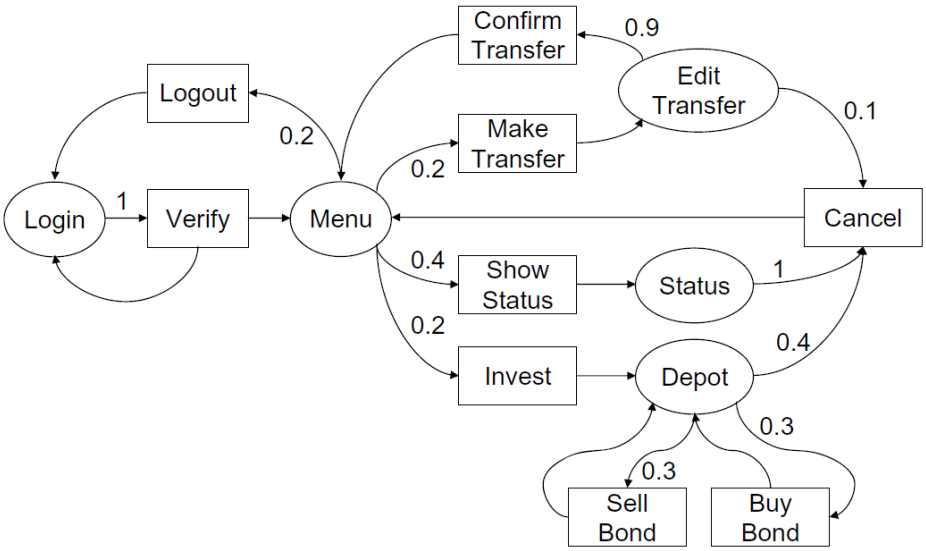


A los efectos de modelar estocásticamente el comportamiento de los usuarios del sistema, se extiende el modelo definiendo distribuciones de probabilidad para los formularios a los que se accede, para los tiempos de permanencia en cada formulario, y para los datos que se ingresan en cada formulario.

En [4] los autores exponen las posibilidades que existen de acuerdo a la forma en la que se estiman estas distribuciones de probabilidad, y proponen un mecanismo de estimación

a través de un grafo bipartito estocástico, donde se asignan las probabilidades de transición contando las ocurrencias de cada una en un historial.

Figura 2-2: Grafo de formularios estocástico para una aplicación bancaria [4]



Como vemos en el ejemplo de la **Figura 2-2** , se agrega a cada formulario del modelo básico la probabilidad condicional de transición de cada página a cada evento alcanzable desde dicha página. Los eventos tienen salida determinista, es decir que cada evento genera una transición hacia un formulario en particular con probabilidad 1. Queda de esta manera definido un proceso estocástico.

2.3 Modelo de Uso Realista (RUM)

Si bien el modelo de grafos de formularios presentado en el párrafo anterior define las bases para la simulación concreta del comportamiento de los usuarios basados en el historial de uso, no resulta en una buena representación de los casos de uso de una aplicación vista a nivel funcional, porque expresa transiciones entre formularios atómicos, y no captura procesos más complejos de los que potencialmente forman parte dichos formularios. Esta es una carencia importante ya que a la hora de simular determinados escenarios, puede no ser suficiente con generar un pedido a una cierta página, ya que

dicho pedido depende de pedidos anteriores que se debieron dar en un orden específico, y el flujo de información es dependiente del proceso completo.

A modo de ejemplo, supongamos que tenemos un proceso para transferir un monto de una cuenta bancaria a otra, que consta de 3 pedidos ordenados a tres formularios distintos:

- Seleccionar la cuenta de origen
- Seleccionar el monto
- Seleccionar la cuenta destino

Podemos ver fácilmente que el proceso de traspaso es válido solamente si la secuencia de pedidos a formularios es la correcta, y además estos formularios podrían aparecer como parte de otros procesos, pero al tratarlos de forma atómica se podrían generar simulaciones que a nivel de negocio son incorrectas.

Un modelo orientado a formularios con mayor expresividad fue presentado en el artículo [5], que define el modelo RUM (*Realistic Usage Model* en inglés). Este modelo es de gran utilidad para describir las secuencias de acciones que forman parte de un proceso de negocio de la aplicación web. Se trata de un modelo sumamente expresivo, y a su vez está fuertemente relacionado con el aspecto transaccional de las aplicaciones web ya que captura el funcionamiento de la aplicación a nivel funcional, que es una propiedad buscada por las razones antes expuestas, sin perder su relación con los pedidos HTTP de bajo nivel.

Un modelo RUM se construye en base a los siguientes conceptos.

PEDIDO es un pedido HTTP de un recurso específico. Se trata del concepto de más bajo nivel manejado por el modelo.

ACCION es una acción concreta efectuada por un usuario, y tiene relacionado al menos un objeto del tipo PEDIDO.

SESION es el concepto que se acerca más al nivel de casos de uso desde una perspectiva de ingeniería de software. Se trata de una secuencia ordenada de objetos ACCION que el usuario debe llevar acabo para ejecutar una funcionalidad específica.

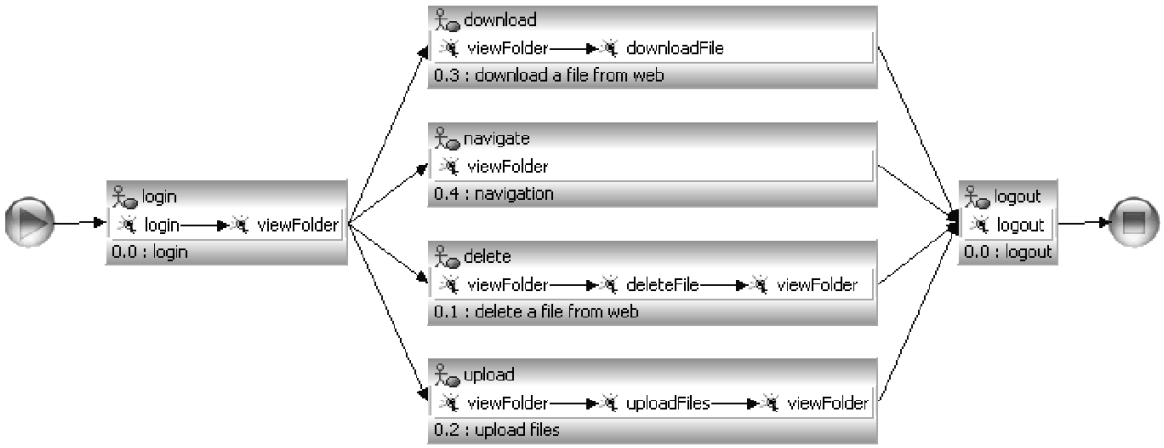
ESCENARIO es un conjunto de sesiones específicas invocadas por un usuario. A fines prácticos se trata de un conjunto de funcionalidades utilizadas por un usuario real en el sistema, ordenadas según su aparición en los registros de uso, es decir una instancia de uso de la aplicación web por parte de un usuario.

En la **Figura 2-3** vemos un ejemplo de una instancia de RUM generada para una aplicación web simple que tiene 4 posibles sesiones (download, delete, upload, navigate). Se pueden observar que dentro de los recuadros que modelan las sesiones, se define la secuencia ordenada de acciones que definen dicha sesión. Las flechas que conectan las sesiones muestran las posibles transiciones, en particular el ciclo de vida de un escenario para esta aplicación incluye el login como primera sesión en todos los casos, luego cualquiera de las 4 sesiones podrán ser invocadas arbitrariamente, y finalmente para cerrar un escenario se tiene la sesión especial de logout.

El ejemplo nos sirve para ver qué papel juegan los conceptos del RUM en un caso real, y brinda una idea de cómo podemos aplicarlo.

El RUM brinda artefactos que permiten modelar cómo se conforman los procesos de negocio del SUT, y cómo mapear dichos procesos a partir de los pedidos de bajo nivel para poder detectar en un registro de uso de la aplicación los procesos que efectivamente el usuario ha ejecutado.

Figura 2-3: Modelo RUM para una aplicación web simple [5]



2.3.1 Modelo Simple de Carga

Una vez creada la instancia de RUM que describe el comportamiento de la aplicación, debemos utilizar un modelo de carga para ejecutar una prueba concreta sobre el sistema.

Para esto el artículo [5] plantea un modelo dependiente del RUM llamado Modelo Simple de Carga (Simple Load Model o *SLM* en inglés), el cuál es expresado mediante un diagrama de relación de la aplicación web. Se trata de un grafo topológico de 3 niveles que describimos a continuación (desde el nivel final hacia el primer nivel).

En el *nivel 3* tenemos los pedidos a ser detectados en el registro de usuario, en el orden en el cual deberán suceder efectivamente para formar la acción a la cual corresponden.

En el *nivel 2* se agrupan los pedidos del nivel 3 en acciones.

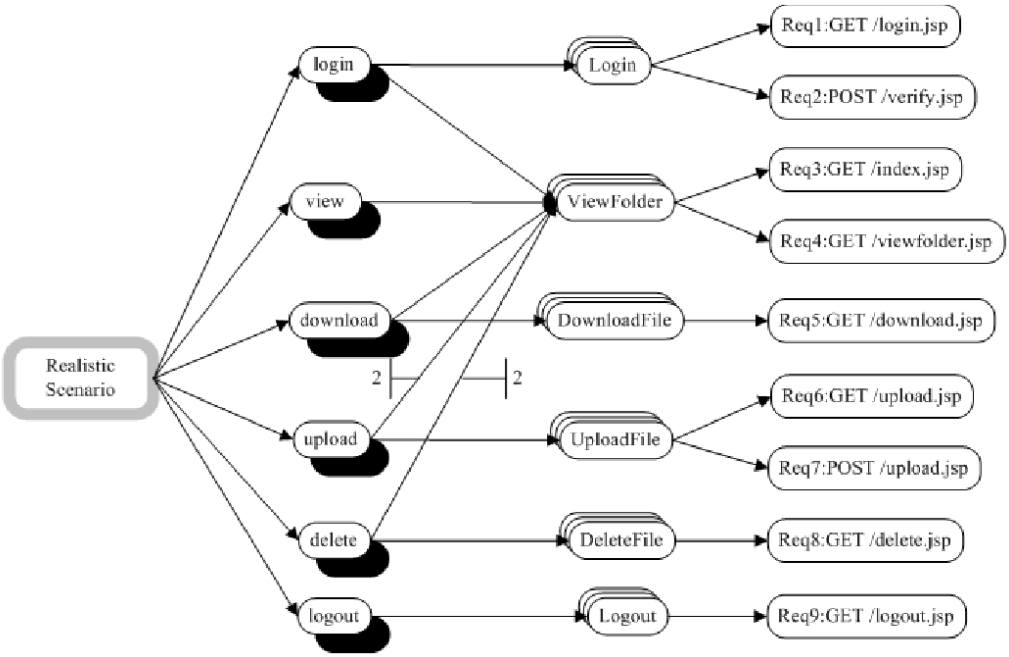
Finalmente, en el *nivel 1* tenemos las sesiones del RUM, que son agrupaciones de las acciones del nivel 2.

En base a este diagrama (que depende fuertemente del RUM) y la lista de pedidos concreta obtenida del registro de usuarios de la aplicación web, el modelo SLM calcula las probabilidades de ocurrencia de cada sesión contabilizando las ocurrencias en el registro de usuario, obteniendo así la probabilidad de ocurrencia para cada sesión concreta según el histórico.

Tabla 2-1: Ejemplo de pedidos detectados en los registros de usuario

Pedido	Url	Ocurrencias
Req1	GET /login.jsp	3
Req2	POST /verify.jsp	3
Req3	GET /index.jsp	130
Req4	GET /viewfolder.jsp	130
Req5	GET /download.jsp	30
Req6	GET /upload.jsp	20
Req7	POST /upload.jsp	20
Req8	GET /delete.jsp	10
Req9	GET /logout.jsp	2

Figura 2-4: Diagrama relacional para el modelo RUM de ejemplo [5]



Calculadas las probabilidades, el modelo ofrece un mecanismo para sortear un grupo de sesiones y formar de esa manera un escenario concreto para simular en el SUT.

El estudio de este modelo nos proporciona una primera aproximación al problema de generar carga realista utilizando los datos del RUM.

2.4 Cadenas de Markov de tiempo continuo

El modelo RUM en conjunto con el SLM son suficientes para modelar los aspectos de la aplicación y el comportamiento de un determinado usuario bajo el SUT, sin embargo el artículo [5] no presenta ninguna técnica para simular un flujo basado en datos estadísticos relevados de una aplicación dinámica.

El artículo [6] presenta un modelo de carga basado en Cadenas de Markov de Tiempo Continuo (CMTC) en donde se definen las páginas web como estados, y se calculan las probabilidades de permanencia o transición a otra página enumerando las posibilidades de abandonar dicha página. El cálculo de las probabilidades se realiza a través de un análisis de cada página de la aplicación, y contando los posibles eventos que pueden llevarnos a abandonar dicha página hacia otra, de volver a la página, o eventualmente terminar la sesión del usuario allí. Se presentan para tal fin un conjunto de definiciones para capturar todos los eventos mencionados, y se realizan los cálculos de probabilidad condicional en base a dichas definiciones.

La principal carencia de esta representación es que se trata de una cadena de orden 1, la cual podría no capturar correctamente el funcionamiento de una aplicación web a nivel de procesos de negocio.

Otro problema del modelo es la dificultad para calcular las distribuciones de probabilidad condicional para los estados de la cadena de Markov en base al funcionamiento de una aplicación web dinámica. Realizar un análisis para aplicaciones dinámicas implica conocer la posibilidad de salida, regreso, reinicio (entre otras) para cada página, que potencialmente puede servir para varios procesos distintos y generar respuestas distintas en base al flujo previo, por lo que no se puede reconocer de una manera directa a qué eventos responde cada aparición de una página. Es por eso que en el artículo se presenta un ejemplo con un sitio web estático donde es sencillo deducir las transiciones y calcular las probabilidades, pero no se presentan ejemplos para aplicaciones dinámicas donde estos cálculos se vuelven complejos.

2.5 Tiempos de espera

Los tiempos de espera son el tiempo que le insume a un usuario real procesar una página obtenida desde el servidor, e invocar otra operación sobre el sistema como consecuencia.

El análisis estadístico de los tiempos de espera (*think times* en inglés) presentan diversas complejidades a la hora de simular el real comportamiento de un usuario. En particular todos los usuarios tienen niveles distintos de conocimiento y diversas velocidades de respuesta basadas por lo general en su experiencia. Además el volumen de datos que obtiene el usuario es variable y afecta el tiempo de respuesta del mismo, sin embargo no resulta simple capturar dichas dependencias.

En [4] se presenta un modelo que estima los tiempos de espera de una manera sencilla utilizando las funciones de densidad para cada formulario, los cuales son construidos en base a la información histórica.

Un concepto interesante de [4] es el de multi-modelos, donde los usuarios se agrupan en diferentes categorías de acuerdo al comportamiento estadístico de sus tiempos de espera. Para cada categoría de usuario se realiza una estimación diferente del modelo de tiempos de espera.

El concepto detrás de esto es el de capturar y diferenciar el comportamiento de los usuarios según qué tan expertos en el uso de la aplicación son, lo cual puede deberse por ejemplo a la asiduidad con la que usan la aplicación cada categoría de usuarios.

2.6 Conclusiones

Nuestro estudio del estado del arte nos hizo transitar por varios modelos para representar el flujo de uso de una aplicación web, los cuales en su mayoría presentan deficiencias a la hora de modelar sitios web dinámicos, con excepción del RUM que resulta en un modelo poco explorado por la comunidad y con gran potencial.

Además, nos permitimos concluir que parece razonable el uso de procesos de Markov para modelar los flujos, dado que existen diversos trabajos que utilizan este mecanismo con buenos resultados.

Finalmente, concluimos en que no existen trabajos previos que unan todos estos conceptos. Si bien existe un prototipo que plantea objetivos similares a los de nuestro trabajo [3], éste tiene como fin la evaluación del rendimiento de servidores de aplicaciones web, y no considera en ningún momento la aplicación bajo prueba (SUT). Además, como veremos más adelante, para modelar los aspectos en, hemos tomado otros caminos que se ajustan mejor a la evaluación del desempeño de aplicaciones, que es en definitiva el objetivo de nuestro trabajo. Finalmente, como detalle interesante, éste artículo hace alusión al problema de simular flujos reales utilizando RUM como un problema abierto, y como veremos en el próximo capítulo, nuestro trabajo presenta un método para su resolución.

3 Modelado del problema y descripción de la solución

En el presente capítulo describimos la solución que elaboramos en este proyecto.

En la sección 3.1 presentamos un modelo para definir cómo se comporta un usuario concreto a nivel abstracto, presentando los conceptos requeridos para entender el planteo del modelo de carga.

En la sección 3.2 explicamos cómo estimamos los parámetros del modelo a partir de los registros de uso de aplicación y del RUM de la aplicación, definiendo de manera constructiva un modelo de carga que se basa en un modelo árbol para generar los escenarios, y un modelo estadístico basado en histogramas para estimar los tiempos de espera.

3.1 Modelado del comportamiento de los usuarios

Los usuarios de una aplicación web pueden ser vistos como una entidad que invoca una secuencia ordenada de operaciones específicas sobre la aplicación a lo largo del tiempo. Estas operaciones, invocadas concurrentemente por todos los usuarios, son las que en definitiva determinan el esfuerzo de cómputo que debe soportar el sistema. En consecuencia, modelar adecuadamente este comportamiento resulta fundamental para realizar una buena prueba de rendimiento.

Cada operación ejecutada por el usuario involucra determinada carga de procesamiento que depende de la operación concreta que se está realizando y el tamaño de los datos intercambiados. En consecuencia, tanto el tipo de operaciones que realizan los usuarios como los volúmenes de datos intervinientes en estas operaciones son variables importantes para modelar su comportamiento. Otra variable que influye sobre el desempeño de un sistema es el tiempo de espera necesario por el usuario para procesar

los datos recibidos y responder en consecuencia (por ejemplo, leer un texto y/o rellenar un formulario). Frente a una misma secuencia de operaciones, un sistema puede ser capaz de responder adecuadamente o no, dependiendo del espacio de tiempo que medie entre las solicitudes de procesamiento asociadas a operaciones consecutivas.

Una consideración similar aplica al orden en que los usuarios realizan las diferentes operaciones. Las mismas operaciones, realizadas en diferente orden, pueden tener un impacto diferente sobre el rendimiento del sistema debido, por ejemplo, al mejor o peor aprovechamiento de los diferentes niveles de jerarquía de memoria (memoria cache, memoria virtual, etc.).

A la luz de estas observaciones, modelamos el comportamiento de un usuario a través de secuencias de ternas $\langle o_i, t_i, d_i \rangle$ donde

- o_i es una operación específica que es invocada por el usuario
- t_i es el tiempo de espera entre la operación o_i y o_{i-1}
- d_i es el conjunto de datos que se intercambia entre cliente y servidor durante la realización de la operación o_i

En nuestro modelo, el tiempo de espera t_i depende únicamente de la operación o_{i-1} y del tiempo transcurrido hasta que se registra la operación o_i, lo que es una simplificación de la realidad ya que dicho tiempo podría no depender únicamente de la operación efectuada anteriormente. Por ejemplo si tenemos un asistente que consta de varios pasos, la primera pantalla puede determinar cuántos datos se cargan en las pantallas siguientes y por ende modificar el tiempo de procesamiento. Sin embargo dichas dependencias no son fáciles de capturar porque están vinculadas a los eventos que a su vez están sujetos al comportamiento de los distintos procesos de negocio definidos por la aplicación, es decir que se debería describir de alguna forma semántica qué decisiones toma la aplicación en cada operación para tomar decisiones sobre cada caso. Se opta por lo tanto por esta simplificación.

La variable d_i es difícil de modelar. Por ejemplo, los datos ingresados por un usuario muchas veces deben cumplir restricciones que dependen del estado del sistema en el momento específico que se realiza la operación (por ejemplo, un número de cliente que esté previamente cargado en una base de datos). Por este motivo, no definimos un

modelo específico para d_i. Dejamos a cargo del analista el suministro de una base de datos que abastece a los usuarios virtuales para ejecutar las simulaciones, y damos por hecho que estos datos son representativos de la realidad y son estadísticamente similares a los que ocurren durante un uso real de la aplicación.

Las operaciones o_i se corresponden uno a uno con casos de uso de la aplicación.

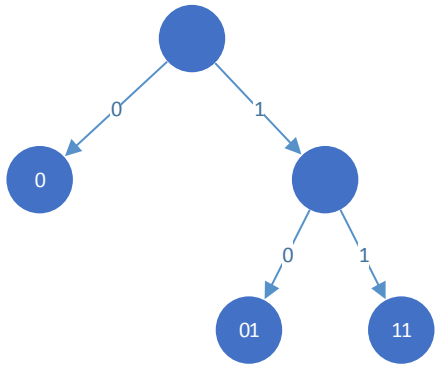
Nuestra herramienta tiene como objetivo generar escenarios potencialmente distintos a los detectados, pero que mantengan las propiedades estadísticas originales, para ello tratamos los escenarios, que son secuencias de sesiones, como una tira de largo arbitrario, y como veremos a continuación se utilizará un modelo árbol para simular nuevas tiras de largo arbitrario.

Para una aplicación determinada existen, por lo tanto, un conjunto finito O de α operaciones posibles que puede realizar un usuario. Modelamos la secuencia de operaciones ejecutada por un usuario mediante un modelo árbol [7] sobre el alfabeto O (también conocido como modelo de Markov de largo variable en estadística), que definimos a continuación.

El modelo árbol se define a través de un árbol α-ario completo finito, donde cada arista que parte de un nodo interno se encuentra etiquetada con un símbolo de los posibles α. Identificamos cada nodo con la concatenación de las etiquetas de las aristas en el camino que va desde el nodo hacia la raíz.

Por ejemplo, supongamos que tenemos un alfabeto formado por el símbolo 0 y 1, entonces un posible árbol resultante de 2 niveles podría ser el que se muestra en la **Figura 3-1**.

Figura 3-1: Árbol ejemplo para $A = \{0,1\}$ y 2 niveles



Para una secuencia lo suficientemente larga, $x = x_1, x_2, \dots, x_n$ decimos que x selecciona el único estado que es sufijo de la secuencia, es decir el estado cuyo nombre sea el más largo talque coincida con los últimos elementos de la tira. Por ejemplo, en el árbol de la **Figura 3-1**, la secuencia $x = 0001001$ selecciona el estado 01, mientras que la secuencia $x = 000100$ selecciona el estado 0.

A efectos de que toda secuencia de símbolos seleccione un estado, independientemente de su largo, se toma por convención que la tira x se encuentra precedida por una cantidad infinita de repeticiones de un símbolo fijo (por ejemplo 0). Dada esta convención para nuestro ejemplo, la tira 1 selecciona el estado 01.

Cada estado s tiene asociada una distribución de probabilidad condicional sobre A, denotada como $P(\cdot\,|s)$, que determina la probabilidad de aparición de un símbolo en una secuencia dados todos los símbolos anteriores. Específicamente, la probabilidad de x_{n+1} está determinada por la distribución condicional asociada al estado seleccionado por $x = x_1, x_2, \dots, x_n$, definiendo por consiguiente un proceso de Markov en el cual la cantidad de símbolos pasados que determinan la distribución del siguiente varía de una posición a otra (de allí el nombre de modelo de Markov de largo variable).

3.2 Estimación de parámetros del modelo

Los registros de aplicación son capturas en bruto de los pedidos HTTP realizados sobre la aplicación web, los cuales presentan las características básicas que provee el protocolo. En particular para nuestro proyecto se escogió como formato a procesar el W3C [8] por ser el más difundido entre los servidores de aplicación web en el mercado, y además presenta un esquema fácilmente procesable.
Por sus características, la información provista es de muy bajo nivel, y en general no presenta información suficiente como para extraer patrones de uso de la aplicación de manera que puedan ser fácilmente reproducibles por usuarios virtuales en la etapa de simulación. Es por esto que tenemos la necesidad de modelar la aplicación con algún mecanismo que a posteriori pudiera ser emparejado con las apariciones concretas de pedidos en los registros de aplicación.

Como vimos en el capítulo 2, el modelo RUM [5] es un modelo muy descriptivo y se asemeja mucho a lo que necesitamos, además de ser fácilmente extensible para agregar más funcionalidades que asistan al mecanismo de detección de sesiones en los registros de usuarios. Es por esto que lo tomamos para el modelado de los casos de uso de la aplicación.

Tenemos entonces 2 componentes bien definidas:

- Modelo RUM
- Registros de aplicación reales

Los casos de uso a nivel de ingeniería de software se corresponderán uno a uno con sesiones del RUM, las cuales a su vez se desglosan hasta traducirse a pedidos web que componen las distintas acciones del modelo. Por lo tanto, la elaboración de este modelo requiere la intervención de un analista con conocimiento del SUT, ya que en algunos casos la correspondencia entre las acciones del RUM y los pedidos web realizados en la aplicación no es trivial.

Una vez construido el RUM podemos buscar las ocurrencias de las distintas sesiones en el registro de aplicación de una manera tan precisa como se desee (dependiendo del nivel de extensión que queramos agregar al RUM), y por consiguiente se puede obtener

a partir de registros de aplicación, un listado de usuarios reales con sus correspondientes sesiones del RUM invocadas.

3.2.1 Construcción del modelo árbol

Para estimar el árbol [7] a partir de una secuencia, comenzamos con un árbol completo de profundidad k, con k constante predefinida.

Para una cadena s y un símbolo a definimos

$$n_s^a = |\{i: 1 \leq i < n, x_i = a, s\ es\ sufijo\ de\ x_{-\infty} \dots x_{i-1}\}|\ ,$$

donde recordamos que la tira x está precedida de infinitas repeticiones de un símbolo fijo. Observamos que para un nodo interno del árbol, s, y un símbolo a, se cumple

$$n_s^a = \sum_{b \in A} n_{bs}^a.$$

Esto significa que habiendo calculado n_s^a para toda hoja s y todo símbolo a, quedan definidos los valores n_u^a para los nodos internos u simplemente recorriendo sus hijos. Definimos luego para cada nodo s

$$n_s = \sum_{a \in A} n_s^a,$$

$$K(s) = -\sum_{a \in A} n_s^a \log_2 \left(\frac{n_s^a}{n_s}\right).$$

Cuando n_s^a o n_s valen 0, se considera por convención que el resultado del logaritmo es 0.

Finalmente fijamos una función de poda en nuestro estimador, denominada f, y para cada nodo s definimos la siguiente función:

$$C(s) = \begin{cases} K(s) + f(n), si\ s\ es\ una\ hoja, \\ mín\left\{K(s) + f(n), \sum_{b \in A} C(bs)\right\}, si\ s\ es\ un\ nodo\ interno. \end{cases}$$

Se calcula $C(s)$ para todo nodo, haciendo una recorrida bottom-up del árbol, y si para un nodo interno s el mínimo de C se alcanza para el término $K(s) + f(n)$, entonces truncamos el árbol en esa posición, convirtiendo por consiguiente el nodo s en una hoja.

La distribución condicional asociada a cada estado s queda definida como

$$P(a|s) = \frac{n_s^a}{n_s}, a \in A.$$

El modelo árbol tiene por consiguiente dos parámetros de ajuste:

1. $k =$ altura máxima del árbol
2. f = función de poda

Donde k define el máximo orden posible del proceso de Markov definido implícitamente por el modelo árbol, y por consiguiente cuando mayor sea su valor, mayor orden podremos capturar para la secuencia con la que construimos el árbol. Por su parte f define la agresividad de la poda, es decir, qué tan a menudo se toma la decisión de cortar un subárbol de un nodo interno. La función f define el grado de libertad que tiene nuestro estimador de generar tiras potencialmente distintas a las que aparecen en los registros de usuario, definiendo un equilibrio entre preservar las estadísticas y aumentar el grado de libertad existente para generar secuencias nuevas.

En nuestro caso, utilizamos una opción que resulta balanceada en su agresividad de poda, que es la función log_2.

Con este procedimiento, a partir de un registro de usuario procesado, podemos sortear una tira de sesiones de largo arbitrario, cuyo comportamiento será similar al de la tira de entrada, y por consiguiente representativo del usuario en cuestión. Esto producirá además escenarios potencialmente distintos a los reales, y por ende generará

secuencias que pueden no haber surgido explícitamente en la aplicación real, dando un valor agregado al modelo de estimación para nuestra realidad.

El mecanismo de estimar el proceso de Markov, en este caso el modelo árbol, y sortear luego tiras de largo arbitrario basado en dicho modelo, puede interpretarse como una simplificación de la técnica de simulación presentada en [9].

En [9] se prueba que las tiras simuladas utilizando este mecanismo cumplen que la diferencia entre la frecuencia de ocurrencias de cualquier patrón en la tira original y cualquier otra tira simulada tiende a 0 con el largo de la tira, es decir que en general cualquier tira producida tendrá una frecuencia de aparición de patrones que tiende a ser la misma que la tira observada.

3.2.2 Implementación para los tiempos de espera

Como vimos anteriormente, los tiempos de espera (*think times* en inglés) son un dato estadísticamente importante para recrear el comportamiento de los usuarios, dado que cada usuario presenta diversas características distintivas en el uso de la aplicación, que pueden ser debidas a su propio nivel de experiencia, o a otras variables ajenas a sí mismo como pueden ser hardware o software que usa, velocidad de conexión, etc.

Debido a la gran diversidad de opciones posibles, modelar los tiempos de espera es una tarea que puede llegar a ser muy compleja.

En nuestro caso modelamos los tiempos de espera procesando las ocurrencias reales de los tiempos agrupados por acciones del RUM y calculamos dichos tiempos a partir de los datos en los registros del uso real. Esto nos provee información con la que podemos generar un modelo estadístico con el que luego sorteamos a demanda los tiempos de espera para una determinada acción dentro de una sesión.

Para realizar el sorteo de los tiempos de espera se aplica la técnica de análisis de histogramas, utilizando las siguientes fórmulas definidas en [10] para calcular la cantidad de contenedores c, y el ancho de los contenedores h.

$$h = \frac{3.5\delta}{m^{1/3}}$$

$$c = \left\lceil \frac{\max t - \min t}{h} \right\rceil$$

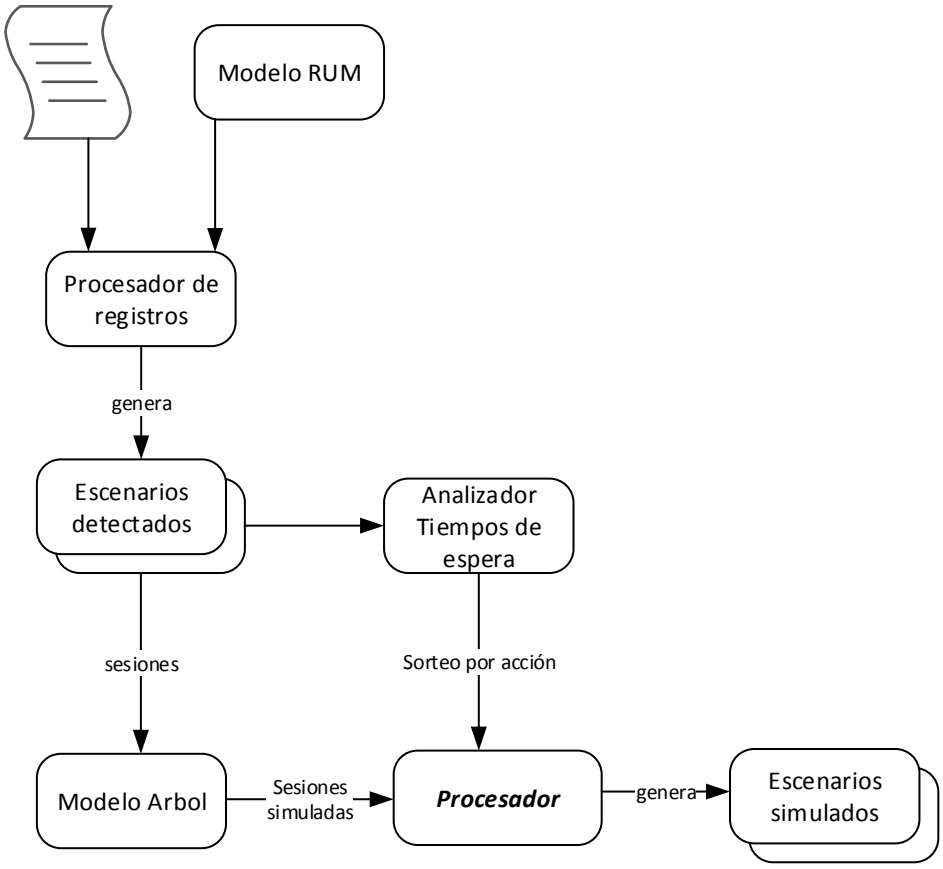
Donde $\max t$ y $\min t$ son respectivamente los tiempos máximos y mínimos obtenidos de la muestra de los tiempos de espera existentes en el registro de uso, δ es la desviación estándar de dicha muestra, y m es la cantidad de datos en la muestra.

La elección de h se denomina la *Regla de referencia normal de Scott (Scott's normal reference rule en inglés)* [10] y permite estimar de una manera sencilla el ancho de los contenedores a partir de una muestra.

3.3 Resumen del proceso completo

En base a lo explicado en el presente capítulo, presentamos en la **Figura 3-2** el proceso completo a través del cual generaremos los escenarios simulados que serán ejecutados por los usuarios virtuales.

Figura 3-2: Flujo completo para simular escenarios



El sistema requiere como entrada los registros de aplicación y el RUM completo para la aplicación web objetivo, y los procesa para detectar escenarios ejecutados por los usuarios reales y sus respectivas sesiones. En base a los datos de tiempo obtenidos del registro de uso, se calculan los histogramas para los tiempos de espera, agrupados por las acciones intervinientes en cada sesión, y se construye el modelo árbol a partir de las sesiones detectadas para simular así nuevas secuencias de sesiones de largo arbitrario. El procesador es el que integra las salidas del modelo árbol y los valores sorteados de los tiempos de espera, y genera escenarios concretos listos para producir usuarios

virtuales que serán ejecutados sobre el SUT. Tenemos entonces un modelo completo para el problema y cada uno de sus componentes.

El procedimiento completo para obtener los usuarios virtuales es el siguiente.

1. Procesar los registros de aplicación para obtener la secuencia de sesiones del RUM realizadas en la aplicación
2. Nombrar cada sesión detectada con un símbolo único (ej: $\{s_1, s_2 \dots s_n\}$) y generar una tira donde los símbolos serán los identificadores de la sesión, en el orden de aparición real (ej: $\{s_1, s_5, s_2, s_5 \dots s_n\}$)
3. Crear el árbol de estimación a partir de la tira generada en 2
4. Con el árbol de estimación, generar N tiras cuyo largo se encuentre en el entorno $[a, b]$, donde N es el número de usuarios virtuales a generar, a y b son parámetros dados.
5. A partir de cada tira generada, identificar la secuencia de sesiones que representan
6. Sortear un tiempo de espera para cada *acción* que forma parte de las sesiones
7. Asignar un escenario para cada usuario virtual

4 Aplicación

De acuerdo al modelo presentado en el capítulo anterior, desarrollamos una aplicación que permite poner en práctica el proceso completo. Se trata de una herramienta para generar y ejecutar escenarios de usuarios virtuales para aplicaciones web a partir de los registros de sistema registrados por el servidor de aplicaciones, los cuales cumplen la premisa de ser representativos del uso real en producción del SUT. El sistema cuenta con asistencia por parte de los desarrolladores, los cuales brindan información de los escenarios definidos a nivel de casos de uso, mapeados a interacciones web. Esta tarea se realiza a través de un esquema diseñado específicamente para el sistema el cual está fuertemente basado en RUM.

También cuenta con un filtro que corre en el servidor web del SUT y permite capturar información complementaria a la contenida en los registros del sistema. Información tal como parámetros enviados en forma de cadenas de consulta (query strings) y datos incluidos en la carga útil (payload) de los pedidos HTTP, así como también las cookies que viajan del servidor al cliente ida y vuelta. Dicha información se vincula al usuario que corresponda a través del ID de sesión que le haya sido otorgada al autenticarse en el sistema y la URL correspondiente al pedido HTTP realizado.

A máximo nivel de abstracción, consideramos el sistema como una “caja” que toma por entrada los registros del servidor de aplicaciones en formato estándar, los datos complementarios provenientes del filtro y el modelo del SUT, y devuelve escenarios de simulación que constan de usuarios virtuales (*vusers*) que son generados en base a un modelo probabilista que cumple determinadas propiedades deseables de los escenarios volcados en los registros. Estos usuarios virtuales son en consecuencia representativos del uso real de la aplicación en producción, y pueden entonces estresar la aplicación de una manera comparable a la que lo harían usuarios reales.

Los escenarios de salida cuentan con toda la secuencia de pedidos HTTP a realizar junto con el tiempo de espera (*think time*) de cada uno, y se cuenta con un proceso que permite obtener los parámetros (query string, payload) a enviar en cada pedido que provienen de los datos almacenados previamente por el filtro.
A partir de lo anterior el módulo encargado de la simulación ya cuenta con todo lo necesario para ejecutar los usuarios virtuales.

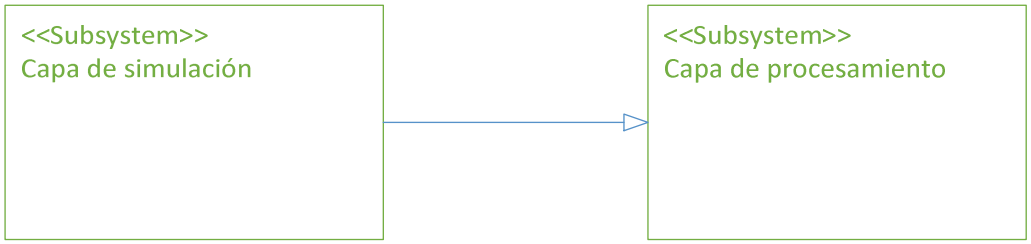
Como resultado de la ejecución se obtienen una serie de datos que permiten al sistema generar reportes para poder analizar los resultados. Estos datos permiten vincular los resultados de los distintos pedidos HTTP ejecutados por el módulo de simulación con el resto de los componentes del sistema y así obtener contexto sobre cada pedido simulado.

Para interactuar con esta "caja" presentamos una Aplicación Web que permite manipular todos los aspectos mencionados hasta ahora sobre el sistema. Dicha aplicación permite ingresar registros (subir un archivo que contenga los registros del servidor de aplicación web), ingresar el modelo del SUT (alta, baja y modificación de datos), configuración de parámetros que permitan ajustar el modelo matemático utilizado para la generación de escenarios, visualizar datos y modelos generados y la ejecución de las simulaciones.

4.1 Arquitectura

El sistema no sigue un esquema arquitectónico predefinido (como ser arquitectura en capas, cliente-servidor, etc.), sin embargo se estructura en dos subsistemas bien diferenciables (**Figura 4-1**).

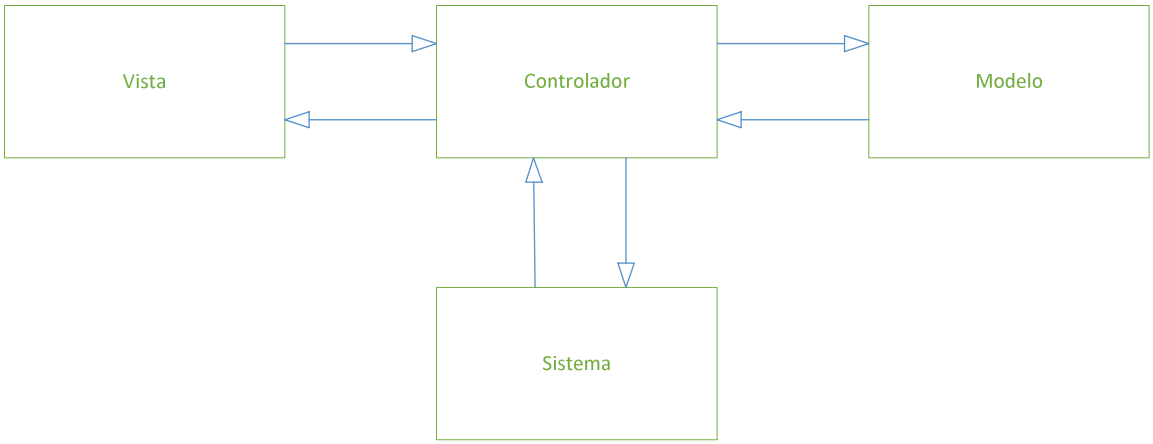
Figura 4-1: Subsistemas



La capa de simulación es la encargada de efectuar la simulación basada en los requisitos definidos por las pruebas de carga. Éste subsistema se comunica con la capa de procesamiento que es la encargada de generar los escenarios de simulación y usuarios virtuales requeridos, basados en los logs de sistema y los datos específicos del SUT.

La aplicación web utilizada para interactuar con el sistema sigue un patrón de arquitectura MVC. En la **Figura 4-2** vemos como se integra dicha aplicación con el sistema.

Figura 4-2: Aplicación Web.



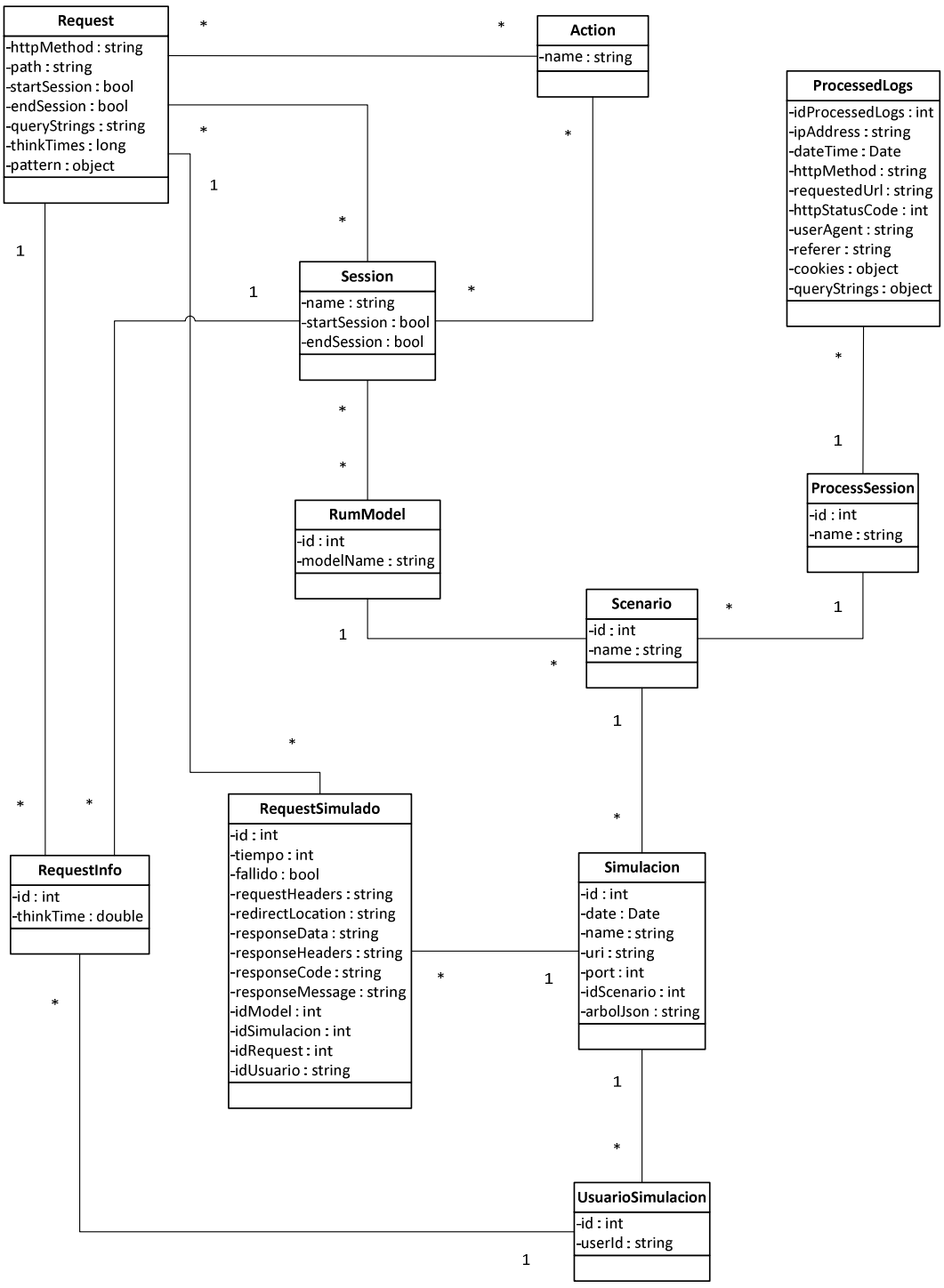
4.2 Módulos

En esta sección exponemos la estructura del sistema, el diagrama de clases y los dos subsistemas en los que se divide la arquitectura, junto con sus componentes y los flujos de ejecución que permiten obtener los resultados.

4.2.1 Diagrama de clases

Presentamos a continuación el diagrama de clases (**Figura 4-3**) y una descripción del mismo.

Figura 4-3: Diagrama de clases



Comenzamos por las clases que representan el modelado del RUM y el procesamiento de registros de aplicación (las clases que forman parte del subsistema “Capa de procesamiento”).

Las clases *Request, Action, Session, Scenario y RumModel* son las utilizadas para representar un modelo RUM a nivel de objetos:

Request: esta clase representa un pedido HTTP. Incluye: el método (POST, GET, PUT, etc.), el path (ej: /instructors/edit/45), startSession y endSession que indican si el *Request* es comienzo o fin de una sesión, queryStrings que son los parámetros enviados en la URL (ej: el parámetro “q” en /search?**q**=valorparametro), thinkTimes que es la lista de tiempos de espera según ocurrencias en los logs y pattern que es una clase del módulo *java.util.regex*.
Una instancia de *Request* pertenece a una sesión y en algunos casos a varias, esto último se debe a que de acuerdo al modelo SLM pueden existir algunos pasos en común entre determinadas sesiones.

Action: esta clase representa las acciones del RUM a nivel del sistema. Contiene un nombre y la lista de *Request* que la integran.

Session: esta clase representa las sesiones del RUM a nivel del sistema. Se le asigna un nombre, contiene la lista de *Action* que la conforma y una referencia a la lista de todos los *Request*.

RumModel: esta clase representa al modelo RUM en sí mismo, lo que permite al sistema soportar múltiples modelos. Contiene un nombre y una lista de *Session* y de *Scenario*.

Antes de presentar la clase *Scenario,* describimos a continuación las clases que permiten el manejo de los registros del log del SUT. Estas clases son:

ProcessedLogs: esta clase representa un registro del archivo de logs. Contiene todos los valores contenidos en una línea de un archivo de logs en formato estándar W3C, que son: ipAddress (dirección ip donde se originó el pedido), dateTime (timestamp del pedido), httpMethod (método HTTP del pedido), requestedUrl (URL del pedido), httpStatusCode (status retornado, ej: 200 OK, 404 Not found, etc.), userAgent (agente

utilizado), referer (URI que identifica la dirección), cookies, queryStrings (parámetros pasados en la URL).

ProcessSession: esta clase representa a un conjunto de registros de logs, cada vez que se sube un archivo de logs, este se procesa generando una lista de *ProcessedLogs* y un *ProcessSession* que la representa. Esto permite al sistema manejar múltiples archivos de logs. Contiene un nombre y la lista de ProcessedLogs que la integra.

Scenario: esta clase representa a los escenarios del RUM a nivel del sistema. Pertenece a un *RumModel* y contiene un *ProcessSession*.
Se le asigna un nombre y a partir de esta clase es que se generan las simulaciones, partiendo del modelo RUM al que pertenece y de los registros de logs que le fueron asignados (los *ProccesedLogs* del *ProcessSession*).

Las siguientes son las clases que forman parte del subsistema “Capa de simulación”:

RequestInfo: esta clase es utilizada para manipular los objetos *Request* con el thinktime asignado. Como el thinktime es un parámetro que se genera a medida que el Request es simulado y un mismo *Request* puede existir en distintas simulaciones con distinto thinktime, se crea esta clase que representa a la instancia de un *Request* en una simulación con el thinktime que le fue asignado a partir del histograma generado con su lista de thinktimes.
Por lo tanto contiene un *Request* y el thinktime correspondiente. También guarda una referencia a la *Session* a la cual corresponde la ejecución del *Request* y pertenece a un *UsuarioSimulacion*.

Simulacion: esta clase representa la simulación en sí misma, guarda una referencia al escenario a simular, el árbol generado por el simulador estocástico a través del atributo arbolJSON y atributos necesarios para la ejecución de pedidos como uri y puerto del SUT. También contiene una referencia a la lista de usuarios a simular, que son objetos de la clase *UsuarioSimulacion*.

UsuarioSimulación: esta clase representa a un *vuser (usuario virtual)* en el sistema. Pertenece a una simulación y una referencia a la lista *RequestInfo* que son los *Request* con su thinktime a ser ejecutados.

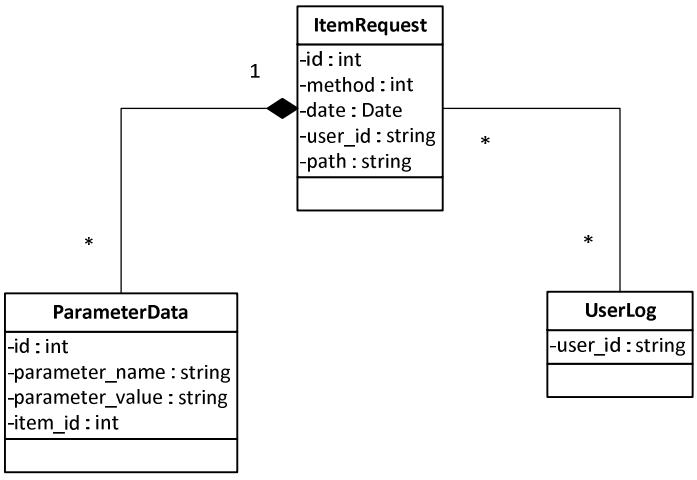
RequestSimulado: esta clase contiene todos los datos resultados de la ejecución de un Request. Estos datos son: tiempo (duración del pedido), fallido (flag que indica si falló o no), requestHeaders (cabeceras del pedido), redirectLocation (URL de redirección, si existe), responseHeaders (cabeceras de la respuesta), responseCode (código de la respuesta, ej: 200, 500, etc.), responseMessage (mensaje de la respuesta, ej: OK, Internal Server Error, etc.).

También guarda una referencia al objeto *Simulacion* desde el cual se ejecutó el *Request* y una referencia al *Request* ejecutado. Esto permite darle el contexto mencionado a los resultados. Teniendo estas referencias y dada la navegabilidad que existe entre los objetos es posible llegar al modelo RUM (sesiones, acciones y pedidos) y a los logs que se utilizaron.

4.2.1.1 Parámetros

A continuación mostramos el diagrama de clases utilizado por el sistema para gestionar a nivel de objetos los datos recabados de las transacciones web efectuadas por los usuarios en la aplicación (**Figura 4-4**).
Estos son los datos almacenados por el filtro mencionado al comienzo de este capítulo y son los que nos permiten rellenar los valores (query strings y payloads) para poder completar los pedidos.

Figura 4-4: Diagrama de clases de los parámetros.



ParameterData: esta clase representa a los parámetros que viajan en cada pedido. Guarda la tupla nombre – valor de cada parámetro representada a través de parameter_name y parameter_value respectivamente y también una referencia al ItemRequest al cual pertenece.

ItemRequest: esta clase representa un pedido que fue realizado por un usuario utilizando el SUT.
Los datos que contiene son: method (método del pedido), date (la fecha en que se realizó), user_id (el id de sesión que le fue asignado al usuario que realizó el pedido) y path (ruta completa del pedido). También guarda una referencia a la lista de ParameterData.

UserLog: esta clase representa al usuario que realizó el pedido, guardando su id de sesión y una referencia a la lista de ItemRequest (pedidos) realizados.
El motivo de crear esta clase es facilitar al módulo encargado del manejo de los datos de parámetros realizar una consulta para chequeo de restricciones que se detalla más adelante en este apartado.

4.2.2 Subsistemas

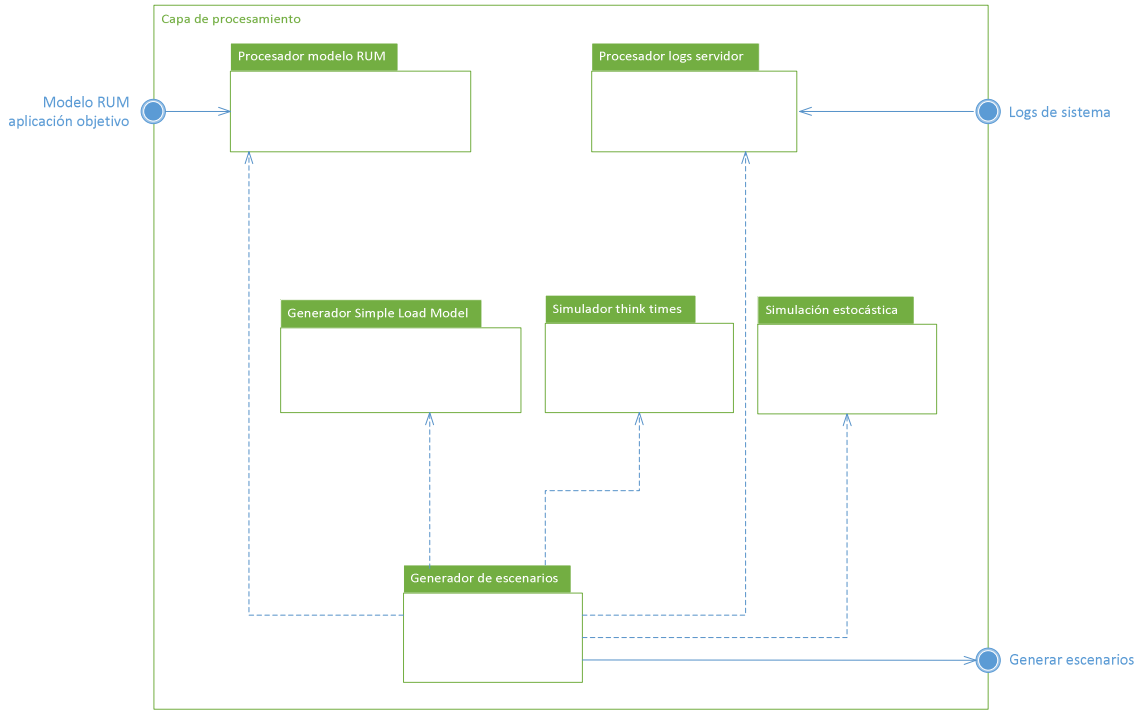
Para los subsistemas definidos en la sección anterior, presentamos a continuación los distintos módulos que forman parte de cada uno.

4.2.2.1 Capa de procesamiento

La capa de procesamiento (**Figura 4-5**) es la encargada de, basándose en los datos de entrada (logs y modelo RUM), generar los escenarios de simulación que posteriormente son utilizados por parte de la capa de simulación.

La parte fundamental del sistema se encuentra implementada en esta capa, por lo que fue utilizada la mayor granularidad posible para definir los módulos con el objetivo de asignar responsabilidades con la mayor cohesión posible y bajar al máximo el acoplamiento entre componentes.

Figura 4-5: Capa de procesamiento.



Descripción.

Para la capa de procesamiento existe un módulo denominado *generador de escenarios,* que se encarga de orquestar los servicios que brindan el resto de los módulos para generar la salida.
Los módulos de entrada son el *procesador de logs* y el *procesador de RUM*. Como se ve en el diagrama estos módulos por separado obtienen los datos de entrada, el archivo de logs y el RUM a modelar respectivamente.

Procesador de logs: modela los datos provenientes de los logs volcados por el servidor del SUT, y brinda los métodos necesarios para manipular y consultar dichos datos.
Se trata básicamente de un mapeo de los registros del archivo de logs a un modelo orientado a objetos. Donde cada registro pertenece a una clase definida (*ProcessedLogs*).
Los logs son ingresados como un archivo en el formato estándar W3C, a partir del cual se corre un proceso que recorre todas las líneas del archivo generando para cada una un

objeto *ProcessedLogs* a nivel del sistema que permite encapsular los datos de cada registro.

Procesador de RUM: obtiene el modelo del SUT y genera las estructuras de consulta y evaluación del mismo para, de manera similar al procesador de logs, obtener un acceso ordenado a los datos de entrada proporcionados.
En este módulo se construye todo el modelo RUM mapeado a clases que definen específicamente cada elemento de un modelo RUM (*Request, Action, Session, RumModel*).
El proceso para ingresar el modelo RUM en el sistema es el siguiente:

- Se crea un nuevo modelo RUM al que se le asigna un nombre (ejemplo: “Contoso University RUM”).
- A partir de dicho modelo se pasan a ingresar los casos de uso que serán las sesiones a nivel de sistema (ejemplo: “Editar estudiante”).
- Para cada sesión se ingresan las acciones que la componen (ejemplo: para “Editar estudiante” podrían ser: “Obtener listado de estudiantes” y “Modificar estudiante”).
- Para cada acción ingresada se asigna la lista de pedidos que se deben realizar. (ejemplo: para “Obtener listado de estudiantes” primero se debe hacer “GET/index” y luego “GET/students” y para “Modificar estudiante”: “GET/student/42/edit” y “POST/student/42/edit”).

Una vez culminado este proceso el sistema ya cuenta con el modelo RUM listo para operar.

Generador Simple Load Model: actúa como factor de cohesión entre el procesador de logs y el procesador de RUM, mapeando los datos obtenidos del log de servidor a instancias de sesiones modeladas en el RUM.
Recibe como entrada los logs procesados y el modelo RUM y brinda dos funcionalidades: la primera es la de ordenar los registros del log por usuarios, y la segunda es matchear las secuencias de los registros del log ordenados con secuencias equivalentes en el modelo RUM y así obtener todas las sesiones.
Continuando el ejemplo presentado más arriba, si al recorrer los logs de un usuario se encuentra la siguiente secuencia:

> … GET/index – GET/students – GET/student/42/edit – POST/student/42/edit …
> Esto matcheará con la sesión “Editar estudiante”.
> Este proceso se continúa hasta obtener todas las sesiones del escenario.
> La lista de sesiones tendrá la forma: Login - … - Editar estudiante - … - Logout.

Simulación estocástica: este módulo tiene la responsabilidad de implementar el modelo estocástico que es disparado para generar un escenario simulado potencialmente distinto al obtenido de los logs como se ha descrito con detalle en el capítulo anterior.
Básicamente recibe una tira de caracteres, cada uno representando una sesión, y a través de un proceso matemático genera las nuevas tiras.

Simulación de tiempos de espera: aquí se implementa la función de asignar un tiempo de espera a cada pedido a realizar en base a la construcción de histogramas como ya se ha mencionado.
Esto lo hace en dos pasos: primero, a partir de los logs ordenados por usuarios y de la lista de pedidos del RUM, le asigna a cada pedido una lista de tiempos de espera según la diferencia de timestamps entre que se realizó el pedido y el pedido siguiente.
Segundo, dado un pedido y su lista de tiempos de espera, se construye un histograma para obtener la distribución de los valores de los tiempos de espera a partir del cual se genera el valor a retornar.

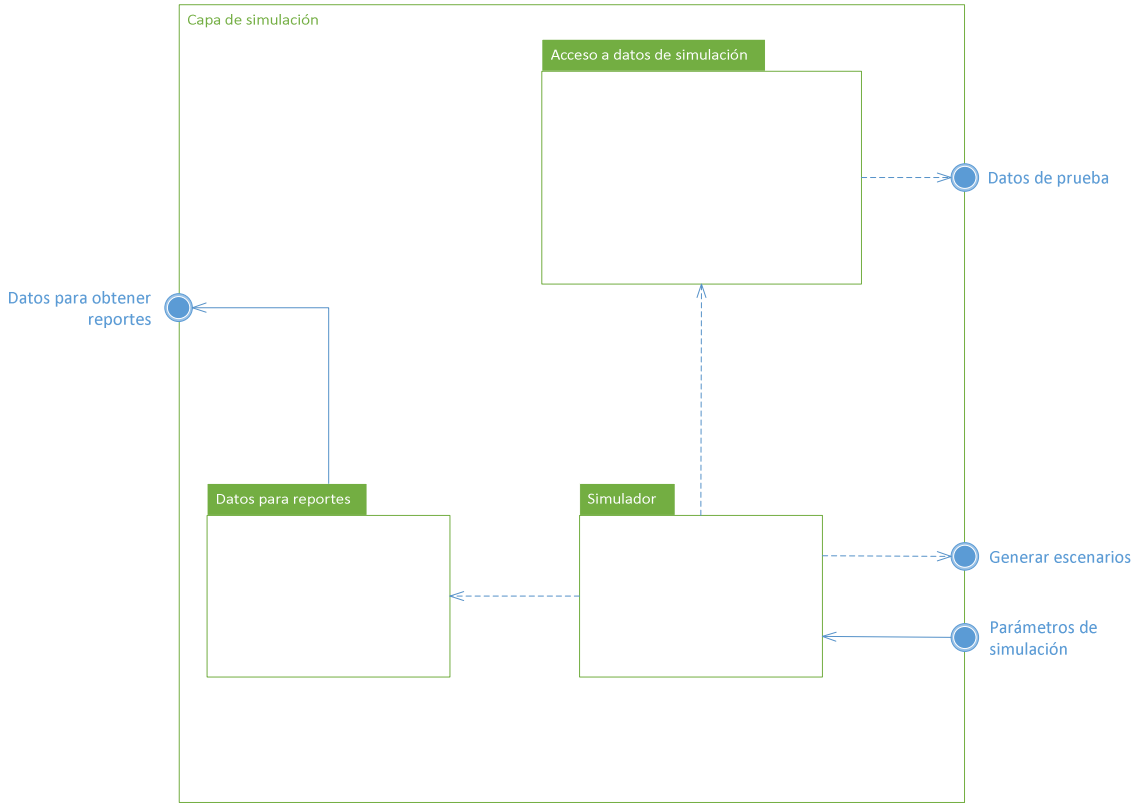
Generador de escenarios: Finalmente y como se habló al comienzo de esta sección, el *generador de escenarios* se encarga de secuenciar las funcionalidades expuestas por el resto de los módulos y provee la lógica adicional para generar la salida, que consta de los *vusers* prontos para ser simulados.
Para la generación de un *vuser*, se efectúa el siguiente flujo: primero se comunica con los dos módulos de entrada para obtener los logs procesados y el modelo RUM que se desea poner a prueba. Utilizando estos datos se procede a llamar al módulo de *generación del simple load model* para generar las secuencias de sesiones que se dieron en los logs ordenados por usuario, de las se elige una y es enviada al módulo estocástico para simular una nueva secuencia. A cada pedido de esta nueva secuencia se le asigna un think time por medio de un llamado al módulo de tiempos de espera. En este punto ya tenemos el escenario simulado, solo resta asignar un id de usuario y obtener así el *vuser* pronto para ejecutar.

Este módulo expone una función, descrita en el esquema como *Generar escenarios* (**Figura 4-5**), que dado una cantidad de usuarios a generar, una mínima y máxima cantidad de sesiones retorna los *vusers* listos para ser procesados por la capa de simulación.

4.2.2.2 Capa de simulación

En esta sección describimos como está compuesta la capa de simulación y los distintos módulos y flujos involucrados en la ejecución de los *vusers*.

Figura 4-6: Capa de simulación.



Descripción.

En el esquema de la capa de simulación (**Figura 4-6**) se muestra cómo está estructurada esta capa, la cual se compone de tres módulos independientes:

Simulador: Es el módulo encargado de ejecutar los *vusers*.
Recibe como entrada los parámetros: URI y puerto a enviar los pedidos, cantidad de usuarios, cantidad mínima y máxima de sesiones en cada escenario. Estos últimos tres parámetros los utilizan para invocar la función de *generar escenarios* brindada por la capa de procesamiento que devuelve la cantidad de escenarios deseada con las características solicitadas.
Se comunica con el módulo de *acceso a datos de simulación* para obtener los parámetros que serán enviados junto con los pedidos y con el módulo de *datos para reportes* para almacenar los datos recabados durante la ejecución de las pruebas.
Para la ejecución de los *vusers* se utiliza JMeter que permite armar los pedidos y retornar datos relevantes a la simulación.

Acceso a datos de simulación: Brinda acceso ordenado a datos para enviar como parámetros en los pedidos ejecutados y se utiliza para chequeo de restricciones.

Datos para reportes: Este módulo se encarga de almacenar y retornar todos los datos recabados durante la ejecución de las pruebas.

A continuación se describe detalladamente el flujo que se realiza para la ejecución de un lote de *vusers*:

- Se obtienen los *vusers* utilizando los parámetros recibidos como entrada (cantidad de usuarios, cantidad máxima y mínima de sesiones) e invocando la función expuesta por el módulo *Generar escenarios* de la capa de procesamiento.
- Por cada *vuser* generado se procede de la siguiente manera:
    - Se crea un hilo de ejecución donde se realizará la simulación de todo el escenario.
    - Se realiza un chequeo de restricciones que permita asegurar la ejecución completa del escenario y se obtiene el id del usuario a utilizar para obtener los parámetros que se usarán en los pedidos.
      Este chequeo se ejecuta por medio de un llamado al módulo de *Acceso a datos de simulación* que toma como entrada la lista de todos los pedidos a realizar y hace un consulta sobre los *datos de prueba* para asegurar la existencia de un usuario que haya ejecutado al menos una vez cada uno

de ellos.
En la sección 4.4 se explica con mayor detalle la implementación de esta consulta.
Los *datos de prueba* son representados a nivel de sistema por las clases *ItemRequest*, *ParameterData* y *UserLog* que fueron presentadas en la subsección anterior.
Entre los usuarios que cumplan esta restricción se sortea uno aleatoriamente.

- El último de los pasos es recorrer la lista de pedidos y ejecutarlos utilizando la herramienta JMeter, en la sección 4.5 presentaremos en detalle cómo se realiza dicha ejecución.
  Finalmente se realiza un system call que pasa el hilo a un estado inactivo (*sleep*) durante el período establecido por el think time que se encuentra en el pedido. *JMeter* brinda un mecanismo para esto, pero se optó por hacerlo de esta manera para tener el control completo sobre la ejecución.
  Este último paso, como ya se ha mencionado, nos da el comportamiento de una ejecución real, en la que un usuario luego de hacer un pedido (por ejemplo obtener un formulario) demorará algún rato en rellenarlo antes de realizar el siguiente pedido (postear el formulario en este caso).

Antes de terminar con la capa de simulación resta presentar un último aspecto, que explica qué es lo que se hace con la toda la información recabada.

En el sistema, a nivel de base de datos, fueron creados dos procedimientos almacenados para hacer una presentación de los datos que se guardaron durante la ejecución de los *vusers*. Se decidió proceder de esta forma para aprovechar el enlace de la ejecución de *JMeter* con el resto de los modelos del sistema y retornar datos uniformes.

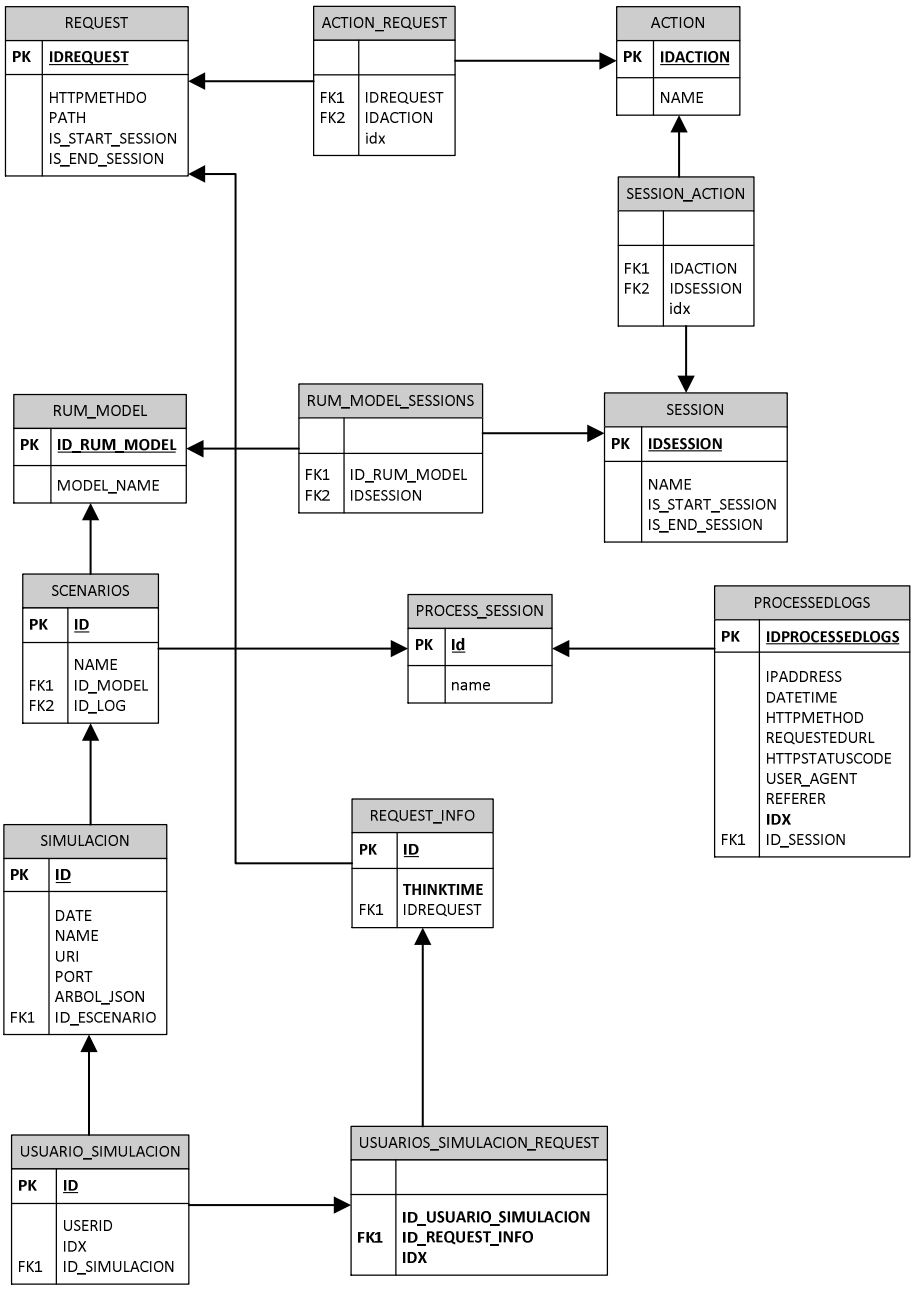
Uno de ellos, llamado *Resumen de requests*, retorna la estadística de cada pedido perteneciente al modelo RUM (por ejemplo /Student/Edit/[0-9]*), los datos presentados son: cuantas de las ejecuciones fueron fallidas, la tasa de éxito, el tiempo promedio de ejecución, el tiempo mínimo y el tiempo máximo (estos últimos dos para visualizar los picos).

El otro, llamado *Resumen de simulación*, presenta la estadística de cada simulación ejecutada por el sistema, los datos retornados son: total de usuarios simulados, total de pedidos fallidos, tasa de éxito, tiempo promedio, tiempo mínimo, tiempo máximo y tiempo total de los pedidos.

4.3 Modelo de datos

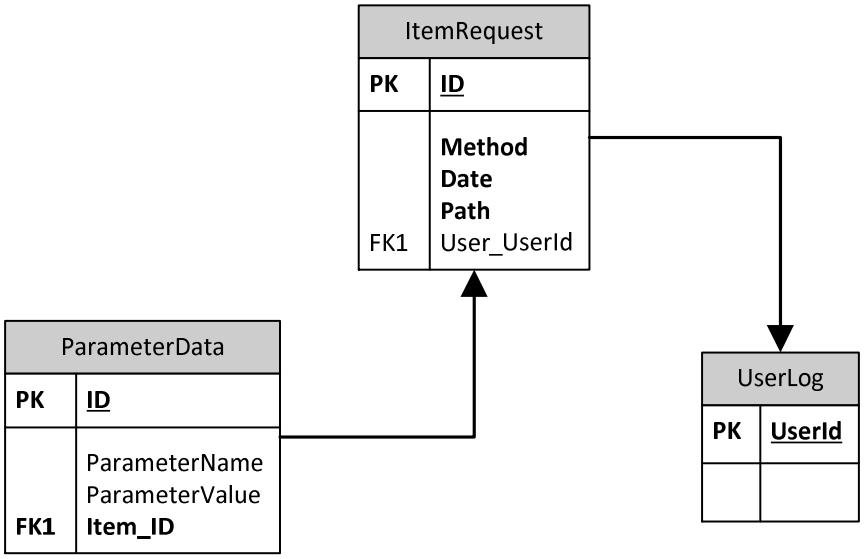
En esta sección presentamos el modelo de datos utilizado, por medio del esquema entidad relación de la base de datos que nos muestra las entidades relevantes del sistema así como sus interrelaciones y propiedades (**Figura 4-7**).

Figura 4-7: Diagrama ER



En la **Figura 4-8** vemos el diagrama ER que esquematiza como está implementado el modelo de datos de los parámetros a nivel de base de datos.
Estas tablas son las utilizadas por el filtro que corre en el servidor web del SUT para almacenar los datos que viajan entre el cliente y el servidor.
Y posteriormente son consultados a nivel del sistema por el módulo de *Acceso a datos de simulación*.

Figura 4-8: Diagrama ER de la estructura utilizada para almacenar parámetros.



4.4 Otros aspectos de implementación

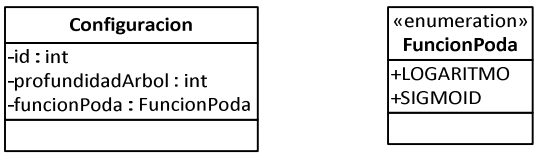
A continuación pasamos a detallar aspectos de implementación que son interesantes para conocer en profundidad el funcionamiento del sistema.

Configuración

Cuando hablamos sobre el módulo de *simulación estocástica* faltó nombrar un aspecto de uso que es el de la configuración de los parámetros que utiliza el modelo matemático.

Esto se logra a nivel de sistema con la clase mostrada en la **Figura 4-9**.

Figura 4-9: Configuración.



Configuracion: esta clase se usa para manipular los valores que se utilizarán para generar el árbol del proceso estocástico (**Figura 4-9**).
Se compone de: id (que la identifica), profundidadArbol (la profundidad del árbol a construir) y funcionPoda (indica la función matemática que se utilizará para podar el árbol).

A nivel de base de datos se representa con la siguiente tabla (**Figura 4-10**):

Figura 4-10: Tabla de configuración.

CONFIGURACION	
PK	**ID**
	PROFUNDIDAD_ARBOL **FUNCION_PODA**

Chequeo de restricciones

En la sección 4.2.2.2 mientras explicamos el flujo utilizado por la capa de simulación para la ejecución de *vusers*, hablamos de la consulta de chequeo de restricciones realizada por el módulo de *Acceso a datos de simulación*.

Esta consulta se realiza a nivel de base de datos consultando en las tablas: *ItemRequest* y *UserLog*.
El objetivo de la consulta es, dada una lista de pedidos HTTP, buscar entre los registros de la tabla *UserLog* aquellos usuarios que hayan ejecutado al menos una vez cada uno de estos pedidos y entre los que cumplan esta condición se sortea aleatoriamente uno (utilizando la clase *java.util.Random*).
Del usuario retornado se toma su id (UserId) que es usada para consultar los parámetros

que son enviados en los pedidos HTTP ejecutados.
Por lo que utilizando dicha id se obtienen las credenciales de *Login* del usuario con el que se realiza la ejecución de todo el escenario hasta ejecutar el *Logout,* como si este estuviera realmente interactuando con el sistema objetivo.
Esto añade consistencia de los datos enviados en los pedidos, ya que al menos nos asegura que para cada pedido a realizar se van a poder obtener parámetros para enviar y además dichos parámetros se van a corresponder con datos que fueron manipulados en el historial de las interacciones del usuario con el SUT.

A nivel de sistema las tablas consultadas son mapeadas a las clases *ItemRequest* y *UserLog* utilizando *Hibernate*.
Y la implementación de la consulta es realizada por medio de las *Criteria* de *Hibernate* que permiten armar consultas complejas.
A nivel de SQL la forma de esta consulta es:

```
SELECT *
FROM UserLog as usr
WHERE
    requests.each |req| // para cada request del escenario
    EXISTS  (SELECT  * FROM ItemRequest as ir
              WHERE ir.request = req AND ir.user = usr)
END
```

Donde “ir.request = req” representa que el método y path de “ir.request” coincidan con los de “req”.

Se busca dentro de los registros de UserLog aquellos que tengan una entrada en la tabla ItemRequest para cada pedido del escenario.

4.5 Uso de JMeter

Para la ejecución de los *vusers* decidimos utilizar el framework JMeter que brinda las características deseadas para este tipo de pruebas.

La decisión se basó en 2 aspectos:

- Simplicidad de uso de la API, es sencillo generar programáticamente la ejecución de los pedidos HTTP junto con los datos (parámetros y cookies) asociados a los mismos y obtener resultados estadísticos de las ejecuciones.

- Posibilidad de aprovechar otras funcionalidades, en la sección 6.2.3 exponemos una forma de explotar JMeter de una manera distinta a la implementación actual.

El flujo ejecutado para realizar los pedidos de los escenarios es el siguiente:

Al recorrer la lista de pedidos a ejecutar, se crea para cada uno, un objeto *HTTPSampler* que es la clase que JMeter utiliza para realizar pedidos HTTP. Este objeto es creado utilizando los datos obtenidos hasta este punto de ejecución que permiten construir la URL completa más el método para que el pedido pueda ser encausado.

Se ejecuta el pedido por medio de un llamado a la función *sample()* obteniendo un nuevo objeto de la clase *HTTPSamplerResult* que contiene todos los datos resultados, de importancia para nuestra prueba. Se llama al módulo de *datos para reportes* para que almacene de forma ordenada los datos, esto es, que también se guarden referencias a los distintos modelos involucrados en la ejecución (simulación, modelo RUM, id de usuario) para poder darle contexto a la información recabada.

Para comprobar efectivamente el comportamiento de los pedidos, utilizamos la herramienta *Wireshark*, con la que vemos los pedidos que salen y entran al sistema y todos los datos que viajan en tiempo real. Esto es de gran ayuda en el proceso de depuración.

5 Caso de prueba

En este capítulo presentaremos un caso de estudio en la utilización de la herramienta a una aplicación real, y describiremos los distintos aspectos involucrados, incluyendo los sub productos generados durante el proceso.

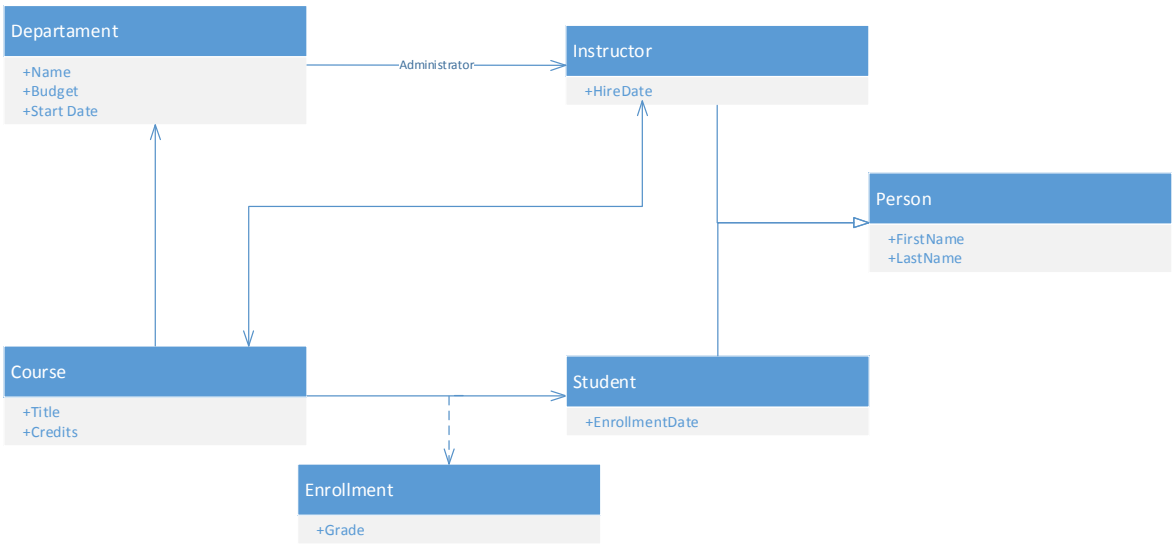
5.1 Contoso University

Contoso University [2] es una aplicación desarrollada en ASP.NET MVC de código libre, cuyo cometido es el de mostrar distintas tecnologías vinculadas con el framework de Microsoft.

Esta aplicación resulta de particular interés para nuestras pruebas dado que el modelo MVC presenta propiedades interesantes para la etapa de reconocimiento de las sesiones del modelo RUM, en particular la relación 1 a 1 de las acciones que forman parte de los casos de uso con los pedidos HTTP. Además, la aplicación es de fácil interpretación a partir de su uso, por lo que construir el RUM a partir de un análisis del sistema en funcionamiento no resulta un problema.

La aplicación se trata de un sitio web ficticio para el manejo de una Universidad. Incluye entre sus funcionalidades la de ABM de estudiantes, ABM de cursos, ABM de instructores y ABM de departamentos y sus respectivas relaciones como se ve en el modelo entidad relación de la **Figura 5-1**.

Figura 5-1: Modelo entidad relación de Contoso University



La aplicación se encuentra instalada sobre una instancia de Internet Information Services (IIS) en el sistema operativo Windows, y será precisamente el IIS el qué recolectará los registros de aplicación reales de la aplicación en formato estándar de World Wide Web Consortium (W3C).

5.1.1 Modelo RUM

A continuación en la **Figura 5-2** y **Figura 5-3** presentamos el modelo RUM de la aplicación con el que procesaremos los registros de aplicación, el mismo fue obtenido mediante el análisis del código fuente.

Figura 5-2: RUM para Estudiantes e Instructores

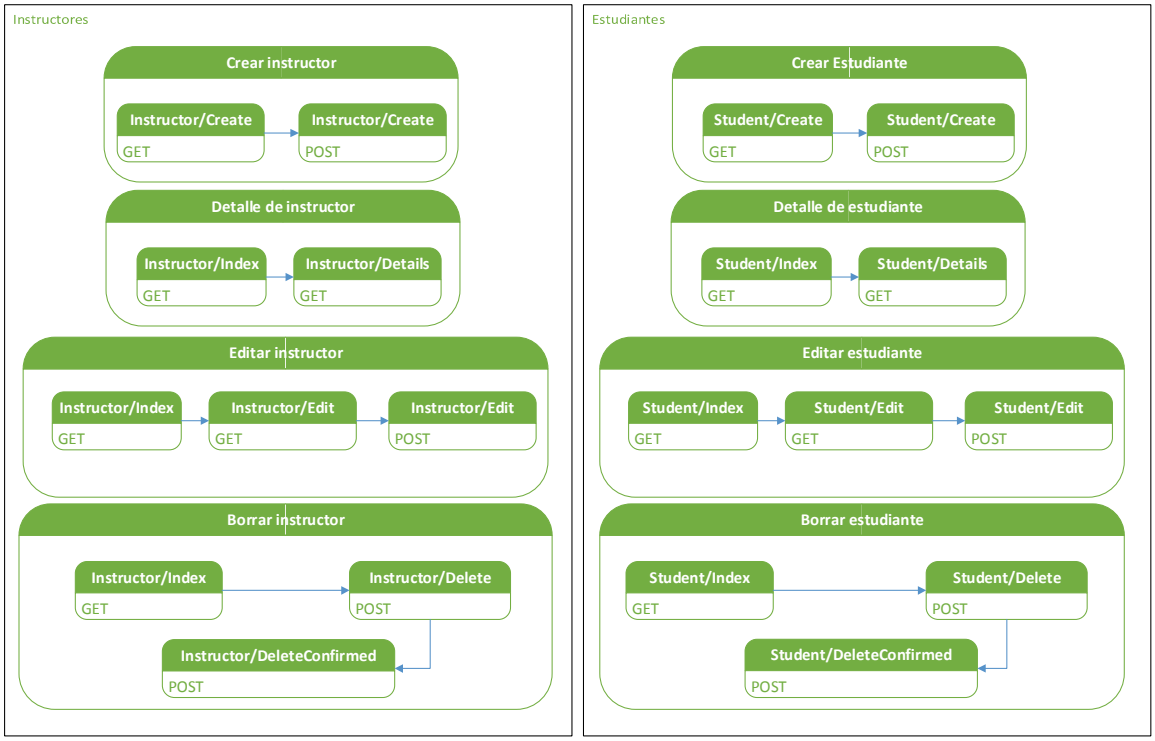
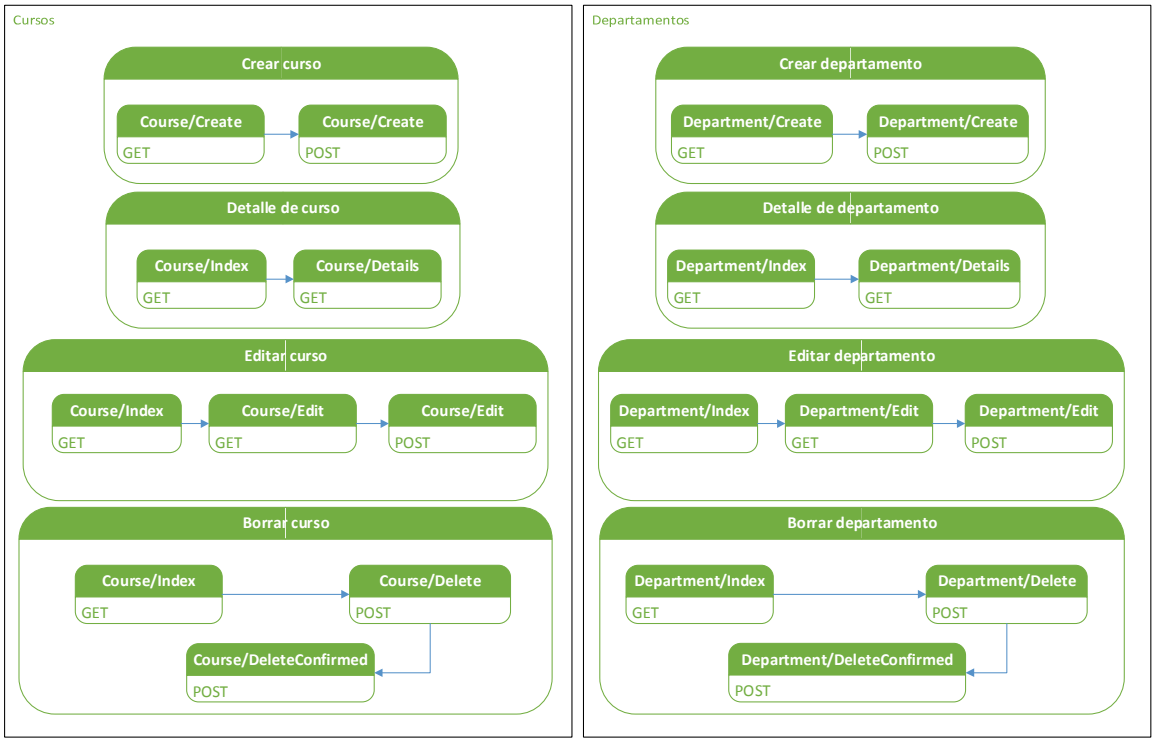


Figura 5-3: RUM para Departamentos y Cursos



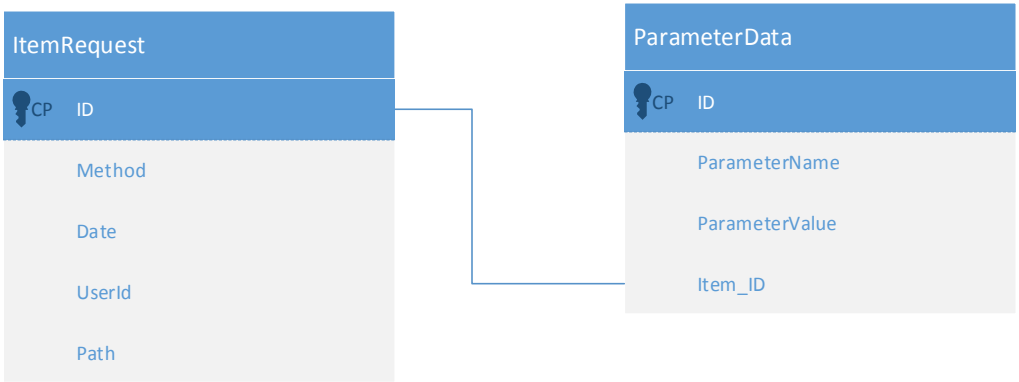
5.1.2 Filtro HTTP

Los escenarios generados por nuestra herramienta, si bien son autodescriptivos, necesitan tener la referencia de los datos que viajan en cada pedido HTTP para poder ejecutar la aplicación contra el SUT. Para ello se implementó un filtro HTTP de .NET dentro de la solución de Contoso University, para capturar todo el tráfico generado.

El filtro se encargará de volcar a una base de datos los pedidos web que realizan los usuarios reales, incluyendo los datos que se intercambian, en particular existe una cookie llamada *ASP.NET_SessionId* que contiene un id único de sesión generado por el framework .net, el cual nos servirá para identificar cuando los pedidos pertenecen a un mismo usuario, y agrupamos así los datos por usuario luego del procesamiento.

Con esta implementación demostramos una técnica para la recolección de datos que constará de 2 etapas, la primera de recolección de datos en una base de datos cuyo esquema se muestra en la **Figura 5-4**, y la segunda de procesamiento y simulación de las pruebas de rendimiento.

Figura 5-4: Modelo entidad relación para la recolección de datos del filtro HTTP



5.1.3 Ejemplo de uso

Generamos una carga real sobre la aplicación Contoso University de manera manual, y obtuvimos los registros de actividad desde el IIS (**Tabla 5-1**), los cuales fueron

procesados por la herramienta, resultando en el siguiente reconocimiento de patrones del RUM.

Tabla 5-1: Sesiones reconocidas por la herramienta en el log de ejemplo

Sesión
Página de estudiantes
Página de estudiantes
Página de estudiantes
Editar estudiante
Lista de estudiantes
Lista de cursos
Obtener datos de curso
Lista de cursos
Lista de estudiantes
Página de estudiantes
Obtener datos de estudiante
Lista de estudiantes
Lista de cursos
Login
Lista de estudiantes
Lista de estudiantes
Editar Instructor
Lista de estudiantes
Obtener datos de estudiante
Lista de estudiantes
Obtener datos de estudiante
Lista de cursos
Lista de estudiantes
Lista de cursos
Lista de instructores
Lista de departamentos
Crear nuevo departamento
Lista de departamentos
Lista de cursos
Lista de estudiantes

Además, los registros de aplicación se procesaron por el módulo de tiempos de espera, obteniendo histogramas para todas las ocurrencias de los pedidos HTTP reconocidos, lo

que utilizaremos luego previo a la etapa de simulación para sortear los tiempos que usaremos en la simulación.

Una vez realizado el reconocimiento, generamos el modelo árbol, asignando un nombre único a cada sesión, y tratando las ocurrencias de la **Tabla 5-1** como si se tratara de una tira de símbolos.

El árbol generado es de profundidad 1, lo que indica que el modelo captura el comportamiento como un proceso de Markov de orden 1. Para cada nodo del árbol se contabilizaron las ocurrencias de la **Tabla 5-2**, con la que calculamos las probabilidades condicionales para cada nodo.

Tabla 5-2: Datos estadísticos del modelo árbol

Estado	Apariciones	
raiz		
	Obtenerdatosdecurso	1
	EditarInstructor	1
	Login	1
	Listadedepartamentos	2
	Paginadeestudiantes	4
	Listadecursos	6
	Editarestudiante	1
	Listadeinstructores	1
	Listadeestudiantes	9
	Crearnuevodepartamento	1
	Obtenerdatosdeestudiante	3
	0	5
Obtenerdatosdecurso		
	Obtenerdatosdecurso	0
	EditarInstructor	0
	Login	0
	Listadedepartamentos	0
	Paginadeestudiantes	0
	Listadecursos	1
	Editarestudiante	0
	Listadeinstructores	0
	Listadeestudiantes	0
	Crearnuevodepartamento	0
	Obtenerdatosdeestudiante	0
	0	0
EditarInstructor		
	Obtenerdatosdecurso	0
	EditarInstructor	0
	Login	0
	Listadedepartamentos	0
	Paginadeestudiantes	0
	Listadecursos	0
	Editarestudiante	0
	Listadeinstructores	0

Listadeestudiantes	1
Crearnuevodepartamento	0
Obtenerdatosdeestudiante	0
0	0

Login

Obtenerdatosdecurso	0
EditarInstructor	0
Login	0
Listadedepartamentos	0
Paginadeestudiantes	0
Listadecursos	0
Editarestudiante	0
Listadeinstructores	0
Listadeestudiantes	1
Crearnuevodepartamento	0
Obtenerdatosdeestudiante	0
0	0

Listadedepartamentos

Obtenerdatosdecurso	0
EditarInstructor	0
Login	0
Listadedepartamentos	0
Paginadeestudiantes	0
Listadecursos	1
Editarestudiante	0
Listadeinstructores	0
Listadeestudiantes	0
Crearnuevodepartamento	1
Obtenerdatosdeestudiante	0
0	0

Paginadeestudiantes

Obtenerdatosdecurso	0
EditarInstructor	0
Login	0
Listadedepartamentos	0
Paginadeestudiantes	2
Listadecursos	0
Editarestudiante	1
Listadeinstructores	0
Listadeestudiantes	0
Crearnuevodepartamento	0
Obtenerdatosdeestudiante	1
0	0

Listadecursos

Obtenerdatosdecurso	0
EditarInstructor	0
Login	1
Listadedepartamentos	0
Paginadeestudiantes	0
Listadecursos	0
Editarestudiante	0
Listadeinstructores	1
Listadeestudiantes	3

Crearnuevodepartamento	0
Obtenerdatosdeestudiante	0
0	0

Editarestudiante

Obtenerdatosdecurso	0
EditarInstructor	0
Login	0
Listadedepartamentos	0
Paginadeestudiantes	0
Listadecursos	0
Editarestudiante	0
Listadeinstructores	0
Listadeestudiantes	1
Crearnuevodepartamento	0
Obtenerdatosdeestudiante	0
0	0

Listadeinstructores

Obtenerdatosdecurso	0
EditarInstructor	0
Login	0
Listadedepartamentos	1
Paginadeestudiantes	0
Listadecursos	0
Editarestudiante	0
Listadeinstructores	0
Listadeestudiantes	0
Crearnuevodepartamento	0
Obtenerdatosdeestudiante	0
0	0

Listadeestudiantes

Obtenerdatosdecurso	0
EditarInstructor	1
Login	0
Listadedepartamentos	0
Paginadeestudiantes	1
Listadecursos	3
Editarestudiante	0
Listadeinstructores	0
Listadeestudiantes	1
Crearnuevodepartamento	0
Obtenerdatosdeestudiante	2
0	0

Crearnuevodepartamento

Obtenerdatosdecurso	0
EditarInstructor	0
Login	0
Listadedepartamentos	1
Paginadeestudiantes	0
Listadecursos	0
Editarestudiante	0
Listadeinstructores	0
Listadeestudiantes	0
Crearnuevodepartamento	0

	Obtenerdatosdeestudiante 0 0 0
Obtenerdatosdeestudiante	Obtenerdatosdecurso 0 EditarInstructor 0 Login 0 Listadedepartamentos 0 Paginadeestudiantes 0 Listadecursos 1 Editarestudiante 0 Listadeinstructores 0 Listadeestudiantes 2 Crearnuevodepartamento 0 Obtenerdatosdeestudiante 0 0 0
0	Obtenerdatosdecurso 0 EditarInstructor 0 Login 0 Listadedepartamentos 0 Paginadeestudiantes 1 Listadecursos 0 Editarestudiante 0 Listadeinstructores 0 Listadeestudiantes 0 Crearnuevodepartamento 0 Obtenerdatosdeestudiante 0 0 0

Algunos apuntes interesantes sobre estos resultados:

1) El estado 0 tiene 1 sola ocurrencia para página de estudiantes, por lo que con probabilidad 1 la simulación irá hacia esa sesión. Esto demuestra que el modelo captura el primer elemento al que accedió el usuario, manteniendo coherencia con la sesión de entrada registrada.
2) La probabilidad de ir hacia el estado 0 es nula desde cualquier sesión, salvo cuando tenemos la tira vacía, lo que ocurre solo una vez al principio.
3) El estado raíz cuenta la cantidad de ocurrencias de cada sesión en el registro procesado.

A partir de estos datos, se ejecutó la simulación de varias tiras, de las que destacaremos los escenarios de la **Tabla 5-3**.

Tabla 5-3: Escenarios simulados

Número	Escenario	
1		
	Página de estudiantes	5304.0
	Editar estudiante	5271.0
	Lista de estudiantes	6511.0
	Lista de cursos	3349.0
	Lista de estudiantes	156.0
	Lista de cursos	3517.0
	Lista de estudiantes	65405.0
	Lista de cursos	3574.0
	Lista de estudiantes	3972.0
	Lista de cursos	10714.0
	Lista de estudiantes	2132.0
	Lista de cursos	3993.0
	Lista de estudiantes	10583.0
	Lista de cursos	2170.0
	Lista de estudiantes	3074.0
2		
	Página de estudiantes	1944.0
	Obtener datos de estudiante	5766.0
	Lista de estudiantes	4474.0
	Obtener datos de estudiante	4523.0
	Lista de estudiantes	5837.0
	Obtener datos de estudiante	4273.0
	Lista de estudiantes	737.0
	Obtener datos de estudiante	4595.0
	Lista de estudiantes	3941.0
	Obtener datos de estudiante	4986.0
	Lista de estudiantes	2811.0
	Obtener datos de estudiante	4786.0
	Lista de estudiantes	4117.0
	Obtener datos de estudiante	4774.0
3		
	Página de estudiantes	896.0
	Editar estudiante	1024.0
	Lista de estudiantes	814.0
	Página de estudiantes	976.0
	Editar estudiante	9587.0
	Lista de estudiantes	1886.0
	Página de estudiantes	661.0
	Editar estudiante	4428.0
	Lista de estudiantes	1445.0
	Página de estudiantes	5507.0
	Editar estudiante	2216.0

Como observamos en la **Tabla 5-3**, a partir del registro de un único usuario se generaron 3 escenarios diferentes cuya ejecución puede ejercitar el SUT de maneras potencialmente distintas.

Si bien la herramienta tiene un módulo de ejecución de pruebas en JMeter, el resultado intermedio obtenido puede ser utilizado por un analista para generar sus propias pruebas a partir de una herramienta externa, lo que convierte este resultado intermedio en un valor agregado para el analista.

6 Conclusiones y trabajos a futuro

6.1 Conclusiones

Desarrollar un plan de pruebas de rendimiento para una herramienta web se trata de un trabajo para el que se requiere conocimiento en muchos campos, desde el estadístico pasando por varios aspectos técnicos, que lo hacen una tarea difícil.

En este trabajo exhibimos estas complejidades, y presentamos estrategias para su resolución, brindando una herramienta completa de punta a punta.

Se implementó una herramienta a modo de prueba de concepto que demuestra que nuestro trabajo modela correctamente el problema, que es aplicable en una aplicación real, y que los resultados que se obtienen (en el contexto de nuestras pruebas) son los esperados.

Además la herramienta entrega resultados intermedios de gran utilidad, producto de la aplicación del procesamiento de datos y el análisis estadístico, brindándole al analista la posibilidad de utilizar dichos resultados para diseñar un plan de pruebas propio, que podrá ser ejecutado con una herramienta externa para pruebas de carga.

Finalmente, con un caso de prueba, se muestra una opción para la recolección de datos del servidor en vivo la cuál puede ser aplicada en una aplicación real, aunque dicho enfoque puede no siempre estar disponible por temas operativos.

6.2 Trabajos a futuro

Presentaremos aquí una serie de aspectos que se podrían realizar en un futuro para emprender investigaciones similares o fortalecer la investigación realizada.

6.2.1 Agregado de información semántica

La generación de una base de datos para ejecutar las pruebas puede ser un problema complejo de resolver. Si bien se presenta en la aplicación de prueba un mecanismo para recolectar dichos datos, en la vida real existen restricciones operativas, como pueden ser la protección de datos confidenciales, que no permiten la aplicación de dicho tipo de técnicas. Este inconveniente podría solucionarse definiendo un estándar de información semántica a ser introducido por los programadores en tiempo de desarrollo o mantenimiento, de manera que un procesador pudiera procesar el tráfico de páginas en vivo y procesar dicha información, o directamente obtener los datos de fuentes externas de información cumpliendo determinadas reglas definidas a nivel semántico (utilizando, por ejemplo RDF).

6.2.2 Modelado de tiempos de procesamiento

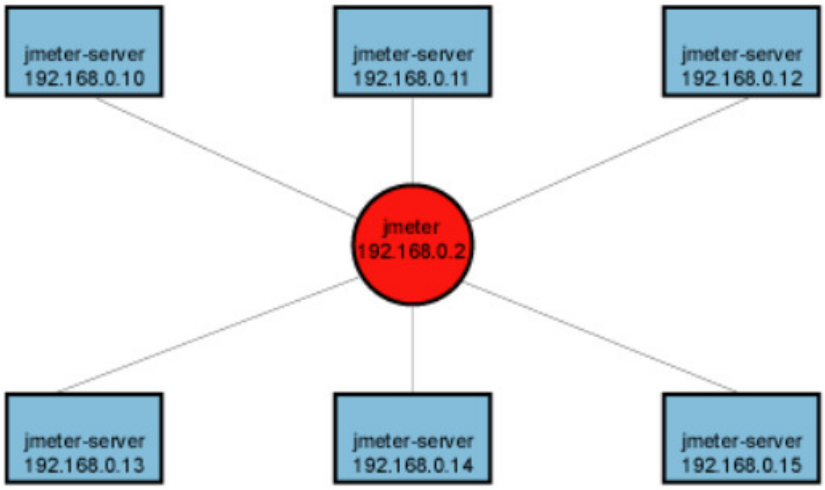
Los tiempos de procesamiento se modelaron mediante histogramas, sin embargo esta no es una técnica del todo exacta para modelar el comportamiento de los usuarios. Una posible mejora sobre este modelo podría ser realizar análisis de clústeres con el comportamiento de los usuarios como se describe en el capítulo 2, agrupando los tiempos de espera en categorías que consideran el nivel de pericia de los usuarios de la aplicación, para que a la hora de sortear los tiempos a simular se puedan generar tiempos de ejecución que dependan del tipo de usuario sorteado, dando un aspecto de consistencia mayor al sorteo, ya que con solamente histogramas por acciones puede suceder por ejemplo que se sorteen tiempos de experto para una sesión, y de novato para otra sesión, dentro del escenario de un mismo usuario, lo cual en la realidad no puede suceder ya que un usuario experto actuará, por lo general, como experto en el resto de sus interacciones con la aplicación.

6.2.3 Agregar características de JMeter

La utilización que el sistema le da a JMeter es la básica para poder realizar pedidos HTTP y manejar los resultados. Consideramos que es posible aprovechar la funcionalidad del testing distribuido.

JMeter permite utilizar múltiples sistemas para realizar las pruebas de carga, a través de un modelo maestro – esclavo (**Figura 6-1** y **Figura 6-2**).

Figura 6-1: Modelo maestro esclavo de JMeter.

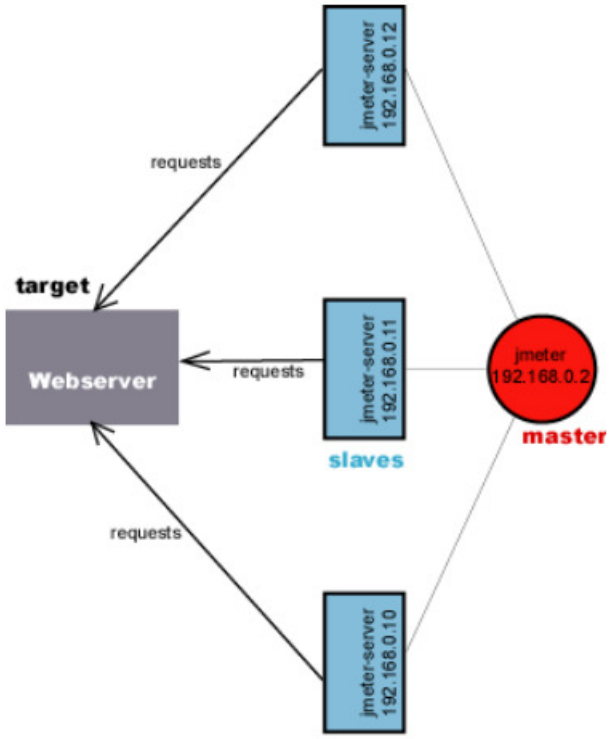


Terminología:

Maestro: el sistema que controla el test.

Esclavo: el sistema que corre el servidor-jmeter, toma los comandos pasados por el maestro y manda pedidos al SUT.

Figura 6-2: Modelo maestro esclavo en acción.



El sistema puede explotar este modelo funcionando como maestro y ejecutando un *vuser* o lote de *vusers* en cada esclavo creado.

Este modelo presenta algunas limitaciones, presentadas en la documentación oficial, que en resumen son:

1. JMeter no puede comunicarse entre subredes que no tengan proxy.
2. Es fácil saturar la IO de la red.
3. A no ser que el servidor sea un sistema multiprocesador grande, es posible que 1-2 clientes sobre carguen el servidor.
4. Un solo cliente JMeter corriendo en una CPU de 2-3 Ghz puede manejar entre 300 y 600 hilos de ejecución dependiendo del tipo de test.

6.2.4 Realizar pruebas exhaustivas

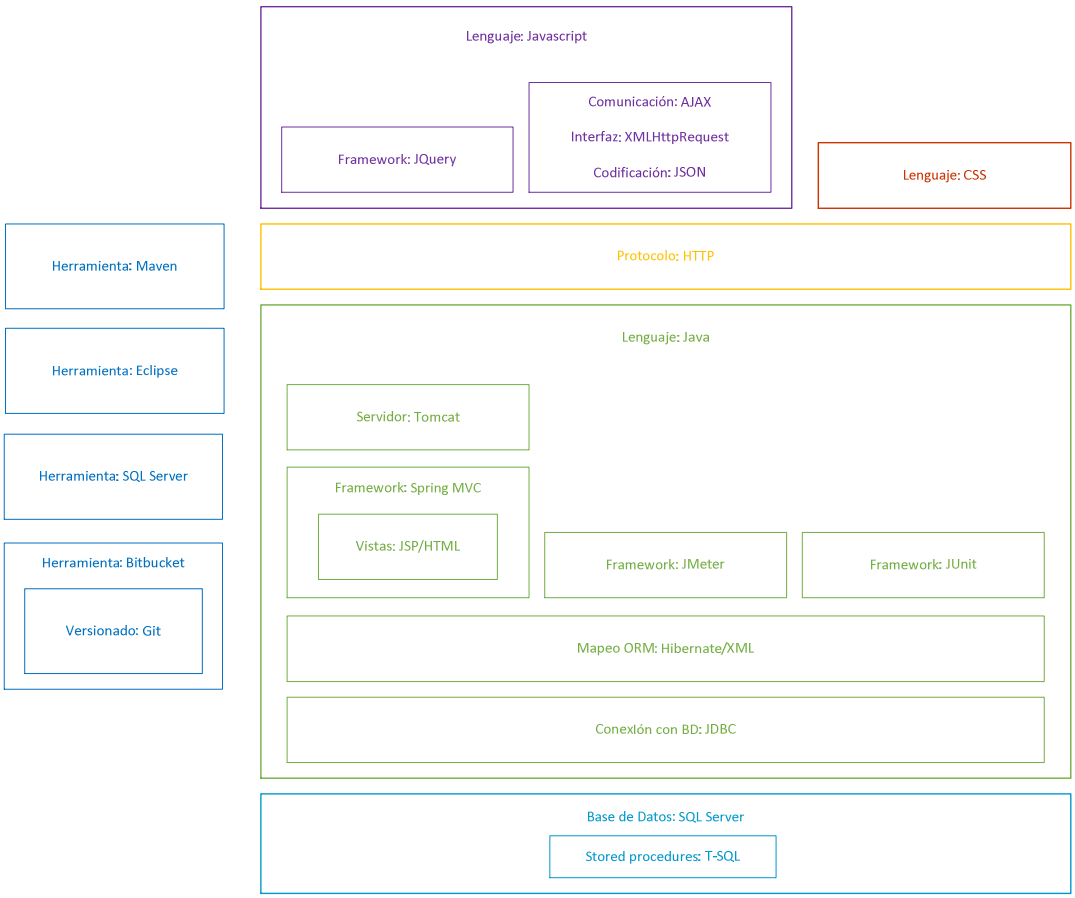
La herramienta aquí presentada fue probada en un contexto acotado el cual demuestra que el concepto es aplicable en el mundo real. Sin embargo sería interesante evaluar el desempeño de la herramienta en ambientes de producción real, con datos reales de uso, y evaluar los resultados obtenidos en ese contexto para deducir de manera más concluyente que el funcionamiento es el deseado.

7 Anexos

7.1 Protocolos, lenguajes y tecnologías

Presentamos a continuación el conjunto de protocolos, lenguajes, frameworks y herramientas que representan la infraestructura a partir de la cual tanto el sistema como la aplicación web fueron construidos.

Figura 7-1: Protocolos, lenguajes, frameworks y herramientas utilizados.



El sistema fue creado como un proyecto Java en *Eclipse Juno* utilizando la herramienta *Maven* para la gestión y construcción del mismo, que utiliza el archivo *Project Object Model (POM)* basado en un formato XML como modelo de configuración.

Se utiliza *SQL Server 2012* como sistema gestor de base de datos y su lenguaje de consulta *T-SQL* para construir procedimientos almacenados.

La lógica del sistema está íntegramente implementada en *Java*, y para algunas funcionalidades empleamos los siguientes componentes como se describe:

Un driver JDBC sobre SQL Server para conectar con la base de datos.

Hibernate configurado en XML para realizar la conversión de datos entre el sistema de tipos orientado a objetos y la base de datos relacional.

El framework *JMeter* para realizar la ejecución de los *vusers* como ya mencionamos anteriormente.

La Aplicación Web la construimos sobre el framework *Spring MVC* utilizando *JSP (Java Server Pages)* para las vistas y *Apache Tomcat* como servidor.
La comunicación con el servidor es principalmente asíncrona a través de *AJAX* y la implementación que brinda el framework *JQuery*, pasando los datos como *JSON.*
Utilizamos hojas de estilo *CSS* para dar una apariencia personalizada.

También utilizamos *JUnit* como framework para realizar pruebas unitarias sobre los componentes del sistema.

Y para el manejo del código usamos *Bitbucket* para el alojamiento del repositorio, con sistema de control de versionado *Git*, de gran ayuda para gestionar la manipulación concurrente de archivos.

8 Referencias

[1] S. T. Help, «Top 15 Performance Testing Tools,» 2013. [En línea]. Available: http://www.softwaretestinghelp.com/performance-testing-tools-load-testing-tools/. [Último acceso: Abril 2015].

[2] Microsoft, «Contoso University MVC NTier,» Microsoft, 7 Noviembre 2014. [En línea]. Available: http://www.asp.net/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application.

[3] J. Zinke, H. Jan y B. Schnor, *servload: Generating Representative Workloads for,* U. o. P. Institute for Computer Science, Ed., http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=6267027.

[4] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth y G. Weber, *Realistic Load Testing of Web Applications,* http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1602358, 2006.

[5] X. Wang, B. Zhou y W. Li, *Model Based Load Testing of Web Applications,* International Symposium on Parallel and Distributed Processing with Applications. http://www.tandfonline.com/doi/ref/10.1080/02533839.2012.726028, 2012.

[6] L. Xu, W. Zhang y L. Chen, *Modeling Users' Visiting Behaviors for Web Load Testing by Continuous Time Markov Chain,* IEEE. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5581372, 2010.

[7] J. Rissanen, «A universal data compression system,» p. http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1056741, 1983.

[8] W3C, «Extended Log File Format,» [En línea]. Available: http://www.w3.org/TR/WD-logfile.html. [Último acceso: Abril 2015].

[9] A. Martín, N. Merhav, G. Seroussi y M. J. Weinberger, «Twice-Universal Simulation of Markov Sources and Individual Sequences,» *IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 56, NO. 9,,* vol. 56, nº 9, p. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4557191, 2010.

[10] D. W. Scott, «On optimal and data-based histograms,» *Biometrika,* pp. http://ie.technion.ac.il/serveng/Recitations/scott-biometrika-1979.pdf, 1979.