Apuntes de Simulación Informática

Ingeniería Informática -II56-

Autor: Rafael Berlanga Llavori

Curso 2006–2007

Índice general

Capítulo 1

Sistemas y Modelos de Simulación

1.1. Introducción

La Simulación es una herramienta de análisis que nos permite sacar conclusiones sin necesidad de trabajar directamente con el *sistema real* que se está simulando. La simulación es especialmente útil cuando no disponemos de dicho sistema real o nos resulta demasiado arriesgado realizar experimentos sobre él.

Posiblemente muchos de vosotros asociéis la simulación con los famosos simuladores de vuelo con el que habréis jugado alguna vez. En general, se podría considerar que muchos juegos de ordenador son simuladores, ya que recrean parte de la realidad. Sin embargo, su intención no es el análisis sino el ocio, y esto es una diferencia bastante importante.

En muchas áreas de ingeniería se utilizan los simuladores como una herramienta de trabajo más. Por ejemplo, en el diseño de nuevos fármacos se suelen utilizar modelos moleculares que sirven para simular mediante un ordenador la interacción entre compuestos químicos. Los ingenieros de automóviles también utilizan modelos computerizados para analizar el impacto de los choques en la seguridad de los viajeros. Seguramente a ti también se te ocurran otros escenarios donde se utiliza un simulador como herramienta de trabajo.

Es importante observar que, al contrario que en los juegos que hemos mencionado al principio, en la simulación existe un motivo especial (objetivo) por el cual se lleva a cabo la simulación. Además, en toda simulación es necesario realizar muchas pruebas para llegar a unas conclusiones, es decir requiere un proceso de experimentación.

Otro tipo de simulación de la que posiblemente habéis oído hablar es la Simulación Numérica. Muchos métodos de integración se basan en este tipo de simulación. Básicamente, estos métodos se basan en la generación de números aleatorios para la obtención de una solución aproximada. Un buen ejemplo de este tipo de simulación es el método de Monte-Carlo, para calcular integrales de forma aproximada. Sin embargo, en la simulación numérica no se simula ningún sistema real, ni existe un proceso de experimentación para sacar conclusiones.

Desde el punto de vista de la Ingeniería Informática lo que nos interesa es simular los sistemas informáticos con los que trabajamos habitualmente, por ejemplo,

sistemas distribuidos, redes de ordenadores, bases de datos, entornos de multiprogramación, etc. En general, nos interesará simular el comportamiento de aquellos Sistemas Informáticos que bien por su complejidad o por su coste elevado no podemos estudiar ni mediante técnicas analíticas (modelos matemáticos directamente resolubles) ni a través de pruebas reales. Así pues, la simulación resultará especialmente útil en la fase de estudio de viabilidad, previa al desarrollo e implantación de un Sistema Informático.

1.1.1. Ciclo de desarrollo de un Simulador

Antes de comenzar a diseñar un simulador debemos tener bien claro cuál es el *objetivo* de la simulación. En un sistema informático, los objetivos de la simulación suelen ser dos: estimar el tiempo medio de respuesta del sistema, y estimar la capacidad máxima del sistema (número máximo de peticiones por unidad de tiempo que puede soportar el sistema).

Ejemplos de objetivos son: estimar el retraso medio de los paquetes en una determinada topología de red, estimar el tiempo medio de respuesta de una Base de Datos distribuida para un perfil de consulta determinado, estimar el número máximo de clientes simultáneos que puede soportar un servidor, etc.

En la figura 3.1 se muestra el ciclo completo de desarrollo de un simulador, el cual pasamos a describir a continuación.

Modelo del Sistema. Una vez fijados los objetivos, debemos analizar la estructura y comportamiento del sistema e intentar plasmarlo en papel. Es decir, debemos diseñar un *modelo* que abstraiga las partes más relevantes del sistema. En esta parte del diseño es muy importante seleccionar bien qué partes del sistema son interesantes modelar, siempre sin perder de vista los objetivos de la simulación. Por ejemplo, en una red de estaciones donde queremos medir el retraso medio de los paquetes sólo debemos modelar las partes del sistema que afectan a dicho retraso, es decir: las colas de procesos que se forman en las estaciones y el tiempo medio de transmisión de un paquete. En las siguientes secciones mostraremos las distintas técnicas de modelado para sistemas que utilizaremos durante el curso.

Modelo Computacional. Cuando el modelo ya ha sido suficientemente especificado (y a ser posible validado por algún experto), pasaremos a la implementación del simulador. Para ello utilizaremos programas, o paquetes de programas, que contengan las rutinas necesarias para realizar nuestro simulador. Entre otras cosas, estos paquetes o programas deben contener como mínimo las siguientes funciones:

- Generadores de números y variables aleatorias.
- Gestor de eventos y/o procesos.
- Funciones de análisis estadístico.

Experimentación con el simulador. Una vez implementado y depurado el simulador, debe diseñarse la batería de pruebas que nos permita extraer los resultados, y a partir de éstos las conclusiones de la simulación. Esta última etapa es la más

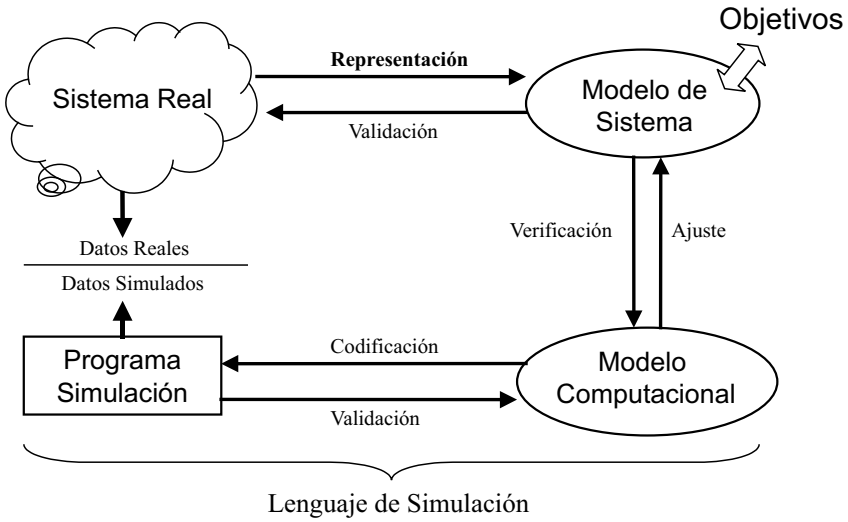


Figura 1.1: Ciclo de desarrollo en Simulación.

importante y delicada de todo el proceso de simulación. Cualquier error en el modelo o su implementación desembocará en un error importante en las conclusiones. Asimismo, un error en la selección de la batería de pruebas también puede conducir a un error en las conclusiones finales. Por esta razón, antes de experimentar con el simulador es muy importante asegurarse de que refleja fielmente el modelo, y que éste no contiene incongruencias con respecto al sistema real. Es decir debe ser *correcto* y *válido*.

1.2. Herramientas para el modelado de sistemas

En esta sección vamos a revisar las principales herramientas de modelado que utilizaremos en este curso para construir los modelos de nuestros simuladores. Antes de entrar en materia, es importante resaltar los aspectos del sistema real que vamos a tener en cuenta durante su modelado:

- *La estructura del sistema*: las partes de las que está compuesto el sistema, y cómo se relacionan entre sí.
- *La dinámica del sistema*: cómo evoluciona el sistema en el tiempo, los cambios que en él acontecen.
- *Los recursos del sistema*: qué partes del sistema son compartidas por distintos agentes y cómo se gestiona su servicio.

En las siguientes secciones vamos a ver distintas herramientas para modelar estos aspectos del sistema real.

1.2.1. Estructura de un sistema

El elemento más básico en un modelo de simulación es la *variable de estado*. Una variable de estado representa una parte del sistema que cambia con el tiempo. Ejemplos de variables de estado son el tamaño de una cola de espera, el número de clientes procesados en un servidor, etc.

Cuando se modela sistemas muy complejos, resulta bastante complicado identificar todas las variables de estado que nos van a interesar para la simulación. Para estos casos, resulta más adecuado utilizar alguna metodología de modelado que sea capaz de representar entidades complejas y sus relaciones. En nuestro caso, la metodología más apropiada y familiar es la Orientación a Objetos, debido principalmente a los siguientes motivos:

- Permiten varios grados de *abstracción* del sistema. Utilizando la modularidad y aplicando refinamientos sucesivos podemos ir representando los distintos niveles de detalle del sistema.
- Podemos ocultar la estructura y ciertos detalles de los objetos (incluida su implementación final).
- Podemos construir nuevos modelos re-utilizando el conocimiento adquirido en otros anteriores.
- *Escalabilidad*, podemos aumentar el modelo añadiendo nuevos módulos sin que ello suponga un mayor esfuerzo.

En este curso, utilizaremos la notación gráfica del modelo lógico de UML (Unified Modelling Language). UML es el lenguaje de modelado más aceptado en el diseño de Sistemas de Información. Soporta una notación gráfica para describir los distintos elementos empleados en Ingeniería del Software: interfaces, clases, objetos, estados, componentes, flujo de datos, etc. En concreto, el modelo lógico de UML constituye la vista estática de los objetos y las clases del modelo.

Clases y Objetos

En cursos anteriores seguramente ya te habrás familiarizado con la terminología de la Orientación a Objetos (OO), sobre todo a nivel de programación. Muchos de los conceptos que ya conoces en programación OO son utilizados de forma directa en el modelo lógico de UML. Por esta razón no vamos a profundizar demasiado en los conceptos básicos del modelo Orientado a Objetos.

Un *clase* es una especificación abstracta de un conjunto de objetos que comparten una serie de *atributos* y un *comportamiento*. Desde el punto de vista de la Simulación, los atributos de los objetos van a representar las *variables de estado* de ese objeto. El comportamiento de una clase se define con una serie de métodos, los cuales permiten manipular el estado de los objetos sin necesidad de conocer su estructura interna. Esta propiedad se conoce como *encapsulación*.

Para describir los métodos utilizaremos la siguiente notación:

$$nombre\_metodo(Clase_1, \ldots, Clase_k) : Clase$$

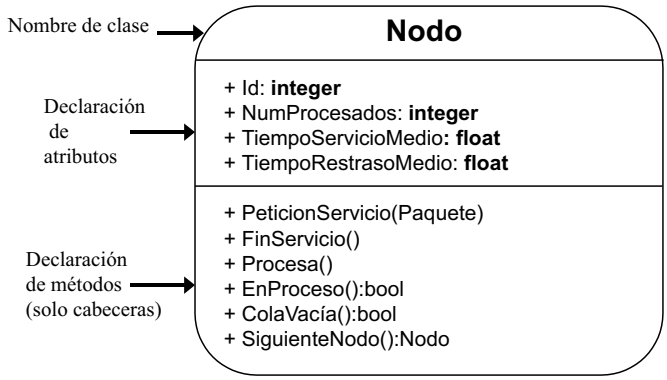


Figura 1.2: Ejemplo de clase en UML.

Fíjate que junto con el nombre del método se especifican los objetos que sirven de entrada y el objeto resultado de su aplicación (todos son opcionales).

En el modelo lógico de UML, una clase se representa como una caja con tres secciones: el nombre de la clase, los atributos de la clase y el comportamiento de la clase (métodos). En la figura 3.2 se muestra un ejemplo de clase en UML, para representar los nodos de una red de ordenadores.

En UML los atributos y métodos se pueden marcar como *públicos* (+), *privados* (-) y *protegidos* (#), para indicar los distintos grados de visibilidad de éstos con respecto al que usa la clase. En el primer caso, se indica que los atributos son visibles para todos, en el segundo caso que no son visibles para las clases externas, y en el último caso se indica que los atributos solo son visibles para las subclases de la clase.

En UML existen otras marcas sobre métodos y atributos, pero no resultan de interés en los modelos de Simulación.

Un *objeto* en UML es simplemente una caja con el nombre de la clase del objeto y su identificador único. A partir de ahora, utilizaremos la siguiente notación: $O.at$ representa el atributo at del objeto O, y $O.m()$ representa el método $m()$ del objeto O. Fíjate que ésta es una notación similar a la utilizada en C++ o en Java.

Relaciones

Las relaciones entre las clases representan la estructura del sistema en forma abstracta. Los tipos de relaciones que tendremos en cuenta en nuestros modelos son las siguientes:

- **Agregación :** representa relaciones de composición (parte-de) entre objetos del modelo. Con esta relación expresaremos por ejemplo que un objeto "red" se compone de objetos "nodo", o que un "servidor" se compone de 3 objetos "discos" y un objeto "CPU". En el diagrama UML podemos anotar la cardinalidad de la relación en los extremos de la línea que representa la relación (ver figura 3.3).

- **Asociación :** representa cualquier tipo de relación entre entidades del modelo. Cada objeto que forma parte de la relación puede tomar un *rol*. Por ejemplo,

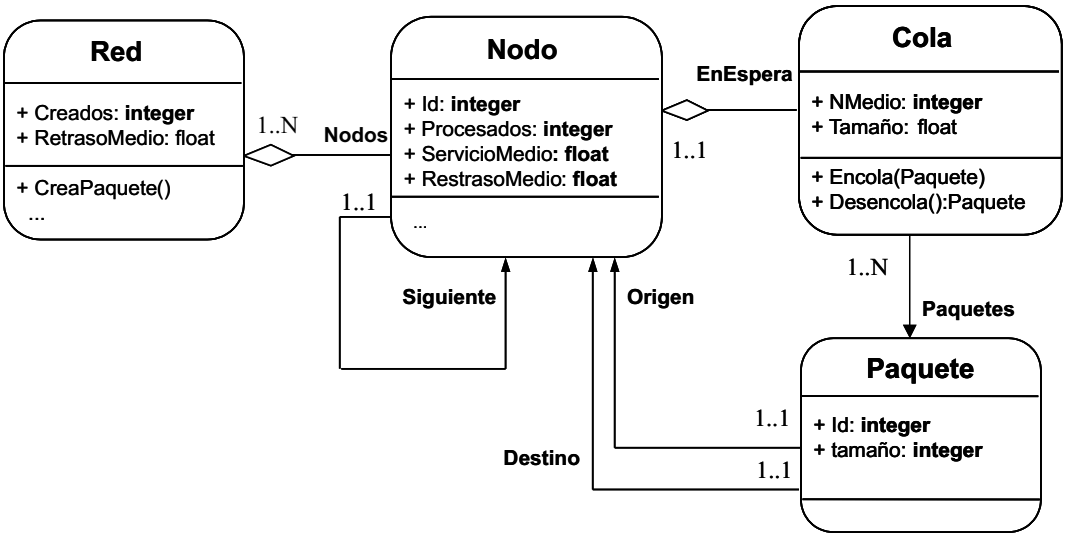


Figura 1.3: Ejemplo de diagrama de clases un UML.

el objeto "navegador" se relaciona con el objeto "servidor Web", el primero toma el rol de "cliente" y el segundo toma el rol de "servidor". Al igual que en la agregación, podemos indicar la cardinalidad de la relación.

- **Herencia :** representa relaciones de herencia entre clases. Una subclase hereda la estructura y comportamiento de su/s superclase/s. En UML la subclase siempre apunta a la superclase.

En la figura 3.3 se muestra un diagrama completo de clases donde se pueden ver varios ejemplos de las relaciones anteriores. El diagrama corresponde al modelo de una red de ordenadores.

Para más información sobre UML, en la dirección http://www.dsic.upv.es/ũml disponéis de un curso completo, así como diversos enlaces a recursos y material sobre el tema.

1.2.2. Diagramas de Sucesos Discretos

Con los diagramas UML podemos reflejar fácilmente las entidades del sistema con sus variables de estado y sus relaciones. En esta sección veremos cómo expresar la evolución de sistema a lo largo del tiempo. Para ello, introduciremos una notación gráfica para representar los sucesos de un sistema y cómo se relacionan éstos a lo largo del tiempo.

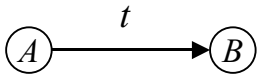
Un *suceso* representa un cambio en alguna variable de estado del modelo. Por simplicidad, asumiremos que estos cambios son instantáneos. Por ejemplo, consideremos los siguientes cambios en el modelo del ejemplo de la figura 3.3:

- *Creación de un paquete*, el cual incrementa la variable de estado $Red.Creados$.
- *Llegada de un paquete a un nodo* N, que incrementa la variable $N.Recibidos$, y modifica la cola de espera $N.Cola$.

- *Procesamiento de un paquete en un nodo* N, que incrementa la variable $N.Procesados$ y modifica la cola de espera $N.Cola$.

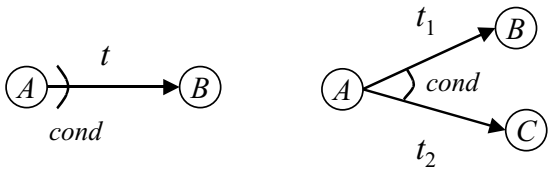
Cada uno de estos cambios representa un *suceso* aislado del sistema. Lo siguiente que tenemos que hacer es relacionar estos sucesos en el tiempo para expresar la dinámica del sistema.

La primera relación básica entre sucesos es la de *planificación* o relación *causa-efecto*. Con esta relación expresamos que después del suceso A va a ocurrir siempre el suceso B después de t unidades de tiempo ($t \geq 0$). Gráficamente lo representaríamos como:



Por ejemplo, la creación de un paquete causa necesariamente su llegada a un nodo de la red, y el tiempo transcurrido entre ambos será el tiempo de transmisión por la red. Como veremos más adelante, el tiempo entre sucesos será muchas veces una variable aleatoria que expresa la variabilidad de éstos.

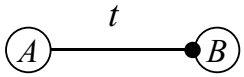
La segunda relación básica es la *planificación condicional*. Con esta relación expresamos que un suceso A causará la ocurrencia de un suceso B en t unidades de tiempo si se cumple una condición *cond* sobre las variables del sistema. Esta relación se representa como sigue:



Un ejemplo de esta relación sería la planificación del suceso de procesamiento de un paquete en un nodo si se cumple que la cola de espera del nodo está vacía.

La condición de planificación *cond* puede contener variables aleatorias para representar causas que se dan con una determinada probabilidad. Por ejemplo, un paquete procesado en un nodo podría tener distintas probabilidades de ir a los distintos nodos de la red.

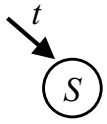
Otra relación importante entre sucesos es la *cancelación de sucesos*, la cual expresa que un suceso A provoca la cancelación de un suceso ya planificado B, transcurridas t unidades de tiempo. Esta relación también puede ser condicional, así como contener variables aleatorias. Su representación es como se muestra a continuación:



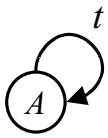
Un ejemplo de cancelación es la interrupción de un proceso debido a un fallo en el nodo, o cuando el tiempo estimado para terminar un proceso cambia debido a un cambio en la carga del nodo.

Además de las relaciones anteriores, existen otros elementos en el diagrama de sucesos que resultan necesarios para la simulación del sistema, a saber:

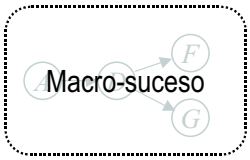
- **Sucesos iniciales:** los cuales representan por un lado las condiciones iniciales de la simulación, y por otro lado indican los puntos de entrada del diagrama de sucesos. Los sucesos iniciales los representaremos colocándoles una flecha de entrada.



- **Sucesos periódicos:** los cuales representan la repetición periódica de un suceso en t unidades de tiempo. Con estos sucesos se modela la llegada de clientes a un sistema, las roturas periódicas de una máquina, etc. Estos sucesos los representaremos como sigue:



- **Macro-sucesos:** se representa en forma de caja, y permite ocultar los sucesos de una parte del modelo. La idea es que a la hora de diseñar un diagrama primero se especifiquen los macro-sucesos y después se vayan refinando éstos por separado. Por ejemplo, todos los sucesos internos que se producen en un nodo podrían representarse con el macro-suceso "Proceso-en-Nodo", ocultando los detalles del mismo.



Ejercicios. Describe mediante diagramas de suceso los siguientes sistemas:

1. Un semáforo con los estados rojo, amarillo y verde.

2. Una máquina expendedora de bebidas. El usuario primero introducirá una cantidad de dinero y después realizará la petición deseada (café, cortado, etc.) Si la cantidad introducida no es suficiente, la máquina solicitará al usuario que introduzca más dinero. En caso contrario, la máquina pasará a preparar la bebida solicitada, cuyo tiempo de preparación dependerá del tipo de bebida. Finalmente, si la cantidad introducida era mayor que el coste de la bebida, la máquina devolverá el cambio correspondiente.

3. Un sistema de cuatro procesadores que distribuye uniformemente las tareas que recibe, según la carga de los procesadores.

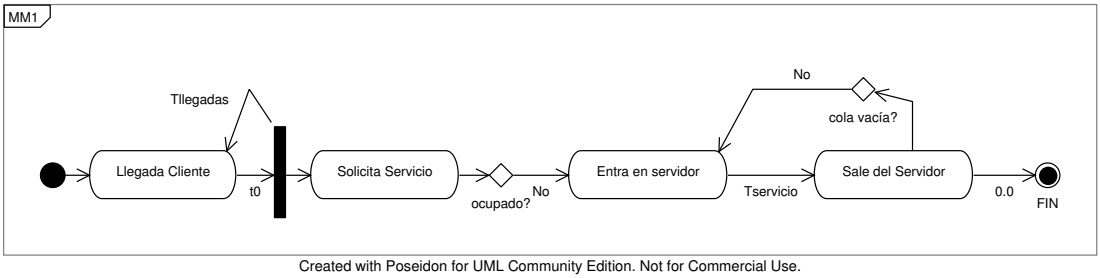


Figura 1.4: Diagrama UML de actividades para una estación de servicio.

4. Un sistema cliente-servidor donde un cliente envia una petición de forma simultánea a cinco servidores.

Aunque UML no soporta directamente la especificación de diagramas de sucesos temporizados como los que hemos visto en esta sección, si que es posible expresar los mismos mediante los denominados *diagramas de actividades*. En la figura 1.4 se muestra un diagrama de este estilo editado con la herramienta Poseidon [1], y puede compararse con el de la figura 1.6 que representa el diagrama de sucesos equivalente.

1.2.3. Recursos del sistema

Un *recurso* es una entidad del sistema que debe ser compartida por varios agentes. En los sistemas informáticos resulta frecuente encontrar recursos compartidos (ej.procesadores, servidores de bases de datos, nodos de una red, etc.), de ahí que sea crucial poder modelarlos de forma correcta.

El modelo más aceptado para representar los recursos de un sistema son las denominadas *redes de colas de espera*. En esta representación, las entidades básicas son las denominadas *estaciones de servicio*, las cuales están conectadas entre sí para expresar el flujo de las peticiones de los agentes (también denominados *clientes*) en los distintos recursos del sistema.

Sobre este modelo se define una teoría estadística, denominada *teoría de colas*, que es utilizada para evaluar analíticamente la eficiencia y fiabilidad de sistemas con recursos. Sin embargo, la aplicación de esta teoría resulta limitada debido a la complejidad de los sistemas reales, y en estos casos se recurre a las técnicas de simulación.

En general, una estación de servicio se caracteriza por los siguientes aspectos:

- La tasa de llegada de clientes a la estación.
- El tiempo que tarda en servirse cada cliente en la estación.
- El número de servidores de la estación.

[1] http://www.gentleware.com/index.php?id=ce

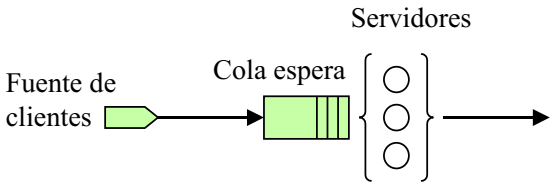


Figura 1.5: Notación gráfica para una estación de servicio.

- La capacidad del sistema, es decir, el número máximo de clientes que pueden esperar a ser servidos.
- La política de servicio, es decir, cómo se va a organizar el procesamiento de los clientes que esperan en la cola.

En la figura 1.5 se muestra la notación gráfica que se utiliza para representar una estación de servicio.

La siguiente tabla muestra el conjunto de variables de estado que se asocia a una estación de servicio de m servidores:

T	Tiempo entre llegadas (variable aleatoria)
$\lambda = 1/E[T]$	Tasa media de llegada (E es el valor esperado de T)
S	Tiempo de servicio de cada trabajo (variable aleatoria)
$\mu = 1/E[S]$	Tasa media de servicio
$I = \frac{\lambda}{\mu}$	Intensidad de tráfico (con $I > m$ el sistema está saturado)
N	Número de trabajos en la estación (variable aleatoria discreta)
N_q	Número de trabajos en espera (variable aleatoria discreta)
N_s	Número de trabajos recibiendo servicio (v.a. discreta)
R	Tiempo de respuesta del sistema (variable aleatoria)
W	Tiempo de espera en cola (variable aleatoria)

En la teoría de colas, una estación de servicio se especifica con la siguiente notación:

Distribución_Llegada/Distribución_Servicio/Número_Servidores

donde cada distribución puede identificarse con uno de los siguientes términos:

M	distribución exponencial (proceso de poisson)
E_k	distribución de una k-Erlang (suma de k procesos de poisson)
D	distribución determinística (tiempo constante)
H_k	distribución hiperexponencial con parámetro k
G	distribución general

Por ejemplo, el modelo M/M/1 representa de una estación de servicio de un servidor, con un proceso de llegada de poisson (exponencial) y un tiempo de servicio exponencial.

Si una estación de servicio es estable ($I \leq m$), entonces se cumples las siguientes fórmulas (fórmulas de Little):

$$\bar{N}_q = \lambda \bar{W}$$

$$\bar{N} = \lambda \bar{R}$$

$$\bar{N} = \bar{N}_q + I$$

Es decir, la media de clientes en la cola y la media de clientes en la estación son directamente proporcionales a las medias de los tiempos de espera y de los tiempos de respuesta, respectivamente. La relación entre ellas es la propia tasa de llegada.

Políticas de servicio

En cuanto a la *política de servicio* de una estación, podemos seleccionar una de las siguientes:

- *Firts Come First Served* (FCFS o FIFO): los clientes se sirven en orden de llegada. Esta política tiene como propiedad que todos los clientes esperan el mismo tiempo de media, independientemente del tiempo de servicio que empleen. Es decir, tanto las tareas largas como las cortas esperan de media lo mismo.

- *Last Come First Served* (LCFS o LIFO): los clientes se sirven en orden inverso al de llegada. Cuando llega un cliente nuevo, este pasa a servirse de inmediato (con apropiación), encolando al cliente que se estaba sirviendo en su caso. Esta política castiga a las tareas más largas, que deben interrumpirse cada vez que llega una nueva tarea. Si la tarea es lo suficientemente corta, no sufrirá interrupciones.

- *Round-Robin* (RR): todos los clientes van sirviéndose en incrementos de tiempo δt hasta completar sus respectivos tiempos de servicio. Al igual que la política anterior, ésta penaliza las tareas largas. De hecho, analíticamente se comprueba que las tareas con un tiempo inferior a $\bar{S^2}/(2 \cdot \bar{S})$ esperan menos tiempo que en el caso de una FCFS. Por contra, todas las que superen dicho tiempo esperarán un tiempo proporcional a su tiempo de servicio, que siempre será mayor que si se procesasen según la FCFS. Esto puede provocar que el sistema se sature con facilidad, dependiendo de la varianza de los tiempos de servicio ($\bar{S^2}$).

- *Processor Sharing* (PS): todos los clientes se sirven simultáneamente, pero con tiempos de servicio proporcionales al número de clientes que se están procesando en ese momento. Esta política es un caso particular de la anterior, donde los incrementos de tiempo δt son prácticamente cero.

- *Service in Random Order* (SIRO): se toma aleatoriamente el siguiente cliente a procesar.

- Con control de *prioridades*, las cuales pueden ser:

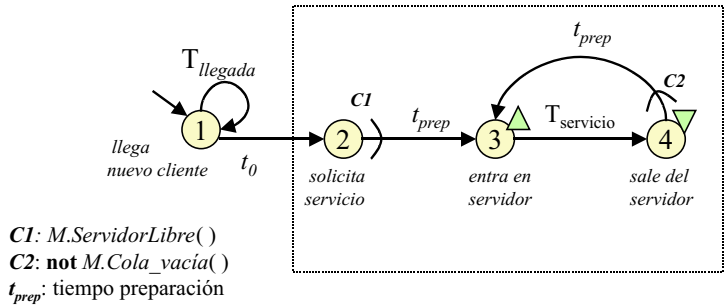


Figura 1.6: Diagrama de sucesos para una estación FCFS

- Expulsiva con reanudación (*preempt*): el cliente con más prioridad interrumpe al que se está sirviendo en ese momento, y el cliente interrumpido se encola y termina lo que le falta.
- Expulsiva con reinicialización: el cliente con más prioridad interrumpe al que se está sirviendo en ese momento, y el cliente interrumpido se encola y empieza de nuevo su servicio.
- No expulsiva: el cliente con más prioridad espera al que se está sirviendo.
- Dinámica: donde la prioridad de un cliente depende del número de veces que ha entrado en la estación.

En la figura 1.6 se muestra el diagrama de sucesos para la política de servicios más común, que es la FCFS, y dejamos como ejercicio realizar los diagramas de sucesos para el resto de políticas de servicio. En esta figura, el suceso ① representa la llegada de clientes, los cuales llegan cada $T_{llegada}$ unidades de tiempo. Cada uno de estos clientes planifica una petición a la estación de servicio (suceso ②). Si existe algún servidor libre, se planifica el procesamiento de la petición en t_{prep} unidades de tiempo. En el suceso ③ comienza el procesamiento de la petición, y en el suceso ④ termina. Por otro lado, hemos incluido dos símbolos en el diagrama de sucesos para representar la toma (Δ) y la liberación (∇) de una estación de servicio por parte de un cliente. Finalmente, cada vez que termina una petición, si la cola de espera no está vacía, se pasa a procesar la siguiente petición.

Ejecución de un diagrama de sucesos. Para ilustrar el funcionamiento de la estación descrita en la figura 1.6, supondremos que sus parámetros son los siguientes: $T_{llegada} = 0{,}4$, $T_{servicio} = 0{,}5$ y $t_{prep} = 0$. A continuación mostramos un fragmento de la traza de ejecución de la estación de servicio. En la primera columna se muestra el tiempo de simulación, en la segunda el suceso que se está ejecutando junto con el cliente implicado, las siguientes dos columnas son las variables de estado de la estación (servidor libre y la cola de peticiones). Finalmente, la última columna representa los sucesos planificados tras la ejecución de cada suceso (el que está marcado con $\surd$ es el siguiente suceso que va a ser ejecutado). En la tabla hemos sombreado los cambios producidos por el suceso en ejecución.

T_{sim}	*Ejecuta* (Suceso/Cliente)	*M.Libre*	*M.Cola*	*Planifica* (Suceso/Cliente, Tiempo)
0.0	-	libre	[ ]	(1/c1, 0.0) √
0.0	1/c1	libre	[ ]	(1/c2, 0.4) (2/c1, 0.0) √
0.0	2/c1	libre	[ ]	(1/c2, 0.4) (3/c1, 0.0) √
0.0	3/c1	ocupado	[ ]	(1/c2, 0.4) √ (4/c1, 0.5)
0.4	1/c2	ocupado	[ ]	(1/c3,0.8) (2/c2,0.4)√ (4/c1,0.5)
0.4	2/c2	ocupado	[c2]	(1/c3,0.8) (4/c1,0.5) √
0.5	4/c1	ocupado	[c2]	(1/c3,0.8) (3/c2,0.5) √
0.5	3/c2	ocupado	[ ]	(1/c3,0.8)√ (4/c2,1.0)
...	...	...	...	...

1.2.4. Cambios contínuos

Hasta ahora nos hemos centrado principalmente a representar sistemas informáticos mediante sucesos discretos y modelado de recursos. Sin embargo, no todos los sistemas pueden ser representados con estas herramientas. En particular, los modelos físicos asumen un tiempo continuo sobre el cual las variables de estado evolucionan también de forma continua. En este contexto los cambios no pueden modelarse como transiciones instantáneas de estado, sino que es necesario modelar los cambios infinitesimales que se van produciendo sobre las variables de estado. A este tipo de simulación se denomina *Simulación Continua*.

Los modelos de simulación continua se representan con un conjunto de variables de estado continuas (x, y, etc.) y las ecuaciones diferenciales que definen sus cambios. Una ecuación diferencial tiene la siguiente forma:

$$b_n \frac{d^n x}{d^n t} + b_{n-1} \frac{d^{n-1} x}{d^{n-1} t} + \ldots + b_1 \frac{dx}{dt} + b_0 = 0$$

En general, las ecuaciones diferenciales de un sistema físico no suele pasar del segundo orden, y por esta razón se utiliza la siguiente notación para expresar las derivadas parciales de primer y segundo orden:

$$\ddot{x} = \frac{d^2 x}{d^2 t}$$

$$\dot{x} = \frac{dx}{dt}$$

En la figura 1.7 se muestra un ejemplo sencillo de modelo de un sistema físico. No vamos a profundizar mas en el tema de Simulación Continua debido principalmente a que no es necesaria para modelar nuestros Sistemas Informáticos.

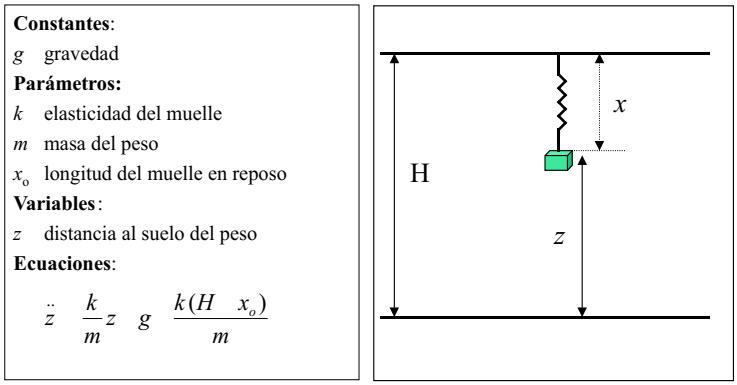


Figura 1.7: Ejemplo de modelo de Simulación Continua

1.3. Ejemplos de modelos

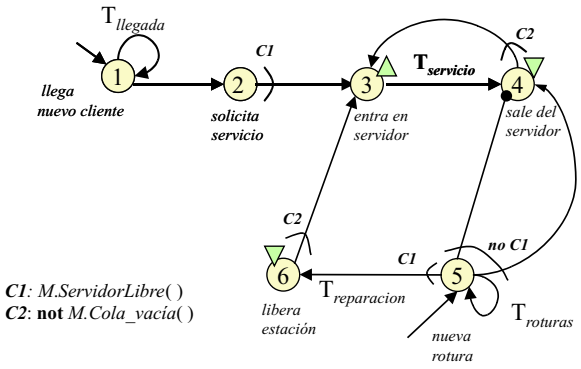
En esta sección vamos a desarrollar dos modelos de ejemplo. El primero es un modelo simple de un sistema genérico con una estación de servicio que sufre interrupciones debido a la presencia de roturas. El segundo ejemplo es un modelo de un sistema más complejo donde se combinan varias estaciones de servicio.

1.3.1. Estación con roturas

Los sistemas informáticos suelen presentar fallos en el funcionamiento de sus componentes que requieren ser modelados para aproximar con mayor exactitud su respuesta real. Ejemplos de fallos en un sistema pueden ser: las roturas físicas de algunas partes, lo que supone poner fuera de servicio esas partes durante mucho tiempo, paradas temporales debidas a ajustes o manteniemiento de alguna componente del sistema, interrupciones temporales en las conexiones de red, etc.

La forma de modelar las roturas suele realizarse extendiendo el modelo de la estación FCFS (o con cualquier otro tipo de política de servicio) con un nuevo suceso periódico (suceso ⑤) que replanifica el suceso de final de servicio (suceso ④) de la figura 1.6). Es decir, cada cierto tiempo (periodo de la rotura) se interrumpe el servicio de la estación, y se replanifica el servicio para dentro de $T_{reparacion} + T'_{servicio}$ unidades de tiempo, donde $T_{reparacion}$ es el tiempo en el que se tarda en reparar la estación, y $T'_{servicio}$ es el nuevo tiempo de servicio del cliente interrumpido. Fíjate todo esto debe realizarse si la estación estaba sirviendo a algún cliente (condición *no c*1). Finalmente, también debe tenerse en cuenta el caso en el que la rotura se produce con la estación vacía. En este caso, se bloquea la estación para que no procese a ningún cliente y se planifica su liberación con el tiempo de reparación (suceso ⑥). Como durante la reparación puede haberse encolado algún cliente, debemos planificar el suceso ③ para pasarlo a servir.

De este modo, el diagrama de sucesos correspondiente a una estación con roturas sería el siguiente:



Observa que la *replanificación* de un suceso se realiza en dos pasos: primero se cancela el suceso y después se replanifica con el nuevo tiempo.

El diagrama anterior puede presentar distintas variantes según sea la política de servicio de la estación. Así, si el cliente interrumpido debe expulsarse, entonces se cancela el suceso ④ y se planifica el suceso ③ con tiempo $T_{reparacion}$ para servir al siguiente cliente de la cola. Si en cambio el cliente interrumpido debe volverse a encolar, entonces planificaríamos el suceso ② con ese cliente.

1.3.2. Un modelo complejo

Supongamos un sistema informático compuesto de cuatro terminales (T1..T4), un procesador (CPU) y dos discos (D1 y D2). Los usuarios lanzan transacciones desde los terminales, donde cada transacción consiste en una serie de visitas a la CPU y a los discos. Hasta que una transacción no ha finalizado, no se lanza una nueva. De este modo, como mucho sólo pueden ejecutarse cuatro transacciones a la vez. Veamos un ejemplo de ejecución:

```
Transacción 1: T1 -> CPU -> D1 -> CPU -> D2 -> CPU -> T1
Transacción 2: T2 -> CPU -> D1 -> CPU -> T2
Transacción 3: T3 -> CPU -> D2 -> CPU -> D2 -> CPU -> D1 -> T3
Transacción 4: T4 -> CPU -> D1 -> CPU -> D1 -> CPU -> T4
```

La primera transacción consiste en un acceso a la CPU, seguido de un acceso al disco D1, otro acceso a la CPU, un acceso al disco D2, otro a la CPU, y finalmente se termina la transacción volviendo a la terminal de origen. Todos estos accesos se realizan de forma secuencial (uno detrás de otro). En cambio, las cuatro transacciones se ejecutan simultáneamente.

Las características de los dispositivos del sistema se resumen a continuación:

- El tiempo medio de servicio en la CPU es de 4ms.
- El tiempo medio de servicio en ambos discos es de 14ms.
- La comunicación entre dispositivos es menor de 1ms.
- La probabilidad de acceso a los discos desde la CPU es la misma.

Por otro lado, estamos interesados en estudiar el impacto que tiene el número medio de visitas a la CPU ($Niter$) en las prestaciones del sistema. Por ejemplo, podemos definir algunos objetivos interesantes con este parámetro:

- Estimar la duración media de las transacciones con $Niter = 3$. Para ello, definimos la variable de estado Ttrans.
- Estimar el número medio de transacciones procesadas para un periodo de tiempo determinado (por ejemplo 100s) y con $Niter = 3$. Para ello, definimos la variable de estado Ntrans
- Mostrar la evolución de las medidas anteriores conforme aumentamos $Niter$.

Modelando los recursos. Volvamos a los ejemplos de las transacciones anteriores. Podemos ver que las cuatro transacciones realizan accesos simultáneos tanto a la CPU como a los dos discos. Sin embargo, estos dispositivos sólo pueden atender una petición a la vez. Entonces ¿qué se hace con el resto de peticiones? ¿en qué orden se procesarán?

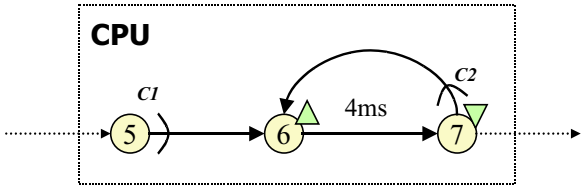
Estos tres dispositivos (CPU, D1 y D2) son los recursos del sistema. En general diremos que un objeto del sistema es un recurso si éste debe ser compartido por varios procesos. Como hemos visto anteriormente, los recursos se modelan como estaciones de servicio, donde hay que identificar una serie de aspectos, entre ellos: la política de servicio, el número de servidores, el tiempo de servicio, la tasa de llegada, su población, etc.

Dado que los tres dispositivos van a procesar las peticiones de una en una, y según su orden de llegada, supondremos que su política de servicio es la FCFS. Así pues, la CPU será una estación de servicio FCFS con un servidor y tiempo de servicio medio de 4ms, y cada uno de los discos será una estación de servicio FCFS con un servidor y tiempo de servicio medio de 14ms.

Observa que en este sistema las tasas de llegada de las estaciones de servicio vienen determinadas por el comportamiento global del sistema, concretamente por el número de peticiones y la longitud de las transacciones.

Diagrama de sucesos. Una vez hemos identificado los recursos, y modelado éstos como estaciones de servicio, tenemos que describir su comportamiento mediante un diagrama de sucesos.

Veamos en primer lugar el diagrama para la CPU:

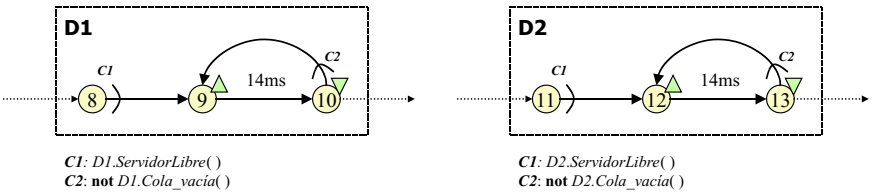


C1: *CPU.ServidorLibre*( )
C2: **not** *CPU.Cola_vacía*( )

Según este diagrama, el procesamiento en la CPU comienza en el suceso ⑤ (los sucesos del ① al ④ los reservamos para los sucesos iniciales). Si la CPU está ocupada,

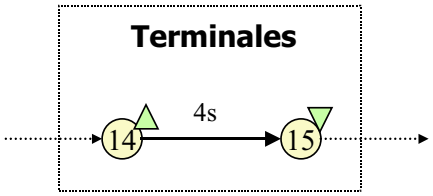
la petición pasará a una cola de espera. En caso contrario, el suceso ⑥ inicia el procesamiento de la petición, y el suceso ⑦ lo termina. El tiempo transcurrido entre estos dos sucesos es el tiempo de servicio (4ms). Observa que el suceso ⑦ planifica el procesamiento de un nuevo cliente si la cola no está vacía.

El diagrama de sucesos para cada uno de los discos es similar al anterior, aunque debemos cambiar el tiempo de servicio que ahora es de 14ms.



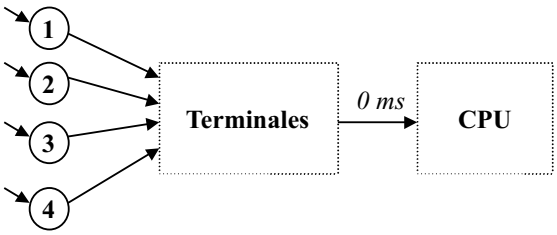
A partir de ahora, en el diagrama de sucesos global omitiremos los detalles de las estaciones de servicio, dibujando solo la caja con el nombre de la estación. Las flechas que apunten a la caja estarán en realidad planificando el primer suceso de la estación (por ejemplo el suceso ⑤ de la CPU). Por otro lado, las flechas que salgan de la caja serán las planificadas por el último suceso de la estación (por ejemplo el suceso ⑦ de la CPU).

Las cuatro terminales del sistema también pueden modelarse como una estación de servicio con cuatro servidores. En este caso no será necesario definir una política de servicio ya que nunca se formarán colas en las terminales. En cuanto al tiempo de servicio, ¿qué representa en las terminales? Dado que una petición de servicio a una terminal representa el inicio de una nueva transacción, el tiempo de servicio de ésta representará el tiempo que transcurre entre dos transacciones distintas (ó el tiempo de preparación de cada nueva transacción). Observa que en la descripción del sistema no se ha contemplado este parámetro; supondremos que éste es de 4000ms. Con todo esto, el diagrama de sucesos de las terminales quedaría como sigue:



Vamos a plantearnos ahora el diagrama de sucesos para todo el sistema. En primer lugar, nos fijaremos en los sucesos iniciales que desencadenan toda la simulación. Éstos serán los que planifiquen las primeras cuatro transacciones que deben lanzarse desde las terminales.

Los cuatro sucesos iniciales planifican el procesamiento en las terminales (suceso ⑭). Como esta estación de servicio tiene cuatro servidores, todas las peticiones serán procesadas simultáneamente, sin que se produzcan colas. A continuación, las terminales lanzan las transacciones que siempre deben comenzar por la CPU. Así pues, la salida de las terminales (suceso ⑮) debe planificar el procesamiento en la CPU (suceso ⑤). Todo lo anterior se representa del siguiente modo:



Observa que en el modelo hemos tomado como unidad mínima de tiempo 1 ms. Todos los tiempos que sean menores de 1 ms se supondrán que valen 0. Así, el tiempo de conexión entre los dispositivos será 0 ms.

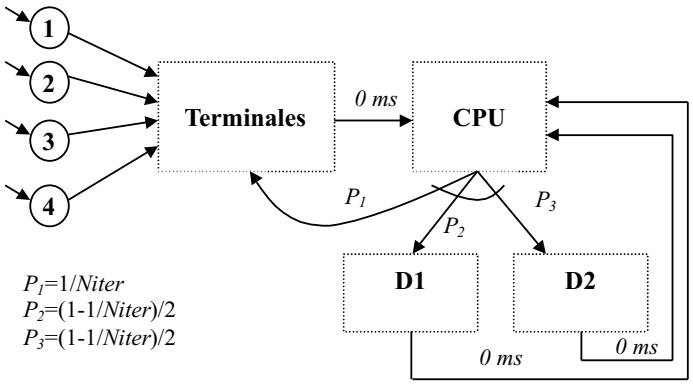
¿Qué sucesos pueden ocurrir a la salida de la CPU? Según la descripción del sistema pueden suceder tres cosas:

1. Que se planifique el final de la transacción, volviendo a las terminales (suceso ⑭)

2. Que se planifique el procesamiento en el disco D1 (suceso ⑧)

3. Que se planifique el procesamiento en el disco D2 (suceso ⑪)

¿Cuándo se seleccionará cada caso? Para responder a esta pregunta es necesario conocer el número medio de interacciones entre los dispositivos durante una transacción. Veamos. Si de media se realizan $Niter$ visitas a la CPU, la probabilidad de terminar una transacción cualquiera será $1/Niter$; y la de seguir la transacción será $1 - 1/Niter$. Dado que el acceso a los discos es equiprobable, esta última probabilidad se dividirá a partes iguales entre los dos discos, es decir que valdrá $(1 - 1/Niter)/2$. Toda esta información se plasma en el diagrama de sucesos como sigue:



Para finalizar el diagrama de sucesos, observa que hemos dibujado dos flechas que partiendo de los discos llegan hasta la CPU. Estas representan el hecho de que siempre hay que volver a la CPU para continuar la transacción.

En la siguiente tabla se resumen todos los sucesos y sus conexiones:

Id	Acción	Vbles. Estado	Planifica
①...④	Inicializa	Todas	14
⑤	Encola petición	CPU.Nq, CPU.W	6
⑥	Comienza servicio	CPU.Ns	7
⑦	Termina servicio	CPU.R	6 y (14, 8 o 11)
⑧	Encola petición	D1.Nq, D1.W	9
⑨	Comienza servicio	D1.Ns	10
⑩	Termina servicio	D1.R	5 y 9
⑪	Encola petición	D2.Nq, D2.W	12
⑫	Comienza servicio	D2.R	13
⑬	Termina servicio	D2.R	12 y 5
⑭	Termina transacción	Ttrans	15
⑮	Comienza transacción	NTrans, Ttrans	5

Ejercicios.

1. Amplia el modelo presentado para tener en cuenta que las *transacciones* pueden acceder además a un servidor remoto con una probabilidad 10 veces menor que a los discos. Supondremos que el tiempo medio para el establecimiento de la conexión remota es de 1000ms, y el tiempo de servicio en el servidor remoto será de 10ms.

2. Modifica el modelo del ejercicio anterior para modelar la situación en la que el tráfico en la red se interrumpe cada 10000ms de media. Cuando esto suceda, todos las peticiones planificadas de conexión al servidor remoto deberán retrasarse en 10ms.

3. Especifica un modelo para una estación de servicio con política *Round Robin*, modificando el modelo básico de una estación de servicio FCFS. Para ello, debes indicar las nuevas variables de estado y dibujar su diagrama de sucesos. Realiza una traza manual del diagrama obtenido, contemplando la posibilidad de que el tiempo de servicio tiende a cero. En este caso estaríamos ante una política de (*Processor Sharing*). ¿Las ejecuciones del modelo propuesto serían eficientes en este caso?

1.4. Bibliografía

M. Fowler "UML distilled a brief guide to the standard modeling language", Addison-Wesley (2000)

D. Ríos Insua et al. "Simulación métodos y aplicaciones", Ed. Ra-Ma, Madrid (1997)

A. M. Law, W. D. Kelton "Simulation Modeling & Analysis", Ed. McGraw-Hill (1984).

R. Jain "The Art of Computer Systems Performance Analysis", Ed. Wiley (1991).

J. J. Pazos et al. "Teoría de Colas y Simulación de Sucesos Discretos", Ed. Pearson/Prentice Hall (2003).

Capítulo 2

Programación de Simuladores

2.1. Introducción

En el tema anterior hemos visto cómo crear modelos de Simulación de sucesos discretos. En este tema veremos cómo implementar un simulador a partir de este tipo de modelos. Afortunadamente, en la actualidad existen numerosos paquetes de software y bibliotecas de funciones que contienen los elementos necesarios para programar un simulador de sucesos discretos. Por lo tanto no será necesario que implementemos desde cero nuestros simuladores.

En este tema vamos a ver dos paradigmas de programación para simulación: la orientada a sucesos y la orientada a procesos. Básicamente, el primero de ellos se basa en la ejecución secuencial del diagrama de sucesos, mientras que el segundo se basa en la identificación de procesos recurrentes en el diagrama de sucesos y su ejecución en paralelo. Como ejemplo del primer paradigma introduciremos la biblioteca SMPL para C, y para el segundo las bibliotecas JavaSim y SimPy para Java y Python respectivamente. En las siguientes secciones vamos a ver con detalle en qué consiste cada uno de estos paradigmas.

2.2. Programación orientada a sucesos

Para ilustrar cómo funciona un simulador orientado a sucesos vamos a estudiar un posible algoritmo para ejecutar un diagrama de sucesos. Tomaremos como ejemplo el diagrama de una estación FCFS (ver figura 2.1), cuyo funcionamiento estudiamos en el tema anterior.

Supondremos que cada suceso se describe con un fragmento de código que actúa sobre las variables de estado afectadas por el suceso. Por ejemplo, en el diagrama de sucesos de una estación FCFS, los sucesos tendrían asociado los siguientes fragmentos de código:

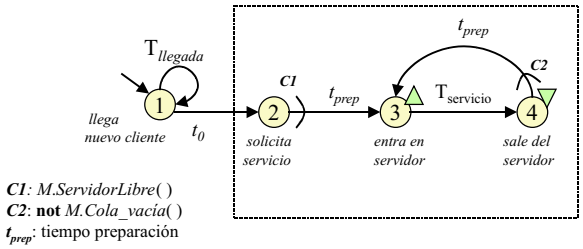


Figura 2.1: Diagrama de sucesos para una estación FCFS.

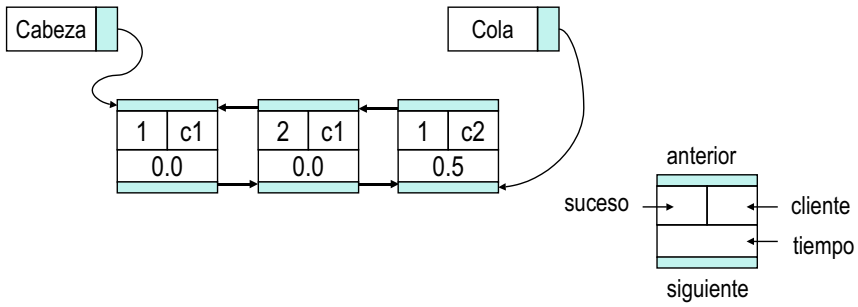


Figura 2.2: Planificador de sucesos basado en una lista doblemente enlazada.

Suceso	Código
①	cliente++;
②	if (! servidorLibre) encola(Q, cliente);
③	servidorLibre=0; desencola(Q, cliente);
④	procesados++; if (colaVacia(Q)) servidorLibre=1;

Para ir ejecutando el diagrama de sucesos, es necesario mantener una estructura de datos con aquellos sucesos que deben ejecutarse en un determinado tiempo (sucesos planificados). Naturalmente, los primeros sucesos que contendrá esta estructura serán los sucesos iniciales de la simulación.

La estructura más utilizada para guardar los sucesos planificados es una lista ordenada por el tiempo de ocurrencia. Así, en la cabeza de la lista siempre estará el siguiente suceso a ser ejecutado por el simulador (ver figura 2.2). Fíjate que junto a cada suceso se indica además el cliente involucrado.

Cuando la cantidad de sucesos planificados es muy grande, entonces debe utilizarse una estructura de datos que acelere la búsqueda e inserción de los sucesos planificados, generalmente una estructura en árbol. La más aceptada es el *montículo*, ya que en la raíz siempre se ubica el suceso con el tiempo de ejecución menor. Recuerda que un montículo es un árbol binario donde el contenido de cada nodo siempre debe ser menor que el de sus hijos.

Volviendo a la ejecución del diagrama de sucesos, un posible algoritmo podría ser el que se muestra en la figura 2.3.

El planificador de sucesos se utilizaría en las líneas 3, 5 y 8. Es decir, buena parte del algoritmo del simulador está formado por la gestión del planificador de sucesos, de ahí la importancia de seleccionar una buena estructura de datos para esta tarea.

Es importante comentar que la condición de fin de simulación depende de los objetivos de la simulación, y puede venir dada bien por el tiempo de simulación o bien por alguna variable de estado, como por ejemplo el número de clientes creados.

Al algoritmo básico anterior habría que añadirle las rutinas de recolección de resultados, así como la presentación de los mismos al finalizar la simulación. Pero esto ya lo analizaremos con más detalle en el tema de tratamiento de resultados.

De todo lo anterior, podemos concluir que un lenguaje de simulación orientado a sucesos debe proveer los siguientes elementos básicos:

1. Un *planificador* de sucesos, que proporcione y gestione la estructura de datos de los sucesos planificados.

2. Un *reloj* para controlar el tiempo de simulación.

```
1   Inicializa tiempo de simulación
2   Inicializa variables de estado
3   Inicializa planificador de sucesos
4   Mientras no FIN-SIMULACIÓN hacer
5       Selecciona el suceso S con menor tiempo de ejecución T
6       Avanza el tiempo de simulación a T
7       Ejecuta el código del suceso seleccionado S
8       Planifica/cancela todos los sucesos que son apuntados en el
9           diagrama de sucesos por S, y que cumplan las condiciones necesarias.
10  Imprime resultados
```

Figura 2.3: Algoritmo para la ejecución de un diagrama de sucesos.

3. Rutinas para la definición e inicialización de variables de estado.
4. Rutinas para la implementación de los sucesos.
5. Generadores para variables aleatorias.
6. Facilidades para el tratamiento de resultados.

Los puntos 2 y 3 se realizan generalmente con el lenguaje de programación que sirve de base al paquete o biblioteca de simulación que estemos utilizando. Por ejemplo, en SMPL las variables de estado son variables de C y los sucesos se implementan con código C.

2.3. SMPL

SMPL es una biblioteca de C que proporciona todas las funciones necesarias para la crear pequeños simuladores de sucesos discretos. Esta biblioteca proporciona principalmente los siguientes elementos:

- La gestión de los sucesos (*events*).
- La gestión de los recursos del sistema (*facilities*).
- La generación de informes.
- La generación de variables aleatorias.

Los tres primeros aspectos están recogidos en la biblioteca `smpl.c` y el último en `rand.c`. Cuando se compila un programa SMPL deben enlazarse ambos ficheros como sigue:

```
gcc -o ejecutable smpl.c rand.c programaSMPL.c -lm
```

Para mostrar las distintas características de este lenguaje vamos a introducir un pequeño ejemplo de SMPL que implementa una estación FCFS de tipo M/M/1, es decir, con un servidor, y con tiempos de llegada y de servicio que siguen una distribución exponencial.

```
#include "smpl.h"

main()
{
     smpl(0,"Simple");
```

```
    FCFS=facility("Estacion",1);
    Ta=1.0;
    Ts=0.8;
    schedule(1,0.0,1);

    while (time()<1000.0)
    {
       cause(&event,&cli);
       switch(event)
       {
          case 1:
             schedule(2,0.0,cli);
             cli++;
             schedule(1,expntl(Ta),cli);
             break;
          case 2:
             if (request(FCFS,cli,0)==0)
             schedule(4,expntl(Ts),cli);
             break;
          /* ojo! el suceso 3 no debe especificarse */
          case 4:
             release(FCFS,cli);
             break;
       }
    }
    report();
}
```

Todo programa en SMPL debe incluir la biblioteca "smpl.h". La estructura de un programa SMPL es siempre la misma:

1. Primero se inicializan las estructuras de datos internas de SMPL invocando al procedimiento:

   `void smpl(int` *Modo* `, char *`*Nombre*`)`

   El *Modo* puede ser interactivo (1) o no (0), y el *Nombre* es una cadena que indetifica el simulador, y que se utiliza en la generación de informes.

2. Después se inicializan todas las variables de estado que hayamos declarado, y se crean todos los recursos del modelo. Para ello utilizaremos la función:

   `int facility(char *`*Nombre* `, int` *NumeroServidores*`)`

   La cadena *Nombre* solo se utiliza para la generación de informes. Lo que a nosotros nos interesa es el identificador numérico que nos devuelva la función, ya que éste será el que utilicemos a lo largo de todo el programa. El argumento *NumeroServidores* permite indicar cuántos servidores tiene la estación de servicio.

3. A continuación se planifican los sucesos iniciales de la simulación. En nuestro ejemplo, el suceso inicial es el suceso ① que implementa la creación de los identificadores de nuevos clientes, (incrementando la variable `cli`). Para planificar un suceso, se utiliza el procedimiento siguiente:

   `void schedule(int` *suceso*`, double` *Tiempo*`, int` *cliente*`)`

   Fíjate que en SMPL los sucesos siempre van ligados a algún cliente, de modo que cuando se planifica un suceso hay que indicar tanto el identificador del suceso como

el del cliente. El argumento *Tiempo* indica el tiempo que debe transcurrir a partir del tiempo de simulación actual para ejecutar el suceso planificado. En otras palabras, corresponde exactamente con el tiempo que ponemos en los arcos de los diagramas de suceso. Ojo, no se trata de un tiempo absoluto, sino relativo.

4. El ciclo de simulación se realiza con un bucle, cuya condición de parada es la condición de fin de simulación. En el ejemplo, la condición de parada es que el tiempo de simulación actual sea igual o superior a 1000.0 unidades de tiempo. Para saber el tiempo actual de simulación utilizamos la función `double time()`.

5. La primera acción del bucle debe ser la selección del siguiente suceso a ejecutar (recuerda el algoritmo de la sección anterior). Esto se realiza con el procedimiento siguiente:

   `void cause(int *`*suceso*`, int *`*cliente*`)`

   En este caso ambos argumentos son de salida. Es decir, el procedimiento nos devuelve el identificador del suceso y el del cliente a ejecutar.

6. Una vez recogido el suceso a ejecutar, debemos seleccionar y ejecutar el código del mismo. Para ello, se utiliza la sentencia `switch`, donde cada caso contiene el código de cada suceso. Recuerda que dentro del código de cada suceso también debe incluirse la planificación/cancelación de los sucesos que están conectados con él en el diagrama de sucesos.

7. Por último, y ya fuera del bucle de simulación, se presentan los resultados. En este caso se ha utilizado el procedimiento `report()`, el cual muestra por pantalla las estadísticas de cada estacion de servicio.

Como habrás podido notar, en SMPL, los sucesos, los clientes y los recursos se identifican siempre con números enteros positivos.

Recursos en SMPL. Uno de los aspectos más destacables de SMPL es que se encarga completamente de la gestión de los recursos del modelo. Es por ello que el suceso ③, asi como todas las planificaciones por él implicadas, no deben ser incluidas en el programa SMPL. Así, el programador sólo debe indicar dónde se realiza la petición de servicio a un recurso (suceso ②) y dónde se libera (suceso ④).

Para la peticion de servicio a un recurso se utiliza la función:

`int request(int `*facility*`, int `*cliente*`, int `*prioridad*`)`

Si la función devuelve 0, significa que la estación está libre, y si devuelve 1, la estación está ocupada. Fíjate que en el caso de que esté libre, en el diagrama de sucesos planificábamos el suceso ③. Pero como este suceso modifica las variables de la estación de servicio, éste debe ser ejecutado internamente por SMPL, y por consiguiente lo omitiremos del programa. De este modo, cuando el servidor esté libre, desde el suceso ② planificamos directamente el suceso ④.

Fíjate también que en el suceso ②, el programa no hace nada si se encuentra la estación ocupada. En realidad, la función `request` además de devolver el estado de la estación, también encola al cliente si encuentra la estación ocupada. Por esta razón no debe programarse nada si la función devuelve un valor distinto de cero.

Otra forma de realizar una petición a una estación consiste en expulsar al cliente que se esté siriviendo en ese momento (petición con derecho de expulsión). Para ello utilizaremos la función `preempt`, en lugar de `request`.

Finalmente, para liberar una estación de servicio utilizaremos el procedimiento:

`void release(int `*facility*`, int `*cliente*`)`

Fíjate que para liberar una estación es necesario conocer el cliente que la está ocupando. Un error corriente en SMPL se produce cuando una estación se libera con el cliente no apropiado. Es por ello que debemos ser especialmente cuidadosos con la planificación de las peticiones y las liberaciones de las estaciones.

Otra función básica de SMPL es la cancelación de sucesos, que se realiza con la siguiente función:

```
int cancel(int suceso)
```

Esta devuelve el cliente al cual se le ha cancelado el suceso. A modo de ejemplo, vamos a mostrar cómo modificar el programa SMPL anterior para incorporar roturas. Concretamente añadimos los sucesos ⑤ y ⑥ (ver sección 1.3.1 del tema anterior) que se definen como sigue:

```
case 5:
   if (status(FCFS)==1){
     cli2=cancel(4);
     schedule(4,Trepara+Tserv,cli2);
   }
   else
     if (request(FCFS,ROTURA,0)
        schedule(6,Trepara,ROTURA);
   Troto=expontl(Troturas);
   while(Troto<Trepara) Troto=expontl(Troturas);
   schedule(5,Troto,ROTURA);
   break;
case 6:
   release(FCFS,ROTURA);
   break;
```

En este caso, debemos añadir al principio del programa las declaraciones de las variables $Trepara$, $Troto$, $cli2$ y $Tserv$. $ROTURA$ es el identificador único de cliente de las roturas, por ejemplo $TROTURA = 0$.

Estado de una estación de servicio. Como has podido comprobar, todo lo concerniente a las estaciones de servicio (variables de estado y sucesos) esta totalmente controlado por SMPL, entonces ¿cómo se puede consultar el estado de las estaciones de servicio?

SMPL también proporciona las funciones necesarias para consultar las variables de estado de una estación de servicio, éstas se resumen en la tabla siguiente.

`int inq(int f)`	Número de trabajos en cola
`int status(int f)`	Estado de la estación (1=ocupada)
`double U(int f)`	Nivel de utilización de la estación
`double B(int f)`	Duración media de ocupación
`double Lq(int f)`	Longitud media de la cola

A modo de ejercicio, puedes extender el programa en SMPL anterior para incluir los sucesos de roturas que vimos en el ejemplo 1.3.1 el tema anterior.

Variables aleatorias en SMPL. En el ejemplo de SMPL anterior hemos utilizado la función `expntl` para representar los tiempos de llegada entre clientes y sus tiempos de servicio. Esta función devuelve una secuencia de números aleatorios que sigue una distribución exponencial (es decir representa una variable aleatoria con distribución exponencial).

```
                    smpl SIMULATION REPORT

 MODEL: Tres colas y tres servidores                TIME:    10006.070
                                                 INTERVAL:   10006.070

                    MEAN BUSY      MEAN QUEUE          OPERATION COUNTS
 FACILITY        UTIL.   PERIOD       LENGTH    RELEASE  PREEMPT  QUEUE
 servidor       0.0306    1.001        0.001        306        0      8
 servidor       0.0476    0.934        0.002        510        0     20
 servidor       0.0182    1.032        0.000        177        0      8
```

Figura 2.4: Ejemplo de salida por pantalla de la función `report()`

Ojo, no debes confundir esta función con la función exponencial `exp(x)` la cual representa la función e^x.

En SMPL tenemos las siguientes funciones para generar valores para variables aleatorias:

`int stream()`	obtiene/selecciona un flujo de números aleatorios
`double ranf()`	genera una uniforme $U(0,1)$
`double uniform(a, b)`	genera una uniforme $U(a,b)$
`double expntl(m)`	genera una exponencial Exp con media m
`double erlang(m, s)`	genera una $Erlang$ con media m y desviación s
`double hyperx(m, s)`	genera una hiperexponencial
`double normal(x, s)`	genera una normal con media m y desviación s
`int random(i, j)`	genera un entero aleatorio en $[i,j]$ con $(i < j)$

Generador de Informes. Con el procedimiento `report()` se imprime pantalla los resultados generados internamente por SMPL. Un ejemplo de informe se muestra en la figura 2.4.

Las estadísticas que recoge SMPL para visualizar con el procedimiento `report()` son las siguientes:

`[UTIL.]`	Porcentaje total de utilización de los servidores (tanto por número de servidores).
`[MEAN BUSY PERIOD]`	Tiempo medio de ocupación de la estación (debe tender al tiempo medio de servicio).
`[MEAN QUEUE LENGTH]`	Tamaño medio de la cola de la estación de servicio.
`[RELEASE]`	Número de operaciones de liberación efectuadas (clientes procesados).
`[PREEMPT]`	Número de clientes expulsados.
`[QUEUE]`	Número de clientes que han tenido que encolarse.

Es importante resaltar el hecho de que estas estadísticas son insuficientes para sacar conclusiones del simulador. Como veremos en el tema de tratamiento de resultados, será necesario analizar la evolución de la varianza de las variables de estado y calcular los intervalos de confianza de los resultados obtenidos con el simulador. Aún así, la información que nos muestra `report()` puede ser útil para depurar el programa. Por ejemplo, podremos comprobar que los tiempos de servicio se aproximan a los que hemos modelado, y que las colas funcionan más o menos de acuerdo a la carga del sistema.

2.4. Programación orientada a procesos

Una de las principales limitaciones de un simulador orientado a sucesos es que los sucesos se ejecutan de forma secuencial, aunque ocurran en el mismo tiempo de simulación. Esto implica que el tiempo de ejecución sea directamente proporcional al número de sucesos ejecutados por el simulador, que suele ser bastante elevado. Además, el planificador de sucesos puede llegar a desbordar la memoria cuando los sucesos ocurren en tiempos muy próximos.

Como alternativa, se propone el concepto de *proceso*, que trata de paliar estas limitaciones. Con los procesos vamos a tratar de identificar las secuencias de sucesos *recurrentes* que pueden ejecutarse en paralelo. Para ello vamos a utilizar el diagrama de sucesos. Por ejemplo, en el diagrama de la figura 2.1 podemos identificar dos secuencias recurrentes: el suceso periódico de llegada de clientes (suceso ①), y la secuencia de sucesos ③ y ④, que representa el procesamiento de los clientes en la estación de servicio. Fíjate que se trata de dos secuencias repetitivas que pueden ejecutarse en paralelo.

De forma esquemática, podríamos expresar estos procesos como sigue:

Proceso 1	**Proceso 2**
Repite	Mientras c2
Ejecuta ①	Ejecuta ③
Espera Ta	Espera Ts
	Ejecuta ④

Según el diagrama de sucesos, Ta es el tiempo entre llegadas, Ts es el tiempo de servicio, y c2 es la condición de que la cola no esté vacía. El código de los sucesos de este ejemplo se describieron al principio del tema.

En los procesos hemos introducido la instrucción Espera para indicar que el proceso se detiene y planifica su reanudación según el tiempo especificado. En realidad, este será tiempo de simulación (según el reloj de la simulación), y no tiene relación directa con el de ejecución (tiempo de CPU). Más tarde veremos como se gestiona la ejecución de estos procesos.

De lo anterior, podemos deducir que un proceso puede estar en uno de los siguientes estados:

- *Activo*, cuando el proceso está ejecutándose.
- *Planificado*, cuando el proceso está detenido esperando su reanudación para un determinado tiempo.
- *Pasivo*, cuando no está ni planificado ni activo, aunque puede ser activado o planificado por otro proceso.
- *Terminado*, cuando el proceso ha finalizado totalmente su ejecución. Un proceso terminado ya no podrá ser nunca más activado ni planificado.

Volviendo al ejemplo anterior, todavía tenemos que ubicar el suceso ② en los procesos anteriores. Dado que la ejecución de este suceso planifica la ejecución del suceso ③, ubicado en el Proceso 2, éste sólo puede colocarse en el Proceso 1. La conexión entre ambos procesos se realiza con la instrucción Activa, la cual permite activar un proceso determinado desde otro proceso distinto. Con esta instrucción estamos implementando la planificación del Proceso 2 desde el Proceso 1. La implementación definitiva de estos dos procesos sería la siguiente:

Proceso 1	**Proceso 2**
1 Repite	1 Repite
2 Ejecuta ①	2 Mientras c2
3 Ejecuta ②	3 Ejecuta ③
4 Si c1	4 Espera Ts
5 Activa Proceso 2	5 Ejecuta ④
6 Espera Ta	6 Cancela

```
1   Inicializa tiempo de simulación
2   Inicializa variables de estado
3   Inicializa planificador de procesos
4   Mientras hayan procesos activos y/o planificados hacer
5      Si no hay proceso activo
6         Selecciona el siguiente proceso planificado con tiempo mínimo T
7         Avanza el tiempo de simulación a T
8         Ejecuta el proceso seleccionado (pasa a activo)
9   Muestra los resultados de la simulación
```

Figura 2.5: Algoritmo para la ejecución de procesos.

Normalmente, los procesos recurrentes se ejecutan de forma indefinida utilizando un bucle infinito. Fíjate que en el Proceso 2 hemos introducido además la instrucción Cancela, para indicar que este proceso pasa a un estado *pasivo* cuando no tenga clientes que procesar (la cola se ha quedado vacía). En este caso, cuando llegue un nuevo cliente en el Proceso 1 se activará de nuevo el Proceso 2, y pasará a procesar los nuevos clientes de la cola.

Es importante destacar que si el Proceso 2 no incluyese el bucle infinito, al volverse a despertar su hilo terminaría (pasaría al estado *terminado*), y no podría volverse a activar nunca más.

Como puedes apreciar, el diagrama de sucesos se ha simplificado a dos procesos: el proceso de llegadas y el proceso de servicios. Con esta nueva visión, conseguimos reducir el número de planificaciones, y además permitimos su ejecución en paralelo.

Para realizar la ejecución de un conjunto de procesos, tendremos que mantener al menos dos estructuras de datos:

- *Procesos Planificados*, que es una lista con los procesos que están esperando su reanudación en un tiempo determinado. Esta estructura es muy parecida a la de los sucesos planificados que vimos en la sección anterior.
- *Procesos Activos*, que contiene los procesos que están siendo ejecutados actualmente.

El algoritmo de ejecución de un simulador con procesos es muy parecido al que hemos visto para los simuladores orientados a sucesos. En la figura 2.5 se resume este algoritmo. El simulador tiene que gestionar ahora los procesos planificados, de tal forma que si en un momento dado no hay ningún proceso activo, se saca del planificador el proceso "dormido" que tenga menor tiempo de simulación, se avanza el reloj de simulación y se activa dicho proceso.

Veamos ahora parte de la ejecución de los procesos del ejemplo mediante la traza de la figura 2.6. Junto a cada proceso hemos indicado la línea de código que se está ejecutando (o planificando) en cada momento. Para esta traza se ha tomado un tiempo entre llegadas de $Ta = 0{,}5$ y un tiempo de servicio de $Ts = 0{,}6$. Se deja como ejercicio completar la traza hasta procesar cuatro clientes. En la traza hemos sombreado los procesos planificados por el proceso activo en esos momentos.

2.5. La biblioteca de JavaSim

La mayor parte de los simuladores y paquetes de software comerciales para la simulación de sucesos discretos se basa en procesos, por ejemplo GPSS, SIMSCRIPT II.5 y SLAM II. Puedes consultar la bibliografía si quieres conocer más detalles de estos lenguajes.

Planificador			Estación		
T_{sim}	*Activo*	*Planificados*	$\sharp Cliente$	*Servidor*	*Cola*
0	P1(1)	-	-	libre	[ ]
0	P1(2)	-	1	”	”
0	P1(3)	-	”		[1]
0	P1(4,5)	P2(1),0.0	”	”	”
0	P1(6)	P2(1),0.0	”	”	”
0		P1(1),0.5			
0	-	P2(1),0.0 $\surd$	”	”	”
0		P1(1),0.5			
0	P2(1,2)	P1(1),0.5	”	”	”
0	P2(3)	P1(1),0.5	”	ocupado	[ ]
0	P2(4)	P1(1),0.5	”	”	”
0		P2(5),0.6			
0	-	P1(1),0.5 $\surd$	”	”	”
0		P2(5),0.6			
0.5	P1(1)	P2(5),0.6	-	”	”
0.5	P1(2)	P2(5),0.6	2	ocupado	”
0.5	P1(3)	P2(5),0.6	”	”	[2]
0.5	P1(4)	P2(5),0.6	”	”	”
0.5	P1(6)	P2(5),0.6	”	”	”
		P1(1),1.0			
0.5	-	P2(5),0.6 $\surd$	-	”	”
		P1(1),1.0			
0.6	P2(5)	P1(1),1.0	1	”	”
0.6	P2(2,3)	P1(1),1.0	2	”	[ ]
0.6	P2(4)	P1(1),1.0	”	”	”
		P2(5),1.2			
...	...	...	...	...	...

Figura 2.6: Traza de ejecución basada en procesos.

Posiblemente, uno de los percusores del paradigma orientado a procesos ha sido SIMULA-67. Este ha sido uno de los primeros lenguajes orientados a objeto con nociones de concurrencia (o pseudo-paralelismo). Además fue percusor de otros lenguajes bien conocidos como Algol-68 y Pascal. El lenguaje que vamos a ver en este tema, JavaSim, está basado en las primitivas propuestas en el lenguaje SIMULA-67.

JavaSim tiene sus raíces en un proyecto de investigación de aplicaciones distribuidas en C++ (C++Sim) [1], y su principal característica es la definición de los procesos de simulación como *hilos* independientes de ejecución (o *threads*).

De forma resumida, un *hilo* consiste en un fragmento de código que se ejecuta de forma *simultánea* al programa (o hilo) que lanzó a ejecutar dicho fragmento. La palabra *simultánea* no debe tomarse al pie de la letra, ya que si el ordenador no tiene varios procesadores no podemos hablar de paralelismo, sino de pseudo-paralelismo. La programación por hilos ha tenido un fuerte auge a partir del desarrollo de entornos multimedia, ya que la detección de sucesos en entornos de usuario (pulsar ratón, animaciones, etc.) se realiza mediante la ejecución simultánea de varios hilos. También en sistemas distribuidos, especialmente los sistemas cliente/servidor, utilizan la programación multi-hilo en los servidores. Así, cada petición de un cliente se procesa en un hilo diferente.

En JavaSim un proceso de simulación es una clase especial de hilo, donde además debe registrarse un tiempo de simulación. A continuación vamos a describir los aspectos más relevantes de JavaSim, dejando que los detalles sean consultados en el manual de referencia.

2.5.1. La clase Thread

Un *hilo* es un objeto que tiene la propiedad de poder ejecutar código de forma concurrente al programa o hilo que lo invoca. El código a ejecutar se especifica dentro de un método especial del objeto; en Java es el método `run()` del interfaz Runnable, el cual es implementado por la clase Thread. Este método sólo se ejecuta cuando se activa el hilo, y se puede *congelar* o *cancelar* a sí mismo o desde otros hilos.

Bajo este enfoque, un programa es en sí mismo un hilo, denominado main, que puede congelarse y terminar como el resto de hilos. Sin embargo, la finalización del hilo main supone la finalización de todos los hilos derivados de su ejecución.

Originalmente, la clase Thread tenía en las versiones iniciales de Java los siguientes métodos para gestionar y sincronizar la ejecución de los hilos de un programa:

- `start()`: el cual inicia la ejecución de un hilo.
- `stop()`: el cual finaliza la ejecución de un hilo, generando una excepción por la interrupción.
- `suspend()`: que congela la ejecución de un hilo.
- `resume()`: que reanuda la ejecución de un hilo congelado.

Todos estos métodos han quedado obsoletos a partir de la versión 1.4 de Java debido a problemas de interbloqueos que se producían de forma aleatoria en la ejecución de programas multi-hilo [2].

El mecanismo alternativo propuesto para la sincronización de hilos consiste en la utilización de los métodos `wait()` y `notify()` de la clase Object, que es la clase de la cual heredan todas las clases de Java, incluida la clase Thread. El método `wait()` permite detener el hilo que se está ejecutando actualmente, mientras que `notify()` permite poner en marcha desde el hilo activo un hilo previamente detenido.

Veamos un ejemplo del uso de estos métodos. En primer lugar definiremos una clase para un hilo que simplemente espera un determinado tiempo y devuelve el control al hilo principal (main).

[1] Proyecto Arjuna:http://www.distribution.cs.ncl.ac.uk/history/

[2] http://java.sun.com/j2se/1.4.1/docs/guide/misc/threadPrimitiveDeprecation.html

```
import java.lang.Thread.*

public class Ciclo extends Thread
{
   public Object cerrojo= new Object();

   public void run() {
      try {
         sleep(2000); //Duerme 2000 ms
         synchronized(cerrojo)
         {
             cerrojo.notify(); //despierta el hilo main
         }
   }
}
```

La forma de activar este hilo desde el hilo principal se realiza como sigue:

```
public class Main
{
   public static void main() {
      Ciclo p= new Ciclo(); //Creamos el hilo
      p.start(); //Arrancamos el hilo
      synchronized(p.cerrojo)
      {
         try
         {
             cerrojo.wait(); //duerme el hilo main
         }
         catch (InterruptedException e)
         {
            System.exit(1); //ejecución interrumpida
         }
      }
      System.exit(0); //finaliza el programa
   }
}
```

2.5.2. La clase SimulationProcess

Los procesos de simulación que hemos definido en las secciones anteriores se implementan con la clase SimulationProcess, que es una subclase de Thread. Su interfaz se resume a continuación:

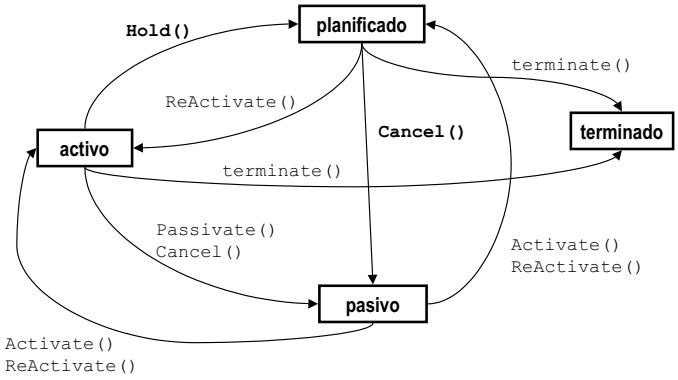


Figura 2.7: Transiciones entre estados de un proceso.

SimulationProcess
+ Object cerrojo
+ void run() + void Activate() + void ActivateBefore(SimulationProcess p) + void ActivateAfter(SimulationProcess p) + void ActivateAt(double AtTime,boolean prior) + void ReActivate() + void Cancel() + void terminate() + void finalize() + final double Time() + final double evtime() + static SimulationProcess current() + static double CurrentTime() + boolean idle() + boolean passivated() + boolean terminated() # void Hold(double t) # void SimulationProcess()

Debido a que el contructor de esta clase está protegido, siempre que queramos implementar un proceso, debermos crear para él una subclase de SimulationProcess. Una vez definida esta clase, el código del proceso se incluirá en su método `run()`.

Buena parte de los métodos anteriores sirven para planificar y coordinar distintos procesos. En la figura 2.7 relacionamos estos métodos con los estados de un proceso.

Otros métodos de interés de esta clase son:

- `CurrentTime()` y `Time()` que nos proporciona el tiempo actual de simulación. La diferencia entre ambos se debe a que el primero puede utilizarse sin instanciar la clase (static).
- `current()` y `next_ev()` devuelven respectivamente el proceso que se está ejecutando actualmente y el siguiente proceso planificado a ser ejecutado (será NULL si no hay planificados).
- `evtime()` devuelve el tiempo en el que el proceso ha planificado su reanudación. Si no está planificado devuelve la constante `Never`.

Ejemplos. A modo de ejemplo vamos a mostrar como se implementarían los dos procesos del ejemplo anterior. Al Proceso 1 que representa el proceso de llegadas le denominaremos Llegadas, y al Proceso 2 que representa el proceso de servicios, lo denominaremos Estacion. En la tabla siguiente se muestra por un lado los atributos de las clases (primera fila), y el método `run()` que implementa su funcionamiento. Recuerda que ambas clases deben ser subclases de SimulationProcess.

Llegadas.java	**Estacion.java**

`public Estacion M;`
`private double Ta=0.5;`
	`public Cola Q = new Cola();`
	`public boolean funciona=false;`
	`public int procesados;`
	`private double Ts=0.6;`

`public void run() {`
`  for(;;){`
`  //Suceso ①`
`  Cliente J = new Cliente();`
 	
`  //Suceso ②`
`  M.Q.Encola(J);`
`  vacia=M.Q.Vacia()`
`  if (Vacia && !M.funciona)`
`  M.Activate();`
`  Hold(Ta);`
`  }`
`}`
	`public void run() {`
	`  for(;;){`
	`  funciona=true;`
	`  while(!Q.Vacia()){`
	`  //Suceso ③`
	`  J=Q.Desencola();`
	`  Hold(Tserv);`
	`  //Suceso ④`
	`  procesados++;`
	`  }`
	`  funciona=false;`
	`  Cancel();`
	`  }`
	`}`

En el código anterior hemos omitido la captura de excepciones (sentencias `try`) para hacerlo más legible. Concretamente, los métodos `Hold` y `Activate` exigen la captura de la excepción `SimulationException`. Esta excepción se lanzará por ejemplo cuando se intente planificar un proceso con tiempos negativos o se intente activar un proceso terminado.

La variable `Estacion M` contiene la referencia al objeto de la clase Estacion que recibe los clientes del proceso de llegadas. Esta referencia puede pasarse en el constructor de Llegadas o bien utilizar una variable estática externa. Fíjate también que la variable `Q` es la variable que representa la cola de la estación.

2.5.3. La clase Scheduler

La clase Scheduler es la encargada de gestionar la ejecución de los procesos. En otras palabras, constituye la estructura de datos que maneja los procesos planificados y activos. Al igual que SimulationProcess, esta clase también constituye un hilo, es decir es una subclase de Thread.

El interfaz de esta clase es como sigue:

Scheduler
+ Scheduler()
+ static void stopSimulation ()
+ static void startSimulation()
+ static boolean simulationReset()
+ static void reset()
+ static double CurrentTime()

Esta clase es abstracta, con lo cual no debe crearse ningún objeto para ella. Para utilizarla procederemos como sigue. Antes de iniciar el ciclo de simulación activaremos el planificador

con `startSimulation()`, y al finalizar la simulación lo detendremos con `stopSimulation()`. En algunos casos, sobre todo para el tratamiento de los resultados, necesitaremos reiniciar varias veces la simulación. Para ello, se utiliza el método `reset()`, y para saber si el simulador se está reiniciando, se usa el método `simulationReset()`.

Por ejemplo, vamos a ver como se implementa la clase que representa el ciclo de simulación para una estación M/M/1.

```
public class Simulador extends SimulationProcess
{
     public void run()
     {
        try {
           //Creamos los procesos
           A = new Llegadas();
           M = new Estacion();
           //Activamos de proceso inicial
           A.Activate();
           //Comenzamos la simulación
           Scheduler.startSimulation();
           while (M.procesados<1000)
              Hold(1000);
           //Finalizamos la simulación
           Scheduler.stopSimulation();
           //Finalizamos los procesos
           A.terminate();
           B.terminate();

           //Despertamos al hilo principal
           synchronized(cerrojo)
              { this.cerrojo.notify(); }
        }
        catch(SimulationException e) {}
        catch(RestartException e) {}
     }
}
```

Como puedes comprobar, el propio simulador es también un proceso, el cual se encarga de arrancar todos los procesos del modelo y de controlar el final de la simulación. Al principio sólo es necesario arrancar los procesos iniciales (que son los que contienen los sucesos iniciales del diagrama de sucesos); en el ejemplo éste es el proceso de llegadas `A`. Una vez iniciada la simulación, el proceso del simulador se dedica únicamente a vigilar periódicamente que la condición de finalización de la simulación se cumple (`M.procesados>=1000`).

Una vez terminada la simulación, el simulador debe notificarlo al planificador (`Scheduler.stopSimulation(`
y terminar todos los procesos que se pusieron en marcha. Finalmente, se retorna el control al hilo principal tal y como indicamos en la sección anterior.

2.5.4. Generación de números aleatorios

Todos los generadores de números aleatorios en JavaSim son subclases de la clase RandomStream, la cual representa la distribución uniforme $U(0,1)$. Esta clase implementa un generador congruencial multiplicativo con $a = 5^5$ y $p = 2^{26}$. En el tema siguiente veremos con detalle cómo se generan números aleatorios con estos parámetros.

El interfaz de la clase es el siguiente:

RandomStream
+ RandomStream()
+ RandomStream(long MGSeed, long LCGSeed)
+ double getNumber()
+ double Error()
# double Uniform();

Lo más destacable de esta clase es el método `getNumber()` que se utiliza para obtener uno a uno los valores de la secuencia de números aleatorios.

Para nuestros simuladores utilizaremos las siguientes clases (constructures) derivadas de RandomStream:

- UniformStream(double a, double b, int StreamSelect), que representa una variable con distribución $U(a, b)$
- ExponentialStream(double Media, int StreamSelect), que representa una distribución exponencial con media $Media$.
- ErlangStream(double Media, double SD, int StreamSelect), que representa una distribución $Erlang$ con media $Media$ y desviación estándar SD.
- HyperExponentialStream(double Media, int StreamSelect), ídem para una distribución exponencial.
- NormalStream(double Media, int StreamSelect), ídem para una distribución normal.

El último argumento de todos estos constructores representa el número de la secuencia aleatoria seleccionada, es decir, la semilla (o primer número de la serie) que utiliza en el generador para crear toda la serie. De este modo, con distintos valores del selector obtenemos secuencias diferentes de números aleatorios que siguen la distribución correspondiente. Este argumento será de especial utilidad en la fase de experimentación, donde necesitaremos realizar varias simulaciones con diferentes secuencias.

Cada vez que necesitemos una variable aleatoria en nuestro programa, crearemos un objeto con la clase que represente su distribución. Después generaremos valores utilizando el método `getNumber()` sobre el objeto creado.

Ejemplo. En el proceso Llegadas del ejemplo anterior podemos incluir un generador de números aleatorios que sigan una distribución exponencial para simular el tiempo entre llegadas de los clientes. Para ello, creamos un objeto de la clase ExponentialStream en el constructor de la clase Llegadas y la usaremos en el método `run()` para la obtención de valores para los tiempos entre llegadas.

```
public class Llegadas extends SimulationProcess
{
     public ExponentialStream Ta = null;

     public Llegadas(double T)
     {
        Ta = new ExponentialStream(T,1));
     }

     public void run()
     {
```

```
        for(;;){
           Cliente J = new Cliente();
           //Suceso ②
           M.Q.Encola(J);
           vacia=M.Q.Vacia()
           if (Vacia && !M.funciona)
              M.Activate();
           Hold( Ta.getNumber() );
     }
}
```

De nuevo hemos omitido por claridad las sentencias `try-catch` para capturar las excepciones producidas por `Hold`, `Activate` y `getNumber`.

2.5.5. Recogiendo resultados

JavaSim también proporciona un juego de clases para recoger resultados de la simulación y realizar estadísticas con ellos. La clase básica es Mean, cuyo interfaz es el siguiente:

Mean
+ Mean()
+ void setValue(double)
+ int numberOfSamples () const
+ double min()
+ double max()
+ double sum()
+ double mean()
+ void print()
+ void *reset()*

Para recoger las distintas observaciones del experimento utilizaremos el método `setValue()`. Una vez recogidos todos los datos, se pueden consultar algunos de sus estimadores estadísticos: número de muestras, mínimo, máximo, suma y media (`mean()`).

Existen otras clases derivadas de Mean que nos permiten extraer otros resultados de caracter estadístico, destacaremos las siguientes:

- Variance, que proporciona los métodos `variance()`, `stdDev()` y `confidence(double)` para determinar respectivamente la varianza, desviación típica e intervalo de confianza de los datos recogidos.
- TimeVariance es subclase de `Variance` y proporciona el método `timeAverage()` para calcular el promedio temporal de los resultados. Es decir, tiene en cuenta el tiempo en el que se ha recogido cada dato para promediar con el tiempo de simulación.

Para más información sobre estas clases y otras que no se han visto en estos apuentes, se recomienda consultar el manual de referencia de JavaSim.

Ejemplo. A modo de ejemplo vamos a modificar el proceso Estacion para tomar datos estadísticos de la cola de espera. Para ello creamos la variable `enCola` que representará la media temporal de los clientes encolados. Esta variable se actualiza en el método `run()`:

```
public class Estacion extends SimulationProcess
{
```

```
    public TimeVariance enCola=new TimeVariance();

    public void run()
    {
       for(;;)
       {
          funciona=true;
          while(!Q.Vacia())
          {
             //Suceso ③
             J=Q.Desencola();
             enCola.setValue(Q.Tamanyo());
             Hold(Ts);
             //Suceso ④
             procesados++;
          }
          funciona=false;
          Cancel();
       }
    }
}
```

Además, habría que realizar también las observaciones cuando se encola cada cliente en el proceso de llegadas:

```
public class Llegadas extends SimulationProcess
{
    public ExponentialStream Ta = null;

    public Llegadas(double T)
    {
       Ta = new ExponentialStream(T,1));
    }

    public void run()
    {
       for(;;){
          Cliente J = new Cliente();
          //Suceso ②
          M.Q.Encola(J);
          enCola.setValue(M.Q.Tamanyo());
          vacia=M.Q.Vacia()
          if (Vacia && !M.funciona)
             M.Activate();
          Hold(Ta.getNumber());
    }
}
```

También podemos estimar el tiempo medio de respuesta del sistema recogiendo los tiempos de entrada y salida de cada cliente. Para ello definimos una variable estática dentro de la clase que controla la simulación (Simulador), a la cual denominamos `TRespuesta`. Los

tiempos de entrada y de salida se toman en el contructor de la clase Cliente y en el método `finished()`, tal y como se muestra a continuación:

```
public class Cliente
{
     public Cliente()
     {
        Tllegada = Scheduler.CurrentTime();
     }

     public void finished()
     {
        Tsalida = Scheduler.CurrentTime();
        TRespuesta.setValue(Tsalida - Tllegada);
     }
     private double Tllegada;
     private double Tsalida;
}
```

Todos estos datos estadísticos se mostrarían en al finalizar el ciclo de simulación, en el cuerpo de la clase Simulador, tal y como se muestra a continuación:

```
public class Simulador extends SimulationProcess
{
     public void run()
     {
        try {
           //Creamos los procesos
           A = new Llegadas();
           M = new Estacion();
           //Activamos de proceso inicial
           A.Activate();
           //Comenzamos la simulación
           Scheduler.startSimulation();
           while (M.procesados<1000)
              Hold(1000);

           System.out.println(``Estadísticas:'');
           System.out.println(``Tiempo medio de respuesta:''+TRespuesta.mean() );
           System.out.println(``Tamaño medio de la cola:''+M.EnCola.mean() );

           //Finalizamos la simulación
           Scheduler.stopSimulation();
           //Finalizamos los procesos
           A.terminate();
           B.terminate();

           //Despertamos al hilo principal
           synchronized(cerrojo)
              { this.cerrojo.notify(); }
        }
        catch(SimulationException e) {}
```

```
        catch(RestartException e) {}
    }
    public static Mean TRespuesta= new Mean() ;
}
```

Es importante destacar que los resultados deben mostrarse antes de terminar la simulación con `stopSimulation()`, ya que de lo contrario los datos mostrados podrían no ser correctos.

La implementación completa de este ejemplo en JavaSim la puedes encontrar en la página Web de la asignatura.

2.6. SimPy

En las secciones anteriores nos hemos centrado en dos paquetes para crear simuladores basados en sucesos (SMPL) y procesos (JavaSim). Como se mencionó anteriormente, existen otros paquetes basados en otros lenguajes, como C++SIM en C++, Simula en Pascal, etc.

En esta sección vamos a describir las pricipales características del paquete de simulación SimPy para Python[3]. Como veremos, este paquete combina elementos de simulación de los lenguajes SMPL y JavaScript. Concretamente, de SMPL toma la gestión de recursos mediante las primitivas request y release, y de JavaSim las primitivas de gestión de procesos.

El bucle de simulación. Un simulador en SimPy debe incluir simpre las siguientes sentencias:

```
from SimPy.Simulation import *

initialize()
simulate(until=10000.0)
```

La primera línea importa la biblioteca de SimPy. La segunda línea inicializa todo el entorno de simulación (reloj, planificador de procesos, etc.) Finalmente, la sentencia `simulate` simula 10000,0 unidades de tiempo. En este caso, la simulación acaba inmediatamente ya que no hay sucesos que ejecutar.

Observa que la simulación siempre se controla con el tiempo de simulación máximo. Si se quiere controlar la simulación con cualquier otra condición, por ejemplo el número de clientes procesados, será necesario consultar periódicamente dicha condición y terminar la simulación cuando ésta se cumpla. Para ello, SimPy proporciona la sentencia stopSimulation() que detiene la simulación desde cualquier parte del programa. También existen otros mecanismos más sofisticados para controlar la simulación, que pueden consultarse en el manual de SimPy.

Procesos. Si queremos incluir sucesos, debemos crear los procesos correspondientes, al igual que en JavaSim. Para ello, crearemos subclases de la clase Process, En esta clase definiremos un método de ejecución que, al contrario que en JavaSim, puede tomar cualquier nombre y puede tener argumentos de entrada.

Por ejemplo, el siguiente proceso simplemente espera 100,0 unidades de tiempo, y cambia la variable de estado fin a True.

[3] http://simpy.sourceforge.net

```
from SimPy.Simulation import *

class Espera(Process):
        def __init__(self):
           Process.__init__(self)
        def run(self):
           yield hold, self, 100.0
           fin=True

initialize()
fin=False
p=Espera()
activate(p,p.run())
simulate(until=10000.0)
```

Para que un proceso funcione como tal, es necesario incluir siempre su constructor (`__init__`), y dentro de él, invocar al contructor de la superclase Process. En el constructor también podrían incluirse las variables de estado locales al proceso. Para activar por primera vez un proceso utilizaremos la sentencia activate, donde indicaremos cuál es el proceso concreto a activar, y el método que debe ejecutarse. Opcionalmente podría indicarse también el tiempo de simulación en el que debe activarse (parámetro at), el cual por defecto es cero.

Si por algún motivo un proceso p se duerme a sí mismo, utilizando yield passivate,self o es cancelado desde otro proceso, utilizando self.cancel(p), debe ser activado de nuevo con la sentencia reactivate(p). De este modo, activate solo debe ser utilizado la primera vez que activamos el proceso.

El siguiente programa es una traducción a SimPy de la implementación en JavaSim de una estación M/M/1. En él se puede ver el funcionamiento de los procesos en SimPy.

```
from SimPy.Simulation import *
from random import Random
class Cliente:
        def __init__(self,id):
           self.id=id
           self.Tinicio=0
        def finaliza(self):
           global TR
           TR+=(now()-self.Tinicio) ###acumula tiempo respuesta

class Estacion(Process):
        def __init__(self,Tservicio):
           Process.__init__(self)
           self.Q=[]
           self.funciona=False
           self.Tservicio=Tservicio
           self.Procesados=0

        def run(self):
           global TR
           while True:
              self.funciona=True
              while self.Q!=[]:
                 J = self.Q[0]
```

```
                del self.Q[0]
                yield hold,self,self.Tservicio
                self.Procesados+=1
                J.finaliza()
                del J
            self.funciona=False
            yield passivate,self

class Llegadas(Process):
        def __init__(self,id,Tmedio):
            Process.__init__(self)
            self.id=id
            self.cuenta=0
            self.Tmedio=Tmedio
            self.g=Random(333555)
        def run(self):
            global MAQ
            while True:
                self.cuenta+=1
                J = Cliente(self.cuenta)
                J.Tinicio=now()
                MAQ.Q.append(J)
                if not MAQ.funciona:
                    reactivate(MAQ)
                yield hold,self,self.g.expovariate(1.0/self.Tmedio)

initialize()
TR=0.0
LL=Llegadas("llegadas", 1.0)
MAQ=Estacion(1,1.0)
activate(LL,LL.run())
activate(MAQ,MAQ.run())
simulate(until=2000.0)

print "Procesados:", MAQ.Procesados
print "Tiempo medio de respuesta:", TR/MAQ.Procesados
```

Recursos. De forma similar a SMPL, SimPy también permite la gestión interna de los recursos. Para ello, proporciona la clase Resource. Las instancias de esta clase representan estaciones de servicio con uno o más servidores, donde la cola de espera puede gestionarse con una disciplina FIFO (por defecto), o mediante prioridades.

Como en SMPL, la gestión de los recursos se realiza mediante las setencias request y release, aunque ahora no será necesario indicar explicitamente el cliente que realiza esas operaciones.

Dado que entre ambas operaciones debe transcurrir el tiempo de servicio, será necesario definir un proceso para incluirlos. Normalmente, denominaremos a dicho proceso Cliente, ya que representará los sucesos que va realizando un cliente en el sistema simulado. A modo de ejemplo, implementaremos la estación M/M/1 utilizando los recursos de SimPy.

```
from SimPy.Simulation import *
from random import Random
```

```
class Cliente(Process):
        def __init__(self,id):
           Process.__init__(self)
           self.id=id
           self.Tinicio=0
        def run(self,MAQ,Tservicio):
           global TR,Procesados
           self.Tinicio=now()
           yield request,self,MAQ
           yield hold,self,Tservicio
           yield release,self,MAQ
           Procesados+=1
           TR+=(now()-self.Tinicio) ###acumula tiempo respuesta

class Llegadas(Process):
        def __init__(self,id,Tmedio):
           Process.__init__(self)
           self.id=id
           self.cuenta=0
           self.Tmedio=Tmedio
           self.g=Random(333555)
        def run(self):
           global MAQ
           while True:
              self.cuenta+=1
              J = Cliente(self.cuenta)
              activate(J,J.run())
              yield hold,self,self.g.expovariate(1.0/self.Tmedio)

initialize()
TR=0.0
Procesados=0
LL=Llegadas("llegadas", 1.0)
MAQ=Resource(capacity=1)
activate(LL,LL.run())
simulate(until=2000.0)

print "Procesados:", Procesados
print "Tiempo medio de respuesta:", TR/Procesados
```

Otras utilidades. SimPy utiliza la biblioteca random de Python para la generación de números y variables aleatorias. Explicaremos éstas con más detalle en el siguiente tema.

Por otro lado, al igual que JavaSim, SimPy proporciona herramientas para recoger resultados, denominados monitores. Éstos se crean instanciando la clase Monitor. Para registrar las observaciones se utiliza el método m.observe(dato), y para inicializar el monitor m.reset(). Durante o al final de la simulación, se pueden obtener estadísticas a partir de estos monitores, concretamente: la suma total de las observaciones (m.total()), el número de observaciones (m.count()), la media de las observaciones (m.mean()), la varianza (m.var()), la media temporal (m.timeAverage()) y un histograma de las observaciones (m.histogram(low,high,nbins)).

Para más información sobre éstas y otras funcionalidades de SimPy podéis consultar su manual en línea.

2.7. Bibliografía

A. M. Law, W. D. Kelton "Simulation Modeling & Analysis". Ed. McGraw-Hill (1984).

R. Jain "The Art of Computer Systems Performance Analysis". Ed. Wiley (1991).

MacDougall "Simulating Computer Systems: Techniques and Tools". MIT Press (1987).

G. Gordon "System Simulation". Editorial Prentice-Hall (1978).

M. C. Little y D. L. McCue "JavaSIM User's Guide", Libre Distribución.

G. L. Heileman "Estructuras de datos, algoritmos y Programación Orientada a Objetos". McGrawHill (1998).

J. García Jalón "Aprenda Java como si estuviera en primero". Escuela Superior de Ingenieros Industriales. Universidad de Navarra (2001).

T. Vignaux y K. Muller "SimPy Manual". http://simpy.sourceforge.net/SimPyDocs/Manual.html. Última visita: 21/10/2005.

Capítulo 3

Generación de variables aleatorias

3.1. Introducción

Un elemento clave para el desarrollo de un simulador es la generación de valores aleatorios para variables con una determinada distribución de probabilidad (*variables aleatorias*). La introducción de variables aleatorias en los modelos de simulación permiten recrear situaciones que no están completamente controladas, o que dependen de algún factor aleatorio. Por ejemplo, como vimos en temas anteriores, el tiempo entre llegadas de clientes en una estación de servicio suele modelarse como una distribución exponencial.

Un generador de valores para una variable aleatoria se consigue en dos pasos. Primero tenemos que definir un generador de números aleatorios distribuidos uniformemente entre 0 y 1. Después, transformaremos esta secuencia para generar los valores de la variable aleatoria deseada.

Un requerimiento importante que deben cumplir los generadores de números aleatorios es que las series generadas sean *reproducibles*, de tal modo que en distintos experimientos podamos utilizar exactamente la misma secuencia de números aleatorios para reproducir las mismas condiciones.

En la actualidad, prácticamente todos los lenguajes de programación proveen alguna función para generar números aleatorios. Aunque podríamos pensar que estos son suficientes para realizar nuestras simulaciones, es necesario que nos aseguremos de la buena calidad de estos generadores. El empleo de un generador que produzca series con fuertes dependencias o correlacciones puede conducir a simulaciones incorrectas y a conclusiones totalmente falsas.

3.2. Generadores de números aleatorios

Un generador de números aleatorios consiste en una función que devuelve los valores de una secuencia de números reales, $(u_1, u_2, \ldots, u_n)$, donde cada $u_i \in [0, 1]$. El primer elemento de la secuencia se le denomina *semilla inicial* de la serie, y a partir de ella debe ser posible generar el resto de la secuencia para que ésta sea *reproducible*.

Entre las propiedades que debe cumplir un buen generador para realizar simulaciones, destacaríamos las siguientes:

- Los valores u_i deben ser independientes y estar idénticamente distribuidos (IID).
- La secuencia generada debe ser bastante larga, con el fin de permitir simulaciones largas y/o con muchas variables aleatorias.

- Debe computarse muy eficientemente, ya que cada simulación podría requerir la generación de una cantidad considerable de números aleatorios, del orden de millones.

La condición de uniformidad debe cumplirse además para todas las subsecuencias de tamaño k de la secuencia generada. Es decir:

- los u_i deben distribuirse uniformemente en $[0, 1]$
- los pares (u_i, u_{i+1}) deben distribuirse uniformemente en el plano $[0, 1] \times [0, 1]$
- los tríos (u_i, u_{i+1}, u_{i+2}) deben distribuirse uniformemente en el cubo $[0, 1]^3$
- etc

Asegurar la k-uniformidad de las secuencias es crucial para los experimentos de simulación. Con ella se evitan correlaciones entre distintas variables que compartan una misma secuencia. Por ejemplo, si en un simulador utilizamos la misma secuencia de números aleatorios para generar los tiempos entre llegadas ($T_{llegadas}$) y los tiempos de servicio ($T_{servicio}$) debemos asegurarnos que los pares consecutivos de valores son independientes, es decir cumplen el criterio de 2-uniformidad. Si esto no se cumple, estaremos introduciendo una correlación no deseada entre ambas variables. Fíjate que el valor máximo de k dependerá del modelo de simulación.

En la siguiente sección vamos a describir los generadores de números aleatorios basados en congruencias, los cuales son los más utilizados en simulación.

3.2.1. Generadores Congruenciales Lineales (GCL)

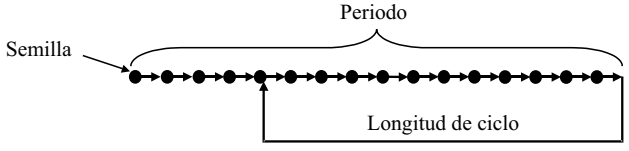
Los generadores congruenciales fueron descubiertos por Lehmer en 1951 cuando observó que los residuos de las potencias sucesivas de un número sugieren un comportamiento aleatorio. La primera propuesta de Lehmer consistió en la siguiente secuencia:

$$x_n = a^n \; mod \; m$$

Es decir, el número n-ésimo de la secuencia se obtiene dividiendo la potencia n-ésima de un entero a por un entero m y tomando el resto. La expresión anterior es equivalente a la siguiente:

$$x_n = a \, x_{n-1} \; mod \; m$$

De este modo se puede obtener cada elemento de la secuencia a partir del elemento anterior, y en primer lugar de un valor inicial x_0 denominado *semilla*.



Muchos de los generadores propuestos actualmente son una generalización del propuesto por Lehmer, donde se incluye un sesgo b en la expresión anterior:

$$x_n = (a \, x_{n-1} + b) \; mod \; m$$

La secuencia así obtenida consiste en una serie de números enteros comprendidos entre 0 y $m - 1$. Las parámetros del generador a, b y m deben de ser números enteros positivos. Por otro lado, cuando $b = 0$ diremos que el generador es *multiplicativo*.

Para obtener una secuencia de números entre 0 y 1 a partir de un GCL, bastará con dividir cada x_n por el módulo m de la serie:

$$u_n = x_n/m$$

A pesar de su simplicidad, la selección adecuada de las constantes (a, b, m) permitirá obtener secuencias largas y aleatorias de un modo muy eficiente.

3.2.2. Selección de parámetros

Como se muestra en la figura anterior, los GLCs producen secuencias cíclicas. Una de las propiedades que nos interesa en Simulación es que el periodo de repetición de las secuencias sea lo más grande posible. Dicho periodo vendrá determinado por los parámetros del generador. Por ejemplo, se ha demostrado que el máximo periodo se alcanza cuando se dan las siguientes condiciones:

- $b \neq 0$
- $mcd(b, m) = 1$ (primos relativos)
- $a = 1 \; mod \; p$ para cada factor primo p de m
- $a = 1 \; mod \; 4$ si 4 divide a m

Además de la longitud de los ciclos, en simulación también nos interesa que el generador sea muy eficiente. Dado que los GLCs multiplicativos ($b = 0$) son los más eficientes de computar, a partir de ahora sólo estudiaremos este tipo de generadores.

Es fácil ver que la operación más costosa en un GLC es la operación del resto (m suele ser un número muy grande). Si tomamos $m = 2^\beta$ la operación de resto resulta obvia, ya que basta retener los β últimos bits del producto $a \cdot x_i$. Sin embargo, la longitud máxima de este tipo de generadores es $m/4$, y se da cuando la semilla es impar y la constante a tiene la forma $8 \cdot i \pm 3$.

La longitud del periodo de un GLC multiplicativo puede llegar a ser $m - 1$ cuando m es un número primo (en este caso los valores de la serie estarán entre 1 y $m - 1$, nunca valdrán cero). Además, el valor de a tiene que ser cuidadosamente seleccionado. A este respecto, se ha demostrado que si a es una raíz primitiva de m, entonces se alcanza el periodo máximo $m - 1$. Esto ocurre cuando $a^n \; mod \; m \neq 1$ para $n = 1, 2, \ldots, m - 2$.

Un ejemplo de GLC multiplicativo con periodo máximo $m - 1$ es el siguiente:

- $a = 7^5 = 16807$
- $b = 0$
- $m = 2^{31} - 1 = 2147483647$

Este ejemplo, de hecho, es el que se ha tomado como estándar para la generación de números aleatorios, debido principalmente a la calidad de las series obtenidas.

Un último factor que influye en la elección de los parámetros de un GLC es la calidad de la serie generada. Como vimos en la introducción, la calidad de una serie se mide por la uniformidad e independencia de los valores de la serie.

A este respecto, una propiedad inherente de los generadores GLC es que producen una estructura reticular cuando se toman subsecuencias de la serie generada. Un ejemplo típico de estructura reticular se muestra en la figura 3.1, donde se consideran los pares de números consecutivos de la serie (u_i, u_{i+1}).

En esta estructura reticular pueden identificarse una serie de líneas (unas con pendientes positivas y otras con pendientes negativas) donde se sitúan todos los pares de la serie. Dependiendo de la distancia entre estas líneas, los pares se distribuyen más o menos uniformemente en el plano. Generalmente, cuanto mayor sea la distancia, peor será el generador.

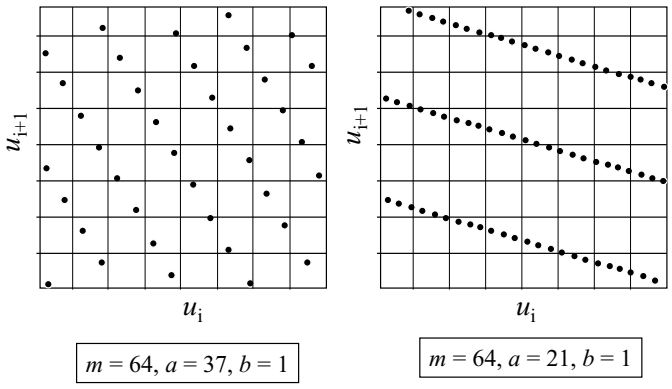


Figura 3.1: Estructura reticular de un GLC

Esta distancia viene determinada por el parámetro a. Así pues, el valor de a va a determinar la uniformidad de los valores, y su selección es crítica para la calidad de la serie final.

A modo de ejemplo, un primer generador GLC de números aleatorios que fue propuesto por IBM en 1960 tenía los siguiente parámetros: $a = 65539$, $b = 0$ y $m = 2^{31}$. Con este generador todas las tripletas de números consecutivos de las series $(5 \cdot 10^{10})$ caen en tan solo 15 planos.

Marsaglia demostró en 1968 que el número máximo de hiperplanos paralelos que puede producir un GLC multiplicativo es $(n!m)^{1/n}$, donde n es la cantidad de números consecutivos de las subsecuencias consideradas. Observa que el número de hiperplanos cae rápidamente conforme aumenta la dimensión del espacio n.

3.2.3. Otras consideraciones

La implementación de un GLC debe realizarse con sumo cuidado, ya que los resultados teóricos obtenidos con unos parámetros determinados podrían no ser ciertos si la implementación no realiza los cálculos exactos. En la implementación debe prestarse especial atención a los redondeos y a los desbordamientos en determinadas operaciones.

En el generador GLC anterior, puede verse que el producto $a \cdot x_i$ puede llegar a $1{,}03 \cdot 2^{45}$, lo cual producirá un desbordamiento, a no ser que se use un tipo entero de al menos 46 bits.

Para estos casos, debemos reformular el cálculo del GLC para evitar el desbordamiento. Una posible solución la propone Schrage (ver [Raj Jain, 1991]), y se resume en el siguiente fragmento de código escrito en Python:

```
int random(x):
  a = 16807
  m = 2147483647
  q = 127773      #m div a
  r = 2836        #m mod a

  x_div_q = x / q
  x_mod_q = x % q
  x_new = a*x_mod_q - r*x_div_q
  if x_new>0:
     x = x_new
  else:
     x = x_new + m
  return x
```

Para comprobar que el generador produce el resultado esperado, el valor para x_{10000} debe ser 1043618065 con $x_0 = 1$.

3.3. Test empíricos

Hemos visto en la sección anterior que los generadores de números aleatorios pueden tener más o menos calidad dependiendo de los parámetros seleccionados. Recordemos que la calidad se mide en función de la independencia y uniformidad de los valores de las secuencias generadas.

Para comprobar la calidad de un generador se han propuesto varios tests empíricos. Si un generador no pasa alguno de estos tests deberemos descartarlo ya que no cumple algún requisito mínimo para asegurar que los valores son IID. Sin embargo, si el generador pasa todos los tests, no podemos asegurar que el generador está completamente libre de sesgos o dependencias. Podría pasar que un nuevo test descubra alguna deficiencia en dicho generador y lo descarte definitivamente. También puede ocurrir que algunas semillas produzcan secuencias que pasen los tests, pero otras no, con lo que habría que descartar el generador.

Los test empíricos se realizan de una forma similar:

1. Primero se realizan una serie de *observaciones* sobre la serie a estudiar.
2. Se define una *estimación* a partir de la distribución que se supone deben seguir las observaciones.
3. Se evalúa la *proximidad* entre las observaciones y la estimación.

Para la última tarea, debido a que trabajamos con números aleatorios se utilizará una distribución conocida para tener en cuenta las variaciones de las medidas calculadas.

A continuación vamos a describir los test empíricos más utilizados para comprobar la bondad de los generadores.

3.3.1. Test χ^2

Este test es el más utilizado para comprobar si una lista de números satisface una determinada distribución. Este test se basa en una partición del espacio muestral en k categorías disjuntas, de modo que cada observación o_i consiste en contar cuántos números de la serie caen en la categoría i-ésima. La estimación de este test se calcula a partir de la función de densidad hipotética. Con esta función se calcula el número de valores estimados e_i que caen en cada categoría.

Para comparar las observaciones con la estimación se utiliza el error cuadrático medio:

$$X = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i}$$

Debido a la aletoriedad de las observaciones, la medida X no va a ser nunca cero, sino que satisface una distribución aleatoria, precisamente denominada chi-cuadrado (χ^2), con $k-1$ grados de libertad. Así, dependiendo del grado de confianza que queramos en el test (α) utilizaremos el valor de la distribución $\chi^2_{[1-\alpha;k-1]}$.

En nuestro caso, si queremos comprobar si un generador de números aleatorios es bueno, compararemos la serie generada de longitud n con la distribución uniforme $U(0,1)$:

- Particionamos el espacio muestral $[0,1]$ en k intervalos disjuntos.
- Calculamos cuántos números de la serie caen en cada intervalo (o_i).
- La estimación es que n/k números caigan en cada intervalo, es decir, $e_i = n/k$.

- A partir de la ecuación anterior, el valor de X se calcularía como sigue:

$$X = \frac{k}{n}\sum_{i=1}^{k}(o_i - \frac{n}{k})^2$$

- Finalmente, si el valor de X es mayor que $\chi^2_{[1-\alpha;k-1]}$, entoces descartaremos el generador.

Una limitación importante de este test es que si los valores estimados e_i no son los mismos para las distintas categorías (cosa que no ocurre para la distribución uniforme), entonces el error será mayor en las categorías de tamaño menor. Para evitar este efecto, tendríamos que dividir el espacio muestral en categorías equiprobables, las cuales podrían tener tamaños diferentes. Esto implica tomar una decisión sobre el número y forma de las categorías o celdas, lo que a veces no resulta fácil. Por otro lado, para que el test sea fiable, cada categoría debe tener al menos 5 elementos.

En general, el test de chi-cuadrado se ha diseñado para muchas muestras (series muy largas) que sigan una distribución aleatoria discreta.

A modo de ejemplo, a continuación mostramos el resultado de este test aplicado a tres secuencias de 1000 números. La primera es simplemente una serie de números consecutivos, la segunda es una serie con los tiempos de respuesta de una estación M/M/2, y la tercera es una serie de números obtenidos con la función `random()` de Python. Las dos primeras series no pueden ser consideradas aleatorias ya que la primera tiene una fuerte correlacióon entre pares consecutivos de valores, y en la segunda el tiempo de respuesta de un cliente depende de los tiempos de respuesta de los clientes anteriores, es decir también existe una fuerte correlación entre los valores (aunque no tan evidente).

Secuencia	**Resultado**
Números consecutivos	0.02
M/M/2	1629.86
Random	8.72

Según este test, una serie se considera buena si el resultado obtenido es menor que 15,987. Observa que el mejor resultado lo obtiene la secuencia de números consecutivos, que por cierto es la menos aleatoria de las tres.

3.3.2. Test Kolgomorov-Smirnov

El test chi-cuadrado se basa en observaciones realizadas sobre la función de probabilidad o densidad. El test de Kolgomorov-Smirnov (K-S) se basa en la función de distribución o acumulativa (CDF).

A partir de una secuencia de números $(x_1, \ldots, x_n)$, las observaciones se obtienen a partir de la función CDF discreta producida por esta serie:

$$F_n(x) = \frac{\sharp\{x_i \leq x\}}{n}$$

Es decir, esta función toma en x el número de elementos de la serie que son menores o iguales a x.

Esta función se compara con otra función estimada $F_e(x)$. Concretamente, mediremos la desviación máxima entre ambas funciones:

$$K^+ = \sqrt{n}\, max_x[F_n(x) - F_e(x)]$$

$$K^- = \sqrt{n}\, max_x[F_e(x) - F_n(x)]$$

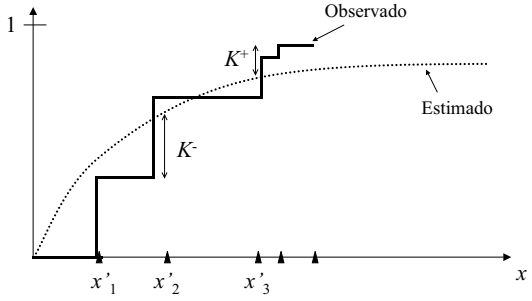


Figura 3.2: Desviaciones máximas en el test K-S

El significado de K^+ y K^- se puede ver en la gráfica de la figura 3.2. El valor de K^+ expresa la máxima diferencia entre ambas funciones cuando la CDF de las observaciones supera a la CDF estimada. K^+ expresa la máxima diferencia entre ambas funciones cuando la CDF estimada supera a la CDF de las observaciones.

Al igual que en el test chi-cuadrado, en este test las desviaciones calculadas siguen una distribución aleatoria, denominada K, con n grados de libertad. Así, dependiendo del grado de confianza α deseado, la desviación máxima deberá ser menor que el valor $K_{[1-\alpha;n]}$.

De nuevo, si queremos comprobar si un generador de números aleatorios es bueno, compararemos la serie generada de longitud n con la distribución uniforme $U(0,1)$, del siguiente modo:

- Ordenamos la serie de menor a mayor $(x'_1, \ldots, x'_n)$.
- Calculamos las desviaciones máximas como sigue:

$$K^+ = \sqrt{n}\, max_j[\frac{j}{n} - x'_j]$$

$$K^- = \sqrt{n}\, max_j[x'_j - \frac{j-1}{n}]$$

- Finalmente, si el valor de K^+ o K^- es mayor que $K_{[1-\alpha;n]}$, entoces descartaremos el generador.

Al contrario que el test chi-cuadrado, el test K-S está diseñado para series con pocas muestras y que siguen una distribución continua. El test K-S hace mejor uso de los datos ya que no necesita tantas muestras como el chi-cuadrado (al menos 5 por cada categoría), y además no necesita diseñar una partición apropiada del espacio muestral. En general, el test K-S es bastante mejor que el chi-cuadrado.

A modo de ejemplo, los resultados obtenidos con las series de la sección anterior serían los siguientes:

Secuencia	**Resultado**
Números consecutivos	0.0316
M/M/2	17.296
Random	0.894

Las secuencias que pasan este test no deben superar el valor 1,48 según las tablas de la distribución K-S. Observa que de nuevo sólo la serie M/M/2 es rechazada.

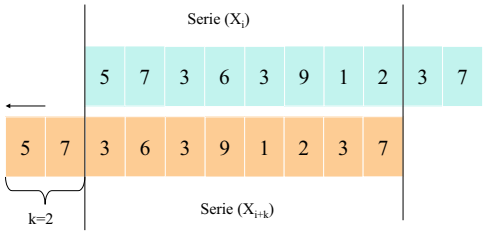


Figura 3.3: Variables X_i y X_{i+k} en un test de autocovarianza

3.3.3. Test de correlación serial

Los dos tests anteriores se aplican para medir la uniformidad de las series generadas por los generadores. Otro aspecto que nos interesa medir en una serie es la dependencia de sus valores. Para ello se proponen dos técnicas básicas:

- Realizar un test espectral de la serie, es decir, mostrar en un plano cómo se distribuyen los pares de valores consecutivos de la serie (u_i, u_{i+1}) (ver figura 3.1).
- Medir la *autocovarianza* de los valores de la serie.

En esta sección nos vamos a centrar en el segundo método. Éste consiste en calcular la covarianza existente entre la serie original y la misma serie desplazada en k posiciones (ver figura 3.3). Si la covarianza no es cero, podremos asegurar que existe alguna dependencia entre los valores de la serie, y en consecuencia tendremos que descartar el generador. Sin embargo, lo contrario no es cierto: si la covarianza es cero, la serie todavía puede presentar dependencias entre sus valores.

Para medir la autocovarianza de una serie aleatoria, calcularemos la covarianza que existe entre los números de la serie que están separados en k posiciones, esto es, entre x_i y x_{i+k}. Esta covarianza se denomina *autocovarianza* de salto k, denotada R_k:

$$R_k = Cov(X_i, X_{i+k}) = E[(X_i - \mu_i) \cdot (X_{i+k} - \mu_{i+k})]$$

En nuestro caso, como la serie de números debe seguir una distribución uniforme $U(0,1)$, R_k se calcularía como sigue:

$$R_k = \frac{1}{n-k} \sum_{i=1}^{n-k} (x_i - \frac{1}{2})(x_{i+k} - \frac{1}{2})$$

Al igual que en los tests anteriores, la medida R_k es una medida aproximada que debe contrastarse con una distribución aleatoria. Para este caso, cuando n es grande, R_k sigue una distribución normal centrada en 0 y con varianza $1/(144(n-k))$.

Así pues, el generador debe descartarse cuando el intervalo de confianza para R_k no contiene el valor 0, ya que demuestra que existe una dependencia importante entre los valores de la serie. Para R_k el intervalo de confianza se calcula como sigue:

$$R_k \pm \frac{z_{1-\alpha/2}}{12\sqrt{n-k}}$$

A modo de ejemplo, tomaremos de nuevo las tres series anteriores y probaremos este test con diferentes valores de k.

Secuencia	$k=1$	$k=2$	$k=3$
Números consecutivos	[0.0797,0.0865]	[0.0796,0.0864]	[0.0794,0.0862]
M/M/2	[0.107,0.114]	[0.107,0.114]	[0.107,0.114]
Random	[-0.0015,0.0052]	[-0.0016,0.0051]	[-0.00081,0.0059]

Según este test, solo pasaría la serie obtenida con `random()`, ya que es la única que incluye el 0 en los intervalos obtenidos.

3.4. Variables aleatorias

En muchos experimientos, incluidos los de simulación, los resultados obtenidos presentan cierta incertidumbre. Generalmente, la representación y manejo de esta incertidumbre se realiza mediante *variables aleatorias*. Partiendo del conjunto de todos los posibles resultados en un experimento, denominado *espacio de muestreo* y denotado como S, una variable aleatoria define una función que asigna a cada valor de S un valor real que representa de alguna forma su probabilidad de suceso. Las variables aleatorias suelen denotarse con letras mayúsculas (ej. X, Y, Z), mientras los valores que pueden tomar se denotan con letras minúsculas (ej. x, y, z)

Diremos que una variable aleatoria es *discreta* si su espacio de muestreo es discreto, es decir, si existe una correspondencia uno a uno con un subconjunto de los números enteros. Si el espacio de muestreo se define sobre los reales, diremos que la variable aleatoria es *continua*. Vamos a analizar en primer lugar las variables aleatorias discretas.

La función que asigna la probabilidad de cada valor de una variable aleatoria X se denomina *función de densidad*:

$$p(x_i) = P(X = x_i)$$

Una propiedad que debe cumplir esta función es que la suma de todas las probabilidades debe valer 1, es decir:

$$\sum_{i=1}^{\infty} p(x_i) = 1$$

Otra función interesante que caracteriza una variable aleatoria es la *función de distribución* o acumulativa (cdf), la cual define la probabilidad de encontrar un valor menor que otro dado x, es decir $P(X \leq x)$. Esta función se define a partir de la de densidad como sigue:

$$F(x) = \sum_{x_i \leq x} p(x_i)$$

Esta función presenta algunas propiedades interesantes:

- Está acotada entre 0 y 1, es decir $0 \leq F(x) \leq 1$
- Es monotona creciente
- $\lim_{x \to \infty} F(x) = 1$ y $\lim_{x \to -\infty} F(x) = 0$

En el caso de una variable aleatoria continua, también asociaremos un función de densidad, denotada $f(x)$, sobre los valores del espacio de muestreo. De forma similar a las variables discretas, la suma sobre todo el espacio de valores debe ser 1, es decir:

$$\int_{-\infty}^{\infty} f(x) \cdot dx = 1$$

Sin embargo, la función $f(x)$ no representa la probabilidad de que $X = x$, ya que ésta valdría siempre 0 ($\int_x^x f(y) \cdot dy = 0$). La nueva interpretación de la función de densidad debe realizarse sobre pequeños intervalos. Así, si tenemos un pequeño incremento $\Delta x \geq 0$, la probabilidad asociada a x será proporcional al área que define la función de densidad sobre el intervalo $[x, x + \Delta x]$, cuanto mayor sea el área mayor será la probabilidad del punto x.

$$P(X \in [x, x + \Delta x]) = \int_{x}^{x+\Delta x} f(y) \cdot dy$$

Sobre las variables aleatorias continuas también podemos definir una función de distribución o acumulativa, la cual representará la probabilidad de encontrar un valor menor que otro dado x:

$$F(x) = \int_{-\infty}^{x} f(y) \cdot dy$$

Esta función presenta las mismas propiedades que las que hemos visto anteriormente para las variables discretas.

Para finalizar, definiremos varios parámetros que caracterizan a cualquier variable aleatoria:

Parámetro	**Discreta**	**Continua**
Valor esperado $E(X)$	$\sum_{j=1}^{\infty} x_j \cdot p(x_j)$	$\int_{-\infty}^{\infty} x \cdot f(x) \cdot dx$
Varianza $Var(X)$	$E(X^2) - E(X)$	$E(X^2) - E(X)$
Desviación Típica (σ)	$\sqrt{Var(X)}$	$\sqrt{Var(X)}$

3.5. Generación de variables aleatorias

Desde el punto de vista de programación, una variable aleatoria es un generador de números aleatorios que sigue una determinada distribución. En el tema anterior vimos cómo generar secuencias de valores aleatorios que siguen una distribución uniforme $U(0, 1)$ mediante los denominados generadores congruenciales lineales. En este tema estudiaremos cómo generar secuencias de valores aleatorios que siguen otras funciones de distribución conocidas y ampliamente utilizadas en simulación.

3.5.1. Método de inversion

Este método se basa en la función inversa de la función de distribución de la variable aleatoria X. Dado que $F(X)$ está acotada entre 0 y 1, podemos generar valores en $U(0, 1)$ y sobre ellos aplicar $F^{-1}(X)$.

Por ejemplo, tomemos la distribución exponencial con media en λ, y cuyas funciones de densidad y distribución son las siguientes:

$$\begin{aligned} f(x) &= \lambda \cdot e^{-\lambda \cdot x} \\ F(x) &= 1 - e^{-\lambda \cdot x} \end{aligned}$$

La función inversa de $F(X)$ sería:

$$F^{-1} = -\frac{1}{\lambda} ln(1 - u)$$

Y el algoritmo para generar valores aleatorios para esta distribución sería :

```
def exponential(media):
    u = random(0,1)
    return -log(1-u)/media
```

El método de la inversa es especialmente útil para generar valores de variables aleatorias discretas. Veamos un ejemplo.

Supongamos que tenemos una variable aleatoria discreta con espacio de muestreo $S = \{1, 2, 3\}$, y cuya función de densidad es la siguiente:

$$p(1) = 0{,}2 \qquad p(2) = 0{,}3 \qquad p(3) = 0{,}5$$

Entonces su función de distribución sería:

$$F(1) = 0{,}2 \qquad F(2) = 0{,}2 + 0{,}3 = 0{,}5 \qquad F(3) = 0{,}5 + 0{,}5 = 1$$

Así, en el eje de F(x) podemos identificar tres intervalos: $(0, 0{,}2]$, $(0{,}2, 0{,}5]$ y $(0{,}5, 1]$. Para obtener valores para esta variable, generaremos un valor para $U(0, 1)$, y determinaremos su inversa a partir del intervalo en el que caiga. Si éste cae en el primer invervalo, el valor de F^{-1} será 1, si en el segundo 2, y si en el tercero 3. Así, el algoritmo para generar valores para esta variable aleatoria sería el siguiente:

```
def genera_discreta(p1, p2, p3):
    u= random(0,1)
    if u<=p1:
       finv = 1
    elif u<=p1+p2:
       finv = 2
    else:
       finv = 3
    return finv
```

Recuerda que este tipo de distribuciones suele aparecer en los diagramas de sucesos, concretamente cuando un suceso planifica la ocurrencia de otros sucesos con una determinada probabilidad.

Como ejercicio se plantea realizar una función para obtener valores aleatorios según una distribución discreta representada con una lista de cualquier longitud (por ejemplo [0.1, 0.3, 0.4, 0.2] o [0.5, 0.3,0.2], etc.)

3.5.2. Método del rechazo

Cuando no podamos calcular la inversa de la función de distribución, lo que sucede en la mayor parte de las distribuciones continuas, entonces tendremos que basarnos en la función de densidad $f(x)$.

El método del rechazo utiliza una función de densidad $g(x)$ para la cual sabemos generar números aleatorios (ej. una uniforme), y que además recubre por completo la función de densidad que queremos obtener, es decir:

$$a \cdot g(x) \geq f(x)$$

El algoritmo para generar valores a partir de $g(x)$ sería el siguiente:

```
repetir
    genera un valor x con densidad g
    genera un valor u con U(0, 1)
hasta que u·a·g(x) <= f(x)
devuelve x
```

Para que funcione eficientemente, el algoritmo debe utilizar una función $g(x)$ sencilla, y que se ajuste lo más posible a la función $f(x)$. De ella dependerá el número de iteraciones del algoritmo.

Veamos un ejemplo presentado en [Raj Jain, 1991]. Supongamos que queremos generar una secuencia de valores para la distribución $beta(2,4)$, cuya función de densidad es:

$$f(x) = 20 \cdot x(1-x)^3$$

Esta función puede ser recubierta por un rectángulo de altura 2,11, que es el máximo de la función. Por lo tanto, tomando $g(x) = U(0,1)$ y $a = 2{,}11$, obtendríamos el siguiente algoritmo:

```
def beta2_4():
    a = 2.11
    x = random(0,1)
    u = random(0,1)
    while (u*a <= (20*x*(1-x)**3)):
        x = random(0,1)
        u = random(0,1)
    return x
```

3.5.3. Método de Convolución

Si X es la suma de dos variables aleatorias Y_1 y Y_2, entonces la función de distribución de X puede obtenerse analíticamente mediante la convolución de las funciones de distribución de Y_1 y Y_2. Este es el principio del método de convolución: para obtener los valores de X bastará con generar 2 valores para Y_1 y Y_2, y devolver su suma.

Es importante destacar que la suma de un conjunto de variables aleatorias (convolución) es un concepto diferente al de la suma de sus funciones de distribución (composición).

Existen varias distribuciones que pueden generarse por convolución:

- Una variable con distribución normal puede obternerse sumando un número suficiente de variables aleatorias con cualquier distribución.
- Una variable con distribución χ^2 con k grados de libertad puede obtenerse sumando los cuadrados de k variables con distribución $N(0,1)$.
- Una variable con distribución Erlang-k puede obtenerse sumando k variables con distribución exponencial.
- La suma de dos variables con distribución $U(0,1)$ forma una variable con distribución triangular.

3.6. Algunas distribuciones útiles

A continuación mostramos brevemente las características de algunas distribuciones continuas que son habitualmente utilizadas en simulación.

Uniforme: denotada como $U(a,b)$ con $b > a$, su función de densidad es:

$$f(x) = \frac{1}{b-a}$$

su media es $\frac{a+b}{2}$, y su varianza es $\frac{(b-a)^2}{12}$.

Podemos generar valores para $U(a,b)$, generando un valor u en $U(0,1)$, y devolviendo $a + (b-a) \cdot u$.

Recuerda que para obtener valores de una distribución uniforme se suelen utilizar los generadores congruenciales que vimos al principio de este tema. Además, a patir de esta distribución pueden obtenerse el resto según los métodos vistos en las secciones anteriores.

Exponencial: denotada como $Exp(a)$, su función de densidad es:

$$f(x) = \frac{1}{a} e^{-x/a}$$

su media es a y su varianza a^2.

Sus valores se generan mediante el método de inversión (sección 3.1).

Dentro de la familia de la distribuciones exponenciales encontramos también la $Erlang(a,m)$, que se obtiene mediante la convolución de m variables exponenciales (de hecho se utiliza para simular la suma de los tiempos de servicio de m servidores en serie); la $Gamma(a,b)$, que es una generalización de las anteriores (donde a juega el papel de m y b es la media de la distribución); y la $Weibull(a,b)$, que generaliza la exponencial en $a = 1$, siendo b la media de la distribución.

Normal: denotada como $N(\mu,\sigma)$, su función de densidad es:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

Su media es μ y su varianza σ^2.

Pueden obtenerse valores para esta distribución por convolución de n ($n = 12$) variables en $U(0,1)$:

$$N(\mu,\sigma) \sim \mu + \sigma \frac{(\sum_{i=1}^{n} u_i) - n/2}{\sqrt{n/12}}$$

Otro método utilizado para generar esta distribución es el de Box-Muller, el cual genera dos valores independientes para $N(\mu,\sigma)$ a partir de dos valores u_1 y u_2 de una uniforme $U(0,1)$:

$$\begin{aligned} x_1 &= \mu + \sigma cos(2\pi u_1)\sqrt{-2ln(u_2)} \\ x_2 &= \mu + \sigma sin(2\pi u_1)\sqrt{-2ln(u_2)} \end{aligned}$$

A partir de la distribución normal se pueden obtener diversas distribuciones similares, como por ejemplo la χ^2, $LogNormal(\mu,\sigma)$, la $t - student$, y la distribución $Cauchy(a,b)$.

Zipf: En el ámbito de los Sistemas de Información es muy frecuente encontrar distribuciones que siguen una función potencia (*Power Law*). Estas distribuciones son discretas, y expresan una relación entre el orden i de la frecuencia de uso (o popularidad) de un objeto con su probabilidad de ocurrencia P_i, a saber: $P_i = 1/i^a$, donde a es aproximadamente 1. Este tipo de distribuciones se observan por ejemplo en la frecuencia de uso de las palabras de un diccionario, en el grado de popularidad de las páginas de la Web, o en la frecuencia de uso en una caché. En los dos últimos casos, el exponente a suele ser menor que 1.

La versión continua de esta distribución es la *Pareto*, que se ha utilizado tradicionalmente en Economía para modelar el reparto de las riquezas (el 20 % de la población dispone del 80 % de las riquezas). En el ámbito de la computación, esta distribución se ha utilizado con éxito para modelar el tamaño de los ficheros de un servidor en Internet (con $a = 1{,}06$). Las funciones de densidad y acumulada de esta distribución son las siguientes:

$$f(x;a) = \frac{a \cdot m^a}{x^{a+1}} \; con \; x > m$$
$$F(x;a) = 1 - (m/x)^a \; conx \geq m$$

En prácticamente todos los libros de simulación podréis encontrar una revisión de las distribuciones más utilizadas en simulación, tanto sus características como el método más apropiado de generar valores.

3.7. Bibliografía

Raj Jain "The Art of Computer Systems Performance Analysis". Ed. Wiley (1991).

David Ríos Insua et al. "Simulación métodos y aplicaciones". Ed. Ra-Ma, Madrid (1997)

A. M. Law, W. D. Kelton "Simulation Modeling & Analysis". Ed. McGraw-Hill (1984).

Capítulo 4

Tratamiento de los resultados

4.1. Introducción

Generalmente, el mayor esfuerzo a realizar en un proyecto de simulación se centra en la implementación del modelo. De hecho, podemos considerar el desarrollo de un simulador como un ejercicio más o menos complicado de programación. Sin embargo, un proyecto de simulación no finaliza con la implementación del simulador. Todo proyecto de simulación debe incluir además una etapa de *experimentación*, la cual consiste en el tratamiento estadístico de los datos de salida del simulador. Recordemos que estos datos están directamente relacionados con los *objetivos* del modelo de simulación. Sin esta fase experimental, las conclusiones que extraígamos carecerán de fundamento y posiblemente sean erróneas.

Desafortunadamente, sobre los datos de salida de un simulador no podemos aplicar directamente las técnicas clásicas de análisis estadístico, debido principalmente a que estas técnicas se basan en que las muestras sean **IID** (Independientes e Identicamente Distribuidas). Como veremos, los datos de salida de un simulador suelen contener fuertes correlaciones y sesgos, y por lo tanto no se cumple esta propiedad.

Pero antes de tratar los resultados de salida del simulador, debemos asegurarnos de que éste sea correcto y válido con respecto al modelo del sistema. El proceso de *validación* consiste en comprobar que las aproximaciones realizadas en el modelo de simulación se ajustan lo más posible a la realidad. Por otro lado, el proceso de *verificación* consiste en comprobar que dichas aproximaciones se implementan de forma correcta. En el proceso de validación deben intervenir expertos en el sistema real que avalen la proximidad de los

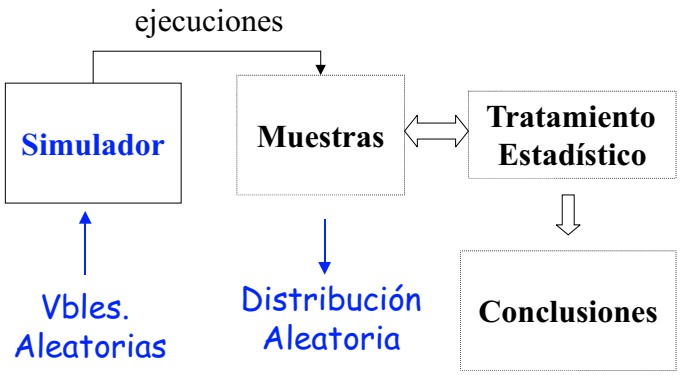


Figura 4.1: Tratamiento de resultados

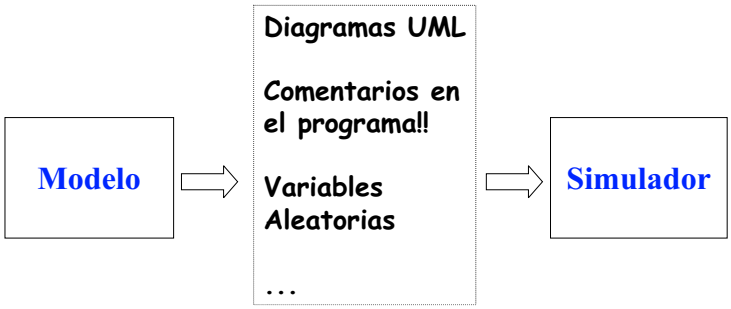


Figura 4.2: Herramientas de validación de modelos

resultados obtenidos con el modelo de simulación y el comportamiento del sistema real. En el proceso de verificación entra en juego todas las técnicas de depuración y verificación de programas que hemos visto a lo largo de la carrera.

4.2. Validación del modelo

Validar el modelo supone comprobar que las asunciones hechas para simplificar el funcionamiento del sistema son aceptables. En ese caso, y si el modelo se implementa correctamente, los resultados del simulador y del sistema real deberían ser parecidos.

Para validar el modelo, se deben comprobar tres aspectos del mismo: las asunciones que simplifican el sistema real, los parámetros de entrada del simulador y las distribuciones utilizadas, y por último los valores obtenidos y las conclusiones que se pueden extraer.

Estos tres aspectos deberían contrastarse con tres fuentes de información: la opinión de expertos en sistemas como el que se está simulando, mediciones sobre el sistema real y datos obtenidos con otros tipos de modelos.

En realidad se debería contrastar los tres aspectos a validar con las tres fuentes de información, aunque esto generalmente no es posible, ya que se suele utilizar técnicas de simulación cuando no se dispone de otras posibilidades. En cambio, suele ser posible la comparación con algunas de las fuentes de información para casos simplificados.

4.3. Verificación o depurado del modelo

Se trata de comprobar que el modelo ha sido implementado correctamente en el programa informático. Para verificar la implementación del modelo se pueden utilizan técnicas típicas del desarrollo de software. Además, existen otras técnicas más específicas para programas de simulación. Destacaremos las siguientes:

Diseño jerárquico y modular. Los modelos de simulación pueden resultar en proyectos software bastante complejos. Por tanto las técnicas generales de desarrollo de software deben ser utilizadas. Si el modelo de simulación se divide en módulos, éstos podrán ser comprobados independientemente. Además se puede modificar el funcionamiento de un módulo sin afectar al resto del programa. Por otro lado, los módulos son más fáciles de verificar, ya que son partes más pequeñas. En la división del modelo en módulos debe quedar clara las variables y funciones (la interfaz) que conecta a los módulos. El diseño jerárquico supone que para desarrollar cada módulo, se identifique una estructura en la que el problema se divide en subproblemas más sencillos. De esta forma se llegaría a partes cuya verificación sería directa. Para aplicar estos principios muchas bibliotecas de simulación se basan en la metodología orientada a objetos.

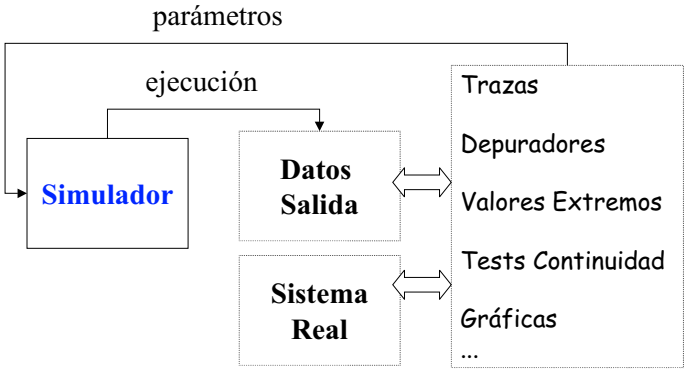


Figura 4.3: Verificación de simuladores

Comprobaciones en el programa. Consiste en poner comprobaciones en el programa que podrán detectar ciertos tipos de error. Por ejemplo, comprobar que una variable supera un valor imposible (por ejemplo, una probabilidad mayor que 1) o que la diferencia entre clientes creados y destruidos es incoherente.

Exposición del programa. Una manera de depurar el programa consiste en explicarlo detalladamente (línea a línea) a una pequeña audiencia. Simplemente el hecho de tener que ordenar las ideas para exponerlas hace que el programador detecte errores. Además, la audiencia también puede detectar errores o aportar sugenrencias.

Modelos determinísticos. El hecho de utilizar variables aleatorias en los programas difulta la detección de errores porque su comportamiento es menos predecible. Una posibilidad es estudiar el funcionamiento sustituyendo las variables aleatorias por comportamientos deterministas (por ejemplo sustituir una distribución aleatoria por la media de la distribución). De esta manera, los resultados son más fáciles de calcular y deben coincidir con el resultado de la simulación.

Ejecución de casos simples. Para encontrar respuestas previsibles y comprobar que coinciden con el resultado de la simulación se puede simplificar los casos a simular. Por ejemplo, ver la respuesta si se simula un único cliente, o reducir el número de servidores en una red. Esto no asegura que el simulador funcione bien en el caso general, pero muchos errores ya se producen en simulaciones muy simplificadas.

Estudio de trazas. En una traza se puede ver la secuencia de cambios en el simulador de forma ordenada temporalmente (en tiempo de la simulación). Las trazas permiten hacer un seguimiento de la simulación y entender los cambios que se producen y los valores que toman las variables de estado. Las bibliotecas de simulación pueden estar preparadas para generar trazas. En otro caso, el programador puede introducirlas en el programa, pero debería ser muy fácil de activar o desactivar su funcionamiento. Una vez que el programa ha sido depurado ya no se utilizan trazas. Las trazas pueden tener diferente nivel de información. Por ejemplo se puede incluir el tipo de evento y el tiempo en que se crea y procesa o también se puede indicar el cliente asociado, los valores de variables de estado modificadas, etc. Otra posibilidad es presentar en la traza solamente información de algún tipo de cliente particular o de algunas partes del sistema simulado. En ocasiones es interesante mostrar gráficamente algún parámetro de la simulación al mismo tiempo que ésta se desarrolla. En este caso

se dispone de menor información que con una traza pero puede ser fácil detectar que la simulación es errónea.

Test de continuidad. Se basan en que pequeños cambios en la entrada de las simulaciones deben traducirse en pequeños cambios en la salida (generalmente con un tendencia coherente respecto a la modificación introducida). Si el efecto en la salida es brusco o no sigue la tendencia esperada, puede haber errores en el programa.

Comprobación de casos extremos. Consiste en comprobar los casos extremos que se consideran en la simulación. Estos casos extremos incluirían tanto los más sencillos de configuración o de carga como los más complicados. Las pruebas pueden no representar casos de funcionamiento típico en el sistema real pero ayudan a detectar errores y situaciones no contempladas en el programa.

Test de consistencia. Consiste en modificar un conjunto de parámetros de la simulación de manera que el resultado deba ser equivalente. Por ejemplo, en una simulación cambiar un servidor por dos servidores idénticos con la mitad de potencia que el inicial. El resultado de la simulación debería ser equivalente. Las semillas de los generadores de números aleatorios tampoco deberían variar el resultado de la simulación. De todas formas esto podría detectarse si se replican los experimentos de simulación y se calcula un intervalo de confianza para los resultados. Los conjuntos de pruebas que se han utilizado para probar un simulador es conveniente almacenarlos con sus resultados. Así se pueden volver a aplicar cada vez que se modifica el simulador.

4.4. Eliminación del transitorio

Una vez validado y verificado el simulador, para extraer los resultados de las simulaciones, surgen otras cuestiones: si la simulación no arranca de un estado estable, ¿en qué momento se alcanza el estado estable? Esta cuestión es interesante porque necesitamos estar seguros de tomar datos en un funcionamiento representativo del sistema. A esta operación se la conoce como *eliminación del transitorio*. Para tener suficientes datos de una simulación, también se debería evaluar la longitud de la simulación. Si es demasiado corta, los datos pueden tener una gran variabilidad y se podrían sacar conclusiones erróneas. Si la simulación es demasiado larga, se puede perder demasiado tiempo en las simulaciones, sin que se aumente la calidad de los resultados. Veamos en primer lugar el problema del transitorio.

En la figura 4.4 se muestra la evolución de la salida de un simulador para distintas ejecuciones (cada curva es una ejecución distinta).

Como puede observarse, para cada ejecución podemos identificar un periodo transitorio, también denominado de calentamiento, producido por la situación inicial del simulador. En muchas ocasiones, este periodo transitorio se produce porque las colas de una red de estaciones están inicialmente vacías. Tras este periodo de transición, el simulador puede llegar a un estado estable (y representativo del sistema), y la varianza de los resultados entre las distintas ejecuciones se reduce notablemente.

Nótese que este periodo transitorio produce una distorsión muy importante en las medidas globales de la salida, tanto en la media como en la varianza, sobre todo si la simulación es demasiado corta. Por consiguiente, es conveniente eliminar de alguna forma este periodo transitorio a la hora de realizar el tratamiento de los resultados.

No hay un método exacto para eliminar el transitorio. Se utilizan diferentes métodos aproximados, los cuales normalmente se basan en que la varianza de los resultados es menor cuando se alcanza el funcionamiento normal que durante el transitorio (esto es cierto si el sistema simulado es estable pero no se cumple en sistemas inestables).

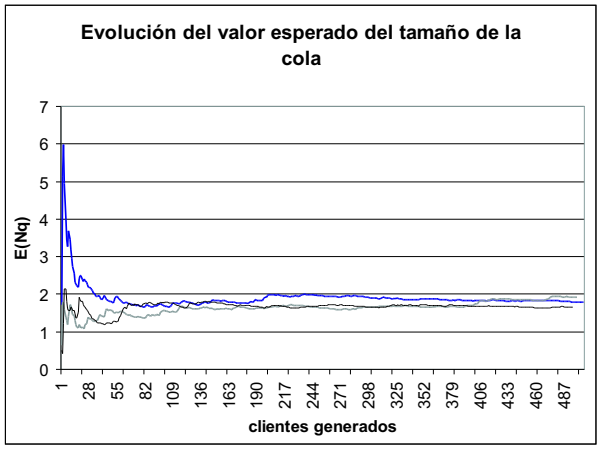


Figura 4.4: Periodo transitorio

4.4.1. Alargar la simulación

Una técnica comúnmente utilizada consiste en alargar la simulación más de lo necesario. De esta forma la influencia de los datos del transitorio se diluye al tomar un mayor número de datos.

Esta forma de actuar tiene dos inconvenientes. Por un lado se desperdicia tiempo y recursos porque la simulación es más larga de lo necesario y por otro no se tiene garantía de estar actuando bien, porque nada indica que se han tomado suficientes datos.

4.4.2. Seleccionar un buen estado inicial

Si se encuentra un estado típico del régimen estable de la simualción, se puede programar directamente ese estado como el inicio de la simulación. En ese caso no sería necesario eliminar el transitorio. El estado inicial representativo se podría buscar a partir de algunas simulaciones previas.

4.4.3. Método del truncado

Como los valores en estado estable oscilarán menos que durante el periodo inicial, una forma de estimar el transitorio es ir comprobando alguna variable de la simulación hasta que el valor de esta variable no sea ni un máximo ni un mínimo del conjunto de valores muestreados.

Ejemplos de variables que se pueden utilizar en un sistema estable, podrían ser el tiempo de respuesta de los clientes o la población media que hay en el sistema. Inicialmente el sistema estará vacío y el número de clientes en el sistema irá aumentando hasta quedar oscilando en un conjunto de valores (la media de ese conjunto será el número medio de clientes en el sistema en el funcionamiento estable; en cambio si el sistema es inestable, el tansitorio no terminaría nunca).

4.4.4. Impacto de eliminar datos iniciales

Este método consiste en eliminar datos iniciales hasta que se estabilice la media de los resultados. Es decir, que se va calculando la media de los resultados al eliminar las i primeras muestras (empezando por $i = 0$). Llegaremos a un valor de i a partir del cual la media calculada del resultado no se modifica.

Como la media del resultado de una simulación tiene variaciones incluso en estado estable, se utiliza la media de varias réplicas de la simulación en las que lo único que cambia son las semillas de los generadores de variables aleatorias.

Supongamos que se hacen m simulaciones y en cada simulación se toman n muestras. x_{ij} es la muestra j de la simulación i. Entonces podemos seguir el siguiente algoritmo:

1. La trayectoria media: $\bar{x}_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij}$ para $j = 1, 2, \ldots, n$
2. Media total: $\bar{\bar{x}} = \frac{1}{n}\sum_{j=1}^{n} \bar{x}_j$
3. Desde $l = 1$ calculamos las medias de los $n - l$ valores: $\bar{\bar{x}}_l = \frac{1}{n-l}\sum_{j=l+1}^{n} \bar{x}_j$
4. Calcular el cambio relativo al eliminar l valores: $\frac{\bar{\bar{x}}_l - \bar{\bar{x}}}{\bar{\bar{x}}}$
5. Repetir los pasos de 3 a 4 variando l desde 1 a $n-1$. Si se va representando el cambio relativo en función de l, se observará que a partir de un valor de l se estabiliza el cambio relativo.

Un inconveniente de este método es que se descarta el transitorio de varias simulaciones.

4.4.5. Medias Batch

En principio es la misma idea que el método anterior. En cambio, en vez de replicar la simulación, lo que se hace es hacer una simulación muy larga y dividirla en pedazos. Cada pedazo se toma como una réplica de la simulación. Únicamente el primer pedazo tendrá transitorio, el resto parten de un estado representativo. El método estudia la varianza de las medias de cada pedazo en función de la talla del pedazo. Una simulación (N observaciones) se divide en m pedazos de talla n. x_{ij} es la muestra j del batch i. Empezando con un n pequeño se siguen los siguientes pasos:

1. Se calcula la media de las muestras de cada batch: $\bar{x}_i = \frac{1}{m}\sum_{j=1}^{n} x_{ij}$ para $i = 1, 2, \ldots, m$.
2. Se calcula la media total: $\bar{\bar{x}} = \frac{1}{m}\sum_{i=1}^{m} \bar{x}_i$
3. Se calcula la varianza de las medias de cada batch en función del tamaño de los batchs: $Var(\bar{x}) = \frac{1}{m-1}\sum_{i=1}^{m}(\bar{x}_i - \bar{\bar{x}})^2$

Incrementando n se repiten los pasos 1 a 3. Se representa la varianza en función de n. El valor de n a partir del cual la varianza siempre decrece, indica el transitorio (hay que tener en cuenta que la curva que forma el transitorio puede tener picos).

4.5. Criterio de parada

Si una simulación es demasiado corta, no nos prodremos fiar de los resultados obtenidos porque pueden representar condiciones particulares. Si es demasiado larga, se utilizan muchos recursos y se puede perder mucho tiempo en los experimentos.

Para tener una idea de la longitud correcta de la simulación lo que se hace es fijar un intervalo de confianza de los resultados y calcular el número de muestras (la longitud de la simulación) para obtener ese intervalo de confianza.

La fórmula del *intervalo de confianza*, dado un nivel de rechazo α para una población de n muestras se calcula como sigue:

$$\bar{x} \pm z_{1-\alpha/2}\sqrt{\frac{Var(x)}{n}}$$

donde $\bar{x}$ es la media de las n muestras, y $Var(x)$ es la varianza de las muestras.

Sin embargo, el uso de la función z para el cálculo del intervalo de confianza solo es válido cuando el número de muestras es grande. En nuestros experimentos estas muestras se extrae de las distintas réplicas de la simulación, que suele ser un número bajo (menos de 10). Por esta razón, se debe calcular los intervalos de confianza con una distribución que se aproxime a z, y ésta es la distribución t_γ (t student) con γ grados de libertad. Esta distribución tiende a la distribución normal $N(0,1)$ cuando γ tiende a infinito.

Con todo esto, el intervalo de confianza se calcula como sigue:

$$\bar{x} \pm t_{n-1,1-\alpha/2}\sqrt{\frac{Var(x)}{n}}$$

La siguiente tabla contiene un fragmento de la tabla de valores críticos de la función t con respecto al número de muestras n y el nivel de rechazo α.

	$1-\alpha/2$		
$n-1$	0.90	0.95	0.99
3	1.638	2.353	4.541
4	1.533	2.132	3.747
5	1.476	2.015	3.365
6	1.440	1.943	3.291
∞	1.282	1.645	2.326

Desgraciadamente, la fórmulas anteriores solo se pueden aplicar si las muestras son independientes. En el caso de las simulaciones las muestras pueden estar correlacionadas. Por ejemplo, si se calcula el tiempo de respuesta que sufren los clientes de una simulación, sus tiempos de respuesta dependen unos de otros si coinciden en las estaciones.

Para solucionar este problema se han propuesto diferentes soluciones.

4.5.1. Réplicas independientes

Se basa en la idea de que los datos de diferentes réplicas son independientes.

Se realizan m réplicas de tamaño $n+n_o$ de la simulación, donde n_o representa el transitorio. Los n_o datos iniciales de cada réplica son descartados. Luego se actúa de la siguiente manera:

1. Calcular la media de cada réplica: $\bar{x}_i = \frac{1}{n}\sum_{j=n_o+1}^{n_o+n} x_{ij}$ para $i = 1, 2, \ldots, m$
2. Calcular la media de todas las réplicas: $\bar{\bar{x}} = \frac{1}{m}\sum_{i=1}^{m} \bar{x}_i$
3. Calcular la varianza de las medias de las réplicas:
   $Var(\bar{x}) = \frac{1}{m-1}\sum_{i=1}^{m}(\bar{x}_i - \bar{\bar{x}})^2$
4. El intervalo de confianza para la media: $\bar{\bar{x}} \pm t_{m-1,1-\alpha/2}\sqrt{\frac{Var(\bar{x})}{m}}$

El intervalo de confianza es inversamente proporcional a $\sqrt{mn}$. Como se pierden n_o datos de cada réplica, el número de réplicas, m, normalmente no es mayor que 10. La longitud de las simulaciones, n, se aumenta hasta tener un intervalo de confianza de la media tan estrecho como sea necesario.

4.5.2. Medias Batch

Se realiza una simulación con $n_o + N$ muestras. Se descartan las n_o muestras iniciales (el transitorio) y las N muestras restantes se dividen en $m = \frac{N}{n}$ partes (lo que serían las réplicas de la simulación).

Empezando por un valor de n pequeño (por ejemplo 1) se actúa igual que en el caso anterior.

Al igual que antes, el intervalo de confianza de la media es inversamente proporcional a $\sqrt{mn}$.

Sin embargo hay un inconveniente. En el método de las réplicas se acepta que cada réplica es independiente de las otras. Esto no es cierto en el caso de usar las medias de partes de una misma simulación, de hecho los pedazos pueden ser más dependientes cuanto más pequeños sean (las muestras que contienen están más cercanas entre sí).

Para resolver este problema se va aumentando n y se recalcula los datos igual que en las réplicas pero además se calcula la covarianza entre medias consecutivas. Se actúa así hasta que la autocovarianza es pequeña comparada con la varianza.

La fórmula de la covarianza es la siguiente:

$$Cov(\bar{x}_i, \bar{x}_{i+1}) = \frac{1}{m-2} \sum_{i=1}^{m-1} (\bar{x}_i - \bar{\bar{x}})(\bar{x}_{i+1} - \bar{\bar{x}})$$

4.5.3. Método de la regeneración

El método se basa en encontrar estados de una simulación, a partir de los cuales la simulación se puede tomar como una simulación independiente que empieza en un mismo estado inicial. Este método no siempre se puede aplicar, ya que los sistemas no siempre presentan puntos de regeneración o bien éstos tienden a presentarse cada vez con menor probabilidad a medida que avanza la simulación.

El método para calcular la varianza con el método de regeneración es un poco más complicado debido a que cada réplica tiene un tamaño diferente.

Supongamos que se dispone de m ciclos de tallas n_1, n_2, ... , n_m. Para calcular el intervalo de confianza:

1. Calcular la suma de cada ciclo: $y_i = \sum_{j=1}^{n_i} x_{ij}$
2. Calcular la media total: $\bar{\bar{x}} = \frac{\sum_{i=1}^{m} y_i}{\sum_{i=1}^{m} n_i}$
3. Calcular la diferencia entre las sumas de ciclos esperadas y observadas: $w_i = y_i - n_i \bar{\bar{x}}$ para $i = 1, 2, \ldots, m$
4. Calcular la varianza de las diferencias: $Var(w) = s_w^2 = \frac{1}{m-1} \sum_{i=1}^{m} w_i^2$
5. Calcular el tamaño medio de los ciclos: $\bar{n} = \frac{1}{m} \sum_{i=1}^{m} n_i$
6. El intervalo de confianza es: $\bar{\bar{x}} \pm t_{m-1,1-\alpha/2} \frac{s_w}{\bar{n}\sqrt{m}}$

Con el método de la regeneración no se eliminan datos iniciales, en cambio presenta algunos inconvenientes que dificultan su aplicación. Puede ser difícil definir los puntos de regeneración. Además en los estados de la simulación se debe ir comprobando si se trata de un punto de regeneración. Otras desventajas es que no se puede predecir el tiempo de la simulación y que los datos estadísticos son menos exactos.

4.6. Bibliografía

A. M. Law, W. D. Kelton "Simulation Modeling & Analysis". Ed. McGraw-Hill (1984).

R. Jain "The Art of Computer Systems Performance Analysis". Ed. Wiley (1991).