Tipos Abstractos de Datos para Desarrollo de Programas de Simulación en C++

Marco Aurelio Alzate Monroy
Universidad Distrital F.J.C.

Resumen

La simulación de eventos discretos resulta una herramienta imprescindible para la evaluación de desempeño de redes de comunicaciones. En este contexto, el uso de un lenguaje de propósito general para desarrollo de programas de simulación tiene dos ventajas principales: el conocimiento previo que el analista tiene del lenguaje y la flexibilidad que se gana al poder desarrollar cualquier modelo con el nivel de precisión deseado. Sin embargo, debe invertirse un gran esfuerzo de programación para resolver problemas generales que nada tienen que ver con el modelo particular que se quiere evaluar, tales como el control del avance del tiempo que permita una secuenciación adecuada de los eventos, la generación de muestras de variables aleatorias para modelamiento de tráfico y otros aspectos no determinísticos del modelo, la adquisición de estadísticas para obtención de medidas de desempeño y el manejo de las colas que se forman en la competencia por los recursos del sistema. Este artículo presenta algunos tipos abstractos de datos en C++ (clases) que resuelven los cuatro problemas mencionados, permitiendo al analista desarrollar sus programas de simulación en C++ mientras concentra su esfuerzo de programación en el modelo particular y no en detalles adicionales. Con el propósito de introducir la herramienta, se hace un repaso de los conceptos principales de la simulación de eventos discretos y, finalmente, se muestran dos ejemplos de aplicación de la herramienta desarrollada.

1. Introducción

Para el correcto diseño de redes de comunicaciones es necesario contar con técnicas adecuadas de evaluación de desempeño, las cuales incluyen la medición directa, el cálculo analítico y el uso de modelos de simulación. Aunque cada enfoque tiene importantes ventajas sobre los demás, la simulación cobra cada vez mayor popularidad debido a que la medición directa requiere la existencia y la disponibilidad de la red, lo cual es una condición poco común en problemas de diseño, mientras que los cálculos analíticos suelen basarse en simplificaciones de las interacciones entre los componentes del sistema, por lo cual sólo se obtienen resultados aproximados que pueden estar muy distantes de la realidad[1].

Para la simulación de redes de comunicaciones se pueden utilizar paquetes de software de aplicación específica como ComNet III o BONeS PlanNet, lenguajes de simulación como SimScript II.5 o GPSS, y lenguajes de programación de propósito general como Pascal, C ó C++. En el primer caso se tiene la máxima facilidad de uso pues, típicamente, se disponen de interfaces gráficas de alto nivel para la descripción de la red y se puede dar inicio a la simulación sin ningún esfuerzo de programación. Sin embargo existen serias limitaciones de flexibilidad pues sólo se pueden estudiar los sistemas que se construyan con los bloques predefinidos en la herramienta[5]. Los lenguajes de simulación, en cambio, tienen mayor flexibilidad pues permiten construir cualquier modelo, pero sacrifican la facilidad de uso al requerir un considerable esfuerzo de programación. Esta opción también tiene la desventaja de que los ingenieros de comunicaciones que vayan a desarrollar los experimentos de simulación deberán aprender un nuevo lenguaje de programación, lo cual puede requerir un tiempo adicional considerable[5]. Por último, la utilización de un lenguaje de programación de propósito general tiene las ventajas de que el ingeniero de comunicaciones puede estar familiarizado con él previamente y de que la flexibilidad es máxima pues se pueden desarrollar los modelos con el nivel de detalle que se desee. Sin embargo, a pesar de estas importantes ventajas, casi nunca se considera la opción de utilizar un lenguaje de propósito general pues el esfuerzo de programación suele ser prohibitivamente excesivo. En efecto, deben resolverse problemas delicados como la generación de variables aleatorias para modelar el tráfico de entrada a la red, el control del avance de tiempo para secuenciar adecuada y eficientemente los eventos, la adquisición de estadísticas sobre las medidas de desempeño de interés y el análisis de los resultados de la simulación[4]. Sin embargo, si se pudiesen obviar estas dificultades de manera que el esfuerzo de programación pudiera concentrarse en el desarrollo del modelo, podrían aprovecharse las ventajas de la flexibilidad y el uso de una herramienta conocida que ofrecen los lenguajes de propósito general.
Como cada vez es más popular el lenguaje de programación C++, en este artículo se reporta la elaboración de unos tipos abstractos de datos (clases en C++) que facilitan el desarrollo de programas de simulación de

eventos discretos en este lenguaje. Después de explicar brevemente en qué consiste la simulación de eventos discretos y describir la herramienta desarrollada, se presentan algunos ejemplos prácticos de su utilización.

2. Simulación de Eventos Discretos

Desde una perspectiva de alto nivel, las redes de telecomunicaciones se pueden observar como unos recursos de comunicación por los que los usuarios generan demandas y unos protocolos o algoritmos distribuidos que controlan la asignación de los recursos de la red para satisfacer dichas demandas[1]. La generación de estas demandas y cada uno de los pasos que sufren dentro de la red para ser atendidas constituyen eventos instantáneos que alteran el estado de la red. Este procesamiento basado por eventos, natural en el contexto de las redes de telecomunicaciones, conduce al método conocido como "Simulación de Eventos Discretos", DES[6].

En los modelos DES, hay un reloj de simulación que se actualiza con la ejecución ordenada de los eventos, donde el Evento es la unidad básica ejecutable. Existe un conjunto de variables de estado que describen al sistema, las cuales se modifican al ejecutarse las rutinas asociadas con cada evento. Para controlar la sucesión de eventos se utiliza una lista de eventos futuros, LEF, en la cual se almacenan los próximos eventos en un orden cronológico ascendente. Las rutinas asociadas con cada evento pueden añadir o eliminar eventos en la LEF, para lo cual utilizan generadores de variables pseudo-aleatorias que representan el tráfico y otras condiciones estocásticas dentro del sistema. Así, el algoritmo de simulación consiste, esencialmente, en extraer de la LEF el evento más próximo, actualizar el reloj de simulación según el tiempo de activación de dicho evento, llamar a la rutina correspondiente al evento seleccionado y repetir el proceso iterativamente (Figura 1)[6].

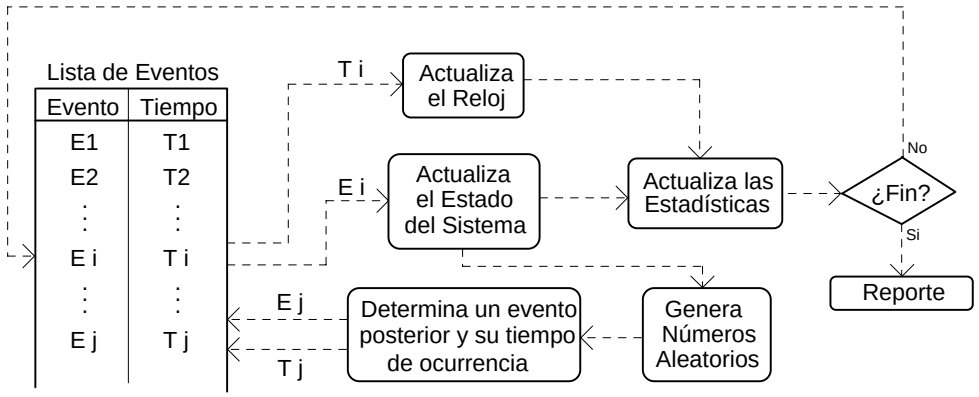

Figura 1. Flujo de una simulación DES

Por ejemplo, durante la simulación de un concentrador de datos en una red de transmisión de paquetes, la lista de eventos consistirá en dos eventos futuros: la llegada del próximo paquete y el fin de transmisión del paquete que se está atendiendo, si hay alguno. Cada vez que ocurre una llegada o un fin de transmisión, el simulador determina cuánto tiempo transcurrirá hasta la próxima vez que este tipo de evento vuelva a suceder, le suma el valor actual del reloj de simulación y almacena esta suma en la lista de eventos futuros. Así, para determinar cuál será el próximo evento, el simulador simplemente debe buscar el mínimo de los tiempos almacenados en la lista[1].

Como se puede observar en la figura 1, muchos de los algoritmos que se deben implementar son comunes para cualquier simulación DES, independientemente del modelo particular que se esté evaluando. Los cuatro problemas básicos que se deben solucionar son la sucesión de eventos adecuadamente ordenada en el tiempo, la generación de variables aleatorias, la adquisición de estadísticas y, en el caso específico de modelos para redes de comunicaciones, el manejo de buffers para administrar las demandas por los recursos[1].

2.1 Control del Avance del Tiempo

Respecto al primer problema, el algoritmo queda planteado en la Figura 1 y constituye el corazón de cualquier programa de simulación DES. Una herramienta de software que implemente este algoritmo deberá disponer de una LEF con algoritmos asociados que permitan la introducción de eventos futuros ordenados de acuerdo con su tiempo de ocurrencia y una rutina principal que extraiga de la LEF el evento más próximo, actualice el reloj de simulación y ejecute la rutina correspondiente al evento seleccionado. La interface con el usuario deberá incluir dos funciones básicas: Activar_Evento, que le permite programar un evento futuro para su posterior ejecución, e Iniciar_Simulación, con la que entrega el control a la rutina principal que desarrolla el algoritmo DES. Cuando el programa retorne de esta rutina, ya la simulación habrá terminado.

Obsérvese cómo, de esta manera, el analista sólo debe preocuparse por definir cuáles son los eventos en su modelo y desarrollar una rutina para cada evento. En el programa principal, dará inicio a la simulación después de activar los primeros eventos, que suelen ser generadores de tráfico. El analista nunca deberá llamar las rutinas de cada evento desde su programa, pues de eso se encargará la rutina de control de tiempo de acuerdo con la secuenciación temporal de los eventos.

El listado 1 muestra las declaraciones de una clase Evento y una clase Lista_de_Eventos, las cuales implementan el algoritmo DES en C++.

```
class Evento                                    // Unidad básica de ejecución en una simulación DES
{
  private:
     friend class Lista_de_Eventos;
     int          Tipo_de_Evento;               // El usuario define los tipos de eventos
     double       Tiempo_de_Activacion;         // Próximo instante en que debe ejecutarse
     Evento       *Siguiente_Evento,            // En la lista de eventos futuros
                  *Evento_Anterior;
     void         Rutina();                     // Rutina donde se realiza el evento
  public:
     Evento(int Tipo);                          // Constructor
};

class Lista_de_Eventos
{
   private:
      Evento *Primero;                          // Primer evento en la lista de eventos futuros
   public:
      Lista_de_Eventos(void)                    // Constructor : La lista empieza vacía
      {  Primero = 0; }
      void Inicia_Simulacion(void);
      void Activa_Evento(Evento *Evento_Activado, double Tiempo_Activacion);
};
```

Listado 1. Declaración de las clases Evento y Lista_de_Eventos

Un Evento tiene un *Tipo_de_Evento* definido por el usuario de acuerdo con su modelo de simulación, un *Tiempo_de_Activación* que corresponde al instante futuro en que se va a ejecutar, unos punteros que lo ubican dentro de la lista de eventos futuros y una *Rutina* donde el usuario incluye los cambios de estado que se producirán en el sistema con la ejecución de este evento, incluyendo la posible activación de otros eventos futuros. Esta rutina es privada puesto que sólo puede ser llamada por el programa de control de tiempo.

El programa de control de tiempo se implementa en el método *Inicia_Simulación* de la clase Lista_de_Eventos. Sólo debe haber un objeto de esta clase, el cual contiene, además del método *Inicia_Simulación*, un puntero al primer evento de la lista y un método *Activa_Evento* para introducir eventos en la lista, en orden cronológico de ejecución. Gracias a que los eventos se guardan debidamente ordenados en la lista de eventos futuros, el próximo evento en ejecutarse siempre será el primer evento de la lista. Cuando la lista esté vacía, termina la ejecución de la función *Inicia_Simulación* y, con ella, termina la simulación completa. El listado 2 muestra la implementación del método *Inicia_Simulacion*.

```
void Lista_de_Eventos::Inicia_Simulacion(void)   // Esta rutina es el corazón de un algoritmo
{                                                // de simulación de eventos discretos:
  Evento *Eptr;
  while(Primero)                                 // Mientras la lista  no esté vacía
  {
     Eptr      = Primero;                        //     Retira el evento más próximo
     Primero   = Primero->Siguiente_Evento;
     if(Primero) Primero->Evento_Anterior = 0;
     Reloj     = Eptr->Tiempo_de_Activacion;    //     Actualiza el reloj de simulación
     (*Eptr).Rutina();                          //     Evalúa el efecto de este evento
  }                                              // Repite hasta agotar la lista
}
```

Listado 2. Implementación del método *Inicia_Simulación* : Algoritmo DES

2.2 Generación de Variables Aleatorias

Pocos fenómenos resultan tan determinísticos como la ejecución de un programa por parte de un computador digital, de manera que no es posible hacer que un computador genere números realmente aleatorios. Sin embargo se han desarrollado algunos métodos numéricos que permiten generar secuencias "pseudoaleatorias" en el sentido de que satisfacen algunas pruebas estadísticas y de que el siguiente número de la secuencia resulta impredecible para quien no conoce los parámetros del método de generación[4]. En general, se busca que estos números estén uniformemente distribuidos en el intervalo [0,1] pues, en principio, se pueden tomar muestras de variables aleatorias con cualquier distribución invirtiendo la respectiva función de distribución acumulativa[3]. En particular, tratándose de generadores pseudo-aleatorios para simulación, se desean las siguientes características[4] :

- Los números generados deben estar uniformemente distribuidos en el intervalo [0,1] y no debe existir ninguna correlación entre ellos.
- El algoritmo debe ser rápido y debe requerir poca memoria.
- El analista debe ser capaz de reproducir con exactitud las secuencias de números obtenidas con anterioridad para poder comparar objetivamente diferentes alternativas de diseño.
- Debe ser posible generar diferentes secuencias simultáneamente, para asignarlas a diferentes fuentes de aleatoriedad en el modelo.

El método más popular que permite satisfacer adecuadamente los cuatro requisitos anteriores es el *Lineal Congruencial*, en el que se obtiene una secuencia de números enteros Z_1, Z_2, ... mediante la relación recursiva[4]

$$Z_i = (aZ_{i-1} + c) \bmod m \qquad (1)$$

donde el módulo *m*, el multiplicador *a*, el incremento *c* y la semilla Z_0 son enteros no negativos. Puesto que Z_i es el residuo de una división por *m*, el número $U_i = Z_i / m$ caerá en el intervalo [0,1). Los parámetros *m*, *a*, *c* y Z_0 se deben escoger cuidadosamente para que los números U_i parezcan muestras de variables aleatorias independientes y uniformemente distribuidas en el intervalo [0,1].

La mayoría de compiladores del lenguaje C++ incluyen en la librería estándar *stdlib*.h una función *rand()* que retorna un número entero pseudo-aletarorio obtenido mediante un generador congruencial con excelentes características estadísticas, así como una función *srand()* para inicializar la semilla. Sin embargo las versiones más populares para computadores personales retornan un número entero entre 0 y 32767, a pesar de que utilizan 32 bits para los parámetros y los cálculos de la relación recursiva (1)[2]. Así pues, los U_i que se obtengan mediante esta función de librería sólo podrán tomar los valores *n*/32768 para *n* entero y no podrán obtenerse valores intermedios entre (*n*-1)/32768 y *n*/32768, a pesar de que la probabilidad de que un número realmente aleatorio caiga en dicho intervalo es $3x10^{-5}$. Esta magnitud puede ser mayor que la probabilidad de bloqueo en alguna central de conmutación, o la fracción de celdas descartadas en una red ATM o la probabilidad de error en una transmisión, etc. Por esta razón, un tipo abstracto de datos que debe incluirse dentro de una librería para simulación es un generador de números aleatorios, con métodos asociados para generar muestras de otras variables aleatorias, y que satisfaga los cuatro criterios previamente mencionados. El listado 3 declara una clase en C++ que cumple con estos requerimientos, basado en el generador UNIRAN de Marse y Roberts y adaptado por Law y Kelton[4].

Obsérvese que el multiplicador se encuentra factorizado y que existe una método privado para operar con variables de tipo *long*. Esto debe hacerse así para controlar el sobreflujo que producen las operaciones de multiplicación, división y residuo cuando se trabaja con números cuya magnitud está en el límite de la representación numérica interna de la CPU (32 bits en el caso de la mayoría de los PCs). Se dispone de diez fuentes independientes de números aleatorios, cuyas semillas se pueden leer o escribir mediante los métodos *Lee_Semilla* e *Inicializa_Semilla*, lo cual garantiza la reproducibilidad de cada secuencia generada. A partir de la función *Aleatorio*, que retorna una muestra independiente de una variable aleatoria uniformemente distribuida entre 0 y 1, se construyen diferentes funciones para obtener muestras de algunas de las principales distribuciones que podrían necesitarse en la simulación de redes de comunicaciones.

```
class Generador_Aleatorio
{
  private:
     long Modulo;
     long Multi1, Multi2;
     long semillas[10];
     long OperacionBasica(long z, long mul);
 public:
     Generador_Aleatorio();
     double Aleatorio(int fuente);
     void   InicializaSemilla(int fuente, long semilla);
     long   LeeSemilla(int fuente);
     double UniformeContinua(int fuente, double inferior, double superior);
     double Exponencial(int fuente, double media);
     double Estandar(int fuente);
     double Normal(int fuente, double media, double varianza);
     double Lognormal(int fuente, double media, double varianza);
     double mErlang(int fuente, double media, int Etapas);
     int    Bernoulli(int fuente, double probabilidad);
     int    UniformeDiscreta(int fuente, int inferior, int superior);
     int    Binomial(int fuente, int Repeticiones, double probabilidad);
     int    Geometrica(int fuente, double probabilidad);
     int    Poisson(int fuente, double media);
};
```

Listado 3. Declaración de una clase para generación de muestras de variables aleatorias

2.3 Adquisición de Estadísticas

Existen dos tipos de medidas de desempeño sobre las cuales se querrían estimar el promedio, la varianza, el máximo, el mínimo, el número de ocurrencias, etc. Las primeras son de parámetro discreto y las segundas son de parámetro continuo[1]. El retardo de los paquetes, por ejemplo, tendrá que acumularse para un número entero de paquetes, mientras que la longitud de un buffer en un conmutador será función continua del tiempo. Si w_i es el retardo del *i*-ésimo paquete y $Q(t)$ es la longitud del buffer en el instante *t*, los respectivos promedios podrán estimarse mediante las siguientes relaciones[1]:

$$E[W] \approx \overline{W} = \frac{1}{N(T)} \sum_{n=0}^{N(T)} w_n \qquad\qquad E[Q] \approx \overline{Q} = \frac{1}{T} \int_0^T Q(t)dt$$

donde se está considerando el intervalo de tiempo [0,T], durante el cual llegaron *N(T)* paquetes. Otras expresiones semejantes podrían encontrarse para otras estadísticas, las cuales deberán evaluarse numéricamente durante la simulación. Este problema puede consumir mucho esfuerzo de programación al desarrollar modelos de simulación en lenguajes de propósito general, por lo que la librería de tipos abstractos de datos para simulación en C++ deberá incluir clases especializadas en la adquisición de estadísticas discretas y continuas. El listado 4 muestra la declaración de dichas clases.

```
class Estadisticas_Discretas
{
  private:
     long   Numero;
     double Promedio, Varianza;
     double Maximo, Minimo;
  public:
     Estadisticas_Discretas();
     double Actualiza(double x);
     double Promedio();
     double Varianza();
     double Maximo();
     double Minimo();
     long   Numero();
     void   Reinicializa();
};
```

```
class Estadisticas_Continuas
{
  private:
     double Ultimo_Instante;
     double Valor_Anterior;
     double Promedio, Varianza;
     double Maximo, Minimo;
  public:
     Estadisticas_Continuas();
     double Actualiza(double x);
     double Promedio();
     double Varianza();
     double Maximo();
     double Minimo();
     void   Reinicializa();
};
```

Listado 4. Declaración de clases para adquisición de estadísticas

Al definir objetos de estas clases, el usuario sólo debe llamar la rutina *Actualiza* cada vez que la variable cambie de valor, y automáticamente se estarán actualizando sus estadísticas, las cuales se podrán leer con las respectivas funciones miembro *Promedio, Varianza, Máximo, Mínimo* y, en el caso de las variables discretas, *Número* de asignaciones. Por supuesto, el usuario siempre tendrá la opción de reinicializar las estadísticas ya sea para ignorar estados transientes, o para evaluar varias réplicas de un mismo modelo durante una sola ejecución del programa, o para cualquier otro propósito que estime conveniente[5].

2.4 Colas de Procesos en Espera de Recursos

Uno de los aspectos que más esfuerzo de programación requieren en la evaluación de modelos de simulación de eventos discretos es el mantenimiento de las colas que se forman cuando los procesos no encuentran disponibles los recursos que requieren (jobs en espera de una CPU que los ejecute, paquetes en espera de un enlace que los transmita, celdas ATM en espera de un permiso que les permita ingresar a la red, etc.)[1]. Así pues, una librería de clases para simulación no podría quedar completa sin un tipo abstracto de dato que implemente los procedimientos correspondientes a una cola. En este reporte se incluyen una clase que ofrece un servicio mínimo para el modelamiento de un recurso (Listado 5).

De acuerdo con dicho listado, un recurso se compone de una o más unidades de servicio que son requeridas en cantidades variables por algunos procesos, los cuales deben esperar en cola si no encuentran suficientes unidades disponibles. La Figura 2 ilustra este concepto.

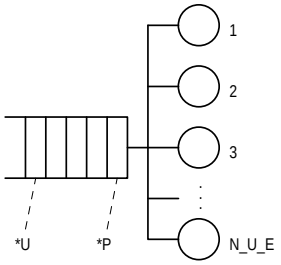

Figura 2. Concepto de Recurso

```
struct Proceso_en_Cola                      // Los procesos en cola son las notificaciones
{                                           // que se hace de los requerimientos de servicio
  double Tiempo_de_Llegada;                 // que no encuentran suficientes unidades disponibles
  int Unidades_Requeridas;                  // Número de unidades que requiere este proceso
  Proceso_en_Cola *Siguiente_Proceso;       // Puntero en la cola del recurso
};

class Recurso                               // Un recurso mantiene una cola de requerimientos
{
  private:
     friend class Evento;
     Proceso_en_Cola *Primero, *Ultimo;     // La cola es una lista encadenada
     Proceso_en_Cola *Siguiente_en_Cola;    // Proceso que salió con la última liberación
     int Fue_Asignado;                      // Indica si el último requerimiento fue servido
     int Unidades_Existentes;               // Número de unidades que componen este recurso
     int Unidades_Libres;                   // Número de unidades disponibles para servicio
     int Procesos_en_Cola;                  // Longitud de la cola
     int Procesos_en_Servicio;              // Número de procesos que reciben servicio
  public:
     Recurso(int Unidades_Existentes);      // Número de unidades que existen de este recurso
     void Requiere(int Unidades_Requeridas);
     void Libera(int Unidades_Liberadas);
};
```

Listado 5. Declaración de una clase para la gestión de colas

Las interacciones con el recurso se hacen mediante las funciones miembro *Requerir* y *Liberar*. Con la primera de ellas se solicita cierta cantidad de unidades del recurso; si está disponible ese número de unidades, se ocupan y se activa la bandera Recurso.Fue_Asignado para que el Evento sepa qué sucedió con su solicitud. Pero si no hay suficientes unidades libres para atender el requerimiento, se crea una notificación de un proceso en cola, en la que se almacena el tiempo de llegada y el número de unidades requeridas. Cuando se liberan algunas unidades del recurso, se hacen disponibles y se verifica si hay alguien en cola esperando por esas unidades liberadas. Si es así, se saca de la cola el primer proceso y se avisa al evento para que tome las medidas pertinentes. Los ejemplos de la siguiente sección ilustrarán mejor este mecanismo de administración de los recursos.

3. Ejemplos de Aplicación

En esta sección se presentan los listados de dos ejemplos sencillos en los que se utiliza la herramienta desarrollada y se evidencia su utilidad en la simplicidad del código. El primer ejemplo corresponde a un multiplexor estadístico modelado como una cola M/M/1 mientras el segundo corresponde a un modelo sencillo de un nodo de conmutación de paquetes.

3.1 Multiplexor Estadístico

Llegan paquetes a una tasa promedio de 10 por segundo según un proceso Poisson y son transmitidos en un tiempo exponencialmente distribuido con promedio 50 ms. Cuando un paquete llega y encuentra el enlace desocupado, lo ocupa durante el tiempo de su transmisión; pero si encuentra el enlace ocupado, el paquete se une a una cola que se atiende en estricto orden de llegada. Este es el típico ejemplo de la cola M/M/1 cuyo programa de simulación se presenta en el listado 6.

Existen dos eventos claramente identificables: La llegada de un nuevo paquete al sistema y la terminación de la transmisión del paquete que se está atendiendo. La variable *Reloj* y la lista de eventos futuros *LEF* se definen en el archivo de implementación de las clases para simulación. Aquí se definen un generador de números aleatorios *RND*, un recurso con una sola unidad llamado *Enlace* y una variable *Retardo* que va a registrar la demora de cada paquete en la cola del multiplexor y a acumular su promedio. Las rutinas donde se evalúan los efectos de cada evento se codifican dentro de la función miembro *Rutina* de la clase *Evento*. En este caso, el evento *LLEGADA* solicita el enlace y el evento *SALIDA* lo libera. La función *Enlace.Requiere* se encarga de poner la solicitud en cola si el enlace no estuviera disponible pero, si lo puede asignar, activa la bandera *Enlace.Fue_Asignado* para que la rutina del evento *LLEGADA* se encargue de activar la próxima salida. En cualquier caso, la próxima llegada se programa dentro de un tiempo exponencialmente distribuido con promedio

0.1 segundos. Por otro lado, la función *Enlace.Libera* se encarga de desocupar el enlace y, si hay alguien en cola dispuesto a ser transmitido, activa la bandera *Enlace.Siguiente_en_Cola*, con la cual la rutina del evento *SALIDA* decide volver a ocupar el enlace y programar la salida del paquete que acabó de salir de cola. Cada vez que un paquete inicia su transmisión (ya sea en *LLEGADA* porque encontró en enlace desocupado o en *SALIDA* porque sacó el primer paquete de la cola) se actualizan las estadísticas del retardo mediante la función *Retardo.Actualiza*, la cual se llama con el tiempo que este paquete permaneció en la cola, *Reloj - Tiempo_de_Llegada*. En el programa principal basta con inicializar el reloj, programar la primera llegada y dar comienzo a la simulación. Cuando el control retorne de la función *LEF.Inicia_Simulación*, ya habrá terminado la simulación y se podrá imprimir el resultado, *Retardo.Promedio*.

En el programa del listado 6 queda clara la funcionalidad de la herramienta desarrollada, pues no fue necesario incluir ninguna línea de código para controlar el avance del tiempo, ni para generar muestras de variables aleatorias, ni para adquirir el promedio del retardo ni para administrar la cola del enlace. Todas estas funciones ya están implementadas en las clases *Evento*, *Lista_de_Eventos*, *Generador_Aleatorio* y *Recurso*.

```
#include <stdlib.h>
#include <iostream.h>
#include "simulaci.h"                          // Librería de clases para simulación

#define LLEGADA 1                              // Tipos de eventos
#define SALIDA  2

extern double              Reloj;              // Reloj de simulación
extern Lista_de_Eventos    LEF;                // Lista de Eventos Futuros (Control de Tiempo)

Generador_Aleatorio        RND;                // Genera tiempos entre llegadas y de servicio
Recurso                    Enlace(1);          // Hay un enlace para la  transmisión de los paquetes
Estadisticas_Discretas     Retardo;            // Acumula el promedio del retardo de los paquetes en cola

void Evento::Rutina(void)                      // Método de la clase Evento proporcionado por
{                                              // el usuario. Debe haber un procedimiento por
  switch(Tipo_de_Evento)                       // cada tipo de evento:
  {
    case LLEGADA :
         {
           Enlace.Requiere(1);                 // Solicita el enlace
           if(Enlace.Fue_Asignado)             // Si se lo asignaron, programa la salida del paquete
           {
             LEF.Activa_Evento(new Evento(SALIDA),RND.Exponencial(1,0.05));
             Retardo.Actualiza(0);             // Este paquete no pasó por la cola
           }                                   // Programa la siguiente llegada (Proceso Poisson)
           if(Reloj<200) LEF.Activa_Evento(new Evento(LLEGADA),RND.Exponencial(0,0.1));
           break;
         }
    case SALIDA :                              // El evento Salida libera el enlace. Pero si
         {                                     // hay paquetes en cola, hace que el primero de
            Enlace.Libera(1);                  // ellos inicie su transmisión
            if(Enlace.Siguiente_en_Cola)
            {                                  // Tiempo que tardó este paquete en cola
              Retardo.Actualiza(Reloj - Enlace.Siguiente_en_Cola->Tiempo_de_Llegada);
              Enlace.Requiere(1);              // Vuelve a ocuparse el enlace : Programa la salida
              LEF.Activa_Evento(new Evento(SALIDA),RND.Exponencial(1,0.05));
            }
            break;
         }
  }
}
```

Listado 6. Programa de Simulación de un Multiplexor Estadístico (continúa)

```
void main(void)
{
  Reloj = 0.0;                                 // Inicializa el reloj de simulación
  LEF.Activa_Evento(new Evento(LLEGADA),0.0);   // Activa la primera llegada
  LEF.Inicia_Simulacion();                     // Da inicio a la simulación
```

```
  cout << Retardo.Promedio();                    // Reporta los resultados
}
```

Listado 6. Programa de Simulación de un Multiplexor Estadístico (continuación)

3.2 Nodo de Conmutación de Paquetes

Los paquetes llegan a un nodo de conmutación según un proceso Poisson con una tasa promedio de 40 por segundo, y allí son atendidos por el software de enrutamiento, quien decide por cual de los dos enlaces de salida debe transmitirse el paquete (Figura 3). El enrutador es capaz de atender 60 paquetes por segundo, con un tiempo de enrutamiento constante para cada paquete. En promedio, el 60% de los paquetes se dirige al primer enlace, el cual tiene una capacidad de 30 paquetes/segundo, y el 40% restante se dirige al segundo enlace que tiene una capacidad de 20 paquetes/segundo, siendo los tiempos de transmisión variables aleatorias exponencialmente distribuidas. Se desea medir el número promedio de paquetes que permanecen en el nodo, incluyendo los que se están transmitiendo.

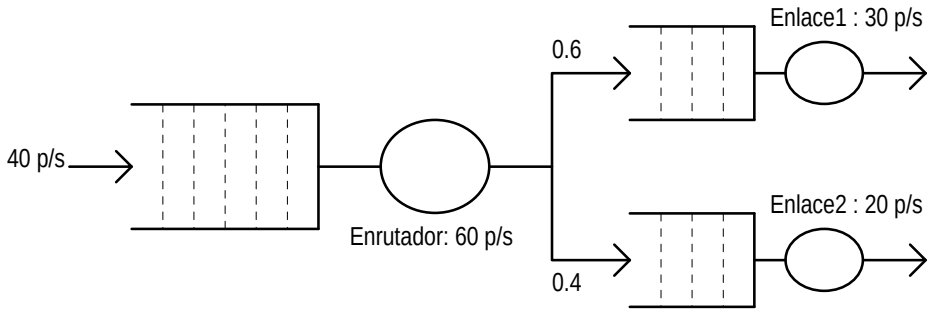

Figura 3. Modelo de un nodo de conmutación

```
#include <stdlib.h>
#include <iostream.h>
#include "simulaci.h"                          // Librería de clases para simulación

#define LLEGADA_ENRUTADOR 1                    // Tipos de eventos
#define SALIDA_ENRUTADOR  2
#define SALIDA_ENLACE1    3
#define SALIDA_ENLACE2     4

extern double             Reloj;               // Reloj de simulación
extern Lista_de_Eventos   LEF;                 // Objeto de Control del Tiempo de Simulación

Generador_Aleatorio       RND;                 // Generador de tiempos entre llegadas y de servicio
Recurso                   Enrutador(1),        // Recursos por lo que compiten los paquetes
                          Enlace1(1),
                          Enlace2(1);
Estadisticas_Continuas    NumPaquetes;         // Acumula el promedio del numero de paquetes en el nodo
long                      Numero_de_Paquetes = 0;// Mantiene la cuenta de cuántos paquetes hay en el nodo

void Evento::Rutina(void)                      // Método de la clase Evento proporcionado por el usuario
{
  switch(Tipo_de_Evento)
  {
    case LLEGADA_ENRUTADOR :                   // Mientras dure la simulación, activa otra llegada con tiempos
         {                                     // entre llegadas exponencialmente distribuidos
           if(Reloj<250)
              LEF.Activa_Evento(new Evento(LLEGADA_ENRUTADOR),RND.Exponencial(0,1.0/40.0));
```

Listado 7. Programa de Simulación de un Nodo de Conmutación de Paquetes (continúa)

```
           NumPaquetes.Actualiza(++Numero_de_Paquetes);  // Hay un paquete más en el sistema
           Enrutador.Requiere(1);              // Solicita servicio del enrutador
           if(Enrutador.Fue_Asignado)          // Programa el tiempo en que liberará este recurso
              LEF.Activa_Evento(new Evento(SALIDA_ENRUTADOR),1.0/60.0);
           break;
         }
```

```
    case SALIDA_ENRUTADOR :
          {
             Enrutador.Libera(1);                  // Libera el enrutador
             if(Enrutador.Siguiente_en_Cola)       // y se lo asigna al primero de la cola (si hay alguien)
             {
               Enrutador.Requiere(1);
               LEF.Activa_Evento(new Evento(SALIDA_ENRUTADOR),1.0/60.0);
             }
             double x = RND.Aleatorio(2);          // Decide por donde enrutar este paquete
             if(x<0.6)
             {
               Enlace1.Requiere(1);
               if(Enlace1.Fue_Asignado)
                  LEF.Activa_Evento(new Evento(SALIDA_ENLACE1),RND.Exponencial(3,1.0/30.0));
             }
             else
             {
               Enlace2.Requiere(1);
               if(Enlace2.Fue_Asignado)
                  LEF.Activa_Evento(new Evento(SALIDA_ENLACE2),RND.Exponencial(4,1.0/20.0));
             }
             break;
          }
    case SALIDA_ENLACE1 :
          {
             NumPaquetes.Actualiza(--Numero_de_Paquetes);    // Hay un paquete menos en el sistema
             Enlace1.Libera(1);                    // Libera este enlace y, si hay alguien en cola,
             if(Enlace1.Siguiente_en_Cola)         // lo asigna al primero de la cola
             {
               Enlace1.Requiere(1);
               LEF.Activa_Evento(new Evento(SALIDA_ENLACE1),RND.Exponencial(3,1.0/30.0));
             }
             break;
          }
    case SALIDA_ENLACE2 :
          {
             NumPaquetes.Actualiza(--Numero_de_Paquetes);    // Hay un paquete menos en el sistema
             Enlace2.Libera(1);                    // Libera este enlace y, si hay alguien en cola,
             if(Enlace2.Siguiente_en_Cola)         // lo asigna al primero de la cola
             {
               Enlace2.Requiere(1);
               LEF.Activa_Evento(new Evento(SALIDA_ENLACE2),RND.Exponencial(4,1.0/20.0));
             }
             break;
          }
  }
}

void main(void)
{
  Reloj = 0.0;
  LEF.Activa_Evento(new Evento(LLEGADA_ENRUTADOR),0.0);
  LEF.Inicia_Simulacion();
  cout << "Permanecieron " << NumPaquetes.Promedio() << " paquetes en promedio\n";
  cout << "Cada paquete se tardó " << NumPaquetes.Promedio()/0.040 << " ms en promedio\n";
}
```

Listado 7. Programa de Simulación de un Nodo de Conmutación de Paquetes (continuación)

4. Conclusiones

Se ha reportado en este artículo una mínima librería de clases en C++ para facilitar el desarrollo de programas de simulación de eventos discretos. Se ha verificado cómo el esfuerzo de programación se ve muy reducido cuando se cuenta con una herramienta que realice la gestión de la lista de eventos futuros, incluyendo el algoritmo DES para el control del avance del tiempo, así como la generación de muestras de variables aleatorias, la adquisición de estadísticas y el manejo de las colas que se forman en la competencia por los recursos. Contando con dichas facilidades, resulta muy ventajoso desarrollar los programas de simulación en un lenguaje conocido y ampliamente documentado, como lo es el lenguaje C++.

Por supuesto, la herramienta presentada aquí es muy incompleta, siendo susceptible de una gran cantidad de mejoras. En primer lugar, desde la misma metodología de desarrollo, debería facilitarse la elaboración de nuevos tipos abstractos de datos que implementen conceptos de polimorfismo y herencia para aprovechar las ventajas de reutilización que ofrece la programación orientada a objetos. Así, por ejemplo, resulta inflexible, además de poco elegante, que las rutinas de cada evento se codifiquen dentro del método *Rutina* de la clase *Evento*, cuando el usuario podría definir sus propias clases que hereden de la clase *Evento*. De la misma manera, la estructura *Proceso_en_Cola* es demasiado rígida, en la medida en que el usuario podría querer almacenar otros parámetros importantes de acuerdo con el modelo particular que esté evaluando.

Los listados completos de los archivos *simulaci.h*, que declara las clases mencionadas, y *simulaci.cpp*, que las implementa, se muestran en los anexos. La esperanza es que el lector los encuentre suficientemente útiles o, al menos, suficientemente claros como para poderles hacer los cambios que considere necesarios para satisfacer sus necesidades particulares de simulación.

Bibliografía

[1] ALZATE , M. “Simulación de Redes de Computadores” Universidad Distrital F.J.C., 1992

[2] BARKAKATI, N. and HIPSON, P. “Visual C++ Developer’s Guide”, Sams publishing, 1993

[3] BULGREN, W. G. “Discrete System Simulation”, Prentice Hall, 1982

[4] LAW, A. and KELTON, W. “Simulation Modeling & Analysis”, 2nd Ed. McGraw-Hill, 1991.

[5] LAW, A. and McCOMAS, M. “Simulation Software for Communications Networks”, IEEE Communications Magazine, Marzo de 1994.

[6] ZEIGLER, B., GARZIA, M. and GARZIA, R. “Discrete-Event Simulation”, IEEE Spectrum, Diciembre de 1986.

[LISTADO UNO : SIMULACI.H]

```
/*************************************************************************************************************
 SIMULACI.H : Archivo de encabezado que declara algunas clases útiles para el desarrollo de programas de simulación de eventos
              discretos en C++. Marco Aurelio Alzate Monroy. Universidad Distrital. Marzo de 1996
*************************************************************************************************************/

/**************************  Clases Para la Realización del Algoritmo DES  ***********************************/
class Evento                                    // Unidad básica de ejecución en una simulación DES
{
  private:
     friend class Lista_de_Eventos;             // Lista de eventos futuros para control del avance del tiempo
     int          Tipo_de_Evento;               // El usuario define los tipos de eventos
     double       Tiempo_de_Activacion;         // Próximo instante en que debe ejecutarse este evento
     Evento       *Siguiente_Evento,            // Punteros para la lista de eventos futuros
                  *Evento_Anterior;
     void         Rutina();                     // Rutina donde se realiza el evento, definida por el usuario
  public:
     Evento(int Tipo);                          // Constructor de objetos Evento
};

class Lista_de_Eventos
{
   private:
      Evento *Primero;                          // Primer evento en la lista de eventos futuros
   public:
      Lista_de_Eventos(void)                    // Constructor : La lista empieza vacía
      {  Primero = 0; }
      void Inicia_Simulacion(void);             // Algoritmo de simulación DES
      void Activa_Evento(Evento *Evento_Activado,       // Introduce un Evento en la Lista de Eventos Futuros
                         double Tiempo_Activacion);
};

/**************************  Clases Para la Obtención de Muestras de Variables Aleatorias  ********************/
class Generador_Aleatorio
{
  private:
     long Modulo;                               // Módulo del generador congruencial multiplicativo
     long Multi1, Multi2;                       // Multiplicador, factorizado para control de sobreflujos
     long semillas[10];                         // Arreglo de 10 semillas para 10 fuentes independientes
     long OperacionBasica(long z, long mul);// Operación básica de multiplicación con control de sobreflujo
 public:
     Generador_Aleatorio();                     // Constructor : Inicializa parámetros
     long   LeeSemilla(       int fuente);  // Obtiene el valor de la semilla de una fuente
     void   InicializaSemilla(int fuente, long   semilla);
     /*****************  Muestras de Algunas Distribuciones Continuas  *************************/
     double Aleatorio(        int fuente);
     double Estandar(         int fuente);
     double Exponencial(      int fuente, double media);
     double UniformeContinua( int fuente, double inferior,     double superior);
     double Normal(           int fuente, double media,        double varianza);
     double Lognormal(        int fuente, double media,        double varianza);
     double mErlang(          int fuente, double media,        int    Etapas);
     /*****************  Muestras de Algunas Distribuciones Discretas  *************************/
     int    Poisson(          int fuente, double media);
     int    Bernoulli(        int fuente, double probabilidad);
     int    Geometrica(       int fuente, double probabilidad);
     int    UniformeDiscreta( int fuente, int    inferior,     int    superior);
     int    Binomial(         int fuente, int    Repeticiones, double probabilidad);
};

/**************************  Clases para la adquisición de estadísticas temoporales  ************************/
class Estadisticas_Discretas
{
  private:
     long   _Numero;                            // Número de elementos que se han considerado
     double _Promedio, Promedio2;               // Valor esperadode la variable y de su cuadrado
     double _Maximo, _Minimo;                   // Valores Máximo y Mínimo que ha tomado la variable
  public:
```

```
      Estadisticas_Discretas();              // Constructor de objetos
      double Actualiza(double x);            // Actualiza las estadísticas según este nuevo valor
      double Promedio();                     // Retorna el Promedio
      double Varianza();                     // Retorna la Varianza
      double Maximo();                       // Retorna el Máximo
      double Minimo();                       // Retorna el Mínimo
      long   Numero();                       // Retorna el número de elementos considerados
      void   Reinicializa();                 // Reinicializa todos los contadores
};
class Estadisticas_Continuas
{
  private:
      double Ultimo_Instante;                // Ultimo instante en el que se dió un cambio en el valor de la variable
      double Valor_Anterior;                 // Valor que tenía la variable antes del cambio actual
      double _Promedio, Promedio2;           // Valor esperadode la variable y de su cuadrado
      double _Maximo, _Minimo;               // Valores Máximo y Mínimo que ha tomado la variable
  public:
      Estadisticas_Continuas();              // Constructor de objetos
      double Actualiza(double x);            // Actualiza las estadísticas según este nuevo valor
      double Promedio();                     // Retorna el Promedio
      double Varianza();                     // Retorna la Varianza
      double Maximo();                       // Retorna el Máximo
      double Minimo();                       // Retorna el Mínimo
      void   Reinicializa();                 // Reinicializa todos los contadores
};

/*******************************   Clases para la Gestión de Colas   ***************************************/
struct Proceso_en_Cola                       // Los procesos en cola son las notificaciones
{                                            // que se hace de los requerimientos de servicio
  double Tiempo_de_Llegada;                  // que no encuentran suficientes unidades disponibles
  int Unidades_Requeridas;                   // Número de unidades que requiere este proceso
  Proceso_en_Cola *Siguiente_Proceso;        // Puntero en la cola del recurso
};
class Recurso                                // Un recurso mantiene una cola de requerimientos
{
  private:
      friend class Evento;
      Proceso_en_Cola *Primero, *Ultimo;     // La cola es una lista encadenada
      Proceso_en_Cola *Siguiente_en_Cola;    // Proceso que salió con la última liberación
      int Fue_Asignado;                      // Indica si el último requerimiento fue servido
      int Unidades_Existentes;               // Número de unidades que componen este recurso
      int Unidades_Libres;                   // Número de unidades disponibles para servicio
      int Procesos_en_Cola;                  // Longitud de la cola
      int Procesos_en_Servicio;              // Número de procesos que reciben servicio
  public:
      Recurso(int Unidades_Existentes);      // Número de unidades que existen de este recurso
      void Requiere(int Unidades_Requeridas);
      void Libera(int Unidades_Liberadas);
};
```

[LISTADO DOS : SIMULACI.CPP]

```
/*****************************************************************************************************************
  SIMULACI.CPP : Archivo de implementacion de las clases declaradas en SIMULACI.H para el desarrollo de programas de
                 simulacion de eventos discretos en C++.  Marco Aurelio Alzate Monroy. Universidad Distrital. Marzo de 1996
*****************************************************************************************************************/
#include <stdlib.h>
#include <math.h>
#include "simulaci.h"

/**********************/
/*   Variables globales   */
/**********************/
double              Reloj;                  // Reloj de Simulación
Lista_de_Eventos    LEF;                    // Lista de Eventos Futuros

/*******************************/
/*    Métodos de la clase Evento    */
/*******************************/
Evento::Evento(int Tipo)                    // Constructor de un objeto evento:
{                                           // Inicializa las variables
   Siguiente_Evento     = 0;
   Evento_Anterior      = 0;
   Tiempo_de_Activacion = 1.7E+308;
   Tipo_de_Evento       = Tipo;
}
// El método Evento::Rutina lo define el usuario de acuerdo con su modelo de simulación

/***************************************/
/*     Métodos de la clase Lista_de_Eventos      */
/***************************************/
void Lista_de_Eventos::Activa_Evento(Evento *Eptr, double TA) // Debe incluirse un nuevo evento en la
{                                           // lista, ordenado por tiempo de activación
   Evento *Temporal1, *Temporal2;
   double TA1, TA2;                         // Tiempos de activación a comparar
   int Bandera = 0;                         // Indica que se debe seguir buscando
   Eptr->Tiempo_de_Activacion = Reloj + TA; // calcula su próximo tiempo de ejecución
   if(!Primero)                             // Si la lista esta vacía, este es el
   {                                        // primer evento
      Primero = Eptr;
      Eptr->Siguiente_Evento = 0;
      Eptr->Evento_Anterior  = 0;
      return;
   }
   Temporal1 = Primero;                     // En otro caso, debe comparar los tiempos
   TA1 = Eptr->Tiempo_de_Activacion;        // de activación
   while(Bandera==0)
   {
     TA2 = Temporal1->Tiempo_de_Activacion;
     if(TA2 > TA1) Bandera = 1;             // Si encuentra un evento con tiempo de activación
     else                                   // mayor, debe insertarse aqui el nuevo evento
     {
       if(Temporal1->Siguiente_Evento==0) Bandera=2;
       else Temporal1=Temporal1->Siguiente_Evento;
     }
   }
   if(Bandera==1)                           // El tiempo de activación de Temporal1 es
   {                                        // mayor que el del evento en cuestión, de
     if(Temporal1==Primero)                 // manera que debe insertarse antes de Temporal1.
     {                                      // Sin embargo, insertar antes del primer
       Primero = Eptr;                      // evento es distinto que insertar entre
       Eptr->Siguiente_Evento = Temporal1;  // dos eventos. En el primer caso, el nuevo
       Eptr->Evento_Anterior = 0;           // evento debe ser el primero de la lista
       Temporal1->Evento_Anterior = Eptr;
     }
     else
     {
       Temporal2 = Temporal1->Evento_Anterior;  // En el segundo caso, deben ajustarse los
       Temporal2->Siguiente_Evento = Eptr;  // apuntadores de los eventos Temporal1 y
       Eptr->Evento_Anterior = Temporal2;   // el evento anterior de Temporal1 para que
       Eptr->Siguiente_Evento = Temporal1;  // el nuevo evento quede insertado entre
```

```
        Temporal1->Evento_Anterior = Eptr;    // ellos dos.
      }
    }
    else                                      // Si ninguno de los eventos de la lista
    {                                         // tiene un tiempo de activacion posterior
      Temporal1->Siguiente_Evento = Eptr;     // al del nuevo evento, el nuevo evento se
      Eptr->Evento_Anterior = Temporal1;      // añade al final de la lista.
      Eptr->Siguiente_Evento = 0;
    }
}

void Lista_de_Eventos::Inicia_Simulacion(void)   // Esta rutina es el corazon de un algoritmo
{                                         // de simulacion de eventos discretos:
  Evento *Eptr;
  while(Primero)                          // Mientras la lista no esté vacía
  {
     Eptr = Primero;                      //     Retira el evento más próximo
     Primero = Primero->Siguiente_Evento;
     if(Primero) Primero->Evento_Anterior = 0;
     Reloj = Eptr->Tiempo_de_Activacion;  //     Actualiza el reloj de simulación
     (*Eptr).Rutina();                    //     Evalúa el efexcto de este evento
     delete Eptr;                         // y repite hasta agotar la lista
  }
}

/*******************************/
/*    Métodos de la clase Recurso    */
/*******************************/
Recurso::Recurso(int Unidades)            // Constructor de un objeto Recurso: Inicializa las variables
{
  Primero              = 0;
  Ultimo               = 0;
  Siguiente_en_Cola    = 0;
  Unidades_Existentes  = Unidades;
  Unidades_Libres      = Unidades;
  Procesos_en_Cola     = 0;
  Procesos_en_Servicio = 0;
  Fue_Asignado         = 0;
}

void Recurso::Requiere(int Unidades_Requeridas)      // Si hay suficientes unidades libres como para
{                                                    // atender este requerimiento, las asigna y activa
   Proceso_en_Cola *Pproceso;                        // la bandera Recurso.Fue_Asignado. En otro caso
   if(Unidades_Libres < Unidades_Requeridas)         // introduce la solicitud en la cola
   {
     Fue_Asignado = 0;                               // No se pueden asignar recursos
     Pproceso = new Proceso_en_Cola;                 // Una nueva solicitud en cola
     Pproceso->Tiempo_de_Llegada = Reloj;            // Registra el momento de llegada
     Pproceso->Unidades_Requeridas = Unidades_Requeridas;  // y el número de recursos solicitados
     if(Primero) Ultimo->Siguiente_Proceso = Pproceso;// Introduce la solicitud en la cola
     else Primero = Pproceso;
     Ultimo = Pproceso;
     Pproceso->Siguiente_Proceso = 0;
     Procesos_en_Cola++;                             // Un proceso más en cola
   }
   else
   {
     Unidades_Libres -= Unidades_Requeridas;         // Asigna los recursos solicitados
     Procesos_en_Servicio++;                         // Un proceso más en cola
     Fue_Asignado = 1;                               // Indica que se puede iniciar servicio
   }
}

void Recurso::Libera(int Unidades_Liberadas)         // Marca como libres los recursos liberados y,
{                                                    // si con ellos puede atender al primero de la cola,
   Procesos_en_Servicio--;                           // lo retira e indica con la bandera Siguiente_en_Cola
   Unidades_Libres += Unidades_Liberadas;
   if(Siguiente_en_Cola) delete Siguiente_en_Cola;
   Siguiente_en_Cola = 0;
   if(Primero)
   {
     if(Primero->Unidades_Requeridas <= Unidades_Libres)
     {
```

```
        Siguiente_en_Cola = Primero;
        Primero = Primero->Siguiente_Proceso;
        Procesos_en_Cola--;
      }
    }
}

/*********************************************************/
/*        Métodos de las clases para adquisición de estadísticas        */
/*********************************************************/
Estadisticas_Discretas::Estadisticas_Discretas(void)   // Constructor : Inicializa contadores
{
  _Numero   = 0;
  _Promedio = 0.0;
  Promedio2 = 0.0;
  _Maximo   = -1.7E+308;
  _Minimo   = +1.7E+308;
}

double Estadisticas_Discretas::Actualiza(double x)     // Con cada nuevo valor calcula el promedio
{                                                      //     (1/N) * SUMA(valor i)   i = 1,2,...,N
  _Promedio = (_Promedio*_Numero + x)/(_Numero+1);     // así como el promedio de los cuadrados
  Promedio2 = (Promedio2*_Numero + x*x)/(++_Numero);   //   (1/N) * SUMA(valor i)^2   i = 1,2,...,N
  if(x>_Maximo) _Maximo = x;                           // el valor  mínimo y el valor  máximo
  if(x<_Minimo) _Minimo = x;                           // Luago actualiza N incrementaándolo en 1
  return(_Promedio);
}

double Estadisticas_Discretas::Promedio(void)          // Métodos del objeto para obtener los valores
{ return _Promedio; }                                  // de las estadísticas acumuladas
double Estadisticas_Discretas::Varianza(void)
{ return Promedio2 - _Promedio*_Promedio; }
double Estadisticas_Discretas::Maximo(void)
{ return _Maximo; }
double Estadisticas_Discretas::Minimo(void)
{ return _Minimo; }
double Estadisticas_Discretas::Numero(void)
{ return _Numero; }
void Estadisticas_Discretas::Reinicializa(void)        // Reinicializa las estadísticas acumuladas
{
  _Numero   = 0;
  _Promedio = 0.0;
  Promedio2 = 0.0;
  _Maximo   = -1.7E+308;
  _Minimo   = +1.7E+308;
}

Estadisticas_Continuas::Estadisticas_Continuas()       // Constructor : Inicializa contadores
{
  Ultimo_Instante = 0.0;
  _Promedio       = 0.0;
  Promedio2       = 0.0;
  Valor_Anterior  = 0.0;
  _Maximo         = -1.7E+308;
  _Minimo         = +1.7E+308;
}

double Estadisticas_Continuas::Actualiza(double x)  // Con cada nuevo valor actualiza el área bajos las
{                                                   // curvas de x(t) y de x^2(t), y noraliza respecto al Reloj
  if(Reloj) _Promedio = (_Promedio *Ultimo_Instante + Valor_Anterior*
                        (Reloj - Ultimo_Instante))/Reloj;
  if(Reloj) Promedio2= (Promedio2*Ultimo_Instante + Valor_Anterior*
                         Valor_Anterior*(Reloj - Ultimo_Instante))/Reloj;
  Ultimo_Instante = Reloj;
  Valor_Anterior  = x;
  if(x>_Maximo) _Maximo = x;
  if(x<_Minimo) _Minimo = x;
  return(_Promedio);
}
double Estadisticas_Continuas::Promedio(void)          // Retorna las estadísticas de interés
{ return _Promedio; }
double Estadisticas_Continuas::Varianza(void)
{ return Promedio2 - _Promedio*_Promedio; }
double Estadisticas_Continuas::Maximo(void)
```

```
{ return _Maximo; }
double Estadisticas_Continuas::Minimo(void)
{ return _Minimo; }
void Estadisticas_Continuas::Reinicializa(void)
{
  Ultimo_Instante  = Reloj;
  _Promedio        = 0.0;
  Promedio2        = 0.0;
  Valor_Anterior   = 0.0;
  _Maximo          = -1.7E+308;
  _Minimo          = +1.7E+308;
}

/*****************************************/
/*      Métodos de la clase Generador_Aleatorio     */
/*****************************************/
Generador_Aleatorio::Generador_Aleatorio(void)
{
  Modulo = 2147483647;                                    // Modulo primo del generador congruencial
  Multi1 = 24112;                                         // El multiplicador es 630360016
  Multi2 = 26143;
  semillas[0] = 1973272912; semillas[1] = 1511192140;     // Semillas separadas por un millón de muestras
  semillas[2] =  726370533; semillas[3] =  336157058;
  semillas[4] = 2122378830; semillas[5] = 1997049139;
  semillas[6] =  498067494; semillas[7] = 1432404475;
  semillas[8] = 1053920743; semillas[9] =  190641742;
}
long Generador_Aleatorio::OperacionBasica(long z, long mul) // Multiplica z*mul sin producir sobreflujo
{
  long lo,hi;
  lo = (z & 65535) * mul;
  hi = (z >> 16) * mul + (lo >> 16);
  z  = ((lo & 65535) - Modulo) + ((hi & 32767) << 16) + (hi >> 15);
  if(z<0) z += Modulo;
  return z;
}

double Generador_Aleatorio::Aleatorio(int fuente)         // Genera muestras independientes y
{                                                         // uniformemente distribuidas entre 0 y 1
  long z;
  z = semillas[fuente];
  z = OperacionBasica(z, Multi1);
  z = OperacionBasica(z, Multi2);
  semillas[fuente] = z;
  return (double)((z >> 7 | 1) + 1) / 16777216.0;
}

void Generador_Aleatorio::InicializaSemilla(int fuente, long semilla)
{                                                         // Asigna el valor indicado a semillas[fuente]
  semillas[fuente] = semilla;
}

long Generador_Aleatorio::LeeSemilla(int fuente)          // Retorna el valor de la semilla de esta fuente
{
  return semillas[fuente];
}

/********** Los siguientes métodos retornan muestras independientes de variables aleatorias con distintas distribuciones ********/
double Generador_Aleatorio::UniformeContinua(int fuente, double inferior, double superior)
{
  return (Aleatorio(fuente)*(superior-inferior) + inferior);
}

double Generador_Aleatorio::Exponencial(int fuente, double media)
{
  double x;
  do
  {
    x = Aleatorio(fuente);
  } while(!x);
  return(-log(x)*media);
}

double Generador_Aleatorio::Estandar(int fuente)
{
```

```
  double u1, u2, v1, v2, w, y, x1;
  static double x2=0.0;
  static int Llamada = 0;

  Llamada = 1 - Llamada;
  if(Llamada==0) return x2;
  do
  {
    u1 = Aleatorio(fuente); v1 = 2.0*u1 - 1.0;
    u2 = Aleatorio(fuente); v2 = 2.0*u2 - 1.0;
    w  = v1*v1 + v2*v2;
  } while(w>1);
  y  = sqrt((-2.0*log(w))/w);
  x1 = v1*y; x2 = v2*y;
  return x1;
}

double Generador_Aleatorio::Normal(int fuente, double media, double varianza)
{
  return (media + sqrt(varianza)*Estandar(fuente));
}

double Generador_Aleatorio::Lognormal(int fuente, double media, double varianza)
{
  double miu, sigma2, y;
  miu    = log((media*media)/sqrt(varianza + media*media));
  sigma2 = log((varianza + media*media)/(media*media));
  y      = Normal(fuente, miu, sigma2);
  return exp(y);
}

double Generador_Aleatorio::mErlang(int fuente, double media, int Etapas)
{
   double producto = 1.0;
   for(int i=0;i<Etapas;i++) producto *=Aleatorio(fuente);
   return ((-media*log(producto))/((double)Etapas));
}

int Generador_Aleatorio::Bernoulli(int fuente, double probabilidad)
{
  if(Aleatorio(fuente)<=probabilidad) return 1;
  return 0;
}

int Generador_Aleatorio::UniformeDiscreta(int fuente, int inferior, int superior)
{
  return (inferior+(int)floor(Aleatorio(fuente)*(double)(superior-inferior+1)));
}

int Generador_Aleatorio::Binomial(int fuente, int Repeticiones, double probabilidad)
{
  int suma=0;
  for (int i=0; i<Repeticiones; i++) suma += Bernoulli(fuente,probabilidad);
  return suma;
}

int Generador_Aleatorio::Geometrica(int fuente, double probabilidad)
{
  return ((int)floor(log(Aleatorio(fuente))/log(1.0-probabilidad)));
}

int Generador_Aleatorio::Poisson(int fuente, double media)
{
  double a, b;
  int i=0;
  a = exp(-media);
  b = Aleatorio(fuente);
  while(b>=a)
  {
    b *= Aleatorio(fuente);
    i++;
  }
  return i;
}
```