UNIVERSIDAD
DE GRANADA

Simulación de Sistemas
grado en ingeniería informática

Ejercicio 2

Modelo Dinámico Discreto

Autor
Vladislav Nikolov Vasilev

Rama
Computación y Sistemas Inteligentes

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2019-2020

Índice

1. Descripción del problema

Un sistema de colas consta de 2 servidores (A y B) dispuestos en serie. Los clientes que acceden al sistema son primero atendidos por el servidor A. Una vez que acaba la atención en el servidor A, pasan a ser atendidos por el servidor B, y cuando terminan de ser atendidos, salen del sistema. Cada servidor tiene una cola de espera FIFO. Todos los tiempos siguen una distribución exponencial. Para las llegadas, dicha distribución tiene una media de 1 minuto. Para el tiempo de servicio en el servidor A se tiene una media de 0.8 minutos, mientras que para el servidor B se tiene una media de 1.2 minutos.

El objetivo es construir un modelo de simulación (en este caso, un modelo dinámico discreto) el cuál permita simular el sistema anteriormente descrito y que además, permita calcular el tiempo de estancia medio de los clientes en el sistema. Adicionalmente, se quiere determinar hasta cuánto tiempo habría que reducir el tiempo de servicio del servidor B, dejando el mismo tiempo de servicio para el servidor A, para conseguir un tiempo medio de estancia inferior a 10 minutos.

2. Grafo de sucesos

Una vez que hemos hecho la descripción del problema, lo primero que nos interesa hacer es determinar los sucesos relevantes que tenemos que modelizar. Si nos paramos a analizar la secuencia de sucesos que se tienen que seguir para atender a un cliente de principio a fin, nos encontramos con lo siguiente:

1. Lo primero que se produce es la llegada del cliente al sistema. Por tanto, debe existir un suceso que represente dicha llegada al sistema.
2. Una vez que el cliente ha llegado, si el servidor A está libre, se inicia la atención del cliente. En caso contrario, el cliente debe esperar su turno en la cola. Por tanto, necesitamos un suceso que represente el inicio de atención en el servidor A.
3. Ya que hemos decidido modelizar el inicio de la atención en el servidor A, también necesitamos modelizar el final de atención. Puede suceder que una vez que se termine de atender el cliente actual queden otros en la cola, con lo cuál se podría pasar a atender el primero. En caso de no haber ninguno, el servidor quedaría libre, de forma que el siguiente cliente no necesitaría esperar en la cola para poder pasar a ser atendido.
4. Lo siguiente que pasaría sería que el cliente pasa a ser atendido por el servidor B, siempre y cuando este esté libre. En caso de que no lo esté, deberá quedarse

esperando en la cola hasta que le toque. Por tanto, necesitamos un suceso que represente el inicio de atención en el servidor B.

5. Tal y como hicimos antes, necesitaremos un suceso que represente el fin de atención en el servidor B. De aquí, tal y como pasaba antes, puede suceder que se pase a atender el siguiente cliente si hay alguien esperando en la cola o que el servidor se quede libre.

6. Finalmente, una vez que ha finalizado la atención en el servidor B, el cliente sale del sistema. Por tanto, necesitamos un suceso que represente la salida del sistema.

Con los seis sucesos anteriormente descritos, podemos construir el grafo de sucesos. Adicionalmente, necesitaríamos un suceso extra que genere la primera llegada. El grafo resultante se puede ver a continuación:

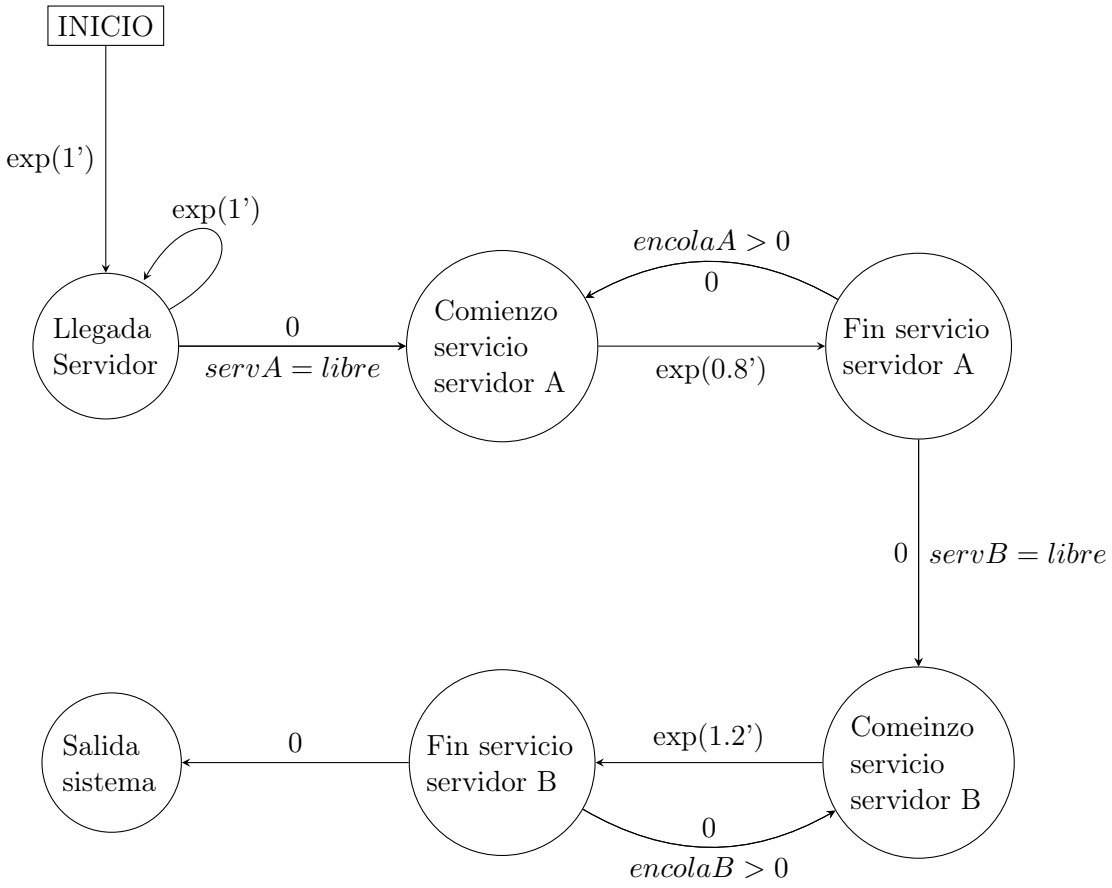

Figura 1: Versión inicial del grafo de sucesos.

El grafo anterior puede ser simplificado ya que hay un conjunto de sucesos a los que solo entran arcos con duración 0. Estos nodos son **Comienzo servicio servidor A**, **Comienzo servicio servidor B** y **Salida sistema**. Si lo simplificamos, eliminando dichos nodos y ajustando los enlaces que salen de los nodos eliminados, obtenemos el siguiente grafo:

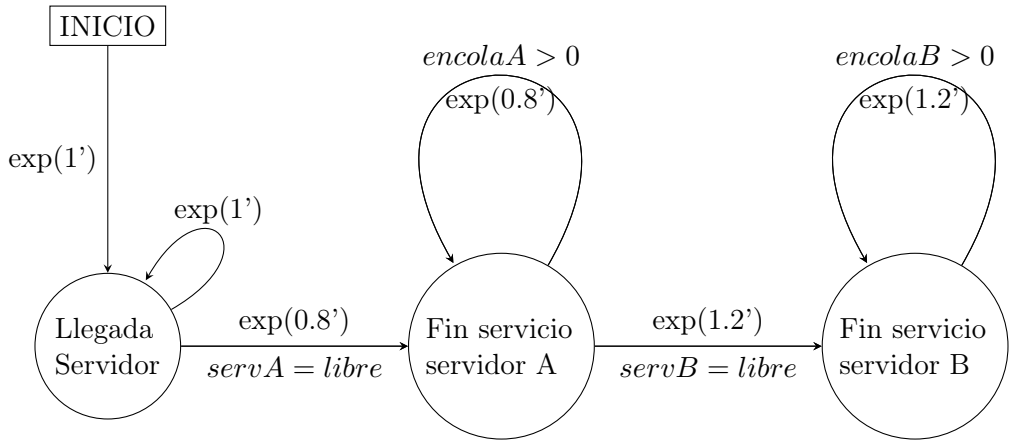

Figura 2: Versión simplificada del grafo de sucesos.

Vemos que este grafo está compuesto por tres sucesos y el suceso de inicio. Las conexiones que salían anteriormente de los nodos eliminados se han conservado, de forma que no se ha perdido ninguna transición.

3. Descripción del modelo

Una vez que hemos visto el grafo de sucesos, vamos a describir algunos aspectos fundamentales del modelo de cara a la implementación de este.

3.1. Consideraciones previas

El tiempo se va a representar en minutos, ya que se ha considerada que es la medida más adecuada según los datos proporcionados en la descripción del problema (los tiempos están dados en minutos). Además, los resultados serán más fácilmente interpretables de esta forma.

Se va a utilizar la técnica del incremento variable del tiempo, ya que es más eficiente que la técnica de incremento fijo, además de que permite obtener unos

mejores resultados que la otra técnica.

En cada ejecución se van a realizar `numSimulaciones` simulaciones. En cada una de ellas se simulará hasta que se produzca un suceso de fin de simulación. Se escogerá el siguiente suceso y se adelantará el tiempo al momento en el que tiene el lugar el suceso, y en función de qué tipo de suceso, se hará una cosa u otra. Es importante destacar que, antes de comenzar cada simulación, se reiniciará el sistema, de forma que las variables clave tengan los valores iniciales que les corresponde y no se tenga información de simulaciones anteriores.

Por último, se quiere destacar que a la hora de hacer la implementación se han juntado las variables y las tareas en una clase, encapsulando por tanto toda la información y la funcionalidad del simulador en una estructura de datos que podamos utilizar.

3.2. Variables de interés

Además de definir los sucesos, otra cosa muy importante que necesitamos determinar son las variables a utilizar. Hay de dos tipos: las variables de estado, las cuales son necesarias para el correcto funcionamiento del sistema, y los contadores estadísticos, los cuáles irán acumulando información con el objetivo de ofrecer información estadística de salida. A continuación, vamos a ver cada tipo de variable.

3.2.1. Variables de estado

Las variables de estado que encontramos son las siguientes:

- `reloj`: Esta variable es esencial, y como su propio nombre indica, representa el reloj, determinando en qué instante de tiempo nos encontramos.
- `enColaServA`: Esta variable indica cuántos clientes hay esperando en la cola del servidor A.
- `enColaServB`: La variable indica cuántos clientes hay esperando en la cola del servidor B.
- `libreServA`: Indica si el servidor A se encuentra libre (`true`) o si está ocupado (`false`).
- `libreServB`: Indica si el servidor B se encuentra libre (`true`) o si está ocupado (`false`).

- `listaSucesos`: Lista que contiene los sucesos próximos. Los sucesos están ordenados por tiempo (el suceso más próximo es el primero de la lista).
- `sucesoActual`: Variable que contiene el suceso actual.
- `tiemposLlegada`: Vector que almacena el tiempo en el que llega cada cliente. El primer elemento representa el tiempo de llegada del cliente más antiguo; el último, el del cliente más reciente. Cuando un cliente sale del sistema, se elimina el primer elemento, ya que ha salido el cliente que llevaba más tiempo en el sistema.
- `finSimulación`: Indica si se ha terminado la simulación (`true`) o si todavía no (`false`).

3.2.2. Contadores estadísticos

Los contadores estadísticos que se han considerado son los siguientes:

- `numClientesAtendidos`: Acumulador que indica cuántos clientes han sido atendidos por el sistema. Cada vez que sale un cliente se incrementa en uno.
- `tTotalClientesSistema`: Acumulador que indica cuánto tiempo en total han pasado en el sistema todos los clientes atendidos.

3.3. Estructura y composición de la lista de sucesos

Hasta ahora hemos estado hablando de los sucesos y de la lista de sucesos, pero no hemos profundizado demasiado en cómo deberían ser ambos elementos.

Por una parte tenemos el **suceso**. Un suceso no es más que una estructura de datos de tipo registro con dos campos:

- `tipoSuceso` Valor entero que representa el tipo de suceso que se va a llevar a cabo.
- `tiempo`: Instante de tiempo en el que se lleva a cabo el suceso.

Por otra parte tenemos la **lista de sucesos**, la cuál contendrá registros del tipo anteriormente especificado. La lista permitirá hacer inserciones por el final y eliminar elementos por el principio. Los elementos tienen que estar ordenados por el tiempo, de forma que el suceso más próximo sea el primero, mientras que el más

lejano el último. Para hacer esto, a la hora de implementar el modelo he utilizado el tipo `list` que ofrece `C++`, el cuál es una lista doblemente enlazada que permite hacer las operaciones de inserción y borrado en tiempo constante.

3.4. Rutinas de interés y estructura de la simulación

Lo último que nos queda es definir cuáles son las rutinas de interés del modelo. Vamos a ver cómo se podrían implementar dichas rutinas. Para cada una de ellas se va a mostrar el código en `C++` con el que se ha implementado, de forma que se tenga la referencia del modelo desarrollado en todo momento.

Por una parte tenemos la generación de sucesos. Podemos hacer la generación de forma genérica para casi todos los tipos de sucesos menos para el suceso **fin de simulación**, para el que podemos hacer una rutina aparte.

La rutina para generar sucesos de forma genérica es la siguiente:

```
// Metodo para generar sucesos
Suceso generarSuceso(int tipoSuceso, double tiempoMedio) const
{
    // Generar nuevo suceso
    Suceso suceso;

    suceso.tipoSuceso = tipoSuceso;
    suceso.tiempo = reloj + generadorExponencial(tiempoMedio);

    return suceso;
}
```

La rutina para generar el suceso fin de simulación es la siguiente:

```
// Metodo para generar el suceso fin de simulacion
Suceso generarSucesoFinSimulacion() const
{
    // Generar nuevo suceso
    Suceso suceso;

    suceso.tipoSuceso = SUCESO_FIN_SIMULACION;
    suceso.tiempo = reloj + TIEMPO_FINAL;

    return suceso;
}
```

Para generar los tiempos de los sucesos se ha utilizado un generador exponencial, el cuál puede ser implementado de la siguiente forma:

```
// Metodo generador exponencial
double generadorExponencial(double media) const
{
```

```
    double u;
    u = (double) random();
    u = (double) (u/(RAND_MAX+1.0));
    return (-media * log(1 - u));
}
```

Una vez que tenemos el suceso, para insertarlo en la lista de sucesos de forma ordenada podemos utilizar la siguiente rutina:

```
// Metodo para insertar un suceso, manteniendo la lista ordenada por tiempo
void insertarSuceso(const Suceso& suceso)
{
    // Insertar el suceso en la lista
    listaSucesos.push_back(suceso);

    // Mantener lista ordenada
    listaSucesos.sort();
}
```

Para poder utilizar la rutina de la forma anterior, tenemos que sobrecargar el operador < del registro `Suceso` o bien definir una función para comparar. En este caso se ha optado por lo primero, y se tiene que el registro tiene la siguiente forma:

```
// Estructura suceso
struct Suceso
{
    int tipoSuceso;
    double tiempo;

    bool operator<(const Suceso& otroSuceso) const
    {
        return tiempo < otroSuceso.tiempo;
    }
};
```

Es necesario tener una rutina para inicializar el modelo de simulación, tanto la primera vez como entre simulación y simulación. Esta rutina se encarga de iniciar todas las variables, tal y como se ha especificado anteriormente. La rutina se puede implementar de la siguiente forma:

```
// Metodo de inicializacion del modelo
void inicializarModelo()
{
    // Inicializar reloj y tiempo parada
    reloj = TIEMPO_INICIAL;
    tiempoParada = TIEMPO_FINAL;

    // Inicializar colas y disponibilidad de los servidores
    enColaServA = enColaServB = 0;
    libreServA = libreServB = true;
```

```
11
12      // Inicializar valores estadisticos
13      numClientesAtendidos = 0;
14      tTotalClientesEnSistema = 0.0;
15
16      // Limpiar lista de sucesos por si ha quedado algun suceso
        anterior
17      listaSucesos.clear();
18
19      // Limpiar lista de tiempos de llegada por si ha quedado alguan
        llegada anterior
20      tiemposLlegadas.clear();
21
22      // Generar suceso inicial e insertarlo en la lista de sucesos
23      Suceso sucesoInicial = generarSuceso(SUCESO_LLEGADA,
        TIEMPO_MEDIO_LLEGADAS);
24      insertarSuceso(sucesoInicial);
25
26      // Generar suceso final e insertarlo en la lista de sucesos
27      Suceso sucesoFinal = generarSucesoFinSimulacion();
28      insertarSuceso(sucesoFinal);
29
30      finSimulacion = false;
31  }
```

También necesitamos tener una rutina para obtener el siguiente suceso y actualizar el reloj al tiempo correspondiente, además de eliminar el suceso obtenido de la lista de sucesos. Para hacer estas tareas, podemos utilizar una rutina como la siguiente:

```
1  // Metodo para obtener el siguiente suceso
2  void siguienteSuceso()
3  {
4      // Obtener suceso y actualizar lista de sucesos y reloj
5      sucesoActual = listaSucesos.front();
6      listaSucesos.pop_front();
7
8      reloj = sucesoActual.tiempo;
9  }
```

De alguna forma tenemos que procesar el suceso actual y determinar qué rutina llamar en función del tipo de suceso. Para hacer esto, podemos utilizar una rutina como la siguiente:

```
1  // Metodo para procesar el suceso actual
2  void procesarSucesoActual()
3  {
4      switch(sucesoActual.tipoSuceso)
5      {
6          case SUCESO_LLEGADA:
7              sucesoLlegadaServidor();
8              break;
```

```
9           case SUCESO_FIN_SERVICIO_SERVIDOR_A:
10               sucesoFinServicioServidorA();
11              break;
12          case SUCESO_FIN_SERVICIO_SERVIDOR_B:
13               sucesoFinServicioServidorB();
14              break;
15          case SUCESO_FIN_SIMULACION:
16               sucesoFinSimulacion();
17              break;
18      }
19  }
```

Cada una de las rutinas anteriores está asociada a un suceso del grafo, menos la rutina `sucesoFinSimulacion()`, la cuál se utiliza para terminar la simulación. A continuación se puede ver qué es lo que se hace en cada una de ellas de forma más detallada:

```
1  // Metodo que representa el suceso llegada de cliente al servidor
2  void sucesoLlegadaServidor()
3  {
4      // Guardar el tiempo en el que ha llegado el cliente
5      // Se guarda al final de la lista de tiempos
6      tiemposLlegadas.push_back(reloj);
7
8      // Generar una nueva llegada
9      Suceso llegada = generarSuceso(SUCESO_LLEGADA,
       TIEMPO_MEDIO_LLEGADAS);
10     insertarSuceso(llegada);
11
12     // Comprobar si el servidor A esta libre y realizar accion
       correspondiente
13     if (libreServA)
14     {
15         libreServA = false;
16
17         Suceso servicioServA = generarSuceso(
       SUCESO_FIN_SERVICIO_SERVIDOR_A, TIEMPO_MEDIO_SERVICIO_SERVIDOR_A);
18         insertarSuceso(servicioServA);
19     }
20     else
21     {
22         enColaServA++;
23     }
24 }
```

```
1  // Metodo que representa el suceso fin de servicio en el servidor A
2  void sucesoFinServicioServidorA()
3  {
4      // Comprobar si hay clientes en la cola del servidor A
5      if (enColaServA > 0)
6      {
7          enColaServA--;
8
```

```
9           Suceso servicioServA = generarSuceso(
    SUCESO_FIN_SERVICIO_SERVIDOR_A, TIEMPO_MEDIO_SERVICIO_SERVIDOR_A);
10          insertarSuceso(servicioServA);
11      }
12      else
13      {
14          libreServA = true;
15      }
16
17      // Comprobar si el servidor B esta libre y realizar accion
    correspondiente
18      if (libreServB)
19      {
20          libreServB = false;
21
22          Suceso servicioServB = generarSuceso(
    SUCESO_FIN_SERVICIO_SERVIDOR_B, TIEMPO_MEDIO_SERVICIO_SERVIDOR_B);
23          insertarSuceso(servicioServB);
24      }
25      else
26      {
27          enColaServB++;
28      }
29  }
```

```
1   // Metodo que representa el suceso fin de servicio en el servidor B
2   void sucesoFinServicioServidorB()
3   {
4       // Incrementar el numero de clientes atendidos y el tiempo total
5       // de los clientes en el servidor
6       numClientesAtendidos++;
7       tTotalClientesEnSistema += reloj - tiemposLlegadas.front();
8
9       // Eliminar tiempo de llegada del cliente mas antiguo (el primero)
10      tiemposLlegadas.pop_front();
11
12      // Comprobar si hay clientes en la cola del servidor A
13      if (enColaServB > 0)
14      {
15          enColaServB--;
16
17          Suceso servicioServB = generarSuceso(
    SUCESO_FIN_SERVICIO_SERVIDOR_B, TIEMPO_MEDIO_SERVICIO_SERVIDOR_B);
18          insertarSuceso(servicioServB);
19      }
20      else
21      {
22          libreServB = true;
23      }
24  }
```

```
1   // Metodo que representa el suceso fin de simulacion
2   void sucesoFinSimulacion()
3   {
```

```
4       finSimulacion = true;
5
6       double tiempoMedioEstancia = calcularTiempoMedioEstancia();
7       tiemposMediosEstancia.push_back(tiempoMedioEstancia);
8   }
```

Podemos ver que la rutina `sucesoFinSimulacion()` calcula el tiempo medio de estancia y lo guarda en un vector. Dicho vector será accedido más adelante para generar el informe. El vector tendrá tamaño `numSimulaciones`. Para calcular el tiemo de estancia medio de los clientes en el servidor, podemos utilizar la siguiente expresión, la cuál relaciona el tiempo total que han pasado todos los clientes atendidos dentro del sistema y el número total de clientes atendidos:

$$tiempoEstanciaMedio = \frac{tTotalClientesSistema}{numClientesAtendidos} \tag{1}$$

Para ayudarse con el cálculo, la rutina anterior utiliza otra rutina que hace el cálculo, la cuál se puede ver a continuación:

```
1   // Metodo para calcular el tiempo de estancia medio al terminar una
        simulacion
2   double calcularTiempoMedioEstancia() const
3   {
4       return tTotalClientesEnSistema / numClientesAtendidos;
5   }
```

Una rutina que nos será útil más adelante es la que permite determinar si se ha producido el fin de la simulación o no. A pesar de que implementar la rutina es trivial, se ofrece la implementación realizada a continuación:

```
1   // Metodo que permite determinar si ha finalizado la simulacion
2   bool esFinSimulacion() const
3   {
4       return finSimulacion;
5   }
```

La última rutina que nos interesa es la generación del informe. El informe contendrá el resultado medio de todas las simualciones junto con su desviación típica. Para ello, se va a utilizar el vector anteriormente mencionado en el que se han ido guardando los resultados medios para cada simulación. A partir de estos, se puede extraer un valor medio global, además de la desviación típica. Podemos generar el informe de la siguiente forma:

```
1   // Metodo para generar el informe final
2   void generarInforme() const
3   {
4       // Obtener suma de los valores medios y suma de los cuadrados de
        los valores medios
```

```
5       double sum = accumulate(tiemposMediosEstancia.begin(),
        tiemposMediosEstancia.end(), 0.0);
6       double sum2 = inner_product(tiemposMediosEstancia.begin(),
7                                   tiemposMediosEstancia.end(),
8                                   tiemposMediosEstancia.begin(),
9                                   0.0);
10
11      // Obtener valores medios
12      double tMedioEstancia = sum / numSimulaciones,
13             desv = sqrt((sum2 - numSimulaciones * tMedioEstancia *
        tMedioEstancia) / (numSimulaciones - 1));
14
15      // Mostrar resultados por pantalla
16      cout << "\n\nRESULTADOS PARA " << numSimulaciones << "
        SIMULACIONES" << endl;
17      cout << "Tiempo medio de estancia: " << tMedioEstancia << " +/- "
        <<  desv << " minutos" << endl;
18  }
```

Por último, nos queda por ver la estructura general del programa de simulación. Simplemente tendríamos que iniciar primero la semilla aleatoria y luego simular `numSimulaciones` veces, reiniciando los valores de las variables en cada simulación. Una vez que se han hecho todas las simulaciones se muestra el informe.

La estructura anteriormente mencionada se puede ver a continuación:

```
1  // Inicializar semilla aleatoria
2  srand(time(NULL));
3
4  // Crear instancia del modelo de simulacion
5  ModeloServidor* modeloServidor = new ModeloServidor(numSimulaciones);
6
7  // Simular numSimulaciones veces
8  for (int i = 0; i < numSimulaciones; i++)
9  {
10     cout << "Simulacion " << i << "..." << endl;
11
12     // Iniciar el modelo antes de simular
13     modeloServidor->inicializarModelo();
14
15     while(!modeloServidor->esFinSimulacion())
16     {
17         modeloServidor->siguienteSuceso();
18         modeloServidor->procesarSucesoActual();
19     }
20 }
21
22 // Generar informe mostrando la media para todas las simulaciones
23 modeloServidor->generarInforme();
```

4. Experimentación y resultados