CAPÍTULO 4: IMPLEMENTACIÓN

Java ha supuesto una revolución en el mundo de la programación. Se dice que Java es a un lenguaje como C lo que en sus comienzos pudo ser C al lenguaje ensamblador. Esto se traduce, por ejemplo, en que un mayor porcentaje del control sobre la máquina lo ejerce la propia plataforma, cuando en C estaba en poder del usuario.

Esto tiene como consecuencia inmediata la protección del sistema frente a códigos mal escritos o programas malintencionados, lo cual supone una gran ventaja. No obstante, esto implica también un coste, que principalmente se manifiesta en la necesidad de equipos más potentes para la realización de tareas similares.

Al ser un lenguaje de más alto nivel que los de tipo procedimental, como C, se ofrecen al programador comandos más potentes que realizan más tareas, asumiendo parte del trabajo del propio desarrollador. Además, al ser Java orientado a la programación en red, se encontraron funciones de gran utilidad en este campo entre las disponibles en la distribución de este lenguaje utilizada [JDK'01]. Todo el código que se escribe en Java está contenido en 'clases'. Todos los objetos que intervengan en una aplicación se crearán según una clase, que definirá sus atributos y sus comportamientos.

4.1. MODELOS DE TRÁFICO

4.1.1. RECURSOS JAVA UTILIZADOS

- Direcciones de Internet

La clase `InetAddress` del paquete `java.net` representa una dirección de Internet según el protocolo IP v4. Dispone de una serie de métodos que facilitan el manejo de este tipo de direcciones en distintas nomenclaturas, a la vez que permiten obtener las direcciones de las máquinas con las que exista conexión en un momento determinado.

- Creación de conexiones TCP

A la hora de manejar conexiones TCP, encontramos dos clases en las que se reúne toda la funcionalidad necesaria.

  - La clase `ServerSocket`

Esta clase se encarga de abrir un canal de comunicación con el exterior a través de la red cuya misión principal es la de escuchar peticiones de conexión y aceptarlas si es necesario. Esto se lleva a cabo con el comando `accept()`, que crea una conexión nueva con el solicitante y continúa a la escucha en espera de nuevas peticiones. Para ajustarse a la duración de la simulación, se utiliza el comando `setSoTimeout()` para hacer que la escucha no se prolongue durante más tiempo del que dura la simulación, con objeto de dejar libre el puerto ocupado para simulaciones sucesivas.

El mecanismo de vigilancia de Java permite indicar que, cualquiera que sea la causa por la que finalice la ejecución del programa en curso, este canal ha de quedar libre para su utilización en cualquier momento, es decir, no quede bloqueado. Esto se lleva a cabo con el comando `close()`.

  - La clase `Socket`

De esta clase son los canales orientados a conexión que se abren mediante petición a un `ServerSocket` o por conexión directa con otro `Socket` que ya se encuentra abierto y cuyo puerto se conoce con anterioridad. Su duración se puede indicar del mismo modo que en el caso anterior, es decir, con el comando `setSoTimeout()`, y se emplea la orden `connect()` para solicitar de aquellos el establecimiento de una conexión nueva.

Para realizar lecturas o escrituras es preciso recurrir al sistema de entrada/salida de Java. Éste permite al programador crear determinados canales con fines específicos a partir de un `Socket`, de modo que sean ellos los encargados de realizar las transacciones necesarias. Este aspecto será abordado más adelante.

- Creación de canales UDP

  - La clase `DatagramSocket`

Puesto que el protocolo UDP es no orientado a conexión, es necesario que ambos extremos de la comunicación conozcan los puertos en los que se va a ubicar cada uno de estos elementos `DatagramSocket`. Se trata de *sockets* parecidos a los empleados en TCP con determinadas características que les permiten ser usados con UDP.

Estos *sockets* no están ligados a ninguna dirección en particular, con lo que cada datagrama que desee enviarse debe contener una dirección y un puerto de destino.

Al contrario que en el caso anterior, con los objetos `DatagramSocket` las lecturas y escrituras se realizan directamente a través del propio *socket*, empleando las órdenes `send()` y `receive()`. El contenido de estos envíos se aloja en objetos de la clase que se aborda en el siguiente apartado.

  - La clase `DatagramPacket`

`DatagramPacket` es un objeto que se crea a partir de un *array* de bytes y es enviado por la red a través de un `DatagramSocket`. En él debe especificarse la dirección IP y el puerto al que va destinado. Su tamaño, incluyendo contenido y cabecera, no debe superar los 65536 bytes, que es el máximo fijado en la definición del protocolo UDP.

- Sistema de entrada/salida de Java

En su librería `java.io`, este lenguaje proporciona una serie de clases específicamente diseñadas para realizar operaciones de entrada/salida. Existe una primera división entre aquellas clases orientadas a byte y las orientadas a carácter. Los caracteres en Java ocupan 16 bits. Esto le permite manejar conjuntos de hasta 2^{16} caracteres, con lo que cubre de sobra las necesidades de escritura de cualquier idioma, cuyos grafos quedan representados en el sistema UNICODE, empleado en este lenguaje.

Las clases orientadas a byte se denominan `InputStream` y `OutputStream`. Existen diferentes versiones de ellas, específicamente implementadas según su utilidad, que puede ir desde el manejo de *arrays* o *buffers* hasta archivos o cadenas de caracteres. En las siguientes figuras se pueden ver las diferentes implementaciones de estas clases.

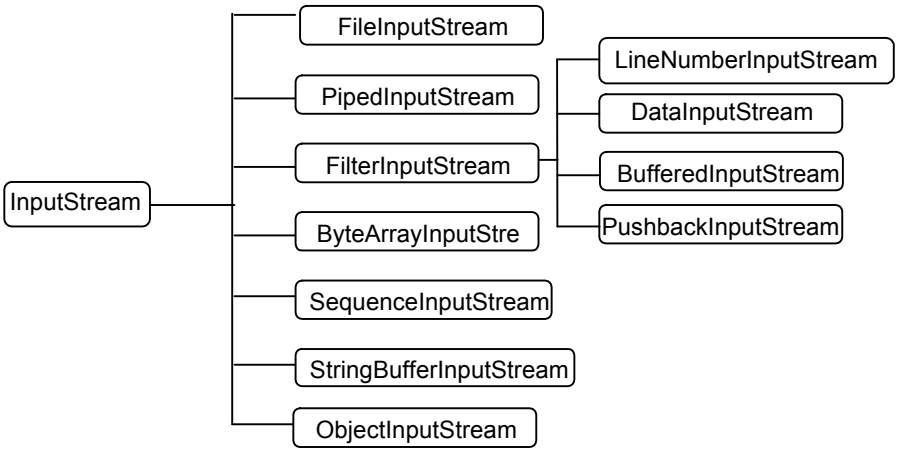


Figura 4.1. La clase InputStream y sus subclases

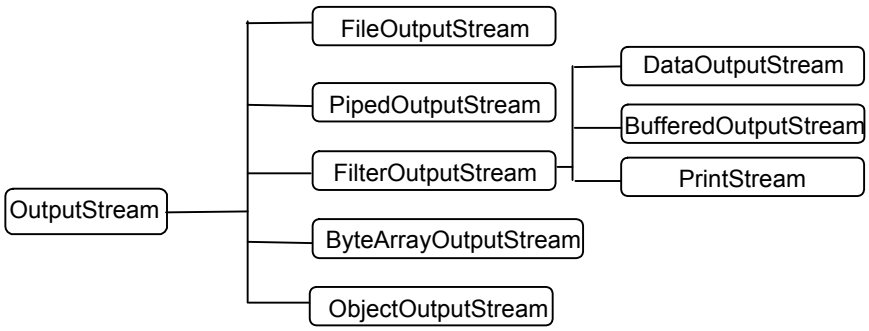


Figura 4.2. La clase OutputStream y sus subclases

Por su parte, las clases orientadas a carácter reciben los nombres de `Reader` y `Writer`. Al igual que en el caso anterior, también están disponibles distintas implementaciones para usos concretos que abarcan desde flujos de caracteres hasta archivos completos. En las siguientes figuras se pueden ver las diferentes implementaciones de estas clases.

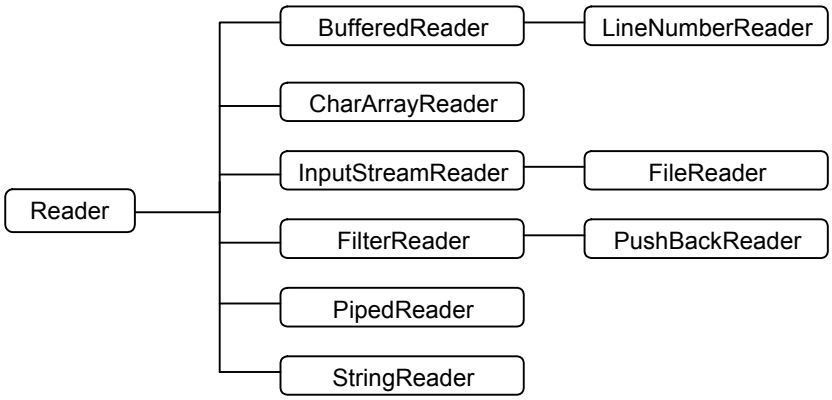


Figura 4.3. La clase Reader y sus subclases

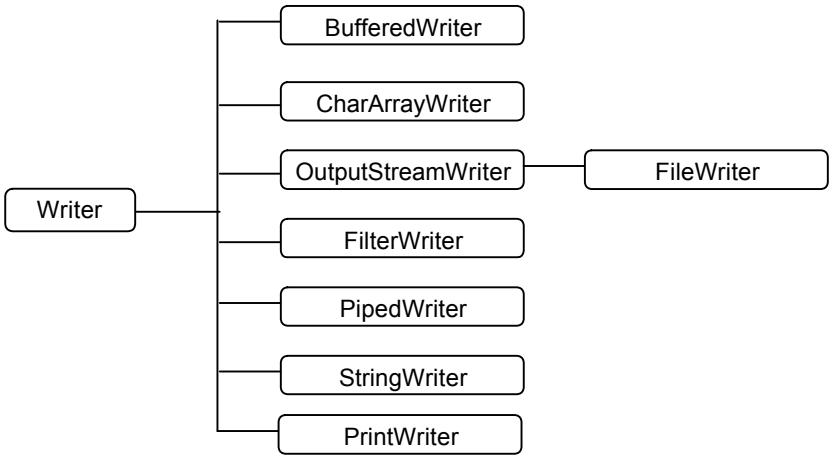


Figura 4.4. La clase Writer y sus subclases

La principal ventaja de todo este sistema radica en la facilidad con la que se pueden crear este tipo de objetos destinados a entrada/salida a partir de un objeto `Socket` o `DatagramSocket`. De este modo, una vez creado el objeto lector o escritor, queda en su poder el control de todas las operaciones de lectura/escritura de los objetos `Socket`, con lo cual el desarrollador puede limitarse a leer o escribir sin preocuparse de la procedencia de los datos que entrega o recibe.

Esta abstracción simplifica en gran medida el proceso de comunicación entre dos máquinas remotas, ya que gran parte de las tareas de control y manejo de contenidos a

transmitir quedan en manos de la plataforma Java, con la consiguiente descarga que esto representa para el programador.

- Medidas del tiempo en Java

Para comprobar las distintas duraciones y periodos de actividad en las simulaciones, la clase `System` dentro del paquete `java.lang` ofrece un método denominado `System.currentTimeMillis()`, que devuelve la cantidad de milisegundos transcurridos desde el 1 de Enero de 1970. A partir de este dato, se puede resolver si un periodo ha finalizado o no.

- Generación de números aleatorios

Con este fin se emplea la clase `Math` del paquete `java.lang`. El método `Math.random()` genera valores en punto flotante de doble precisión comprendidos entre 0.0 y 1.0. La semilla empleada proviene de la clase `java.util.Random`, que en su constructor sin parámetros emplea la hora actual en milisegundos como valor de semilla. Otro método matemático empleado es `Math.round()`, que convierte un valor de doble precisión en un entero de larga longitud.

- Creación de procesos

Una de las facetas que ha aportado más comodidad para el desarrollador con respecto a los lenguajes procedimentales es la creación de procesos y su control. En este sentido, Java dota de una mayor seguridad a la programación, siempre a costa del consumo de recursos, mediante el incremento de la vigilancia al manejar procesos concurrentes.

Para ello, el paquete `java.lang` aporta la clase `Thread`, que ofrece junto a sus métodos una solución muy completa para los procesos en lo que a creación, manejo, pausa, reanudación y destrucción se refiere.

Uno de los usos más significativos que ha tenido esta clase en la aplicación implementada ha sido la realización de las esperas entre dos elementos consecutivos de un

modelo. La llamada a `Thread.sleep()` permitía detener la ejecución del proceso en curso, para ceder el control a otros que estuvieran presentes en ese momento y recuperarlo una vez transcurrido el tiempo indicado en la llamada mencionada.

- Cadenas de texto dinámicas

Para la creación del fichero de resultados, se empleó una combinación de técnicas, la primera de las cuales era la del manejo de cadenas de texto dinámicas. La clase `StringBuffer` del paquete `java.lang` permite la inserción de textos de forma incremental gracias a su método `append()`. De este modo, simplemente se ha de añadir el nuevo dato a la cadena en cuanto esté disponible.

- Control sobre recursos compartidos

Para evitar conflictos entre varios procesos concurrentes que deseen acceder a un mismo recurso a la vez, en este caso `StringBuffer`, Java emplea la palabra reservada `synchronized`. Al hacer que un método sea de este tipo, sólo un proceso a la vez puede acceder a un recurso contemplado en él. Esto permite, por ejemplo, que varias sesiones simultáneas puedan escribir sus datos sin conflictos. El primer proceso que accede a un recurso mantiene el control sobre él de forma exclusiva y cualquier otro que desee utilizarlo debe esperar a que sea liberado para acceder a él.

- Manejo de ficheros

Para efectuar cualquier tipo de operación con ficheros, se dispone de la clase `File` dentro del paquete `java.io`. Esta clase ofrece métodos para realizar todo tipo de operaciones en lo que a archivos y directorios se refiere, siempre que el sistema de seguridad lo permita.

La combinación de esta clase con el sistema de entrada/salida de Java convierte el almacenamiento de los datos obtenidos durante las simulaciones en una tarea relativamente fácil. Para ello, existe la clase `FileWriter` (ver figura 4.4), que permite el volcado del contenido de un objeto de tipo `StringBuffer` de una forma sencilla.

La combinación de estas técnicas permite ir tomando datos de forma incremental sin necesidad de realizar accesos al disco, tremendamente costosos en comparación con las operaciones realizadas en memoria. De este modo, bastará con realizar dos accesos al disco, uno al crear el fichero y otro al acabar la simulación, si se ha solicitado un fichero de resultados.

- Objetos de parámetros de configuración

Los valores de los parámetros deseados para una determinada simulación, junto con otros datos necesarios para el correcto funcionamiento de los programas, van incluidos en una serie de objetos denominados "de configuración". Para cada modelo existen dos objetos según los ficheros `ConfiguracionCliente*.java` y `ConfiguracionServidor*.java`. Estos objetos son generados en la máquina cliente y son suministrados a los programas que ejecutan las simulaciones, tanto localmente como en el lado del servidor. Este último proceso se aborda más adelante en este capítulo, en la explicación de la implementación de la parte que concierne a los *servlets*.

- Puertos

Tanto la conexión de control como cada uno de los modelos utilizan números de puerto determinados para su ejecución. Estos números están almacenados en una clase denominada `Constantes`, incluida en los archivos `.jar` junto a todos los demás componentes de la aplicación. Si en algún momento se desea emplear números de puerto diferentes a los mencionados, basta con modificar el fichero `Constantes.java`, perteneciente al código fuente del programa desarrollado, compilarlo e incluirlo en el archivo `.jar` correspondiente.

- Distribuciones de probabilidad adicionales

En el caso en que se deseara añadir al programa una nueva distribución de probabilidad, los pasos a seguir serían los siguientes (se recomienda poseer ciertas nociones de Java): Tomar del código fuente cualquiera de las distribuciones existentes, reconocibles al estar

incluidas en ficheros de la forma `Distribucion*.java`. Realizar los cambios necesarios en el procesamiento y nombre de la clase, así como en el nombre del fichero correspondiente, según las normas del lenguaje Java. Posteriormente, incluir esta distribución entre las disponibles en el fichero `PanelDistribucion.java` (`EPanelDistribucion.java` para la versión inglesa), así como crear un nuevo panel con los parámetros a solicitar a imagen de los contenidos en los ficheros `PanelDatos*.java` (`EPanelDatos*.java` para la versión inglesa). Por último, incorporar el nuevo modelo en el fichero `CrearDistribución.java`, compilar y volver a generar el archivo `.jar` correspondiente.

4.1.2. MODELO SEMI-MARKOVIANO DE DOS ESTADOS

Todos los modelos implementados tienen en común determinadas formas de proceder, que serán explicadas en este modelo genérico, siendo esta explicación igualmente válida para el resto de ellos.

Para comenzar, resulta apropiado citar la máxima bajo la cual se desarrolla toda la programación existente en lenguaje Java [Eckel'00]: *"En Java, todo lo que hay son objetos"*.

El cliente comienza, como todos los demás, por su propio constructor. El constructor es un fragmento del código de una clase que se ejecuta al crear un objeto de esa clase. En este caso, para crear un objeto de tipo `ClienteGenerico`, al igual que los demás, es preciso pasar como argumento al constructor el objeto de configuración en el que figuran los parámetros que necesita conocer. En este caso, estos parámetros son el protocolo sobre el que se efectuará la simulación, la dirección IP de la máquina del servidor, el panel donde hay que hacer las indicaciones luminosas y la duración de la simulación.

Una vez calculado el instante en el que deberá finalizar la presente simulación, se continúa la ejecución según el protocolo indicado por el usuario. Con objeto de respetar al máximo la temporización, se efectúan frecuentes comprobaciones sobre la necesidad de permanecer en un estado o avanzar al siguiente, ya sea este último continuando o

finalizando la simulación. Si en cualquiera de estas comprobaciones se detecta el vencimiento del tiempo de simulación, se procederá a la finalización de ésta de forma automática. Este cierre se hace siempre de forma segura, es decir, asegurando que todos los canales y conexiones que fueron abiertos, queden convenientemente cerrados y preparados para la siguiente simulación.

En el caso en el que el protocolo elegido haya sido UDP, el cliente ha de enviar un paquete de longitud '1' al servidor, con objeto de que éste pueda extraer de él la dirección IP y el puerto al que debe mandar los paquetes generados. Si el protocolo es TCP, esta medida no será necesaria, ya que al realizar la conexión con el servidor, queda automáticamente fijado el destino de los paquetes.

Tanto en caso de utilizar un protocolo como otro, a la recepción de un paquete seguirá de manera inmediata la indicación luminosa de este hecho al interfaz de usuario. Esto se lleva a cabo lanzando un proceso paralelo que mantendrá encendido el indicador de color verde durante un periodo de 200 milisegundos. Al ser un proceso aparte, no será excesivo el coste del aviso luminoso al usuario a través del interfaz gráfico.

Como ya ha sido mencionado, los números de puerto, al igual que otras constantes, vienen definidos en un fichero llamado `Constantes.java`, con lo cual, cualquier actualización en este sentido se verá reflejada en el resto del código de forma automática.

En muchos casos, entre ellos éste en el que nos encontramos, unos elementos de la simulación presentan mayor complejidad que otros. Las causas de una mayor complejidad son las siguientes:

- Debido al sistema de seguridad que rodea a los *applets*, el cliente está sujeto a severas restricciones, como el hecho de no tener acceso al sistema de archivos de la máquina donde se ejecuta. Esto provoca que las medidas deban realizarse únicamente desde el servidor, donde pueden ser almacenadas en un archivo para su posterior descarga. Para ello es necesario crear un lugar donde almacenar temporalmente las medidas en memoria, ir añadiendo los valores obtenidos y volcar los datos a un fichero al finalizar la simulación.

- Ser el elemento generador de tráfico de la pareja cliente-servidor le confiere mayor complejidad. En la mayoría de los casos, sólo uno de los dos agentes genera el tráfico, lo cual le obliga a manejar distribuciones de probabilidad, generar valores de forma iterativa, realizar envíos y esperar durante ciertos intervalos de tiempo, mientras que el otro agente se limita a esperar dichos envíos.

- Generar procesos concurrentes para el manejo de tráfico agregado es otro de los elementos que añade complejidad a un agente.

En el caso del modelo semi-markoviano de dos estados, la mayor complejidad la posee el servidor, que genera todos los instantes y tamaños, toma nota de los resultados y los archiva posteriormente en un fichero, mientras que el cliente se limita a recibir todos los paquetes hasta que llega el envío de longitud nula con el que se indica el fin de la simulación.

El servidor, que ha de generar valores a partir de distribuciones de probabilidad, invoca su método `crearDistribuciones()`, que prepara los parámetros para ir extrayendo los valores que se utilizarán. También crea el archivo en el que se almacenarán los resultados, que se rellenará al final de la simulación con los datos introducidos en la cadena de texto dinámica destinada a tal efecto.Para observar el esquema de funcionamiento de los procesos implicados en el modelo de tráfico genérico, se incluyen los siguientes flujogramas:

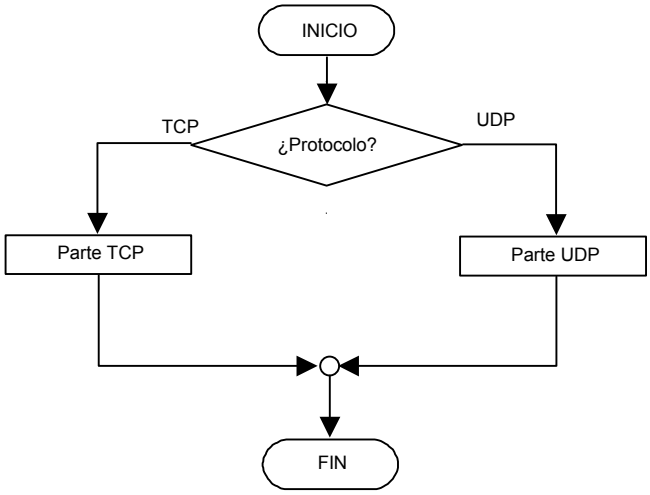


Figura 4.5. Flujograma del cliente de tráfico genérico semi-markoviano de dos estados

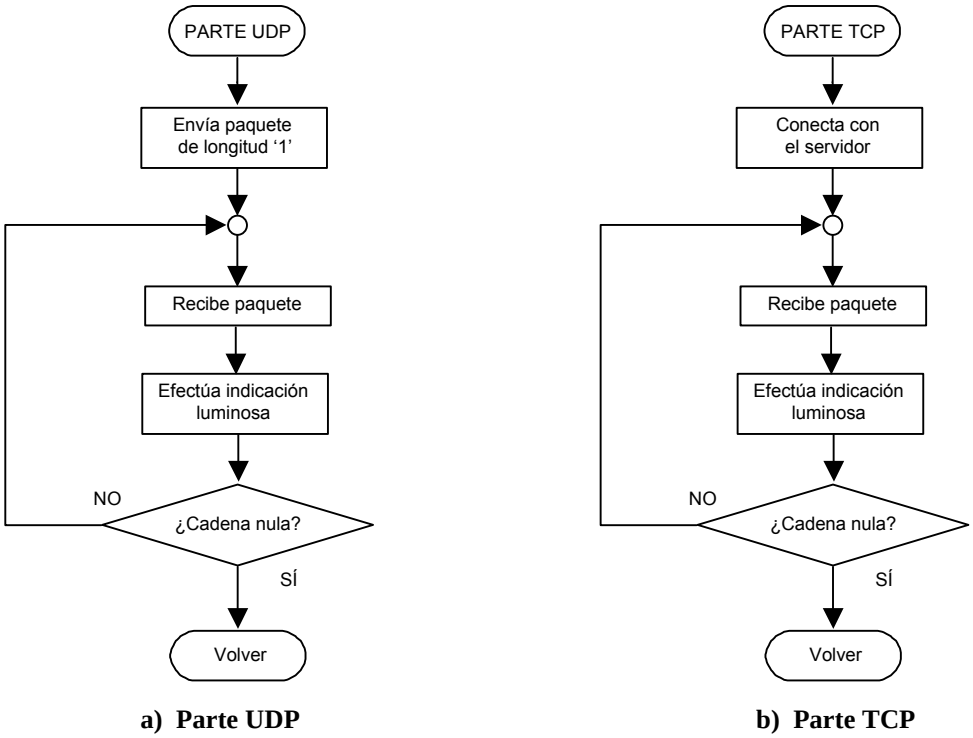


Figura 4.6. Flujogramas de los segmentos del cliente genérico dependientes del protocolo

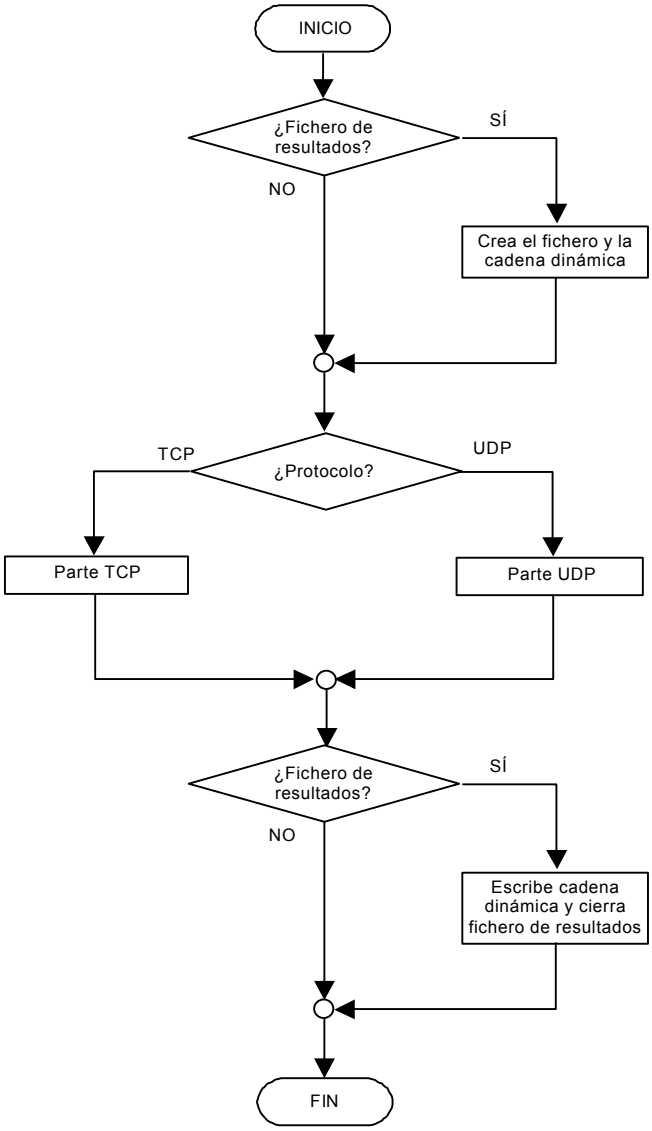


Figura 4.7. Flujograma del servidor de tráfico genérico semi-markoviano de dos estados

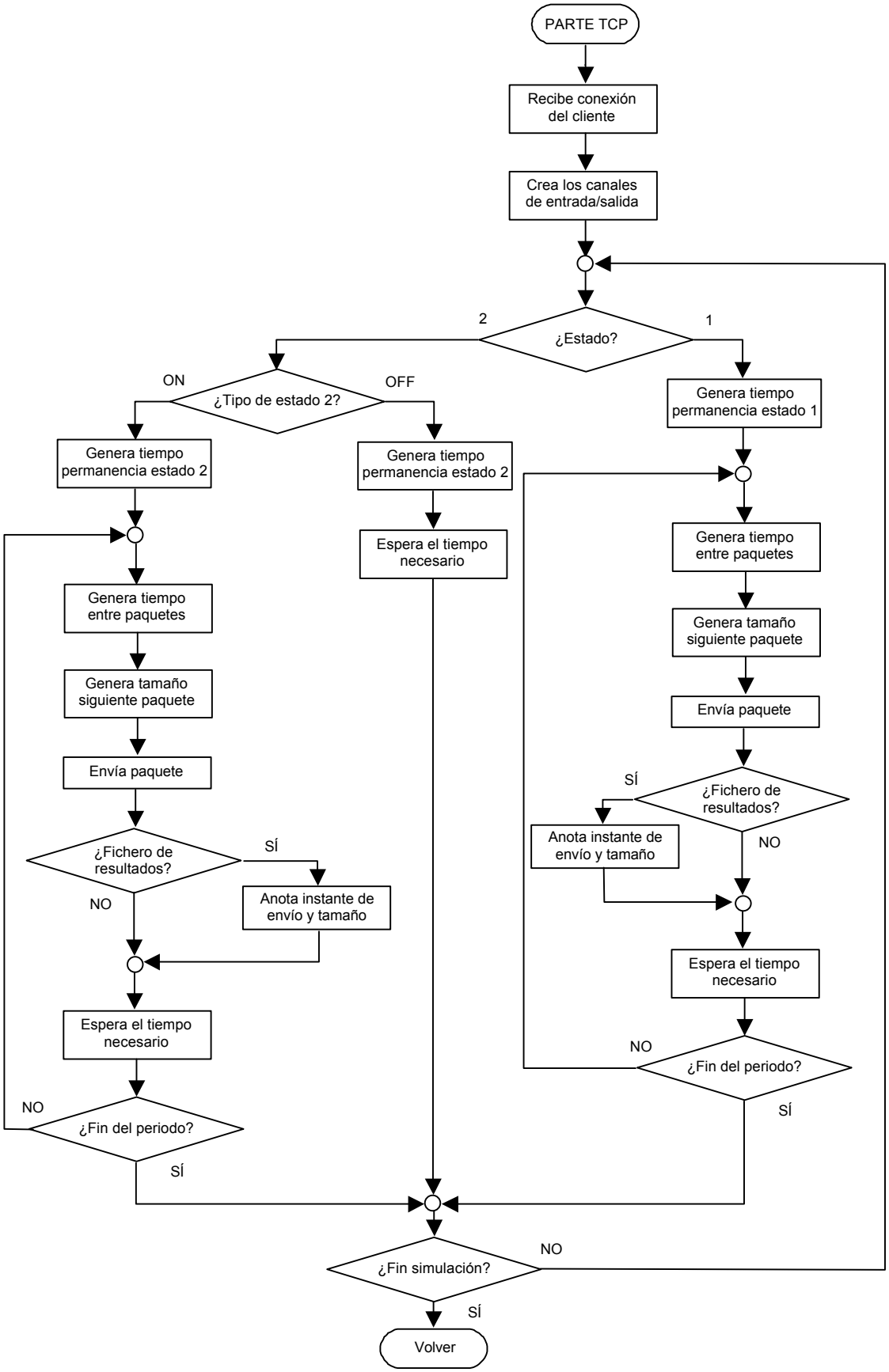


Figura 4.8. Flujograma del segmento del servidor genérico sobre protocolo TCP

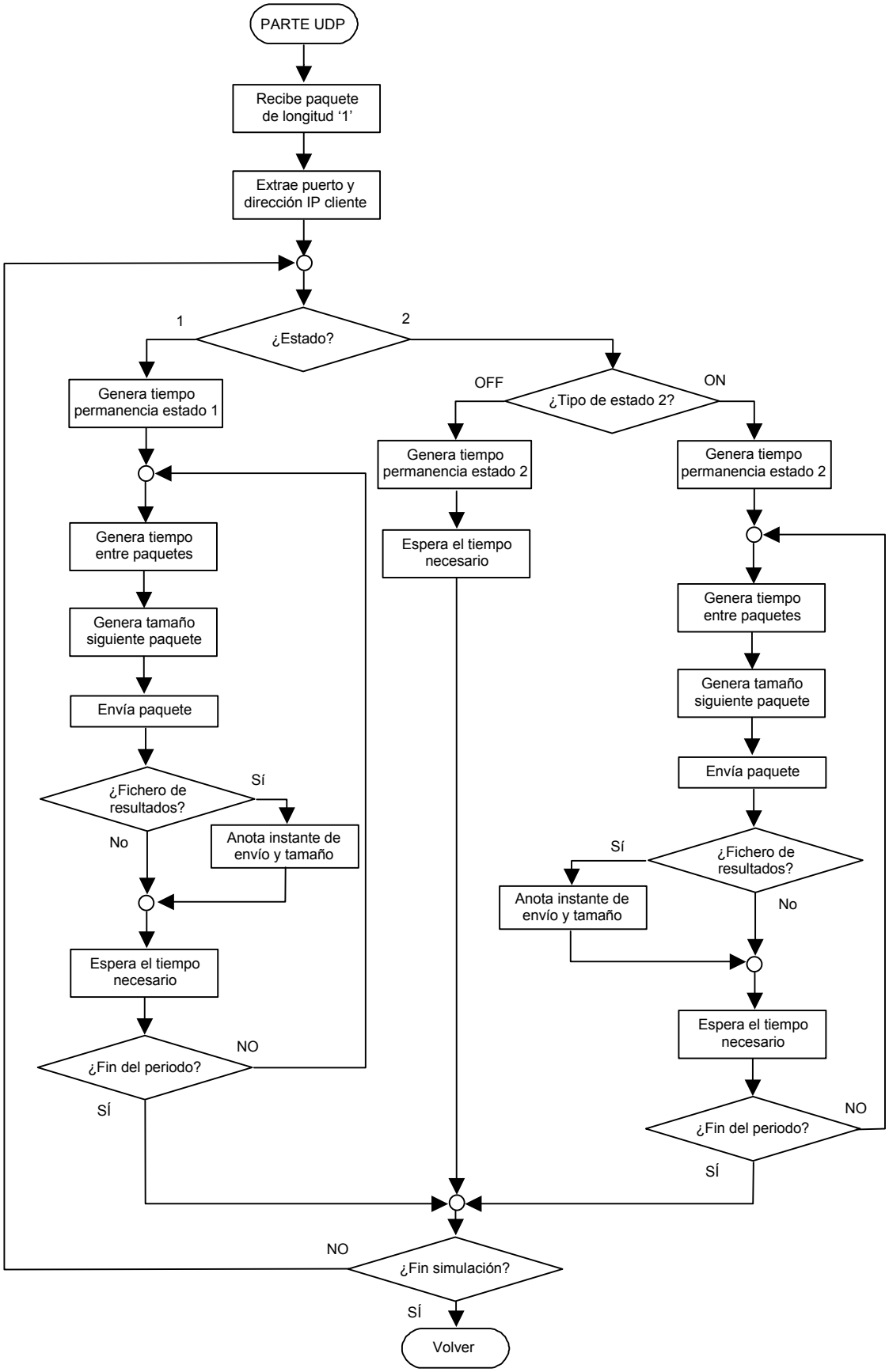


Figura 4.9. Flujograma del segmento del servidor genérico sobre protocolo UDP

4.1.3. MODELO DE CORREO ELECTRÓNICO

Es un modelo conceptualmente sencillo que funciona sobre el protocolo TCP. El cliente se encarga de generar mensajes con las características indicadas a través del interfaz de usuario. Mientras no se alcance el final de la simulación, el cliente irá generando mensajes con o sin *attach* y los enviará al servidor. Es capaz de generar tráfico agregado mediante la creación de procesos independientes.

A continuación se muestran los flujogramas que reflejan este modelo:

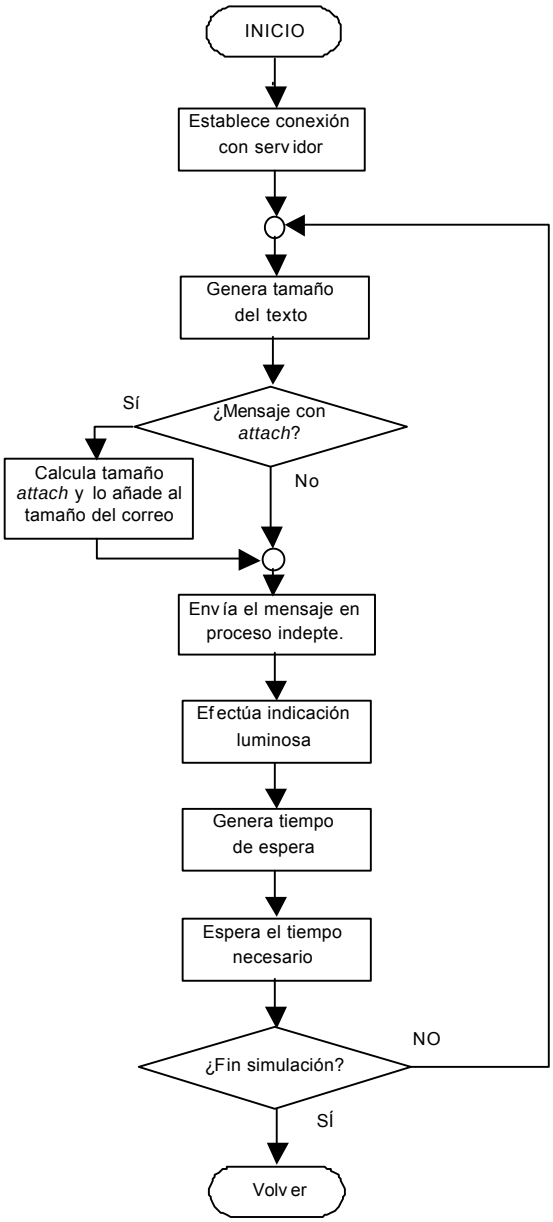
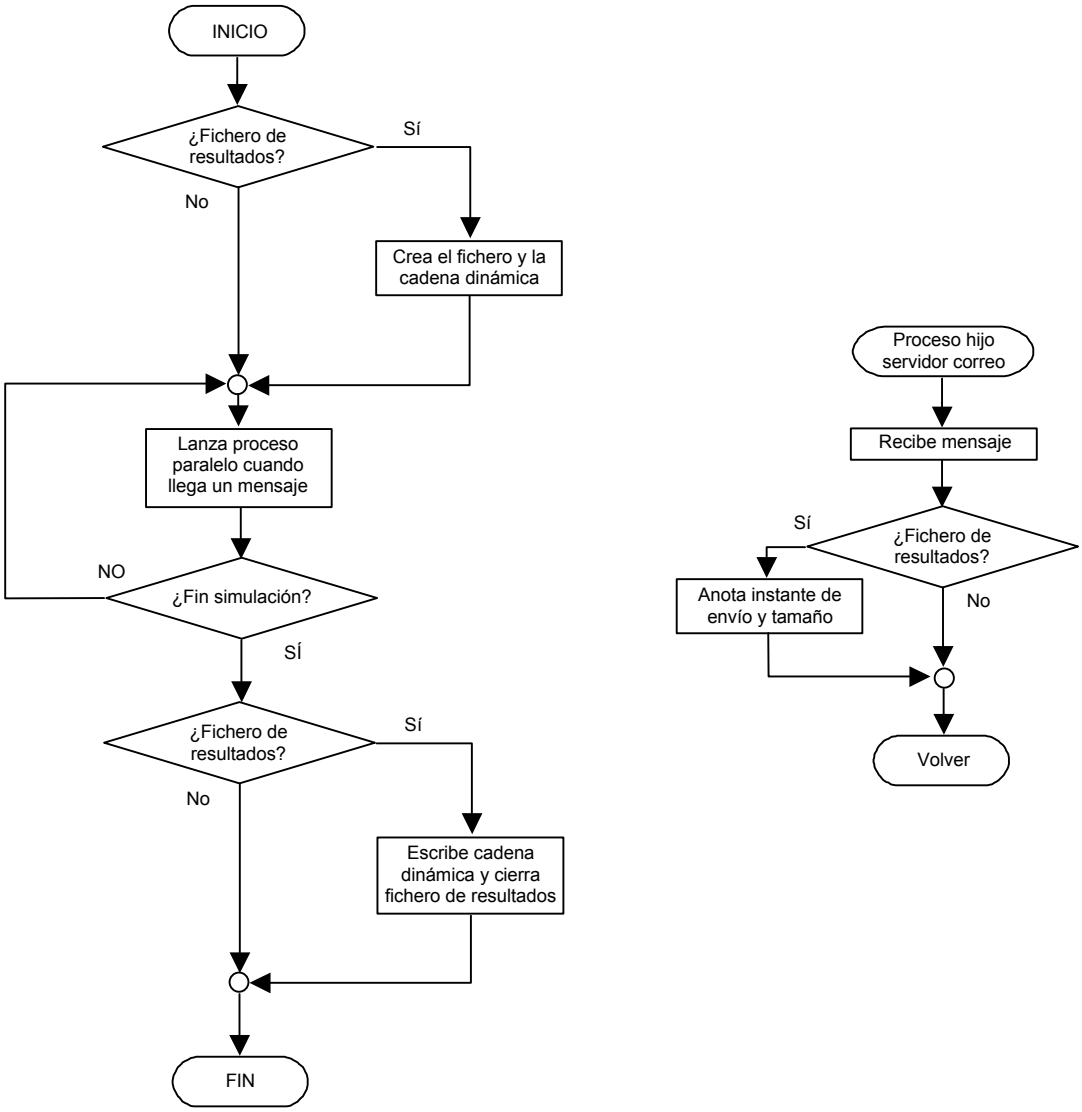


Figura 4.10. Flujograma del cliente del modelo de correo electrónico

Por su parte, en el momento en que se detecte la llegada de un nuevo mensaje, el servidor lanzará un proceso paralelo que se encargará de recibirlo, anotando el instante de llegada y el tamaño del mensaje si así lo solicita el usuario.



a) Proceso servidor global **b) Proceso independiente receptor de un mensaje**

Figura 4.11. Flujogramas del servidor del modelo de correo electrónico

4.1.4. MODELO DE TRANSFERENCIA DE FICHEROS

Este modelo de tráfico FTP posee la característica de generar tráfico agregado. Esto se lleva a cabo mediante la creación de procesos independientes cuya misión es la de generar el tráfico solicitado. Funciona sobre el protocolo TCP.

Las simulaciones se realizan en base a tres niveles diferentes. El cliente crea sesiones separadas un determinado periodo de tiempo, que contendrán cierto número de transferencias. Al generarse los procesos de forma independiente, es posible que varias sesiones y transferencias coexistan temporalmente, generándose así tráfico agregado.

A continuación se presentan los flujogramas correspondientes a los distintos procesos en el cliente y en el servidor:

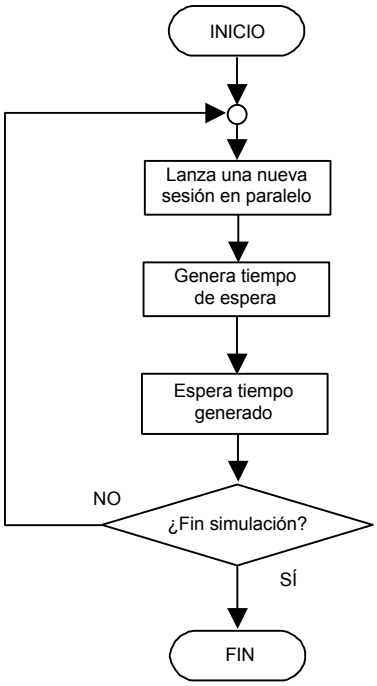


Figura 4.12. Flujograma del proceso global del cliente FTP

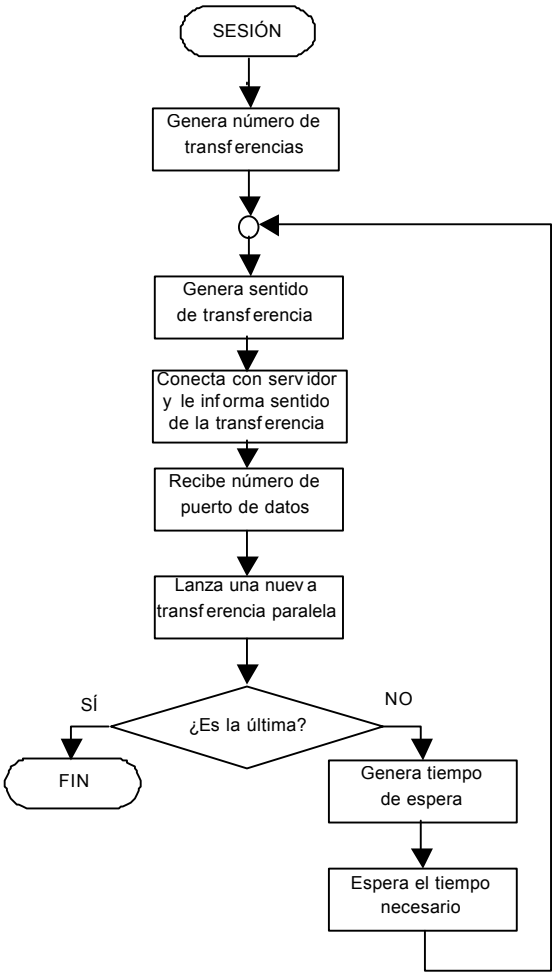


Figura 4.13. Flujograma de una sesión en el cliente FTP

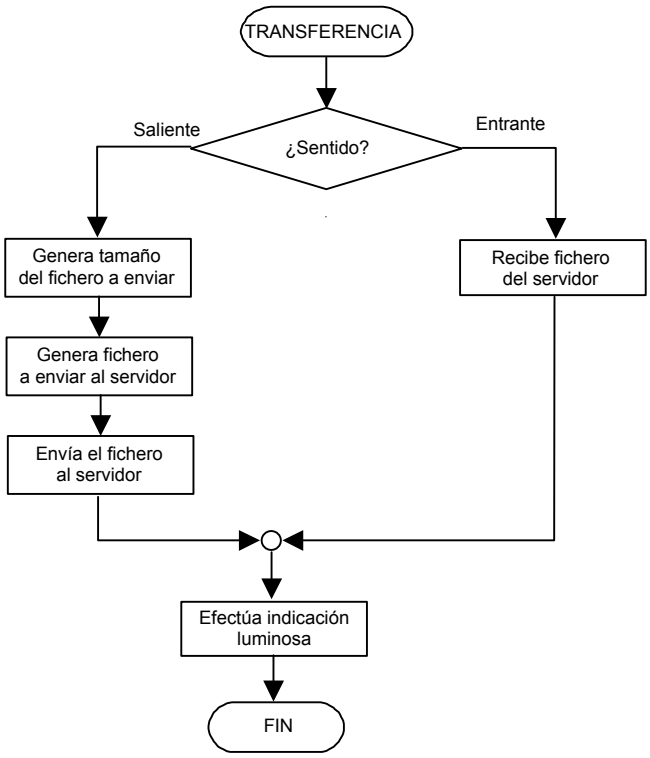


Figura 4.14. Flujograma de una transferencia en el cliente FTP

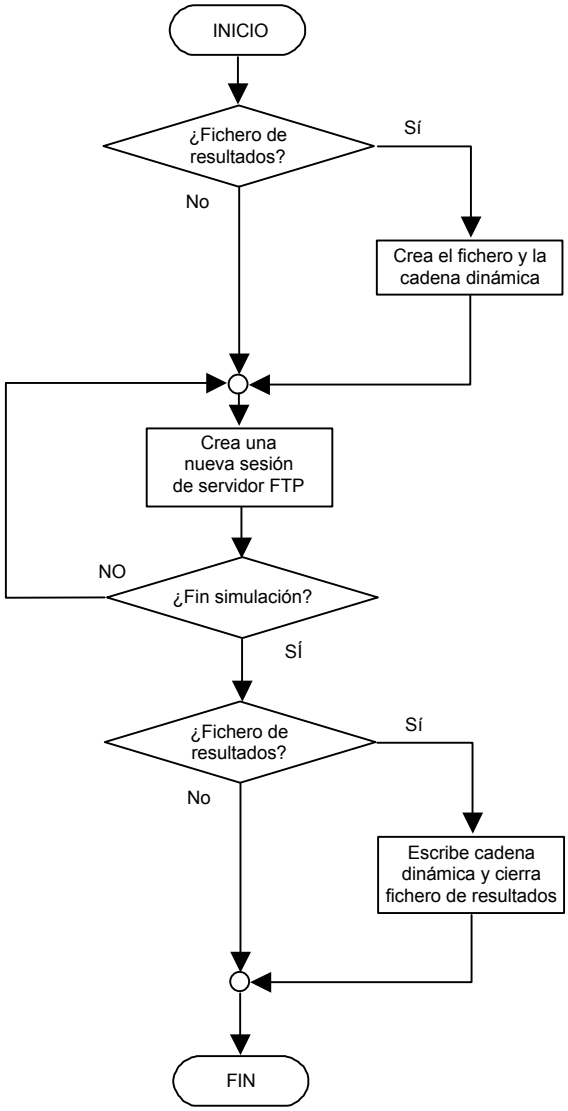


Figura 4.15. Flujograma del proceso global del servidor del modelo FTP

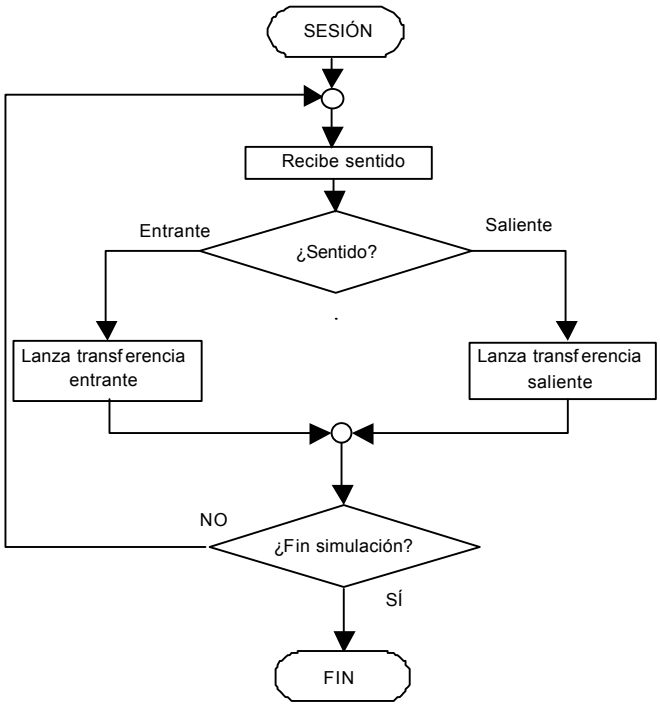


Figura 4.16. Flujograma de una sesión en el servidor FTP

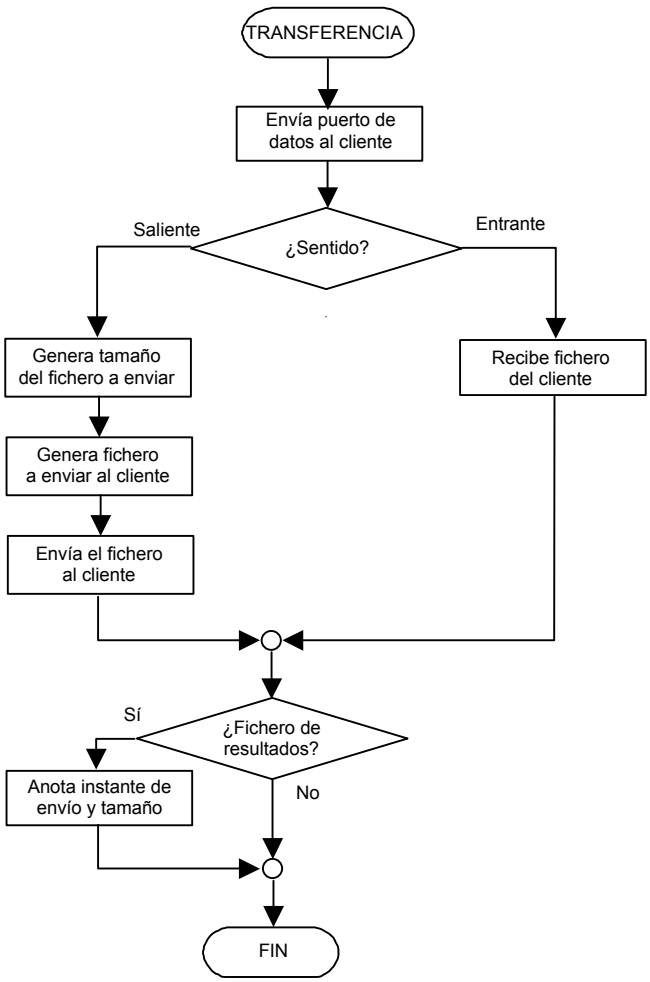


Figura 4.17. Flujograma de una transferencia en el servidor FTP

4.1.5. MODELO DE VÍDEO

Este modelo simula una transmisión de tráfico de vídeo en tiempo real sobre el protocolo UDP. El cliente inicia el proceso enviando un paquete de tamaño '1' al servidor, que lo utilizará para conocer la dirección IP y el puerto al que debe mandar la información solicitada.

Seguidamente figuran los flujogramas que representan los procesos de este modelo:

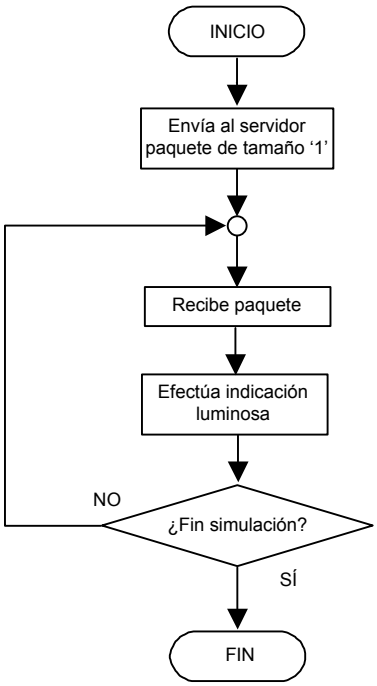


Figura 4.18. Flujograma del cliente en el modelo de tráfico de vídeo

El servidor presenta casi toda la complejidad del modelo. Tarea suya es la realización de todos los cálculos, desde lapsos de tiempo hasta la definición de todos los tamaños necesarios.

Entre los cómputos a realizar se encuentran complejos modelos, como el autorregresivo AR(n), o elaboradas estructuras como el grupo de imágenes (GOP) que se maneja en el modelo propuesto por el grupo MPEG (ver apartado 2.5).

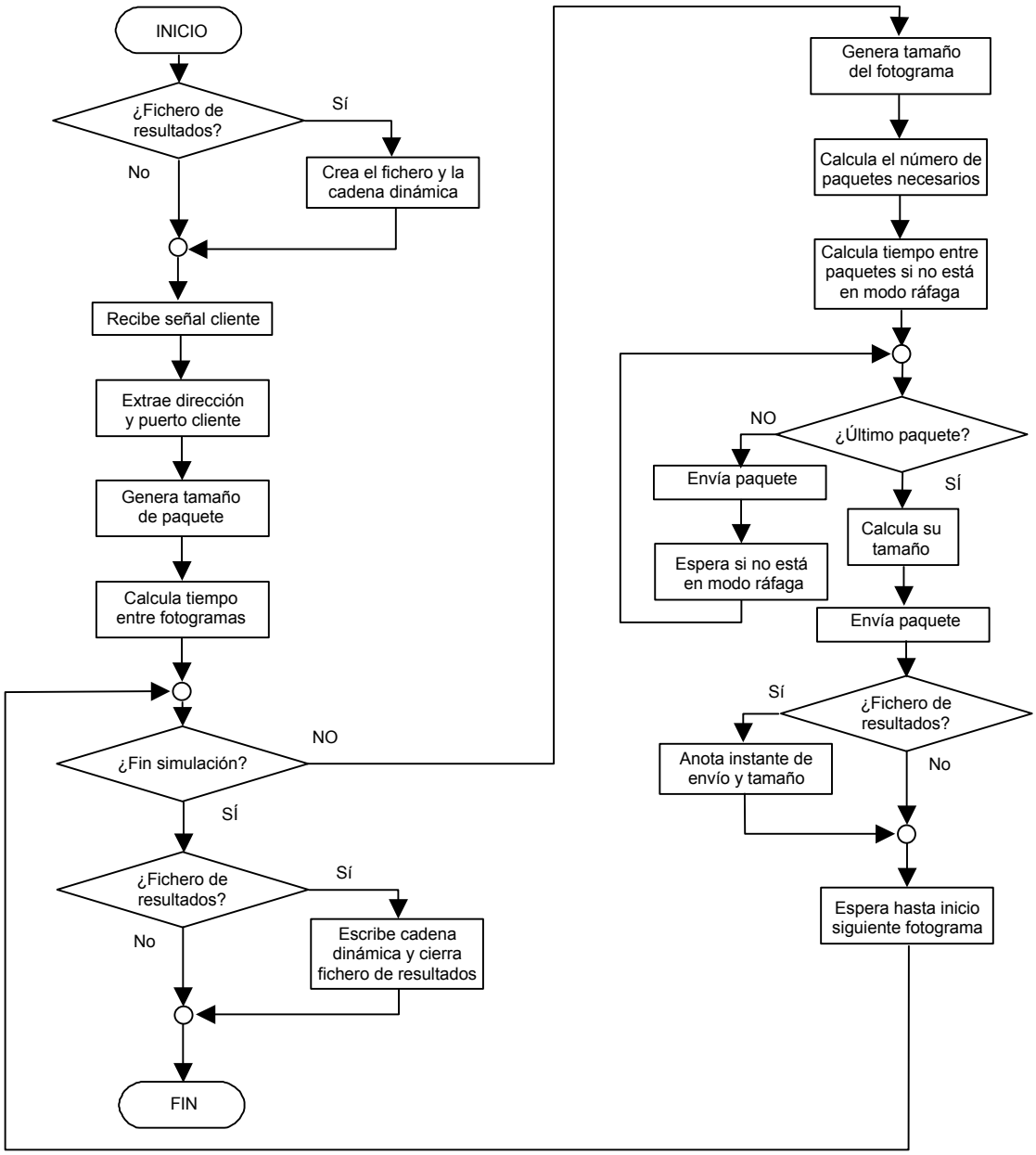


Figura 4.19. Flujograma del servidor en el modelo de tráfico de vídeo

4.1.6. MODELO DE VOZ

Implementado sobre el protocolo UDP, este modelo presenta complejidades similares en el cliente y en el servidor. Para la indicación de cambio de estado se reserva el paquete de longitud nula, mientras que el paquete de longitud '1' se emplea para indicar el final de la simulación. A continuación se muestran los flujogramas correspondientes a los procesos cliente y servidor.

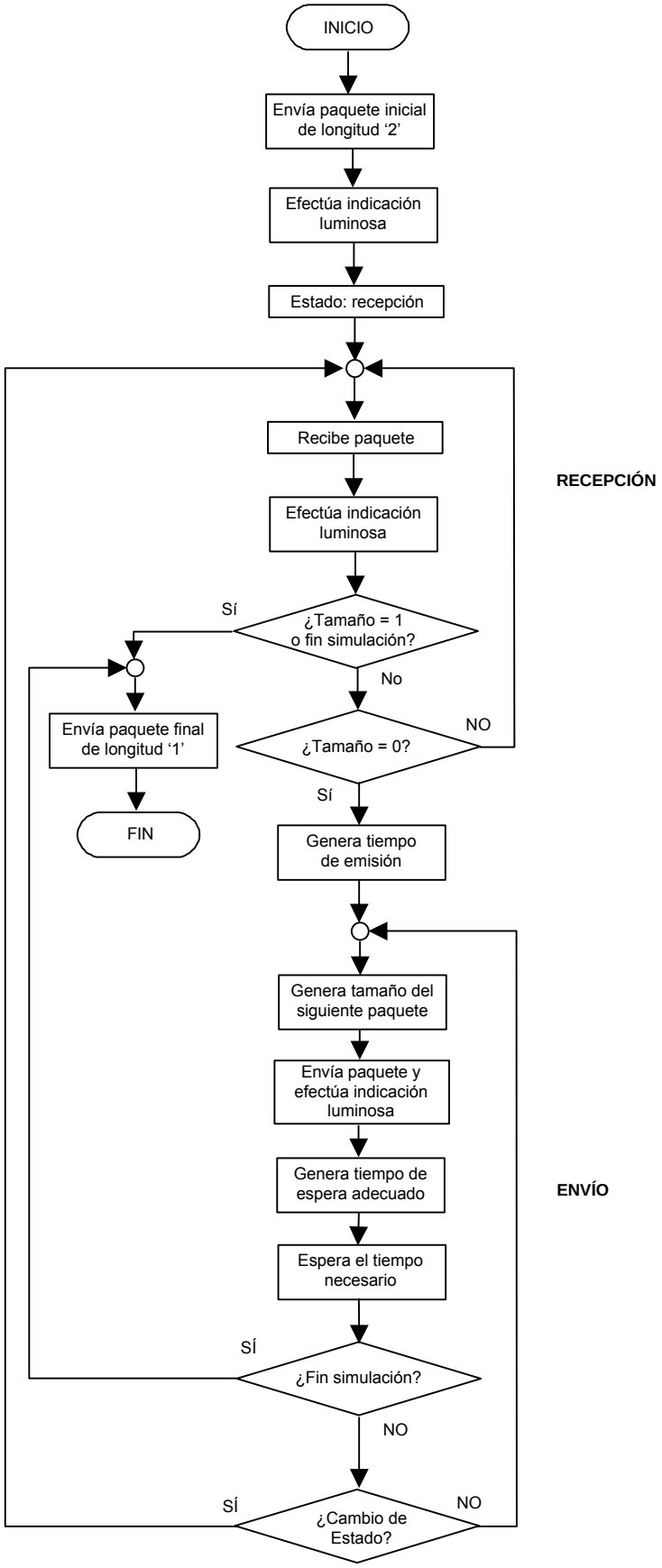


Figura 4.20. Flujograma del cliente en el modelo de tráfico de voz

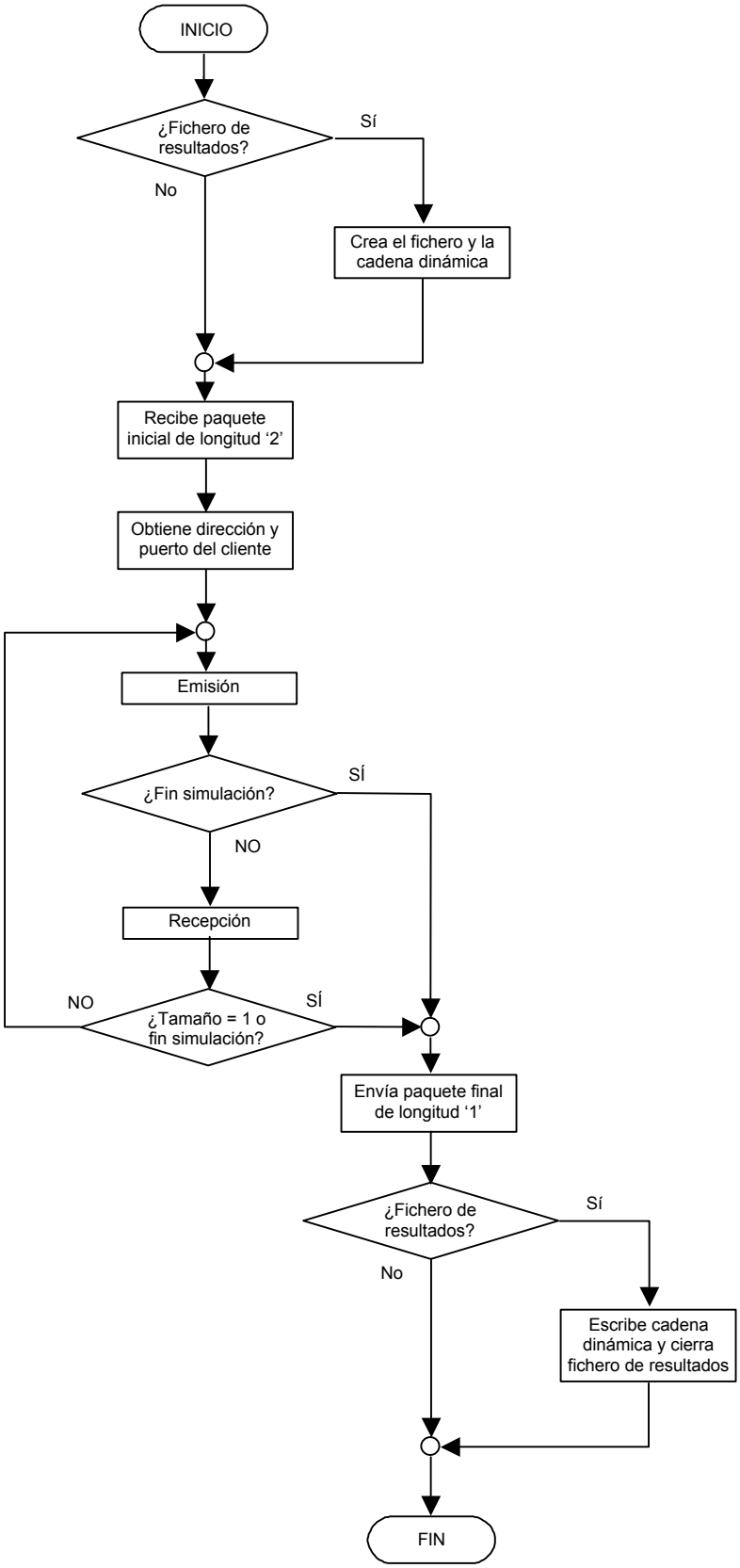


Figura 4.21. Flujograma del servidor en el modelo de tráfico de voz (simplificado)

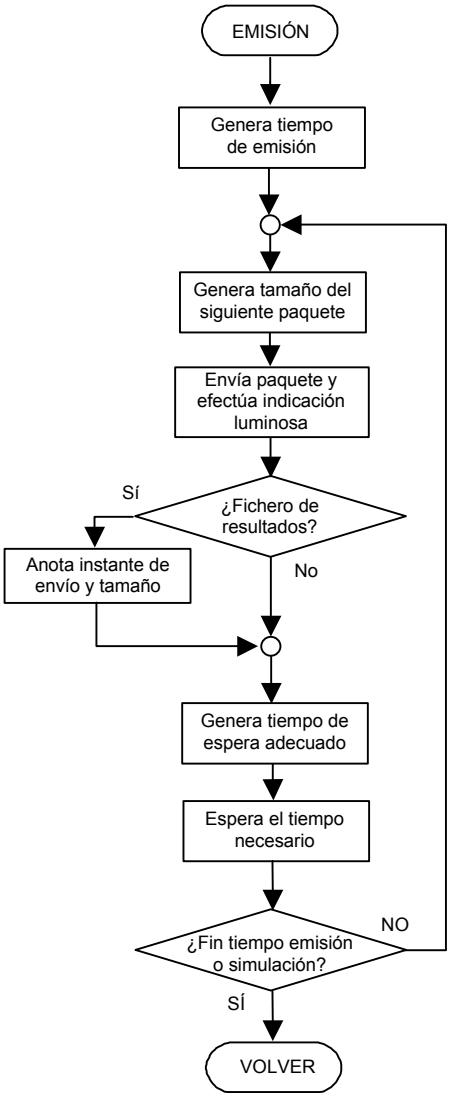


Figura 4.22. Flujograma del estado 'EMISIÓN' en el servidor de voz

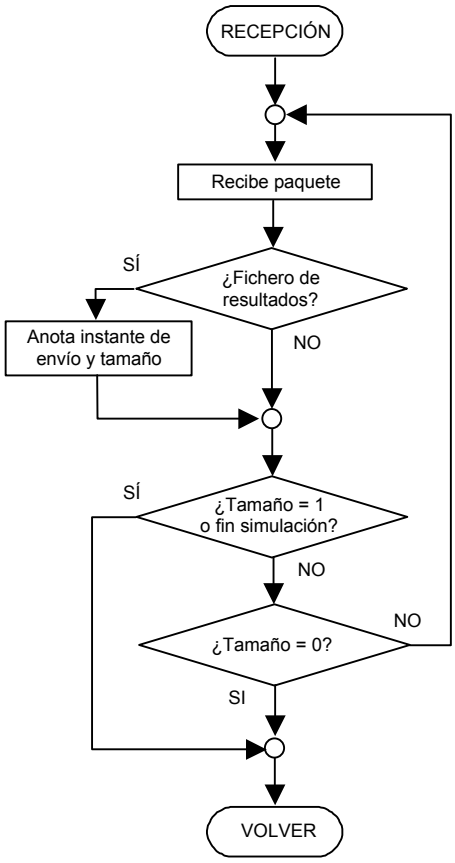


Figura 4.23. Flujograma del estado 'RECEPCIÓN' en el servidor de voz

4.1.7. MODELO DE TRÁFICO WWW

Este modelo está diseñado, al igual que el de FTP, para la generación de tráfico agregado, mediante la creación de procesos independientes que ejecutan las tareas correspondientes a los distintos niveles de abstracción contemplados.

A la hora de simular el nivel de paquete, si esto se solicita, el tamaño de la transferencia a realizar durante una conexión se envía dividido hasta que queda por enviar un fragmento menor que el tamaño de paquete manejado. En ese caso, se enviará un paquete de tamaño menor, suficiente para completar el envío solicitado. Funciona sobre el protocolo TCP.

A continuación se muestran los flujogramas de los procesos del modelo WWW:

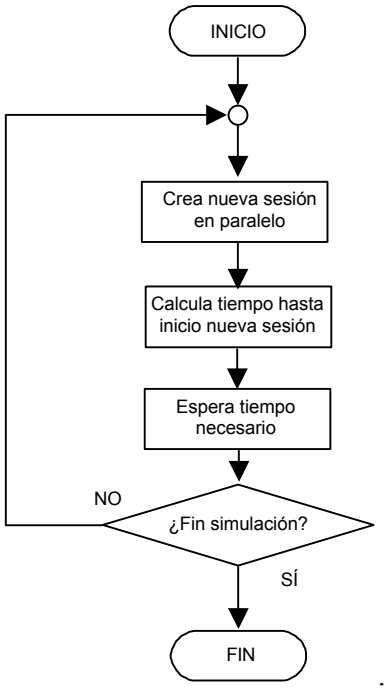


Figura 4.24. Flujograma del proceso global en el cliente de tráfico WWW

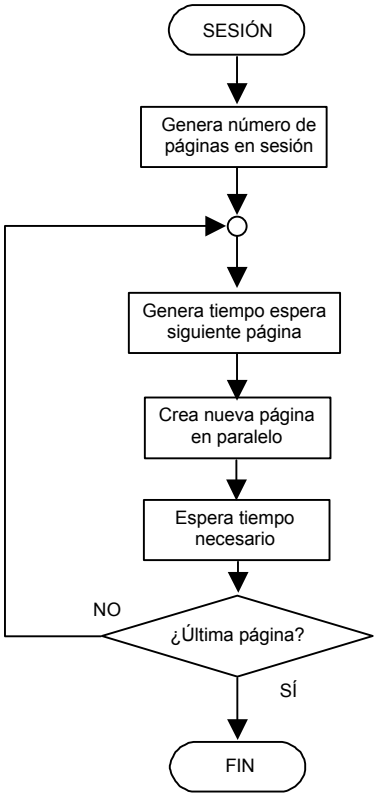


Figura 4.25. Flujograma de una sesión en el cliente de tráfico WWW

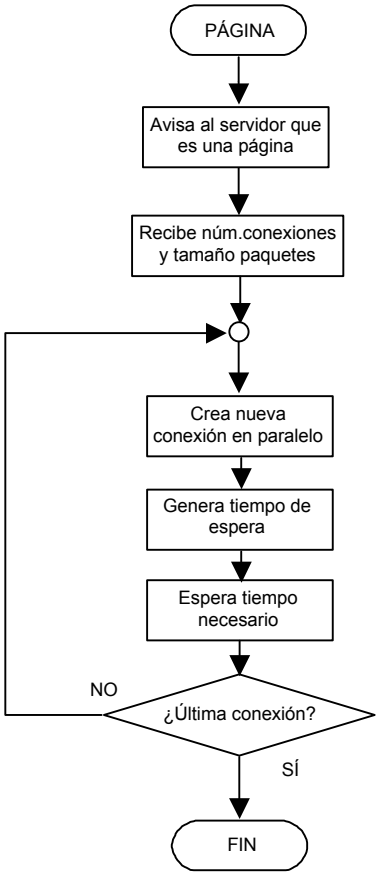


Figura 4.26. Flujograma de una página en el cliente de tráfico WWW

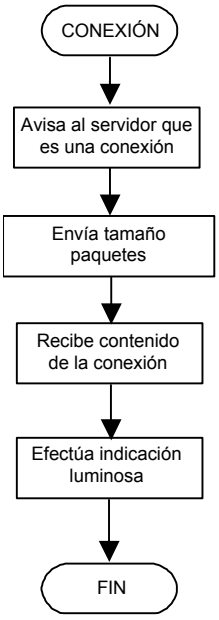


Figura 4.27. Flujograma de una conexión en el cliente de tráfico WWW

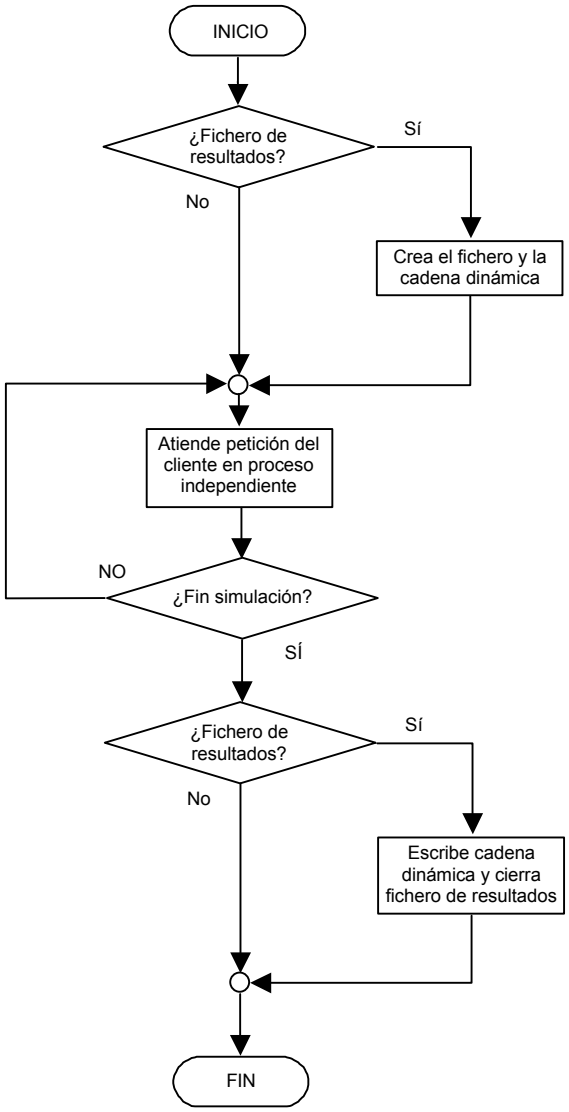


Figura 4.28. Flujograma del proceso global en el servidor de tráfico WWW

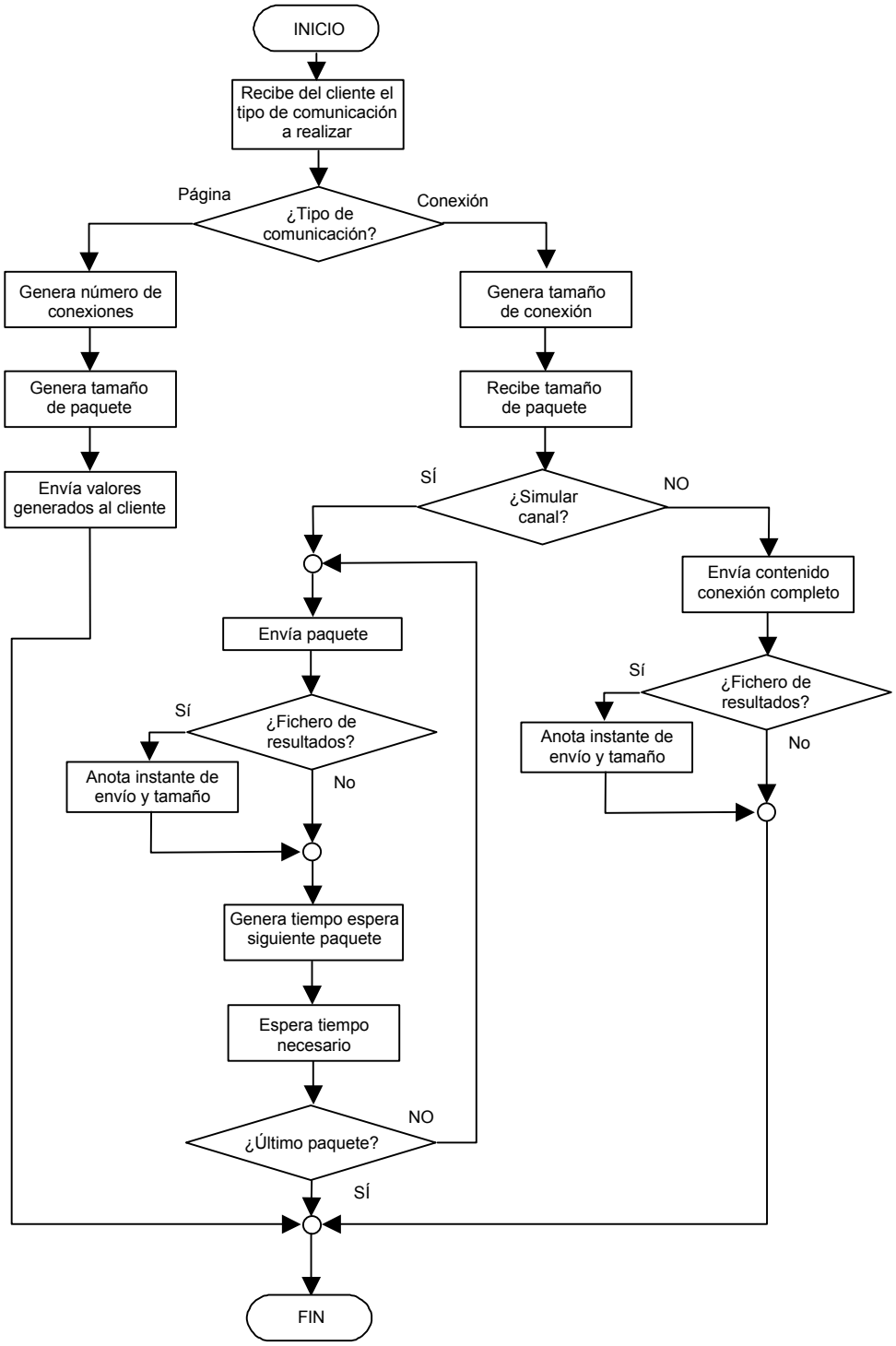


Figura 4.29. Flujograma del proceso de atención a una petición del cliente en el servidor WWW

4.2. INTERFAZ DE USUARIO

4.2.1. COMPONENTES JAVA UTILIZADOS

Para la realización de los interfaces de usuario de esta aplicación, se empleó la tecnología de más reciente aparición dentro de la plataforma Java, denominada *swing*. Las clases que la componen son totalmente configurables y proporcionan una serie de métodos que permiten realizar cualquier tipo de operación necesaria con ellas. A continuación se describen las clases pertenecientes al paquete `javax.swing` utilizadas.

- `JApplet`

Se trata de un componente situado en la parte alta de la jerarquía de *swing*. Proporciona un poderoso mecanismo mediante el cual se puede extender enormemente la funcionalidad de un programa navegador mediante su inclusión en cualquier página *web*.

- `JPanel`

Son contenedores de tipo intermedio, ya que pueden introducirse tanto dentro de un `JApplet`, u otro `JPanel`, como acoger a otros componentes en su interior. Resultan tremendamente útiles dentro de un interfaz de usuario también por su capacidad de anidamiento.

- `JButton`

Estos elementos son los botones que permiten realizar indicaciones y tomar opciones entre otras muchas utilidades. Entre ellos están:

  - `JButton`: Botón sencillo para realizar indicaciones.
  - `JCheckBox`: Casilla de dos estados (marcada o sin marcar).
  - `JRadioButton`: Otra versión de indicador de estados. Pueden asociarse de forma que sólo uno de entre varios esté activado.

- `JComboBox`

Elemento empleado para la realización de listas desplegables. Permite mostrar el componente seleccionado al replegarse, al igual que puede ser configurado de diversas maneras.

- `JTextField`

Consiste en un cuadro de texto destinado a recoger valores introducidos desde el teclado. Están diseñados para recibir pequeñas cantidades de texto.

- `JLabel`

Son etiquetas que permiten colocar porciones de texto a modo de título, por ejemplo, para indicar la naturaleza de otros componentes del interfaz de usuario. También pueden albergar imágenes.

- `ImageIcon`

Clase que representa la imagen que se va a utilizar en otro componente. Se puede crear una instancia de esta clase en base, por ejemplo, a un fichero de imagen o a una imagen representada por una URL.

- `JToolTip`

Consisten en pequeños carteles que aparecen cuando el puntero del ratón queda detenido sobre un componente. Contienen información que puede resultar de utilidad para el usuario mientras éste se hace preguntas acerca del componente sobre el que ha colocado el puntero del ratón.

4.2.2. COMPONENTES ESPECÍFICOS DISEÑADOS

En este apartado se efectúa una breve descripción de una serie de pequeños componentes que se han diseñado con objeto de componer cada uno de los interfaces de usuario que forman parte de cada modelo en la aplicación desarrollada.

- `PanelAR`

Panel que se muestra en el modelo de vídeo cuando se selecciona, como opción para generar valores, el modelo autorregresivo.

Figura 4.30. Aspecto en el interfaz de usuario del componente PanelAR.

- `PanelBoolean`

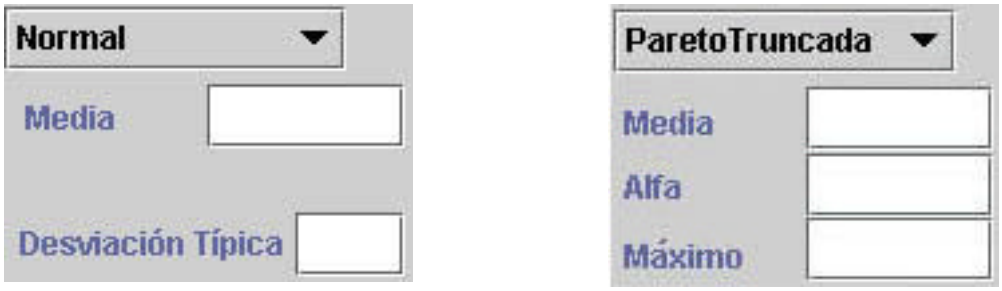
Presenta una característica que puede ser seleccionada o no.

a) No seleccionado **b) Seleccionado**

Figura 4.31. Panel de selección de una característica con dos opciones

- `PanelDatos`

Solicitan al usuario los valores de los parámetros de la distribución de probabilidad elegida. Se presentan dos ejemplos significativos.



a) `PanelDatosNormal` b) `PanelDatosParetoTruncada`

Figura 4.32. Ejemplos de paneles de datos dependientes de la distribución elegida

- `PanelDuración`

Común a todos los modelos, solicita del usuario la duración de la simulación en segundos.

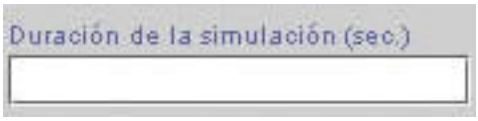


Figura 4.33. Panel donde introducir la duración de la simulación

- `PanelEstructuraGop`

Contiene las casillas necesarias para que el usuario indique al programa las proporciones según las que se generarán los GOP durante la simulación.

Figura 4.34. Panel para indicar la estructura del GOP

- `PanelProtocolo` y `PanelTipoEstado2`

Consisten en una pareja de botones que simbolizan una alternativa a una decisión cada uno. Se diferencian de los `PanelBoolean` en que, mientras estos últimos se refieren a la elección o no elección de una característica, `PanelProtocolo` y `PanelTipoEstado2` ofrecen dos alternativas diferentes con entidad propia.

a) `PanelProtocolo` **b) `PanelTipoEstado2`**

Figura 4.35. Paneles para seleccionar entre dos opciones diferentes

4.3. SERVLETS

A la hora de desarrollar una aplicación en un entorno *web* con cierta complejidad en la parte del servidor, la tecnología *servlet* de Java ofrece una extensión de la funcionalidad de un servidor con multitud de ventajas, como se ha explicado en capítulos anteriores.

En el siguiente gráfico se muestra una breve sinopsis del marco en que trabajan los *servlets*:

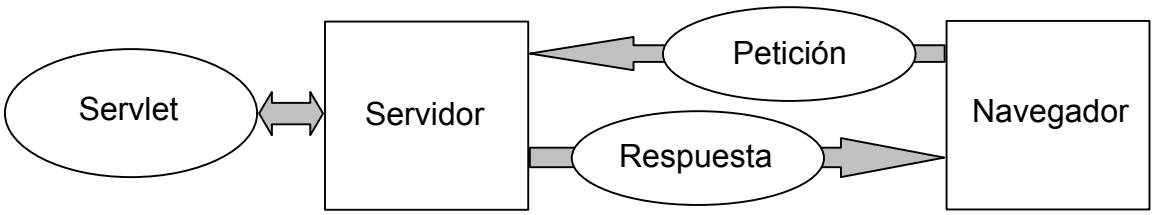


Figura 4.36. Esquema básico de funcionamiento de un *servlet*

4.3.1. COMPONENTES JAVA UTILIZADOS

- `HTTPServlet`

Contenida en el paquete `javax.servlet.http`, la clase `HTTPServlet` continúa la cadena de herencias de la arquitectura *servlet*, mostrada en la siguiente figura:

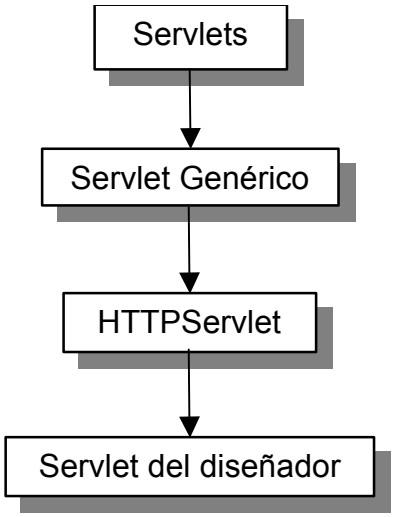


Figura 4.37. Jerarquía de herencias de la arquitectura Java servlet.

Esta clase aporta una base sobre la que construir *servlets* especialmente diseñados para una aplicación *web*. Cualquier subclase de `HTTPServlet` debe implementar al menos uno de sus métodos de servicio, comúnmente `doGet()` y `doPost()`, que serán los encargados de manejar los comandos `GET` y `POST` del protocolo HTTP recibidos a modo de petición.

- Paso de parámetros a los métodos de servicio

Los métodos de servicio recién mencionados reciben como argumentos objetos de tipo `HTTPServletRequest` y `HTTPServletResponse`, que forman parte del paquete `javax.servlet`. El primero de ellos contiene los parámetros del comando HTTP mediante el cual se ha invocado al *servlet* donde figuran. Proporciona una serie de métodos indicados para extraer esos parámetros según sea conveniente en cada momento. El

segundo, por su parte, ofrece un conjunto de métodos con los que el *servlet* puede generar una respuesta de tipo HTTP que será devuelta al cliente que lo invocó.

- `ObjectInputStream` y `ObjectOutputStream`

Pertenecientes al paquete `java.io`, forman parte del sistema de entrada/salida de Java. Tienen una función muy específica en la programación de *servlets*, que consiste en el envío y recepción de Objetos completos entre dos clases separadas. En la aplicación desarrollada existen unos objetos "de configuración" (véase apartado 4.1.1) que contienen, entre otras cosas, todas las características del lado servidor de la simulación que el usuario indica a través del interfaz gráfico. Estos objetos, al implementar el interfaz `java.io.Serializable`, pueden ser leídos y escritos de forma binaria por clases distintas, que pueden estar ubicadas en máquinas diferentes.

De este modo, la clase encargada de ello en el cliente (`Invocador*.java`) crea un canal `ObjectOutputStream` a partir de una conexión que veremos más adelante y efectúa una llamada al método `writeObject()` de dicho canal. A su vez, el *servlet* que está siendo ejecutado ha creado un canal `ObjectInputStream` a partir de la misma conexión (que le une al anterior) e invoca el método `readObject()`. Así, el objeto de configuración aparece en la máquina del servidor en el mismo estado en que se encontraba en el cliente, listo para ser utilizado a voluntad.

- `URL`

Perteneciente al paquete `java.net`, la clase `URL` representa un recurso disponible en Internet (*Uniform Resource Locator*). Es una forma de simbolizar la situación de archivos, directorios o elementos más complicados como referencias a una base de datos o a motores de búsqueda de contenidos.

- `URLConnection`

También incluída en el paquete java.net, la clase `URLConnection` aporta un mecanismo para efectuar conexiones con recursos representados por objetos de tipo `URL`.

Proporciona una extensa gama de métodos que permiten realizar una gran cantidad de operaciones en base a ella. Representa una conexión entre quien la crea y el objeto representado por la `URL` a la que se refiere. Gracias al sistema de entrada/salida de Java, es posible crear distintos canales en base a una conexión de este tipo.

4.3.2. IMPLEMENTACIÓN DE LOS SERVLETS

Una vez que el cliente ha reunido los parámetros necesarios para la simulación, llega el turno de los *servlets*. Es el momento en que se hace necesaria la extensión de funcionalidad que ellos aportan al servidor.

En el cliente, la clase invocadora mencionada anteriormente crea un objeto de tipo `URL` con la dirección del *servlet* que corresponde a su modelo de tráfico (`Servlet*.java`). Seguidamente, establece una conexión en base a dicha `URL` con el método `URLConnection.openConnection()`. Con el propósito de mandar el objeto de configuración, se crea un canal de tipo `ObjectOutputStream`, en el que se escribe el objeto completo. En el servidor, mientras tanto, el *servlet* ha sido invocado al conectar con la `URL` que lo representa y ha efectuado la lectura del objeto, que, por consiguiente, se encuentra disponible también en el lado del servidor.

Seguidamente, el cliente crea un canal de entrada de texto de tipo `BufferedReader`, mientras que el *servlet* crea uno de salida de texto de tipo `PrintWriter`, a través del cual hace llegar al cliente una confirmación de que todo ha ido bien, por el momento. En ese instante, ambos extremos proceden a crear la conexión de control, tras lo cual, el cliente activa el indicador luminoso de color rojo, que permanecerá encendido hasta el fin de la simulación. Una vez alcanzado este punto, cada extremo lanza el proceso encargado de generar el tráfico propiamente dicho, es decir, se pone en marcha la simulación.

Cuando llega el momento de finalizar la simulación en curso, bien porque haya acabado el tiempo solicitado, bien porque el usuario haya accionado el botón de detención, se envía la orden de corte a través de los canales para envío y recepción de texto creados en base a

la conexión de control. Entonces, cada uno de los extremos procederá a detener los procesos correspondientes y cerrar convenientemente los canales creados con objeto de que estén disponibles para una nueva simulación lo más pronto posible. Para indicar esto, el cliente apagará el indicador luminoso de color rojo, que había permanecido encendido desde el comienzo de la simulación.