7. El programa OperAIT.

Los capítulos anteriores han descrito cada una de los componentes que forman la aplicación objetivo, tanto módulos utilizados en el desarrollo de la misma como el controlador modificado de la tarjeta RDSI que permite utilizar las facilidades de red proporcionadas por la centralita Neris 64S. El programa final OperAIT es muy sencillo de implementar utilizando todas estas herramientas, por lo que ocupa pocas líneas de código y fue rápidamente implementado.

La mayor parte del código está contenida en el módulo `datos.o`, que se ocupa de iniciar el sistema, cargar la configuración y ejecutar el algoritmo de procesamiento de nodos[9]. Aparte de este módulo, existe un módulo llamado `main.c` que contiene las definiciones de funciones necesarias para poder utilizar la AVM CAPI ADK, establecer las rutinas de manejo de señales y ejecutar el bucle principal de programa.

7.1. Iniciación del sistema.

Previamente a la recepción de la primera llamada, es necesario realizar varias tareas de iniciación del sistema:

- Registrar la aplicación con CAPI 2.0.
- Establecer las rutinas de captura de las señales `HUP` y `TERM`.
- Iniciar la CAPI AVM ADK.
- Cargar la configuración inicial e iniciar la máquina de estados de comunicación con el usuario.

Una vez todas estas tareas hayan finalizado correctamente, el programa podrá comenzar a recibir llamadas y procesar los eventos recibidos desde el teclado del teléfono del usuario siguiendo el algoritmo de comunicación con el mismo.

Las tres primeras tareas de iniciación se realizan en el módulo `main.o`. Las dos últimas se realizan en el módulo `datos.o`.

---

[9] Este algoritmo define el comportamiento del programa frente a los eventos recibidos del usuario según la configuración activa.

7.1.1. Registro de la aplicación con CAPI 2.0.

El registro de la aplicación indica a CAPI 2.0 que OperAIT va a utilizar las facilidades de la misma, y causa que el controlador CAPI 2.0 inicie sus variables internas para comenzar el envío y recepción de mensajes CAPI 2.0.

El registro de la aplicación se realiza mediante la llamada a la función `RegisterCAPI`, implementada importada de la biblioteca AVM CAPI ADK.

7.1.2. Rutinas de captura de señales.

Existe dos señales que OperAIT captura y procesa: `HUP` y `TERM`. La recepción de la señal `HUP` se interpreta como una orden de reconfiguración, y fuerza una nueva lectura del archivo de configuración tan pronto como no exista ninguna llamada activa. La señal `TERM` indica la orden de finalizar la aplicación, y a su recepción la aplicación cierra todas las conexiones, libera la memoria y finaliza su ejecución.

La señal `HUP` es capturada por la función `ReloadConfiguration`, y su único cometido es indicar la orden de reconfiguración estableciendo a 1 la variable `ReloadConf`. Más tarde, en el bucle principal del programa, se comprueba el valor de esta variable y, si no existe ninguna llamada activa, se procesa la nueva configuración.

La función `StopProcess` captura la señal `TERM` y realiza una llamada a la función `DestroyAll`, que libera toda la memoria y cierra las conexiones. Posteriormente `StopProcess` se libera el registro con CAPI 2.0 y termina el programa mediante la función `exit`.

7.1.3. Iniciación de la biblioteca CAPI AVM ADK.

La biblioteca CAPI AVM ADK debe ser iniciada, lo que se realiza mediante la llamada a la función `InitConnectionIDHandling`. Al retorno de esta función, posteriormente a la carga de la configuración de OperAIT, podrá realizarse la llamada a la función `Listen` para comenzar a recibir llamadas.

7.1.4. Configuración inicial e iniciación de la máquina de estados.

La función `configure` gestiona todos los intentos de configuración, tanto la configuración inicial del sistema como posteriores configuraciones *en caliente* del mismo. Para ello realiza una llamada a la función `InitData` y notifica el intento de reconfiguración a través de la salida de errores, indicando si hubo fallo o si la reconfiguración se realizó correctamente.

La función `InitData` inicia las variables que utilizará la máquina de estados principal -que procesa los eventos de teclado enviados por el usuario y actúa según la configuración- y procesa el archivo de configuración. Para procesar el archivo de configuración, busca primero en el directorio de trabajo de la aplicación y, si no lo encuentra, en el directorio `/etc`. Si se trata de un intento de reconfiguración fallido, la configuración anterior se mantiene, indicándose la condición de error mediante la salida de errores.

Una de las operaciones más importantes de la función `InitData`, una vez el archivo de configuración se ha procesado correctamente, es abrir el archivo que almacenará la monitorización del sistema. Cuando este archivo haya sido abierto correctamente, `InitData` duplica la salida de errores al descriptor de archivo asociado al mismo y cierra la salida de errores. De este modo se consigue que, a partir de entonces, todos los mensajes que iban a la salida de errores se almacenen en este archivo. Por ello todos los mensajes de monitorización se dirigen a la salida de errores. En cambio, si existiera un error en la configuración inicial del sistema éste se mostraría por la salida de errores, puesto que no se duplica el descriptor de fichero en tal caso.

Una vez la función `InitData` ha retornado correctamente, el programa principal realiza la llamada a la función `Listen` y comienza el bucle infinito que procesa las llamadas de los usuarios.

7.2. Definición de las funciones de *callback*.

En el capítulo 4 se comentó que para utilizar la biblioteca AVM CAPI ADK es necesario definir un grupo de funciones de *callback*, cuya declaración se encuentra en el archivo `amvcapiadk.h`. La lista completa de funciones, así como el evento que causa su ejecución, se enumera en la siguiente tabla.

Nombre	*Evento*
`MainDataAvailable`	Recepción de un bloque de datos
`MainDataConf`	Recepción del mensaje DATA_B3_CONF[10]
`MainStateChange`	Cambio de estado en la conexión
`MainIncomingCall`	Llamada entrante
`MainDTMFReceived`	Recepción de tonos DTMF
`MainRedirReceived`	Confirmación de un intento de transferencia de llamada

Tabla 9. Funciones de callback *y eventos asociados.*

[10] El mensaje CAPI 2.0(y el correspondiente Q.931) DATA_B3_CONF confirma la recepción por parte de la red de un mensaje de datos enviado por el usuario a través de la interfaz RDSI.

Todas estas funciones tienen su definición en el archivo `main.c` de la aplicación OperAIT, aunque `MainDataConf` simplemente retorna sin realizar ninguna acción.

7.2.1. MainDataAvailable.

La recepción de datos vocales es continua una vez se ha establecido una conexión. Por ello, el evento de recepción de datos vocales se utiliza para activar el algoritmo de procesamiento de nodos, que decidirá si es necesario enviar nuevos datos, intentar una transferencia de llamada o terminar la conexión. También la recepción de tonos vocales causa la ejecución del algoritmo de procesamiento de nodos, como se verá más adelante.

Cuando se reciben datos vocales de la red, la función `MainDataAvailable` realiza las siguientes tareas:

- Descarta los datos vocales [11].
- Realiza una llamada a la función `GetPCMData`, la cual almacena en el *buffer* de nombre `PCM` -que recibe como parámetro- los datos vocales PCM siguientes a enviar a la red. Para ello, `GetPCMData` utiliza parte del algoritmo de procesamiento de nodos, que será descrito más adelante.
- Realiza la conversión de los datos PCM al formato utilizado por CAPI 2.0 mediante una llamada a la función `PCMaRDSI`.
- Envía los datos a la red mediante la función `SendData`.

Siguiendo este comportamiento, se enviarán datos vocales al mismo ritmo que se reciben, por lo que no es necesario establecer temporizadores para conseguir un envío de datos a velocidad constante.

7.2.2. MainStateChange.

Esta función realiza dos acciones importantes. Por un lado, comprueba si el estado al que pasa una conexión es `Disconnected` (desconectado) y, si es así, reinicia las variables que almacenan el estado de esa conexión realizando una llamada a la función `RemoveConnection`.

Por otra parte, comprueba si el estado de la conexión es `Connected` (conectado) y, si es así, realiza una llamada a la función `FacilityRequest` indicando el tipo de facilidad solicitada mediante la constante `FacilityDTMFStart`, pasada como parámetro. A partir de entonces se comenzará a recibir eventos de tonos DTMF para dicha comunicación.

[11] Recordar que los tonos DTMF se reciben como eventos distintos a través de la función `MainDTMFReceived`, por lo que los datos vocales no se utilizan.

7.2.3. MainIncomingCall.

Cuando se recibe un mensaje de llamada entrante, la función `MainIncomingCall` comprueba si existe un canal libre y, si es así, acepta la llamada mediante una llamada a la función `AnswerCall`.

7.2.4. MainDTMFReceived.

El otro evento que puede causar la ejecución del algoritmo de procesamiento de nodos es la recepción de tonos de teclado desde la red.

La función `MainDTMFReceived`, para cada uno de los códigos de teclado recibidos, realiza una llamada a la función `ProcessOption`. Esta función comprueba si la tecla pulsada, junto con el conjunto de teclas pulsadas desde la última comprobación realizada, corresponde a un código de opción válido.

Para ello, `ProcessOption` comprueba la situación del usuario dentro del menú de opciones, definido en la configuración de la aplicación OperAIT. Si no es un código válido, se continuará procesando el primer nodo hermano del actual. Si sí lo es, se continúa procesando el nodo hijo de aquél que almacenara el código válido.

7.2.5. MainRedirReceived.

Esta función simplemente comprueba el valor de retorno recibido para el intento de transferencia de llamada realizado y, si corresponde, notifica el error a través de la salida de errores.

7.3. Algoritmo de procesamiento de nodos.

El algoritmo de procesamiento de nodos determina el comportamiento del programa ante los eventos recibidos de la red, una vez se ha establecido una conexión, utilizando para ello la información de configuración obtenida del archivo de configuración XML.

Este algoritmo es bastante sencillo, pero gracias a la versatilidad de la sintaxis de configuración puede conseguirse que el programa presente a los usuarios menús de opciones variados, según la configuración activa.

La implementación del algoritmo se encuentra divida en dos partes, en las funciones `ProcessNode` y `ProcessOption`, dentro del módulo `datos.o`. Cada una de estas partes gestiona la recepción de dos eventos distintos: llegada de datos vocales y detección de tonos DTMF, ambos enviados por el otro extremo de la comunicación.

7.3.1. Recepción de datos vocales.

Anteriormente se ha comentado que la función `GetPCMData`, llamada a su vez por la función `MainDataReceived` cuando se reciben datos vocales, ejecuta el algoritmo de procesamiento de nodos para conseguir los siguientes datos vocales a enviar a la red.

`GetPCMData` utiliza una cola donde se almacenan los nombres de los archivos que contienen los datos vocales a reproducir. Cada vez que se ejecuta la función, se comprueba si aún hay datos por enviar del archivo actual y los almacena para su envío en el *buffer* `PCM`. Cuando se llega al final del archivo, se cierra el mismo: la siguiente vez que `GetPCMData` se ejecute se abrirá el siguiente archivo de la cola y se comenzará a enviar sus datos. Hasta aquí no se ha utilizado el algoritmo de procesamiento de nodos.

Cuando no hay más archivos en la cola, `GetPCMData` realiza una llamada a la función `ProcessNode`, que es la función que realmente implementa la primera parte del algoritmo de procesamiento de nodos.

Existen tres casos diferenciados en esta parte según el tipo de nodo a procesar: nodo con sonido asociado, transferencia de llamada o salto a otro nodo. El siguiente diagrama de flujo representa a grandes rasgos el comportamiento de la función `ProcessNode`.

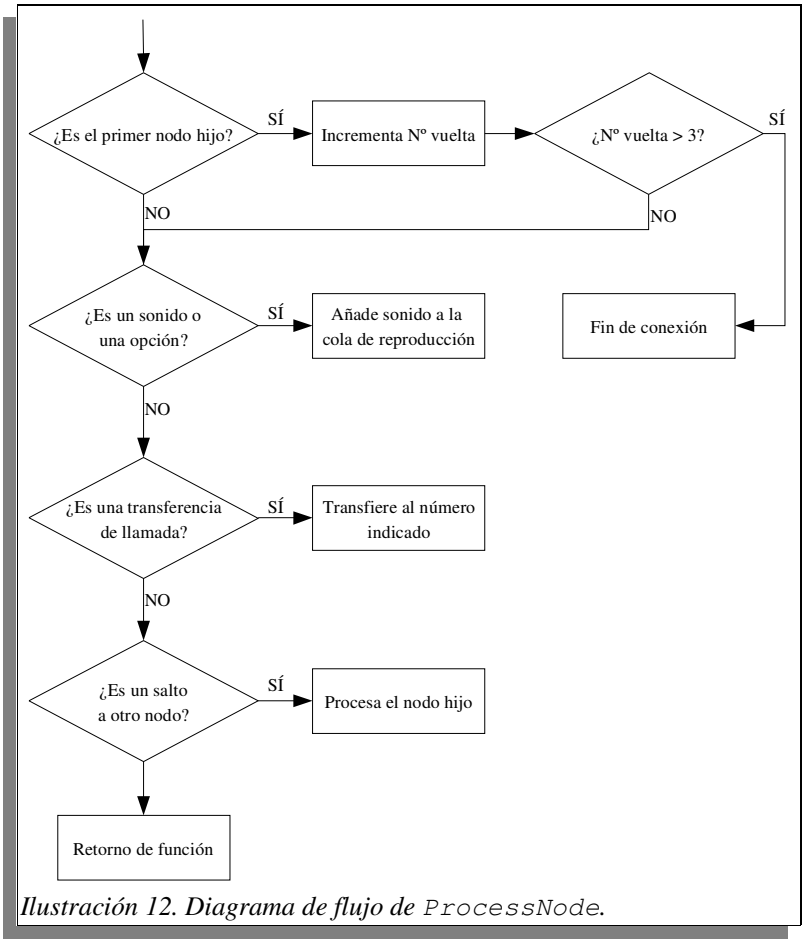

Ilustración 12. Diagrama de flujo de `ProcessNode`.

En el caso de que el nodo a procesar se trate de un nodo de tipo sonido o acción, se añadirá a la cola de reproducción el nombre del archivo de sonido o -si es una acción- los nombres de los archivos que almacenan el mensaje "Pulse" y los códigos de teclado asociado a la opción[12].

Por ejemplo, si el siguiente nodo es de tipo opción y el campo code asociado almacena el valor "15", se almacenarían en la cola los nombres de archivo contenidos en las marcas XML `wavpress`, `wav1` y `wav5`, en este orden.

Si el nodo a procesar es de tipo transferencia de llamada, se realizará una llamada a la función `FacilityRequest`, pasando como parámetro el tipo de facilidad asociado a una transferencia de llamada: `Facility_SS_Redir`. El valor de retorno del intento de transferencia será notificado posteriormente al programa mediante la función de *callback* `MainRedirReceived`.

[12] Los nombres de estos archivos se obtuvieron anteriormente de la información obtenida del archivo de configuración XML.

Por último, si el nodo es de tipo salto a otro nodo[13], la función almacena el valor del nodo a procesar en las variables de estado de la conexión y se llama recursivamente para seguir procesando el árbol. Hay que destacar que ni el módulo de configuración ni el programa comprueban la existencia de bucles de sentencias `goto` sin salida, por lo que un archivo de configuración mal construido podría provocar un bucle infinito de recursividad que acabaría llenando la memoria reservada a la pila de programa y finalizando la aplicación.

7.3.2. Recepción de eventos DTMF.

Como se ha visto anteriormente, los eventos de recepción de tonos de teclado se notifican, una vez se ha solicitado el uso de esta facilidad, a través de la llamada a función `MainDTMFReceived`. Esta función realiza una llamada a la función `ProcessOption` por cada pulsación de tecla detectada.

El siguiente diagrama de flujo representa el comportamiento de la función `ProcessOption`. Esta parte del algoritmo de procesamiento de nodos es algo más compleja que la anterior.

[13] Este tipo de nodo proviene de una etiqueta XML `goto` en el archivo de configuración.

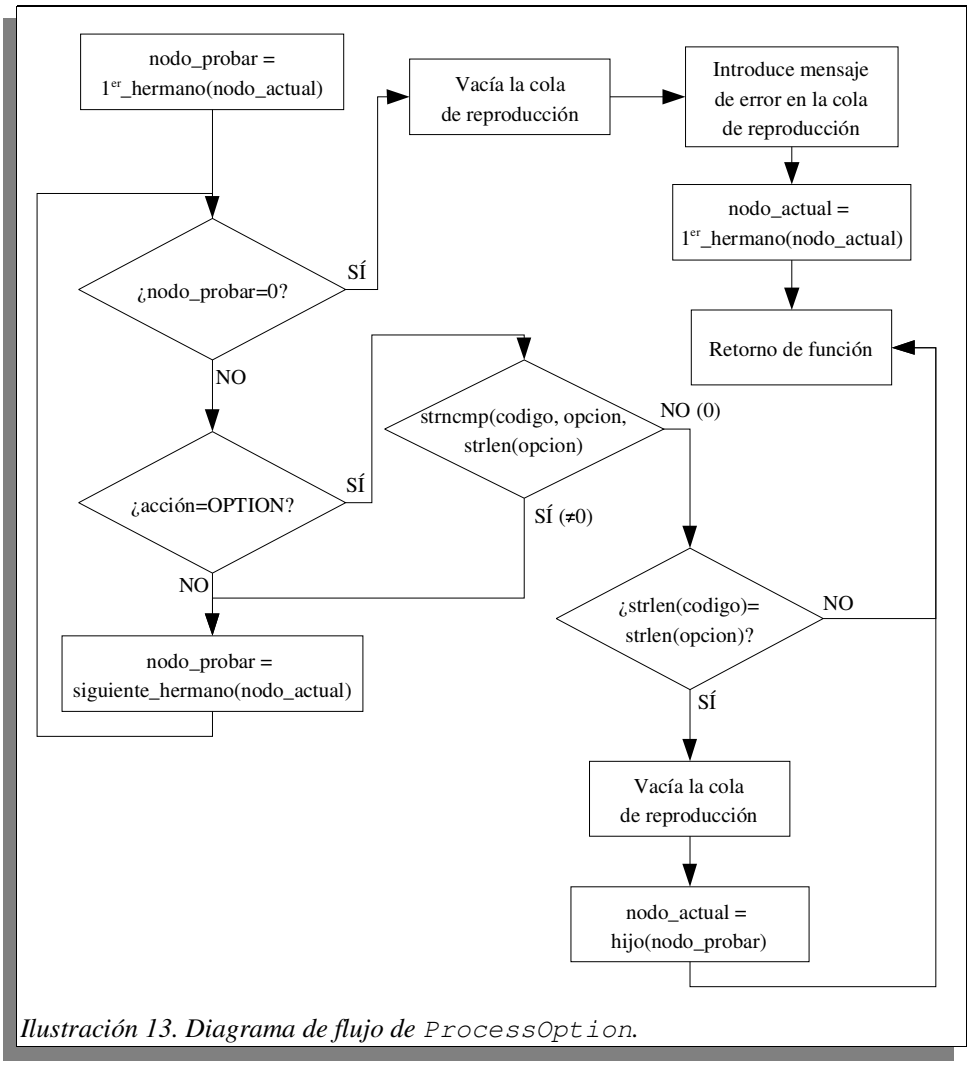

Ilustración 13. Diagrama de flujo de `ProcessOption`.

Para almacenar todas las cifras del código introducido por el usuario se utiliza una tabla de caracteres, que se vacía cada vez que el usuario selecciona una opción correcta o se produce un error por opción inexistente. Esta tabla se representa en el diagrama de flujo con el nombre `opcion`.

El algoritmo se sitúa en el primer nodo hermano del nodo procesándose actualmente y recorre todos los hermanos buscando nodos de tipo `OPTION`. Cada vez que encuentra un nodo de este tipo, comprueba si las teclas pulsadas por el usuario hasta este instante coinciden con los primeros caracteres del campo `code` del nodo[14].

[14] Esto se representa en el diagrama de flujo con la función `strncmp`. Hay que recordar que `strncmp` devuelve un valor 0 si los caracteres coinciden, lo que en el diagrama desviaría el flujo de programa hacia la rama marcada como NO(0).

Si los caracteres no coinciden, sigue recorriendo los nodos en busca de coincidencias. Si no encontrase coincidencia en ninguno de los hermanos, purgaría la cola de reproducción de sonidos y almacenaría en la misma el mensaje "Opción no válida", asociado a la marca XML `wavinvaloption`.

Si en alguno de los nodos encuentra coincidencia, comprueba seguidamente si la longitud del campo `code` y de `option` coinciden. Si es así, el usuario ha pulsado todas las teclas correspondiente a un código de opción correcto, por lo que vacía la cola de sonidos y modifica el nodo actual para que se procese el nodo hijo de `nodo_probar`[15]. Si no es así, el usuario deberá aún pulsar más teclas del teléfono para poder identificar si se ha introducido un código correcto o un código inexistente, por lo que la función retorna sin realizar ninguna acción más.

7.3.3. Consideraciones sobre el campo code.

Hay que hacer notar un efecto no evidente que puede ocurrir debido al algoritmo utilizado: si se configura el sistema de modo que el campo `code` de un nodo coincide con los primeros caracteres del campo `code` de otro hermano, sólo el que aparezca primero en el archivo de configuración podrá ser seleccionado por el usuario. Existen tres escenarios posibles: dos campos iguales, el primer campo es el de mayor longitud y el segundo campo es el de mayor longitud.

Si los dos campos son iguales o el segundo campo de mayor longitud que el primero, la segunda opción será inaccesible al usuario. Esto es así porque el primer nodo será seleccionado en cuanto se pulse la última tecla del campo `code`, pasando seguidamente el sistema a procesar su nodo hijo, lo que impide que el segundo nodo pueda ser seleccionado. Un ejemplo XML de esta situación sería el siguiente:

```
<option code="1">
      <redir>25</redir>
</option>
<option code="15">
      <redir>26</redir>
</option>
```

El segundo caso posible se da cuando el primer nodo tiene un campo `code` de mayor longitud que el segundo. En tal caso, el usuario no podrá seleccionar la segunda opción, puesto que el sistema quedará a la espera de otras teclas que completen la longitud de la

[15] Es decir, el nodo hijo de aquél cuyo campo `code` coincide en su totalidad con el código pulsado por el usuario en el teclado del teléfono.

opción hasta que ésta iguale la longitud del campo `code` del primer nodo. Un ejemplo de configuración XML para este caso es el siguiente:

```
<option code="15">
      <redir>25</redir>
</option>
<option code="1">
      <redir>26</redir>
</option>
```

En el ejemplo anterior, si el usuario pulsa la tecla 1, espera seleccionar la segunda opción, pero el algoritmo de procesamiento de nodos detecta que el primer carácter coincide con el primer carácter de "15", por lo que la función `ProcessNode` retorna sin llegar a procesar el campo `code` igual a "1".

Estos dos comportamientos pueden desconcertar a los usuarios, por lo que, para evitarlos, **se recomienda que todos los campos `code` de nodos hermanos tengan la misma longitud**. Así, si hay 10 nodos hermanos y que se desea numerar comenzando por 1, pueden usarse las combinaciones "01", "02", "03"... hasta "10".

7.4. Bucle principal de programa.

La función main de la aplicación OperAIT sólo realiza dos operaciones: iniciar el sistema y ejecutar el bucle principal de programa.

La iniciación del sistema se ha comentado anteriormente, y se realiza a través de llamadas a las funciones `RegisterCAPI`, `signal`, `InitConnectionIDHandling` y `configure`.

El siguiente diagrama de flujo representa el bucle principal de programa.

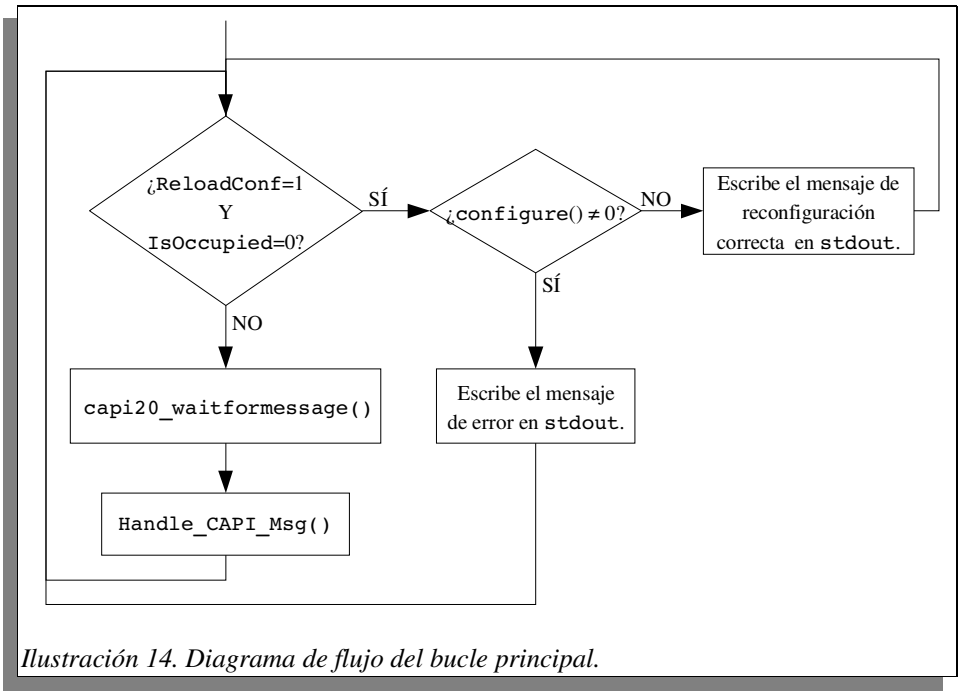

Ilustración 14. Diagrama de flujo del bucle principal.

El primer condicional comprueba si existe una orden de reconfiguración del sistema pendiente. La orden de reconfiguración se codifica con un valor distinto de cero en la variable `ReloadConf`, valor almacenado por la función que captura la señal `HUP`, `ReloadConfiguration`. Puesto que la configuración afecta a todas las conexiones, también se comprueba que no exista ninguna conexión activa mediante la llamada a la función `IsOccupied`.

Si existe orden de reconfiguración y no hay conexiones activas, se procesa de nuevo el archivo de configuración mediante la llamada a la función `configure`. Según el valor de retorno de esta función, se escribe en la salida de errores el mensaje de error o reconfiguración correcta.

Si no existe una orden de reconfiguración o, aunque se haya recibido esta orden, existe alguna conexión activa, se realiza una llamada a la función `capi20_waitformessage`. Esta función es similar a realizar una llamada a `select()` sobre un descriptor de fichero[16]: el programa pasa al estado `SLEEP` (dormir) hasta que recibe un mensaje de CAPI 2.0.

Cuando la función `capi20_waitformessage` retorna, existirá un mensaje CAPI 2.0 en la cola de mensajes del programa. Se realiza entonces una llamada a la función `Handle_CAPI_Msg`, que procesa el mensaje, utilizando la máquina de estados implementada por la biblioteca AVM CAPI ADK. Según el mensaje recibido, la máquina de estados

[16] De hecho, la implementación de `capi20_waitformessage` utiliza una llamada a la función `select` sobre el archivo `/dev/capi20`.

realizará llamadas a las funciones de *callback*, de forma que el programa ejecutará el algoritmo de procesamiento de nodos.