Capítulo 8

Interfaz de configuración

Índice del capítulo

8.1. Introducción

Hasta este momento, a lo largo de este documento, hemos presentado una serie de tecnologías y utilidades que intentan subsanar las carencias motivadoras de este Proyecto ya planteadas en su introducción. Dentro de este Proyecto de Fin de Carrera hemos querido buscar una solución que intente, en medida de lo posible, facilitar al máximo la implantanción en entornos «en estado productivo» de las distintas tecnologías que se han estudiado. Gracias al uso de los *paquetes Debian* y la automatización de su instalación se facilita en gran parte la labor de la primera implantación de las utilidades seleccionadas en nuestro sistema.

Sin embargo, la complejidad de uso y sobre todo de configuración de algunas de las aplicaciones que se han ido presentando en este documento impiden un uso simple y automatizado y a la vez flexible y potente de dichas aplicaciones. Esto anula en cierto modo el interés que estas utilidades pueden tener para su implantación tanto en entornos que ya se encuentran en funcionamiento como en sistemas que comiencen a desarrollarse.

Para solucionar parte de esta complejidad se ha desarrollado dentro de este Proyecto una interfaz de configuración para el manejo de algunas de las utilidades estudiadas. De esta forma se añade al Proyecto una fase radicalmente

distinta de desarrollo de una aplicación a la vez que se persigue lograr un producto más sólido que facilite el manejo de las utilidades seleccionadas.

En este capítulo abordaremos algunos asuntos de importancia de cara al desarrollo de dicha interfaz. En primer lugar realizaremos un estudio de viabilidad de la solución propuesta. A continuación se planteará el análisis de los requisitos de la aplicación que se pretende desarrollar. En el tercer punto comentaremos aspectos de diseño de la interfaz. Finalmente, en el apéndice E, "*Interfaz de configuración*", se verán, entre otros, aspectos relativos a la instalación y configuración de dicha interfaz así como detalles de la implementación de la misma.

8.2. Estudio de viabilidad

En esta fase, puesto que se trata de la realización de un Proyecto de Fin de Carrera, es obvio que se pretende realizar desarrollos allá donde sea necesario, así como reutilizar productos estándares (tales como librerías y componentes de *software*) para facilitar la construcción y propiciar la finalización del mismo en un tiempo razonable. Se tendrá en cuenta por tanto la inclusión de productos estándares y reutilizables allá donde sea posible y cuya licencia de uso lo permitan.

En una vista general del Proyecto se observa que el objeto de trabajo está dividido en dos partes bien diferenciadas, las cuales son:

1. Estudio de problemática y planteamiento de posibles soluciones
2. Interfaz de configuración para un usuario humano

La primera parte se centra más en el estudio de componentes *software* estándares y en la configuración de los mismos sobre una plataforma de sistema operativo (en nuestro caso **Debian Sarge**), buscando su personalización para incluir únicamente lo necesario para cumplir los requisitos de este Proyecto (por motivos de seguridad, rendimiento y espacio) y automatización en la medida de lo posible tanto en el proceso de instalación (con objeto de facilitar su empleo como producto final) como en el funcionamiento autónomo. Mientras tanto, la segunda parte del Proyecto que ahora tratamos tiene un componente importante de desarrollo.

El objetivo de este Proyecto, como ya se ha comentado, es el estudio de una serie de herramientas y sistemas de monitorización que proporcionen al Administrador de Red información adecuada sobre el estado de la misma con la finalidad de solventar la problemática planteada en la introducción de este documento. Dentro del Proyecto, la interfaz de configuración tiene, por una parte, la misión de permitir a este Administrador la revisión del estado de funcionamiento y, por otra, la realización de tareas de configuración y ajuste de parámetros.

Con la interfaz se trata de generar un producto a medida (ya que se ha visto que no hay disponible en la actualidad una aplicación que realice la tarea requerida con el nivel de detalle y comodidad deseados, y más aún ajustándose a los componentes *software* elegidos). La interfaz no cubrirá todos los aspectos

tratados en este Proyecto, sino que se han escogido tan sólo algunas de las utilidades estudiadas. Los motivos que han llevado a la selección son:

1. Escoger utilidades de fácil integración en un entorno «en estado productivo». Con esto se busca evitar aplicaciones cuya instalación y configuración sean poco automatizables y muy particularizadas, lo que sería poco conveniente en sistemas y redes ya operativos y de cierta envergadura.

2. Seleccionar aplicaciones fácilmente escalables, de manera que un aumento en el tamaño de la red o en el sistema no oblige a un pesado proceso de reconfiguración tanto en el sistema que albergue la interfaz, en el nuevo sistema que se integra en la red o en cualquiera de los sistemas que ya lo integran.

3. Combinar utilidades que de manera conjunta ofrezcan una funcionalidad completa e interesante.

4. Evitar utilidades y funcionalidades de menor interés y que no propicien la finalización del Proyecto en un tiempo razonable.

Por estos motivos, la interfaz de configuración contempla a las siguientes utilidades:

1. Recolector de *NetFlow flow-capture*.

2. Analizador *FlowScan*, con su módulo *CUFlow*.

Para la implementación se contará con el apoyo de librerías de clases estándares existentes (extraídas de repositorios de código reutilizable publicados en Internet, puestos a disposición de la comunidad *Open-Source* por grupos de desarrollo voluntarios y sin ánimo de lucro (como suele ser común el los proyectos GPL)). Estas librerías son:

- *PEAR::Config*: facilita el procesamiento de ficheros de configuración de texto en formatos conocidos y estándares (tales como XML, INI, o el estándar genérico «tipo Apache»). Véase [72].

- *patForms*: librería de generación de formularios *Web*, y de ayuda al procesamiento de los datos introducidos a través de los mismos. Véase [73].

- *patTemplates*: librería para el manejo de plantillas *Web*, fácilmente acoplable con patForms. Véase [74].

- *patError*: gestión de errores en la aplicación, utilizada exhaustivamente por patForms. Véase [75].

Estas librerías (desarrolladas en el lenguaje de *scripts* PHP) facilitan enormemente la tarea de, por un lado leer la información de configuración a partir de los ficheros de texto de los programas antes mencionados (en este caso mediante *PEAR::Config*), y por otro generar el formulario *Web* (con *patForms*, y las librerías auxiliares) que presentará una representación gráfica del contenido de dichos ficheros ante el usuario, ofreciéndole la posibilidad de modificar su contenido.

En el sentido inverso de la comunicación, el usuario puede introducir los valores que desee y enviarlos de vuelta a la aplicación (a la capa de lógica de negocio), pudiendo ser leídos con facilidad (de nuevo con la ayuda de *patForms*) y transformarlos para introducirlos posteriormente en los ficheros de configuración necesarios, con la ayuda de *PEAR::Config*.

Por tanto, vemos que con el modelo de componentes descrito, que se adapta perfectamente a la arquitectura de sistema propuesta (tanto en cuestion de sistema operativo Linux como de arquitectura de ejecución de tres capas), se consigue realizar la implementación de un sistema funcional en un tiempo de desarrollo adecuado, y con la funcionalidad requerida al completo.

8.3. Análisis

En esta fase y a partir de la elaboración de un catálogo de los requisitos del sistema (tanto funcionales como no funcionales) se persigue obtener un modelo de la interfaz del sistema que describa su funcionamiento y estructura.

8.3.1. Catálogo de requisitos de la interfaz

8.3.1.1. Requisitos funcionales

Mediante reuniones del equipo de trabajo y el estudio de las necesidades a las que se pretende dar solución con este proyecto, se identifican (referentes únicamente a la interfaz de configuración) los siguientes requisitos funcionales:

RF1: El sistema debe presentar el estado actual de funcionamiento de cada uno de los componentes *software* que pretende controlar.

RF2: El sistema debe presentar al usuario la configuración actual con la que se encuentran funcionando cada uno de los Componentes *software*.

RF3: El sistema debe permitir al usuario la modificación de los parámetros de configuración que se le ofrezca de cada uno de los componentes *software*.

RF4: El sistema debe detectar errores en la introducción de los datos de configuración (valores incorrectos, datos incoherentes, etc.) y evitar introducirlos en la configuración de los componentes *software* (a fin de evitar un mal funcionamiento de los mismos). Así mismo, si se da esta situación, debe presentar un mensaje de error al usuario advirtiendo de este hecho.

RF5: El sistema permitirá al usuario el control sobre la ejecución de un componente *software* íntimamente relacionado con el módulo objeto de configuración. En concreto, debe permitir al usuario el inicio y parada inmediatos de un componente *software*, así como la configuración en el sistema operativo de su inicio o parada automáticos (es decir, del arranque del servicio).

8.3.1.2. Requisitos no funcionales

Mediante la revisión del entorno tecnológico sobre el que operará la aplicación, y las necesidades y restricciones técnicas con las que se cuenta, se ha

elaborado el siguiente catálogo de requisitos no funcionales:

Requisitos sobre la arquitectura y usabilidad:

RNF1: El sistema debe funcionar sobre una arquitectura *Web* de tres capas, y estar desarrollado al completo de manera compatible con el lenguaje PHP y el servidor *Web* sobre el que éste se ejecuta.

RNF2: El sistema presentará al usuario un menú o índice que le llevará a las «pantallas» de configuración de cada uno de los módulos independientes (que controlan a su vez los distintos componentes *software* específicos).

RNF3: Los módulos emplearán un mecanismo de comunicación entre sí que les permita compartir cierta información (principalmente datos de configuración) con el objetivo de que el usuario no tenga que introducir un mismo parámetro en múltiples ocasiones (en varias pantallas diferentes). De esta forma se evitan posibles errores por inconsistencia en las configuraciones de distintos componentes *software*.

Este último requisito, tras estudiar lo que supondrá para el desarrollo de la aplicación, nos lleva a determinar un nuevo requisito no funcional (que catalogaremos también como requisito sobre la arquitectura), que resulta de importancia en cuanto a cómo afectará a dicho método de desarrollo:

RNF4: Se empleará Orientación a Objetos en la implementación de los módulos, con objeto de facilitar la reutilización de código y el paso de mensajes entre módulos.

Ello se debe a que la orientación a objetos permitirá independizar con mayor facilidad unos módulos de otros (mediante la propiedad del encapsulamiento), unificará interfaces, y permitirá el paso de mensajes: así, cuando un módulo requiera que otro adquiera y procese cierta información, se podrá realizar mediante este mecanismo con cierta facilidad.

RNF5: Se empleará XML preferentemente como lenguaje de descripción de estructuras de datos estáticas configurables y/o parametrizables (por ejemplo: para la descripción de los interfaces de usuario, y en los ficheros propios de configuración).

8.3.2. Modelo del sistema

Se busca en este punto crear un modelo que represente el sistema desde un punto de vista no detallado, como corresponde al proceso de Análisis, si bien los requisitos, y en especial el RNF4, dan ya de por sí cierta información acerca de cómo se deberá construir el sistema.

Para representar el sistema se han identificado los elementos de los que constaría y, modelándolos mediante Clases, se muestra en la Figura 8.1 un diagrama de clases del sistema (o, más formalmente, «Modelo de objetos de Análisis»), que muestra a alto nivel las diferentes clases y las relaciones entre ellas. En este modelo se ha conservado la estructura del PFC de *Óliver López Yela* y *Jose Carlos Ramírez Pérez Anubix: Servidor de seguridad perimetral.*

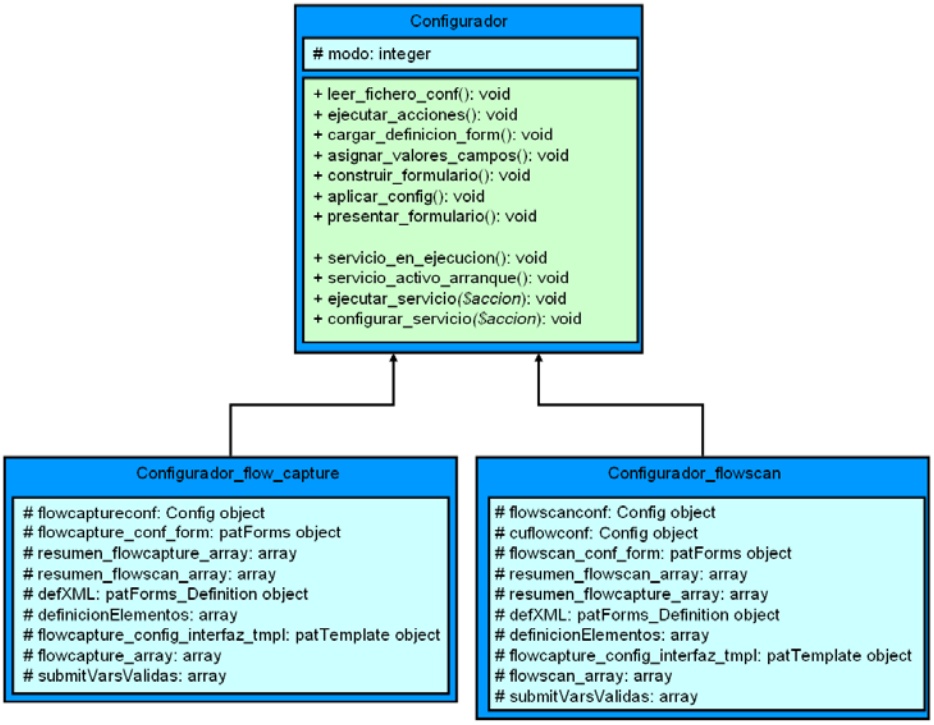

Figura 8.1: Diagrama de clases de la interfaz

8.3.2.1. Descripción del modelo

Como se puede ver en este diagrama, se tiene una clase principal **Configurador**, de la cual heredan una serie de clases hijas. Estas clases hijas tienen como misión particularizar la implementación para cada caso en concreto.

La clase **Configurador** es una abstracción del proceso de diálogo entre el usuario y el módulo que desea configurar (si bien, también podría entenderse como el módulo en sí mismo).

Las restantes clases hijas, que heredan de **Configurador**, tienen como misión implementar el funcionamiento de la clase para cada módulo en concreto, esto es: **Configurador_flow_capture** para el módulo del recolector *flow-capture*, **Configurador_flowscan** para el módulo del analizador *FlowScan*, etc... Esto permite, por ejemplo, en el futuro sustituir cualquiera de estas clases hijas por una implementación específica para otro componente de Software diferente (por ejemplo, otro recolector que no sea *flow-capture*), sin tener que modificar ni la implementación de los demás módulos, ni la interfaz que se emplea para dialogar con el módulo.

8.3.2.2. Justificación

Por los requisitos del sistema, como se ha visto anteriormente, podrá ser necesario que los diferentes módulos se comuniquen entre sí para proporcionar-

se mútuamente información de configuración. Esto implica que, una vez que se realicen cambios en la configuración de un módulo, los restantes deban leer estos cambios para «refrescar» su configuración. Esto se implementa mediante el paso de mensajes: el módulo con el que dialoga el usuario, y que realiza cambios en su configuración, comunica a los demás módulos que deben refrescar su configuración (para introducir cambios en los diferentes ficheros de configuración que se ven afectados).

Esta es la razón de ser del atributo *modo*, de la clase padre **Configurador**: cuando un módulo necesite refrescar su configuración, se instanciará en «modo Refrescar» (`MO_MODO_REFRESCAR`), en caso contrario, se instanciará en «modo Normal» (`MO_MODO_NORMAL`). Esto indicará al objeto internamente en qué «modo de funcionamiento» se encuentra, lo cual será de especial importancia en cuanto al procesamiento de la interfaz de usuario, como veremos posteriormente. Ello es debido a que un módulo que está en «modo Refrescar», no debe procesar información alguna referente a la interfaz de usuario (ya que en ese momento el usuario está dialogando con otro módulo). Este mecanismo evitará errores, y el desarrollador sabrá a «qué atenerse» en cada situación.

8.3.2.3. Descripción de atributos y métodos

Como se puede ver en el diagrama de la Figura 8.1, las clases hijas tienen en común que poseen dos tipos de atributos importantes: *config* y *form*. El significado de los mismos es el siguiente:

- *config* es la representación en memoria de la estructura y contenido del fichero de configuración.
- *form* es la representación de la estructura y contenido de los campos visibles en la interfaz de usuario.

Realmente se tratan de objetos (clases), que más tarde se verá cómo se han implementado (en este caso, utilizando librerías estándares disponibles). La idea es que un módulo tiene como principal función la transformación de los parámetros de los ficheros de configuración en campos visibles en el formulario y configurables por el usuario (y también el proceso inverso). Por tanto debe almacenar internamente tanto el conjunto de parámetros de configuración tal y como se almacenan en el correspondiente fichero, como el conjunto de campos que contiene la interfaz de usuario (con sus tipos, restricciones y formato de visualización concreto). De ahí la existencia de estos dos atributos.

La instanciación de cada uno de estos objetos, con la estructura de datos concreta que necesite cada módulo, será responsabilidad de dicha subclase.

En cuanto a los métodos, podemos agruparlos en:

1. Métodos de configuración

   - *leer_fichero_conf()* : lee el fichero (o ficheros) de configuración correspondientes al módulo, valida su sintaxis y semántica, e introduce su contenido en el atributo `config` del objeto.
   - *ejecutar_acciones()*: ejecuta las acciones inmediatas que precise el módulo durante su configuración (como arrancar o parar servicios) o las que el usuario solicite a través de la interfaz (acciones como: activar el servicio, establecerlo en el arranque, etc...)
   - *cargar_definicion_form()*: carga la definición del formulario (campos, tipos, apariencia) y genera la correspondiente estructura en memoria.
   - *asignar_valores_campos()*: rellena esta estructura en memoria con los valores leídos del fichero de configuración, realizando las transformaciones que sean necesarias.
   - *construir_formulario()*: crea el formulario gráfico que el usuario verá en pantalla, conteniendo toda la información anteriormente recopilada.
   - *aplicar_config()*: una vez que el usuario ha introducido los valores deseados, y envía estos al sistema, se capturan los mismos, se validan y se transforman para introducirlos de nuevo en los ficheros de configuración que sea necesario.
   - *presentar_formulario()*: se presenta finalmente el formulario al usuario, con los posibles mensajes de error fruto de las validaciones anteriores anexos al mismo.

   Como se puede ver, estos módulos se podrían ejecutar de una forma prácticamente secuencial en el flujo normal de ejecución de un módulo. Esto se ha diseñado así expresamente, con el objetivo de hacer el código más sencillo e inteligible, a la vez que fácilmente modificable y ampliable. Teniendo ésto en cuenta, la ejecución de dichos métodos siguen el flujo de datos normal de:

   Fichero de configuración → Validar y transformar → Cargar formulario → Rellenar formulario → Presentar formulario.

   Pero también es válido para este otro flujo «inverso»:

   Datos introducidos por el usuario → Validar y transformar → Cargar formulario → Rellenar formulario → Escribir configuración → Presentar resultados.

   Como se puede ver, lo único que varía es el origen (y el destino) de los datos, siendo gran parte del resto del proceso común a ambos flujos de datos. Ello es debido a que para poder escribir los datos, es necesario leerlos antes. Dicho de otro modo, el usuario introduce modificaciones en los datos que se leen de la configuración. Por tanto el flujo lectura-proceso-escritura se deberá seguir realizando en cada transacción de forma casi idéntica, ya sea sólo para mostrar datos, como para realizar modificaciones en los mismos. También es común en ambos flujos la presentación del

formulario al final. Es este proceso global lo que se implementa en los métodos enumerados anteriormente.

2. Métodos de control de servicios

   - *servicio_en_ejecucion()*: consulta si un servicio del sistema se encuentra actualmente en ejecución.
   - *servicio_activo_arranque()*: consulta si un servicio del sistema está configurado para arrancarse en el inicio.
   - *ejecutar_servicio(accion)*: realiza las acciones inmediatas necesarias para poner en marcha, o detener la ejecución, de un servicio del sistema.
   - *configurar_servicio(accion)*: realiza las acciones diferidas necesarias para configurar, en el arranque del sistema, la ejecución (o no) de un servicio.

   En cuanto a estos métodos, como es lógico, en cada módulo se implementarán para controlar el servicio del sistema que realiza la ejecución del componente *software* íntimamente relacionado con el módulo. Por ejemplo, en el caso del módulo del recolector *flow-capture*, estos métodos controlarán la ejecución del servicio *flow-capture* en el sistema.

8.4. Diseño

8.4.1. Consideraciones sobre la arquitectura del sistema

8.4.1.1. Definición de formularios en XML

Como se comentó en el Análisis de Requisitos, y en concreto en el requisito no funcional RNF5, se ha decidido emplear preferentemente el lenguaje XML para describir las estructuras de datos referentes a los interfaces de usuario (formularios de configuración).

En un compromiso entre mantener este principio y la flexibilidad que proporciona, así como no aumentar demasiado la carga del sistema en cuanto al procesamiento de los formularios y la transformación de los mismos se refiere, sin menospreciar la conveniencia de poder personalizar la apariencia de la aplicación de una forma simple, se ha optado por un esquema mixto que utiliza tanto XML como HTML para la descripción de los formularios (pantallas) de configuración del sistema, estando repartida la responsabilidad de cada lenguaje como a continuación se indica:

- HTML para las plantillas (es decir, la apariencia)
- XML para los elementos estructurados (independientes de la apariencia)

Es decir, que por un lado se emplearán estas plantillas HTML (esto es, código HTML en el que aparecen ciertas «marcas» o etiquetas donde se insertará el contenido dinámico posteriormente) que representan enteramente la apariencia de los formularios (y al tener una estructura interna muy similar a la que tendrán finalmente, una vez renderizados, resultan fácilmente personalizables), y por otro lado ficheros XML donde se describen los campos que aparecerán en los

formularios y los metadatos asociados a ellos (es decir, datos que describen estos campos, como son por ejemplo: el tipo, longitud, nombre, descripción, atributos...).

1. Plantillas HTML

   La idea es que el código HTML sea fácilmente procesable por *patForms* (el núcleo de procesamiento de formularios de la aplicación) de forma que éste pueda incluir los elementos dinámicos que necesite.

   Tras estudiar las posibilidades existentes, se decide el uso de *patTemplates*, que entre otras cosas es capaz de realizar sustitución de etiquetas dentro del código HTML, así como la repetición de secciones de código una o múltiples veces (lo cual es útil cuando se tiene un número indeterminado de controles).

   Su total integración con *patForms* facilita en gran medida la implementación, ya que actúa como una especie de filtro de salida o renderizador a la hora de generar la salida HTML, sustituyendo automáticamente las etiquetas que aparecen en la plantilla en el momento de generar el formulario.

2. Definiciones XML

   Se utilizará *patForms_Definition*, una clase auxiliar de *patForms* que realiza, con ayuda de *PEAR::XML_Serializer*, una serialización y deserialización de la estructura en memoria de la definición de los formularios. Esta estructura serializada está descrita en XML y por tanto nos servirá como base para cargar una definición inicial de los mismos a partir de un fichero.

Veremos más detalles sobre la implementación concreta de estas clases en la sección de E.2, "*Construcción de la interfaz*", del apéndice E, "*Interfaz de configuración*".