Proyecto Fin de Grado
Grado en Ingeniería de las Tecnología de Telecomunicación

Sensor de monitorización SNMP integrado en RedBorder

Autor: Alberto Jódar Borrego
Tutor: Pablo Nebrera Herrera

Dep. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Proyecto Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Sensor de monitorización SNMP integrado en RedBorder

Autor:
Alberto Jódar Borrego

Tutor:
Pablo Nebrera Herrera

Dep. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Proyecto Fin de Grado: Sensor de monitorización SNMP integrado en RedBorder

Autor: Alberto Jódar Borrego

Tutor: Pablo Nebrera Herrera

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

"El mayor riesgo es no tomar ningún riesgo"

Mark Zuckerberg.

Agradecimientos

No son pocas las personas que han formado parte de esto. Todas y cada una de ellas han sido fundamental, como cartas en un castillo de naipes, un castillo que no está terminado, un castillo que crecerá sin lugar a dudas, que caerá y volverá a levantarse, pero un castillo con una base sólida que siempre estará presente. Es a esa base a la que tengo que dedicar estos agradecimientos.

A mis padres y mi hermano, por ofrecerme un sitio donde crecer y vivir de la mejor manera posible, siendo feliz. En especial a mis padres, nunca podré pagarles el esfuerzo para que yo pueda hoy escribir estas líneas.

A las mejores personas que nunca podré conocer, a mi grupo de amigos, las personas que conocí sin saber hablar y que 20 años después siguen ahí todos y cada uno, compartiendo los mejores momentos de mi vida. Ellos son parte de mi vida, de todos mis fracasos y de todos mis éxitos.

A las dos personas con las que compartí los mejores momentos durante la carrera. Todo el esfuerzo, todo lo aprendido y la pasión que he adquirido por la profesión es gracias a ellos. Juntos nos hemos convertido en entusiastas de nuestro trabajo.

Por último, a los profesores que han hecho de los conocimientos adquiridos una pasión. En especial a mi tutor, Pablo Nebrera y al grupo de trabajo de Eneo Tecnología. Seis meses trabajando con ellos para aprender y disfrutar de la profesión más bonita del mundo

Este trabajo es parte de todos y cada uno de ellos. Os estaré eternamente agradecido.

Alberto Jódar Borrego

Sevilla, 2016

Resumen

Para entender este proyecto, es necesario disminuir el zoom desde el que se mira y englobarlo en el proyecto para el que se ha desarrollado.

SIGMONA[1] es un proyecto encargado de integrar todos los conocimientos de red actuales (computación, enrutado y arquitectura de red) en la red LTE / EPC móvil de banda ancha (3GPP). La misión del equipo de ENEO Tecnología[2] será la de ofrecer y adaptar sus herramientas de seguridad en red para proporcionar una defensa activa sobre las posibles vulnerabilidades además de ofrecer una actuación ante la aparición de fallos de seguridad.

Imag. 1 Logo proyecto SIGMONA

Focalizando el trabajo del equipo de ENEO Tecnología en la primera fase del proyecto, vemos el siguiente esquema, el cual podemos estructurar en "Recopilación", "Procesamiento" y "Actuación".

[1] Proyecto SIGMONA (TSI-100102-2013-85) cofinanciado por Ministerio de Industria, Energía y Turismo.
[2] ENEO TECNOLOGÍA. Más información en: http://www.nextel.es/eneo-tecnologia-s-l/

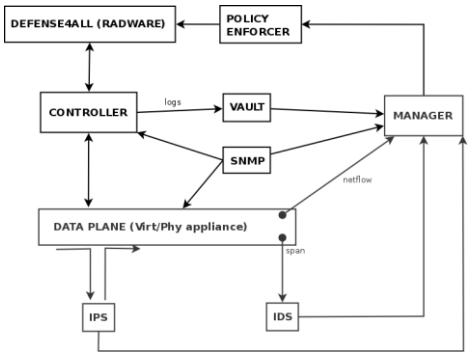


Fig. 1 Esquema proyecto de ENEO T. en SIGMONA

En la recopilación de datos nos encargaremos de desarrolar una sonda snmp, que reciba los objetos a monitorizar y algunos parámetros adicionales. Esta sonda recogerá periódicamente los datos y los incorporará a un flujo de datos mediante técnicas "Big Data". A partir de estos datos y de otros que se incluirán en la primera fase del proyecto (como la información de los logs), el manager actuará sobre la red a través de la recién liberada herramienta "Defense4All"[3].

La primera fase del desarrollo de este proyecto será una fase de investigación. Buscando una herramienta apta para nuestro propósito, trabajaremos principalmente con la herramienta de monitorización y de software libre "Zabbix"[4], más cocretamente con su versión "Zabbix proxy".

La simplicidad de "Zabbix proxy" con respecto a su producto principal "Zabbix Server" nos hizo pensar en esta herramieta como una posible solución a nuestro problema. Tras su intento de adaptación y de modelado para adaptarla a nuestro proyecto, observamos varias barreras que, para lo que buscábamos, se antojaban insalvables.

Continuando con la investigación y tras trabajar con varias herramientas de software libre y diferentes proyectos y librerías subidos a la plataforma GitHub, de forma unánime, el equipo de Eneo Tecnología ecargado de los objetivos del proyecto SIGMONA, decidimos partir desde cero en el desarrollo de la sonda, adaptándola así a las necesidades concretas del proyecto.

Tras la toma de decisión en un punto decisivo en este proyecto, planteamos las características deseadas en la herramienta, que de forma general son las siguientes:

- La sonda debe de recibir del manager la configuración y los objetos a monitorizar mediante SNMP. Siguiendo la dinámica de trabajo del equipo de ENEO Tecnología, se concluyó que estos ficheros tendrían un formato JSON, por lo que la sonda SNMP se encargará de parsearlos y procesarlos.
- Esta configuración deberá poder ser recargada en cualquier momento de la ejecución del programa. La sonda por tanto estará a la escuha de las señales definidas para la recarga de la configuración.
- Al trabajar en una red de gran extensión, el número de objetos a monitorizar será muy elevado. Para sacar el máximo rendimiento a la máquina donde se ejecute la sonda, se hace indispensable que el funcionamiento de esta sea multihilo.
- Además de su ejecución multihilo, se hace indispensable usar la funcionalidad asíncrona de la librería "net-snmp" para aumentar el rendimeinto de la sonda.
- La incorporación de los datos recibidos al flujo de datos del manager deberá ser mediante la librería "librdkafka"[5], ya que es la usada por las herramientas desarrollladas por ENEO Tecnología.

[3] Defense4All: Herramieta software libre de OpenDaylight. Más información en: https://wiki.opendaylight.org/view/Main_Page
[4] Zabbix: Herramienta monitorización software libre. Más información en: http://www.zabbix.com/
[5] Librería que implementa el protocolo de mensajería Apache Kafka. Más información en: https://github.com/edenhill/librdkafka

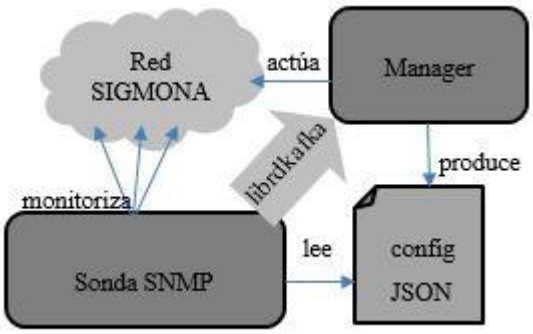


Fig. 2 Esquema básico de funcionamiento sonda SNMP

Observando el esquema general del proyecto, entendemos el objetivo final de esta herramienta. Una herramienta pensada para estar siempre en ejecución incorporando un gran fujo de datos al manager, que será el encargado de su procesado y de realizar una acción en la red si así se requiere

Teniendo estas como características principales, el resto del proyecto es el desarrollo de la herramienta, adaptándola en todo momento a las necesidades de la empresa y del proyecto para el que se desarrolla.

Abstract

This project is a way between all alternatives in the snmp monitoring in order to adapt a tool to the data flow of RedBorder[6].

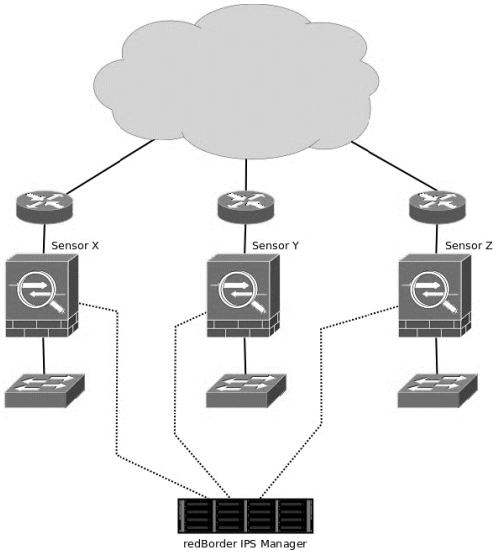


Fig. 3 RedBorder diagram

All starts in **Eneo Tecnología** where they are developing a network security tool called RedBorder in cooperation with other companies like **Nextel SA**[7]. RedBorder consists in a high number of sensors witch gather some type of information across the network which is send to a central process called **RedBorder manager**.

The RedBorder manager process is where this amount of data is processed in order to make differents actions on the net and where they show the results to users in a web frontend.

This project is how we made a RedBorder sensor which compile information using SNMP protocol. The main idea was use a free software snmp tool called **Zabbix**[8]. The strategy of ENEO is adapt this types of free software tools to the work flow of RedBorder, working on the necessary parts for the integration.

[6] https://redborder.com/
[7] http://www.nextel.es/
[8] http://www.zabbix.com/

Our tool is going to be used for a project in which ENEO is working on, SIGMONA[9]. The SIGMONA project, SDN Concept in Generalized Mobile Network Architectures, will study network architectures and functions for evolution of the LTE/EPC (3GPP) mobile networks. The objective of ENEO is to implement RedBorder into the SIGMONA´s network in order to manage, prevent and protect important security risks.

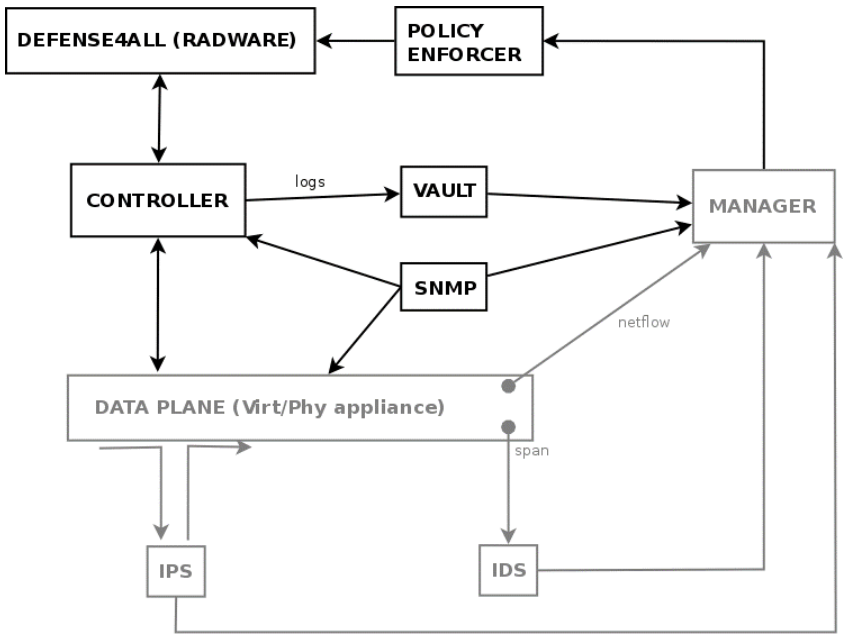


Fig. 4 RedBorder in SIGMONA

As we can see in **Fig 3**, there are two importants parts in the gathering data process. **Vault** in which system log information will be sent to the manager, and **SNMP**, a tool which gather all monitored information from the network using this protocol.

During the project we are going to see how some problems made us to reconsider the possibility of use Zabbix, so we started to look for other alternatives. One of them, and the one we took, was to develop the tool using some basic libraries, like **net-snmp,** which help us with an abstraction of the SNMP monitoring.

Now, we are going to describe all the alternatives and how we develop the final solution.

[9] https://www.celticplus.eu/project-sigmona/

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Fig. 53 Esquema de ficheros del proyecto 131

ÍNDICE DE CÓDIGO

1 INTRODUCCIÓN

1.1 Orígen

Decir que la idea inical de este proyecto nace con una clase puede parecer algo precipitado, pero así es. Conocí a Pablo Nebrera, tutor de este proyecto, en la primera clase que recibí en la carrera. Llegar con nervios y expectante a la primera clase de universidad y encontrar a un profesor entusiasta, comuncativo y muy agradable es una suerte que no todos tienen.

Pero no fue hasta mi tercer año de carrera cuando volvímos a ejercer de maestro y alumno, pero esta vez en la asignatura de seguridad. Es aquí cuando descubrí gracias a él y al profesor Godofredo Fernández que quería dedicarme a la seguridad, que era eso lo que buscaba desde que entré en la carrera.

Finalmente y como yo deseaba desde un principio, Pablo Nebrera me tutelaría el proyecto fin de carrera. Me habló de su empresa, Eneo Tecnología, del producto que desarrollan, RedBorder, y de la cantidad de

cosas que les queda por desarrollar. Muchas de esas tareas pendientes estaban en proceso, desarrollándose por el equipo de Eneo Tecnología. Otras muchas tareas se encontraban en la cola, y es ahí dónde Pablo me ofreció un gran abanico de posibilidades. Y otras muchas, frutos del carácter entusiasta y de la gran ambición de Pablo, eran ideas en el aire, las cuales estoy seguro serán el presente de su empresa en no mucho tiempo.

De ese abanico de posibilidades encontré lo que finalmente se ha convertido en este proyecto. El uso de técnicas Big Data, el trabajo con un protocolo con el que ya estaba familiarizado (SNMP) y la idea de su futura incorporación a un proyecto de nivel europeo como es el proyecto SIGMONA, hicieron para mí que este proyecto fuera el más atractivo de entre todos los que se me ofrecían.

Inmediatamente nos pusimos manos a la obra. En una reunión con Jaime Nebrera, miembro de ENEO Tecnología y encargado de mantener una visión global de los proyectos de la empresa, hablamos del proyecto, de las herramientas por las que empezar y la metodología a seguir. Es en ese punto dónde podemos poner la línea de inicio de este proyecto.

1.2 Fase de investigación

1.2.1 Necesidades a cubrir

La herramienta que buscamos debe cumplir las siguientes especificaciones:

- Debemos de poder modificar su salida. El manager de RedBorder (elementos que detallaremos posteriormente), usando el protocolo Apache Kafka, irá leyendo de una cola de mensajes. Estos mensajes serán producidos por la herramienta, luego debemos poder modificar su salida para que mande los mensajes a dicha cola.
- El funcionamiento de la herramienta debe de ser asíncrono y multihilo. Debido a que va a trabajar en redes de gran extensión, el flujo de datos que va a manejar va a ser muy elevado. Para sacar el máximo rendimiento a la máquina donde va a lanzarse la herramienta, se hace indispensable que su funcionamiento sea multihilo. Para evitar bloqueos que puedan comprometer el funcionamiento de la herramienta, debe de funcionar el envio y recepción de mensajes SNMP de forma asíncrona.
- La herramienta ha de ser de software libre. Dada la política y filosofía de ENEO Tecnología, cuya máxima es la del uso y adaptación de herramientas de software libre, debemos usar el código abierto de la herramienta encontrada y adaptarlo para satisfacer las necesidades.

Con estos puntos como premisas iniciamos el trabajo de investigación.

1.2.2 Zabbix proxy y Zabbix server

Tras la primera reunión, Jaime Nebrera me puso al día en el estado de desarrollo en el que se encontraba el proyecto SIGMONA por parte de ENEO Tecnolgía. Prácticamente a cero.

ZABBIX
The Ultimate Open Source Monitoring Solution

Imag. 2 Logo Zabbix

Para el desarrollo de la monitorización SNMP encontramos rápidamente Zabbix[10]. Esta herramienta de software libre, usada por multitud de empresas de IT[11] cumplía las principales necesidades. Tenía un funcionamiento asíncrono y multihilo y monitorizaba periódicamente una lista de objetos.

[10] Web oficial: http://www.zabbix.com/
[11] Empresas que trabajan con Zabbix: http://www.zabbix.com/users.php

Concretamente, Zabbix contaba con una herramienta que se asemejaba al objetivo final de nuestro proyecto, esta era "Zabbix proxy".

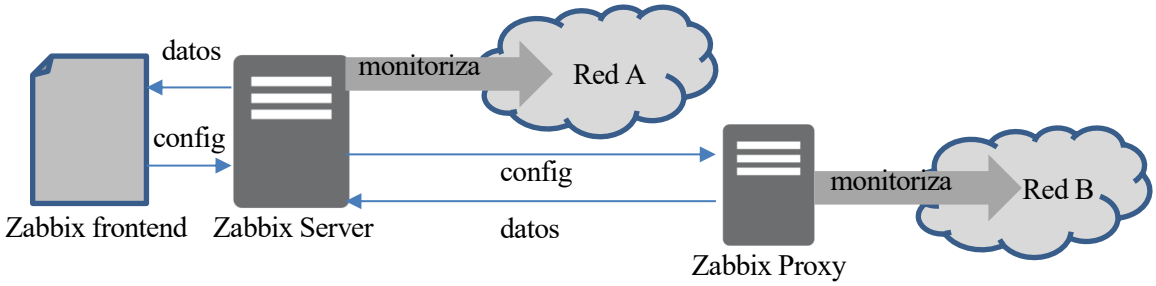


Fig. 5 Funcionamiento Zabbix Server y Zabbix Proxy

Como vemos, Zabbix, en su monitorización SNMP, cuenta fundamentalmente con estos dos productos.

Zabbix Server es el producto central. Se encargará de procesar la configuración, de realizar la monitorización sobre una red y de mantener un "frontend" para la interacción con el usuario. En este frontend mostrará los datos de monitorización, y de aquí recibirá por parte del usuario los datos a monitorizar. A su vez, mantendrá la comunicación con el otro producto de Zabbix, Zabbix proxy.

Zabbix proxy es una versión reducida de Zabbix Server. Su única función será la de recibir la configuración del server, monitorizar los objetos, y enviar estos datos de nuevo al server para que este pueda presentarlos en su "frontend".

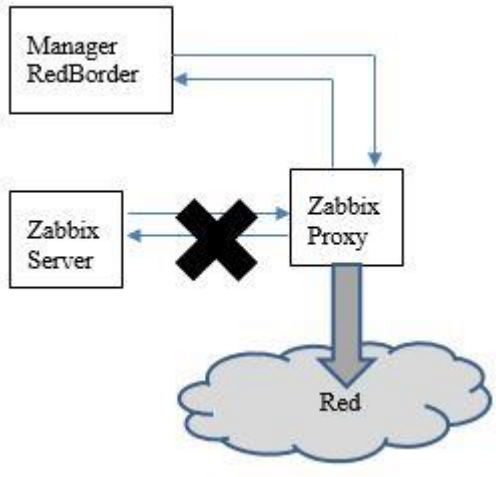


Fig. 6 Esquema sustitución Zabbix Server por Manager RedBorder

Entendiendo el funcionamiento de Zabbix proxy, podemos ver un funcionamiento similar al que nosotros buscamos.

Sustituyendo la dependencia de Zabbix Proxy con el Zabbix Server para que este reciba la configuración del Manager de RedBorder y a su vez le mande los datos de monitorización, tenemos el objetivo que buscamos con el proyecto.

A su vez, seguimos la filosofía de trabajo de Eneo Tecnología, que es la de usar herramientas de software libre y adaptarlas al funcionamiento de RedBorder.

1.2.3 Problemas en Zabbix

Comenzamos pues a trabajar con Zabbix. La idea era la siguiente, usar Zabbix proxy y adaptarlo para que recibiera la configuración del Manager de RedBorder y redirigir la salida de datos hacia el manager.

Tras un tiempo de trabajo en Zabbix, en su código, y tras varios intentos de adaptación, encontramos los siguientes problemas:

- Zabbix, tanto en su versión server como en su versión proxy, mantiene una base de datos local, una base

de datos que almacena periódicamente los últimos datos de la monitorización y que usa como prevención y para evitar pérdida de datos ante posibles fallos de comunicación con el server. Esto, desde nuestro punto de vista, más que una ventaja y una prevención, consistía en introducir un punto de fallo adicional, y era algo que se debía evitar.

- Una funcionalidad de Zabbix para incorporar otra medidad de prevención es la que ellos denominan "KeepAlive". Consiste en una comunicación constante y periódica entre el server y el proxy con el objeto de detectar pérdida de conectividad. El server manda mensajes de "keep alive" de manera periódica al proxy y este, a su recepción, mandará la respectiva respuesta. Esto es una funcionalidad que es deseable quitar, ya que todo proceso que realiza al server, en nuestra adaptación, deberá realizarlo el manager.
- El problema principal. La salida de datos de Zabbix proxy, en nuestra adaptación, deseamos que sea mediante el protocolo Apache Kafka. Esto nos planteó dos posibles soluciones:
    - Modificar el código de Zabbix. A pesar de parecer una solución no viable, la salida de datos dentro de la estructura del código de Zabbix proxy se reduce a una función, la cual localizamos y con la que empezamos a trabajar. Una vez hecha esta opción, habría que contactar con el equipo de Zabbix para que incluyera la funcionalidad de enviar a través de Apache Kafka, proporcionándoles el código desarrollado
    - Diseñar una herramienta que se "cuele" en la comunicación entre un server y un proxy, y mande una copia de los datos por Apache Kafka al manager de RedBorder. Esto planteaba el siguiente problema. Para realizar esto, necesitamos correr un Zabbix server y un Zabbix proxy. La configuración debe surgir del manager de Redborder, pero debe de llegar al proxy a través del Zabbix server. En la figura 5 se ilustra el escenario planteado.

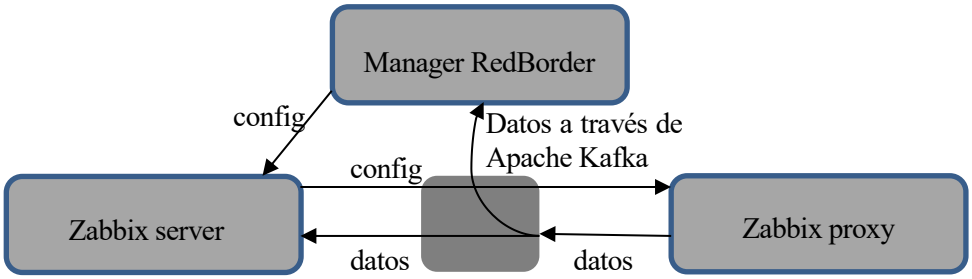


Fig. 7 Esquema de flujo de datos con la herramienta a desarrollar

Analizados todos estos inconvenientes que no esperábamos encontrar en un inicio y consultando con el equipo de ENEO Tecnología, decidimos que dedicar unos días a la búsqueda de otra herramienta podría ser de gran ayuda para tomar una decisión respecto a la continuación del proyecto.

1.2.4 Soluciones alternativas

Con las características de un inicio (multihilo y asíncrono principalmente), nos dispusimos a buscar posibles soluciones ajenas a Zabbix. Las opciones destacadas de entre todas las encontradas fueron:

- RFC[12], una solución de monitorización en Perl. Inicialmente nos fijamos en esta solución por su simplicidad. Una herramienta que cumplía el objetivo inicial, monitorizaba de manera asíncrona y el programa era multihilo. Nuestro trabajo consistiría en adaptar su salida para incorporar los mensajes al

[12] Más información: http://www.perlmonks.org/?node=664360

flujo de datos del manager.

- Rated[13], una herramienta de monitorización en C. Subida a gitgub como una parte del proyecto RTG[14], un sistema de monitorización de alto rendimiento muy conocido y con garantías de ofrecer un buen funcionamiento.
- Ruby-snmp[15]. A pesar de no ser una herramienta y ser una librería para Ruby, ofrece una simplicidad en el manejo de datos en ASN1 y de las PDUS de envio y de respuesta, que hacían de esta opción una posible solución a nuestro problema.
- Net-Snmp[16]. La famosa librería para C que implementa el protocolo SNMP inicialmente quedó descartada, ya que implicaba la programación de una herramienta de polling casi en su totalidad. Después de ver varias herramientas y tras el "desencanto" de Zabbix, incluimos a esta como una posibilidad.

Planteamos la siguiente tabla para la toma de decisión de la herramienta a utilizar:

Herramienta	Ventajas	Inconvenientes
Zabbix	• Herramienta ampliamente usada y testeada • Solución de monitorización similar a nuestro objetivo final • Alto rendimiento • Herramienta en C	• Uso de una base de datos local • Necesidad de añadir un módulo que modifique la salida • Interacción server-proxy que incluye una complicación a la hora de su aadaptación al manager de RedBorder
RFC	• Monitorización asíncrona y de alto rendimiento • Simplicidad del código, lo que nos permitiría trabajar con él y adaptarlo al resto de necesidades • Perl ofrece modulos ampliamente útiles en el manejo de datos	• Uso de un lenguaje de programación poco familiar • Se estudiaría la modificación de la herramienta para que sea multihilo • Necesidad de añadir la salida en Apache Kafka, usando para ellos librerías[17] no usadas por el equipo de ENEO Tecnología
Rated	• Poller asíncrono y multihilo • Programado en C • Equipo de desarrollo activo	• Uso de base de datos local • Cambio de formato de salida no trivial
Ruby-snmp	• Ofrece funcionamiento asíncrono de SNMP • Simplicidad de envío y	• Al ser una librería, se necesitaría desarrollar la aplicación al completo

[13] Más información: https://github.com/mprovost/rated
[14] Más información: http://rtg.sourceforge.net/
[15] Más información: https://github.com/hallidave/ruby-snmp
[16] Más información: http://www.net-snmp.org/
[17] Recomendada por el equipo de desarrollo de Apache Kafka: https://github.com/TrackingSoft/Kafka

	recepción PDU de SNMP	• Ruby es un lenguaje de programación desconocido. Tiempo de adquisición de conocimientos • Necesidad de usar librerías[18] para Kafka desconocidas para el equipo de ENO Tecnologia.
Net-snmp	• Librería más usada para el desarrollo de aplicaciones con SNMP • Familiarizada durante la carrera[19]. Multiples ejemplos y ayuda en la web • Programación en C	• Necesidad de desarrollar completamente la aplicación de polling. Desde los ficheros de configuración hasta la salida de datos

Tabla 1 Ventajas e inconvenientes de las opciones de monitorización disponibles

1.2.5 Solución final

Llegados a este punto sólo quedaba la toma de decisiones. Este fue un punto crítico y vital en el desarrollo del proyecto.

La decisión pasaba por proseguir con Zabbix con todo el trabajo de modificación y adaptación que ello implicaba, trabajar con "rated snmp" completando y mejorando aquellos aspectos en los que la herramienta no cumplía con nuestros requisitos, o desarrollar la herramienta nosotros mismos, adaptándola de esta forma a nuestras necesidades concretas (usando la librería net-snmp).

Finalmente coincidimos en que la mejor solución pasaba por desarrollar la herramienta. Sabiamos que esto iba a acortar los objetivos del proyecto y ahora pasaba a ser un proyecto en el que se dedicaba la totalidad del tiempo a desarrollar la herramienta. A pesar de esto, coincidimos que era la mejor solución, ya que ninguna herramienta nos proporcionaba las características que buscamos.

1.3 Fase de desarrollo

En el anexo 1 se puede encontrar el código desarrolado para esta herramienta. En estos apartados vamos a ver de una manera superficial la estructura del programa desarrollado.

1.3.1 Librerías

Tomada la decision de desarrollar nosotros mismos la herramienta, coincidimos inmediatamente en que íba a ser más productivo el uso de la librería net-snmp por varios motivos. El primero de ellos es que podemos desarrollarla en C, ya que, al ser un lenguaje ampliamente conocido y trabajado, nos iba a ahorrar tiempo de familiarización y adquisición de conceptos. El segundo y más importante, y es un aspecto muy apreciado durante el desarrollo de la herramienta, es que, al ser una librería de gran popularidad, la documentación aportada por el equipo de desarrollo, además de la información encontrada por diferentes personas a lo largo de la red, es muy amplia, facilitando así la solución de los problemas encontrados.

[18] Recomendada por el equipo de desarrollo de Apache Kafka: https://github.com/joekiller/jruby-kafka
[19] Familiarización con net-snmp durante la asignatura de Gestión de redes durante el curso 3 del grado

Sin entrar a profundizar, que lo haremos en capítulos posteriores, el uso que le queremos dar a la librería es el siguiente:

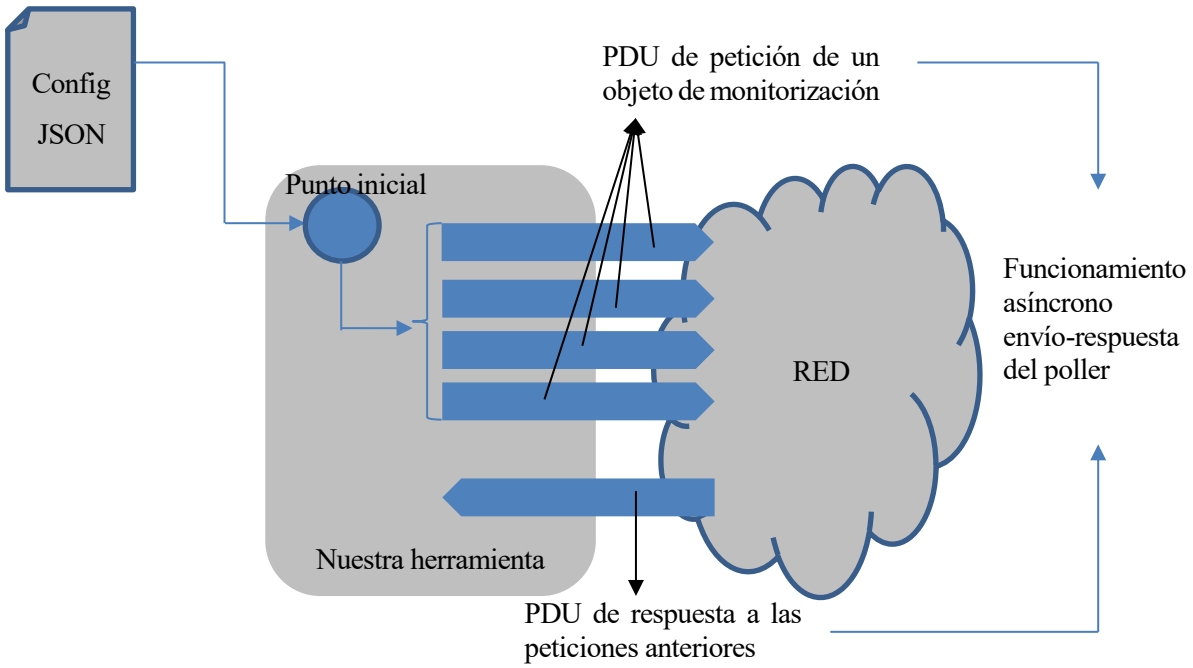


Fig. 8 Esquema básico del funcionamiento asíncrono y multihilo de la herramienta

La librería net-snmp nos permite satisfacer las necesidades que pretendemos cubrir con nuestra herramienta. Podemos encontrar un ejemplo de trabajo asíncrono en la propia documentación de la librería.

Simplificando a un caso, mínimo aunque extensible, en el que solo monitorizamos un objeto de un host con un solo hilo, con la librería podemos funcionar de la siguiente forma:

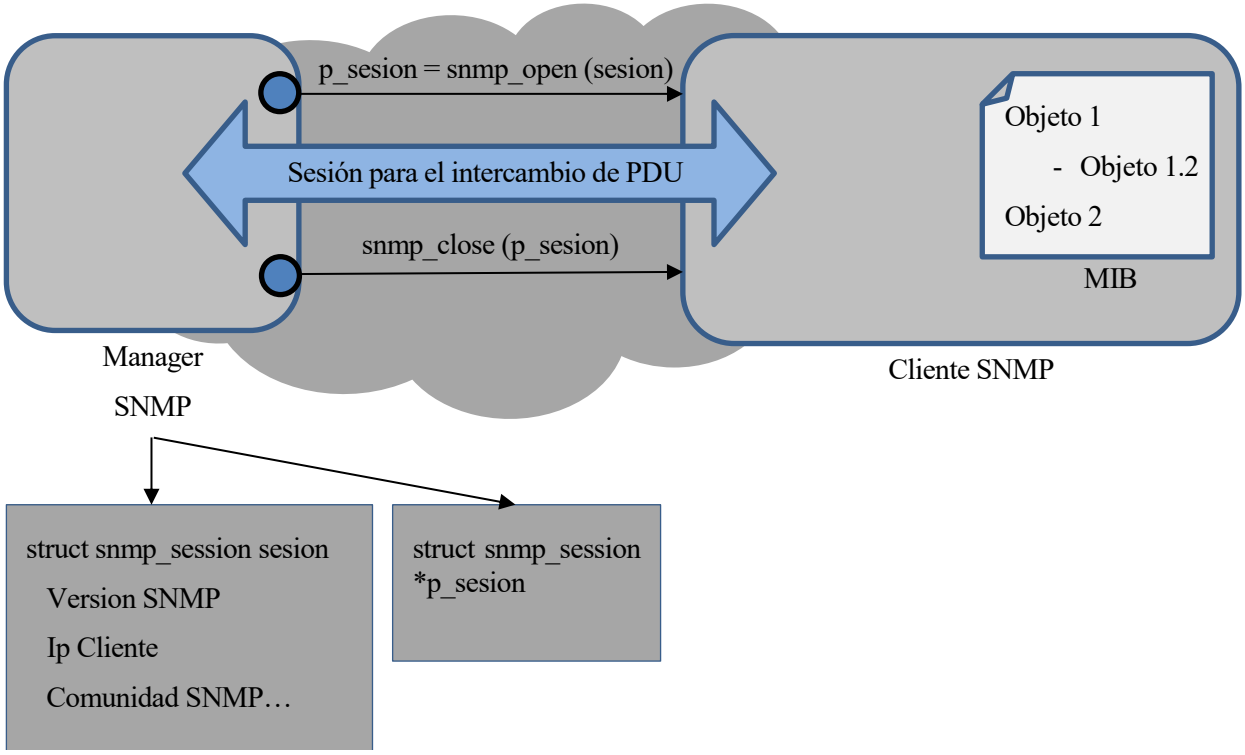


Fig. 9 Esquema funcionamiento básico de net-snmp

El funcionamiento fundamental consiste en:

- Inicio de sesión entre el manager y el cliente (en un caso más amplio, la sesión será abierta con cada uno de los clientes).
- Envio de peticiones y espera de recepción de la respuesta de forma asíncrona.
- Cierre de sesión.

La librería net-snmp requiere de un paso previo al envío de PDU de petición y respuesta. Se require la apertura de una sesión entre el manager y el cliente.

Para abrir esta sesión disponemos de la función de dicha librería denominada "**snmp_open**", y es la misma para todos los casos de funcionamiento mediante SNMP a través de esta librería. Su funcionamiento se detallará en capítulos posteriores, pero a grandes rasgos podemos decir que una sesión se inicia tomando los datos de una estructura, la cual servirá además para almacenar datos de esta misma sesión tras ser abierta (ver en Fig. 7). A través de esta sesión, y de los datos almacenados en su estructura, realizaremos los envios de las PDU de petición de los objetos del cliente.

Una vez abierta la sesión nos serviremos de las funciones de envio y recepción de respuesta asíncronas pproporcionadas por net-snmp y que son ampliamente detalladas en su documentación y ejemplos[20].

El funcionamiento asíncrono se basa en indicar, dentro de la información proporcionada para abrir la sesión, una función "handler"[21] que será llamada cuando se detecte la recepción de una PDU de respuesta. Así pués, una vez enviadas las PDUs de consulta, y teniendo registrado el handler para la recepción, no debemos de bloquear la ejecución para la espera de una respuesta, sino que ante la llegada de esta, se ejecuará la función que procesará los datos sin interrumpir el funcionamiento principal del programa.

Adicionalmente y para que se pueda lanzar la función handler, necesitamos el uso de la función "select"[22], que sirve para escuchar los descriptores de ficheros (lo detallaremos más adelante) hasta que haya alguno disponible, es decir, hasta que se haya recibido algún paquete. Es en este momento cuando se lanza la función handler y se procesa la PDU de respuesta.

[20] Ejemplo de alicación asíncrona proporcionado por la documentación de net-snmp: http://www.net-snmp.org/wiki/index.php/TUT:Simple_Async_Application

[21] "Handler" o manejador es una función que se llamará ante la ocurrencia de determinado tipo de evento, habiendo sido registrado previamente

[22] Más información de la función select: http://linux.die.net/man/2/select

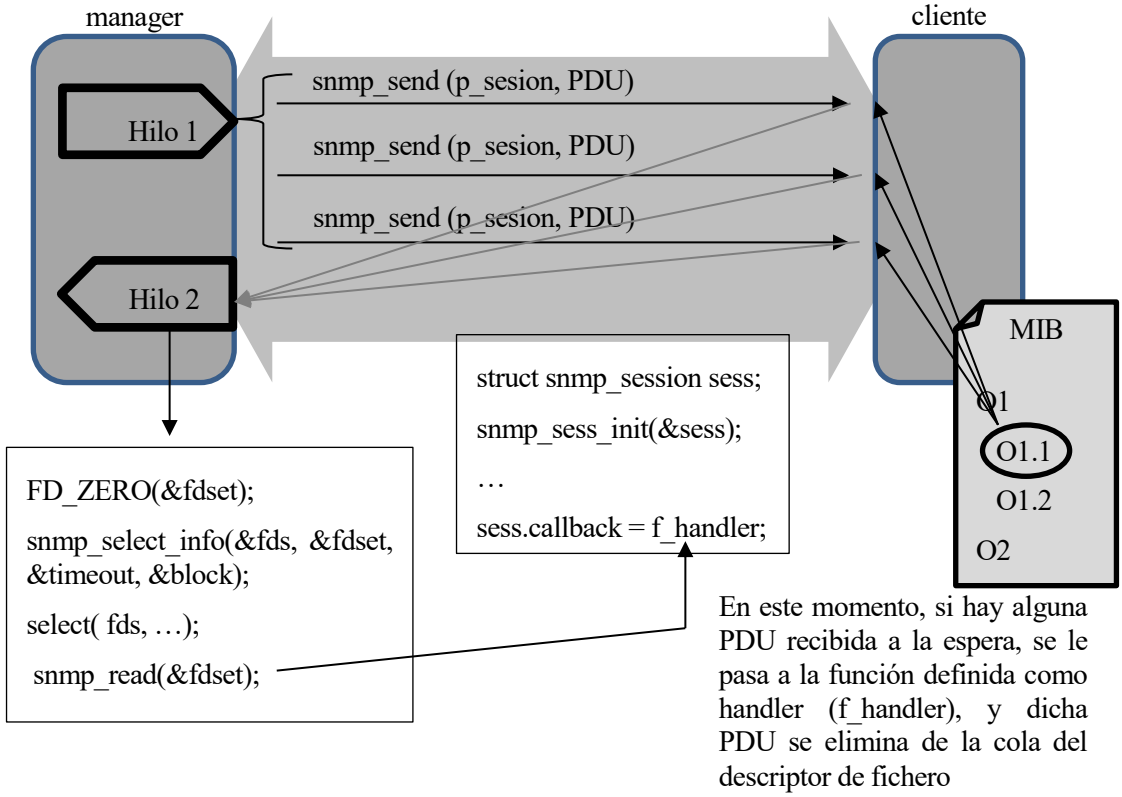


Fig. 10 Funcionamiento asíncrono con la librería net-snmp

Aunque el código será detallado y complementado en posteriores capítulos, podemos comprender el funcionamiento asíncrono con el cual podemos trabajar con esta librería (ver Fig.8). Mientras que uno o mas hilos se encargarán del envío de PDU, otro hilo se encargará de leer del descriptor de fichero, lo que hace que por cada lectura se llame a la función definida como handler, que se encargará de procesar los datos de la respuesta.

Como podemos ver en la figura anterior, "Hilo 2" que es el usado para la lectura de las respuestas, debe hacer una menejo de los selectores de ficheros de entradas para detectar que se ha recibido una respuesta. El código detallado y su funcionamiento se detallarán en capítulos posteriores.

1.3.2 Estructura programa

Visto de una manera superficial el funcionamiento de la librería net-snmp, pasaremos a un rápido repaso a la herramienta final y a su funcionamiento.

Podemos destacar tres acciones funcdamentales en su funcionamiento. Comenzaremos con la lectura de la configuración (tanto configuración general como de los objetos a monitorizar). Tras esto la herramienta comenzará su parte fundamental, que es la de monitorizar los objetos deseados. Y por último deberemos de procesar un cierre seguro en cuanto a memoria ya que el uso de hilos y de memoria compartida así lo requiere.

1.3.2.1 Parte 1: Configuración

Uno de los puntos más determinantes, y con uno de los que se ha invertido más tiempo, es la recepción, parseado y aplicación de los parámetros de configuración.

Existen parámetros en el uso de la herramienta que no son fijos y que deben ser recibidos por el manager. Por

ejemplo, el tiempo entre peticiones, el número de hilos que debe lanzar la herramienta o los equipos y objetos a monitorizar. Esta configuración será recibida del manager en un fichero y es trabajo de la herramienta parsearlo y aplicar su configuración.

Teniendo en mente el objetivo final de la herramienta, es decir, una visión en la que nuestra herramienta estará en continua ejecución y tendrá que modificar su configuración multitud de ocasiones durante su ejecución, se dará una solución a una recarga de la configuración sin tener que hacer un cierre de la herramienta.

Para todo esto se hace necesario distinguir entre dos tipos de configuración.

1.3.2.1.1 Configuración general

La configuración más elemental (se detallará en capítulos posteriores), la configuración indispensable para comenzar a ejecutar la herramienta, vendrá separada del resto.

Recargar la configuración general implica un tratamiento especial, ya que los hilos deberán ser correctamente parados y relanzados de nuevo, la estructura de los datos puede cambiar… entre otras tareas. Este tipo de configuración, la cuál requiere cambios fundamentales en el programa es muy costosa, a diferencia por ejemplo de la configuración que indica qué objetos queremos monitorizar.

Para evitar que se recargue la configuración general con todo lo que implica, cuando en la mayor parte de los casos, se desea modificar parámetros que no afectan a esta, se separa la configuración en un fichero a parte. Se creará una función manejadora, que ante un determinado tipo de señal, iniciará el proceso de recarga de la configuración general, parseando en primer lugar el fichero, parando los hilos, y relanzando estos con la nueva configuración.

1.3.2.1.2 Objetos a monitorizar

El resto de la configuración que se recibe del manager, y es la más importante y la cual vamos a tener que recargar en multitud de ocasiones, es la de los objetos y hosts de la red a monitorizar. A diferencia de la anterior, estos cambios implican una actuación mucho menos en el funcionamiento de la herramienta, y la recarga de este tipo de configuración será más frecuente y en consecuencia más rápida y menos costosa.

El manager indicará, a través de otro fichero de configuración, la lista de hosts y sus objetos que se han de monitorizar.

Sin entrar a profundizar y para tener una idea global, la estructura del fichero de monitorización será la siguiente:

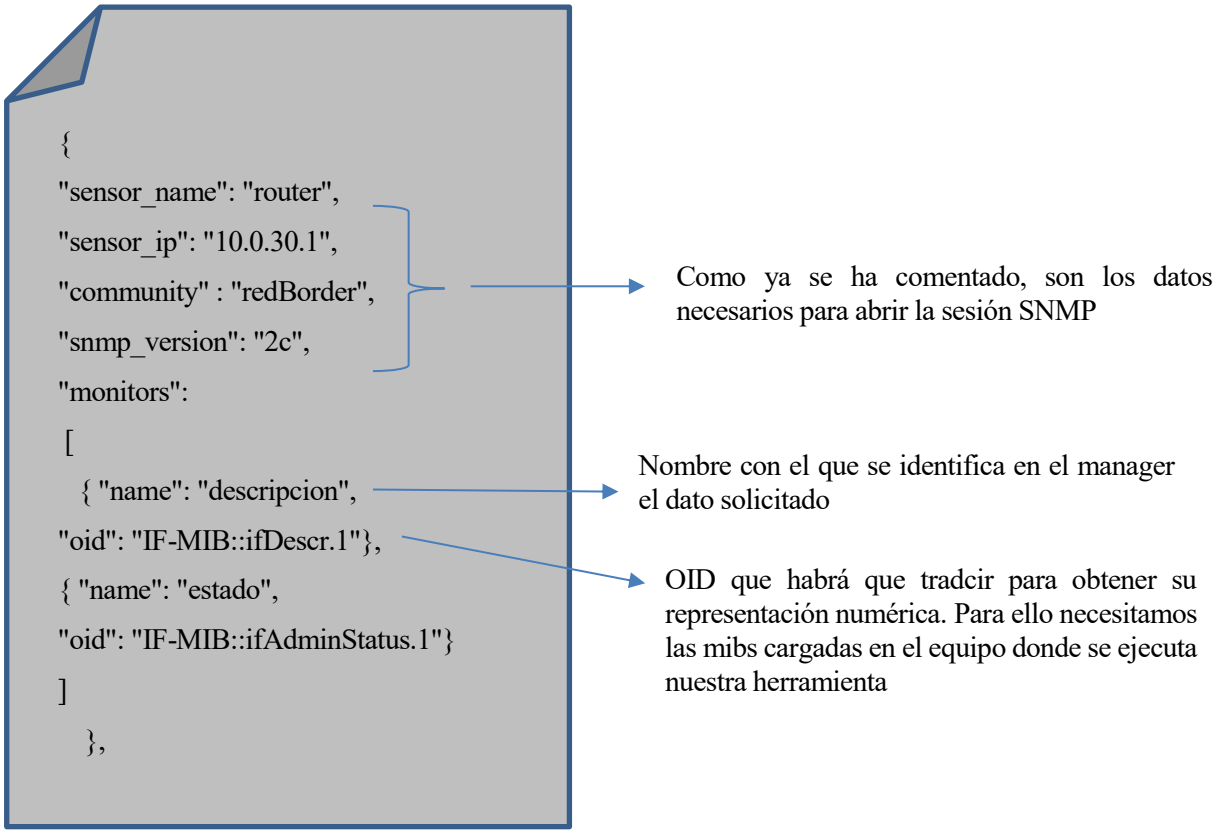


Fig. 11 Ejemplo básico de la configuración de monitorización

Tendremos un objeto por cada host. Cada uno de estos objetos constará de una información necesaria para abrir una sesión snmp, y por último, una lista de los objetos a monitorizar. Podemos ver un ejemplo simple en la Fig. 9.

Tras el parseo de este fichero de configuración se opta, por comodidad de uso y por ser eficientes en memoria, por la creación de una estructura dinámica, a través de la cual los hilos accedan a la información de monitorización.

De esta forma, los hilos accederán a la estructura dinámica paraobtener la información con la que iniciar sesión con cada host y crear y enviar sus peticiones.

Esta estructura será una lista enlazada de los diferentes hosts a monitorizar. Echemos un vistazo rápido.

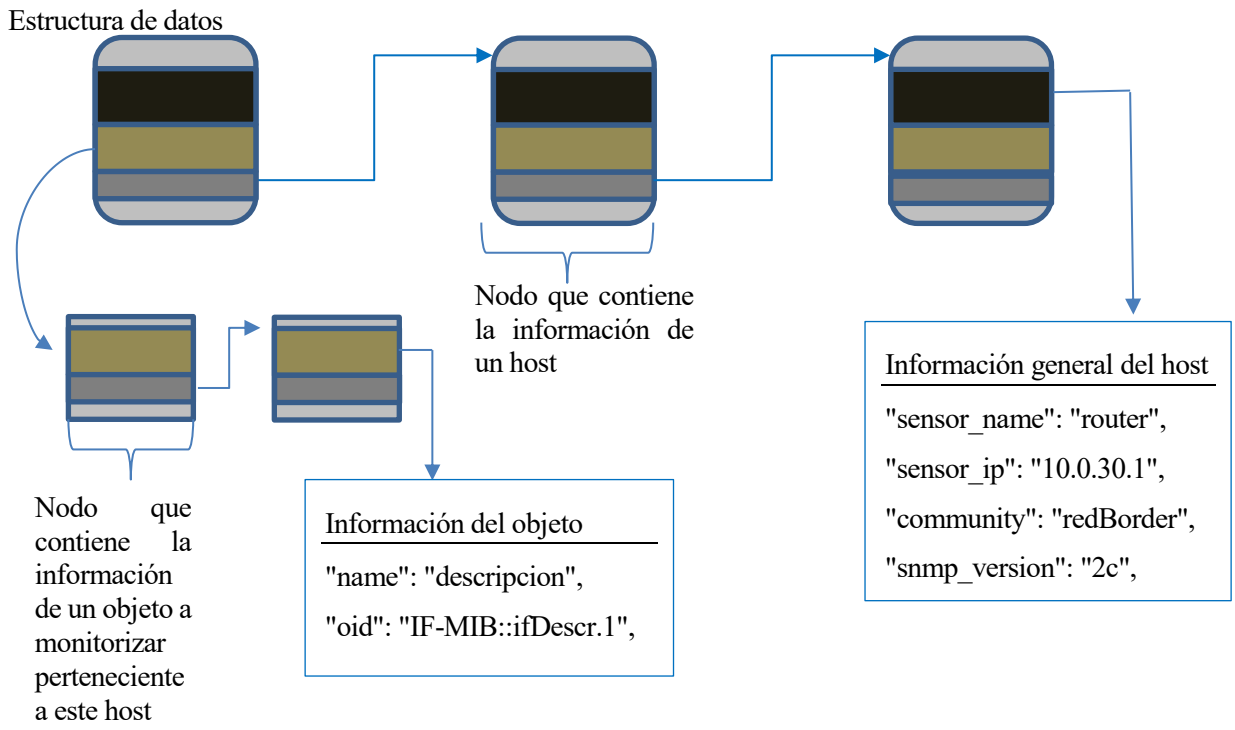


Fig. 12 Resumen de la estructura de datos interna

Aunque la estructura es más compleja y detallada, esta ilustración nos sirve para tener una idea de la lista enlazada que será recorrida por cada hilo, y que contendrá toda la información necesaria para el envío de las PDU.

1.3.2.2 Parte 2: Monitorización

El funcionamiento del envío-respuesta con la librería net-snmp ha sido resumido en páginas anteriores. En nuestra herramienta tendremos una gran cantidad de hosts y un número determinado de hilos, de los cuales dispondremos para enviar PDU a lo largo de la red hacia todos los hosts a monitorizar.

Aun así debemos solucionar un problema fundamental. ¿Qué hosts monitoriza cada hilo y cómo es sabido esto por los diferentes hilos?

Todos los hilos van a acceder a la misma estructura de datos, cuyo puntero será pasado como parámetro. Nuestro objetivo es dividir/trozear la lista enlazada para que cada hilo se encargue de un número de hosts y la carga sea repartida.

Esto será posible gracias a que se ha incluido una variable adicional en la estructura de datos. Esta variable actuará como bandera, que indicará cuando un hilo debe de dejar de monitorizar. Por lo tanto, cada hilo entrará en la estructura de datos y monitorizará progresivamente todos los hosts hasta que encuentre en uno de ellos dicha bandera levantada.

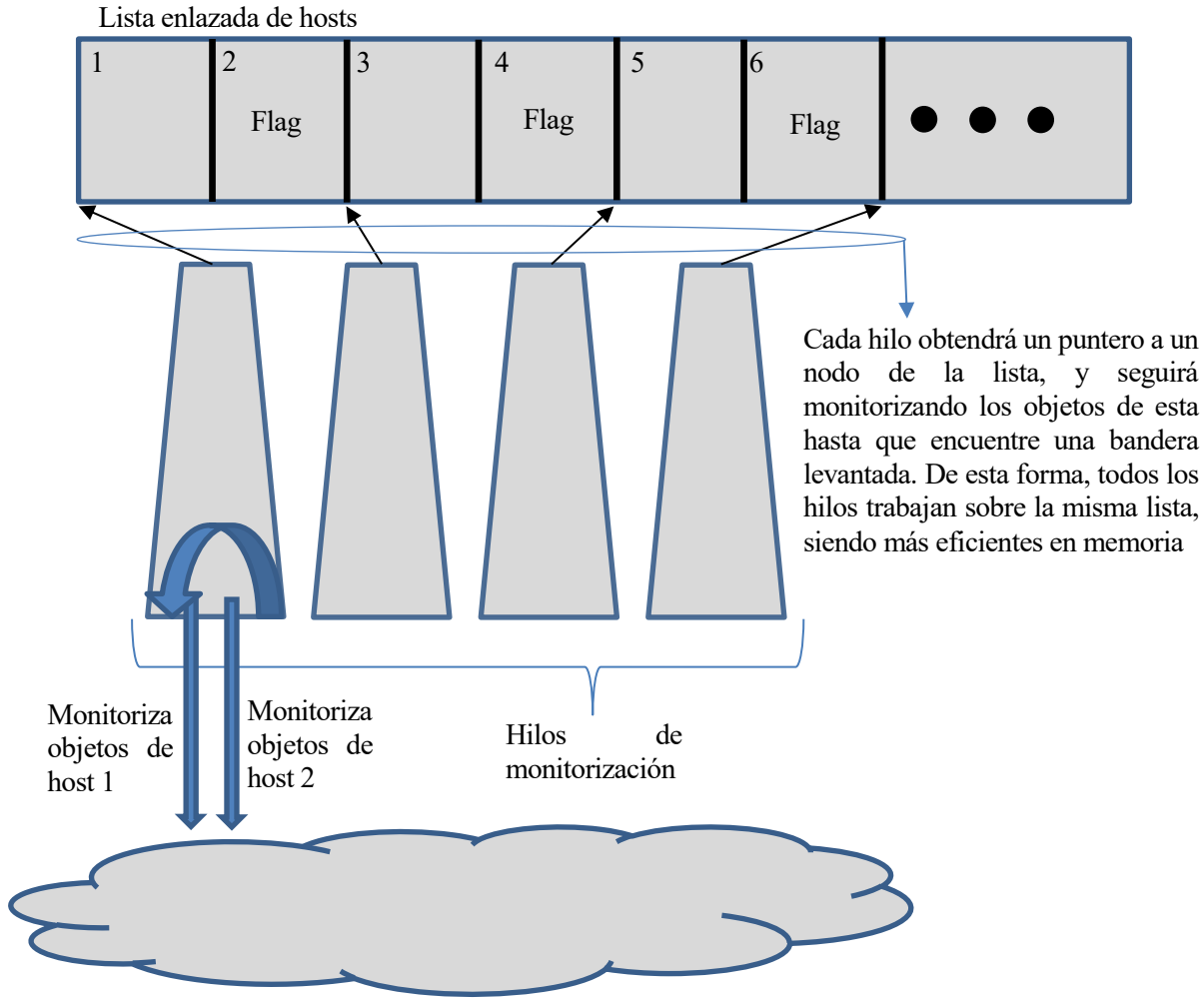


Fig. 13 Funcionamiento de la partición de la lista enlazada

De esta forma, cada hilo accederá a la misma estructura. Puede parecer que se hace necesario una protección del acceso a memoria por parte de los hilos, pero en ningún momento dos hilos van a acceder al mismo nodo, ya que no se incluye redundancia.

El procedimiento de cada hilo será monitorizar cada host avanzando en su lista de OID, y al terminar, pasar al siguiente hosts hasta que encuentre la bandera levantada. Cuando esto ocurra, se esperará el tiempo indicado entre monitorización y volverá al punto de la lista pasado com parámetro y repetirá el proceso.

En puntos posteriores se detallará este proceso, analizando las complicaciones y barreras que se han superado

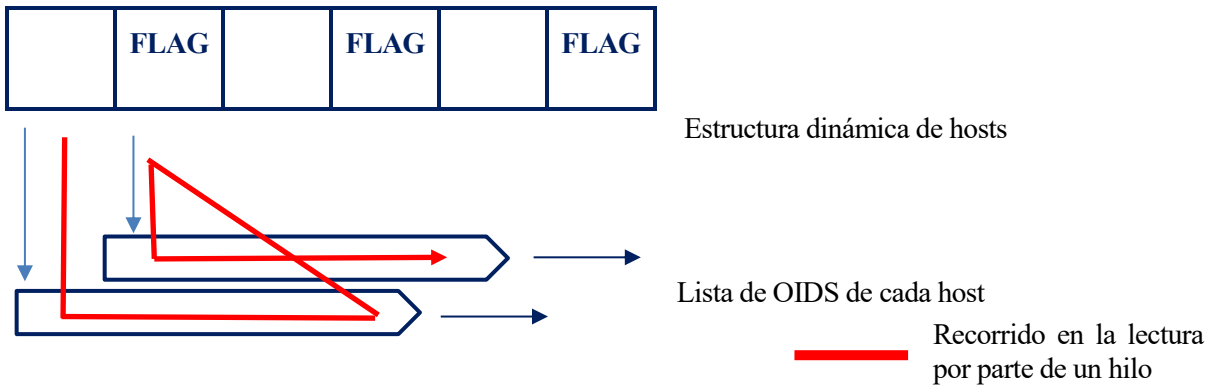


Fig. 14 Ilustración del recorrido por la estructura de un hilo

durante el desarrollo del código.

1.3.2.3 Parte 3: Cierre

Se puede observar que el proceso de monitorización no tiene fin. Cuando se llega al final de la monitorización, se espera el tiempo indicado y se vuelve a empezar.

A pesar de no ser un deseo expreso del funcionamiento de la herramienta, necesitamos que esta se cierre de una manera segura. Para ello, en cuanto se recibe la señal de cierre se llama a un manejador que realizará lo siguiente:

- Se indica a los hilos que deben de finalizar su monitorización.
- Los hilos, en cuanto pueden finalizar de manera segura la monitorización, terminan.
- El programa principal, tras terminar de esperar a todos los hilos, pasa a liberar la memoria.
- Una vez finalizados todos estos procesos, el programa termina.

1.3.3 Objetivos alcanzados

Posteriormente se detallarán los objetivos alcanzados. Principalmente son:

- Funcionamiento asíncrono y multihilo de la herramienta
- Varios puntos de prevención de fallos durante la ejecución
- Recarga dinámica de la configuración
- Funcionamiento continuo e ininterrumpido, con periodos de monitorización
- Funcionamiento de SNMP versión 1 y 2c
- Envío usando portocolo de mensajería Apache Kafka
- Sistema de logs para las notificaciones y errores

Debido a que el desarrollo de la herramienta nos obligó a reducir el alcance del proyecto, se quedan en el tintero los siguientes aspectos:

- Intregración con Zookeeper
- Empaquetado de datos de monitorización, facilitando la lectura del manager
- Representación de los datos en el frontend de RedBorder

1.4 Conclusiones

Sirva esta introducción para tener una visión global del proyecto. En los siguientes apartados documentaremos y detallaremos cada unos de los aspectos aquí mencionados.

El resultado final es satisfactorio. Personalmente y tras el bache inicial de Zabbix, me siento orgulloso del trabajo realizado. A la espera de la implantación e integración en el proyecto SIGMONA, y de completar el trabajo realizado con partes complementarias, se plantan las bases de la herramienta de monitorización SNMP que usará el equipo de ENEO Tecnología en sus proyectos.

2 MOTIVACIÓN INICIAL

Texto elaborado por Alberto Jódar Borrego, alumno de la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla

A inicio de este curso, cuando Pablo Nebrera y yo acordamos la realización de este proyecto, todo era demasiado nuevo para mí. No sabía lo que hacían en ENEO Tecnología, no sabía que era RedBorder y lo más importante, aún no sabía que era realmente dedicarse a la seguridad TIC.

Pablo, Jaime Nebrera[23] y yo mantuvimos una reunión en la que me presentaron el "modus operandi" de ENEO Tecnología. Conocí su producto "estrella", RedBorder y los hitos que habían alcanzado, lo cual representaban grandes avances en el mundo de la seguridad TIC.

Ellos mantenían su producto, RedBorder, como un producto ambicioso. De una forma general, podemos decir que RedBorder es un cerebro en una red, cerebro que recopila datos de todas fuentes de emisión posibles, los procesa, actúa sobre la red en caso necesario, presenta unas medidas de prevención, una defensa activa y pasiva… Y todo esto lo presentan al usuario a través de un frontend[24].

[23] Jaime Nebrera, cofundador de ENEO Tecnología junto a su hermano Pablo Nebrera (tutor de este proyecto), trabaja como CEO y director de tecnología.

[24] Interfaz frontal, es la parte del software que interactúa con los usuarios.

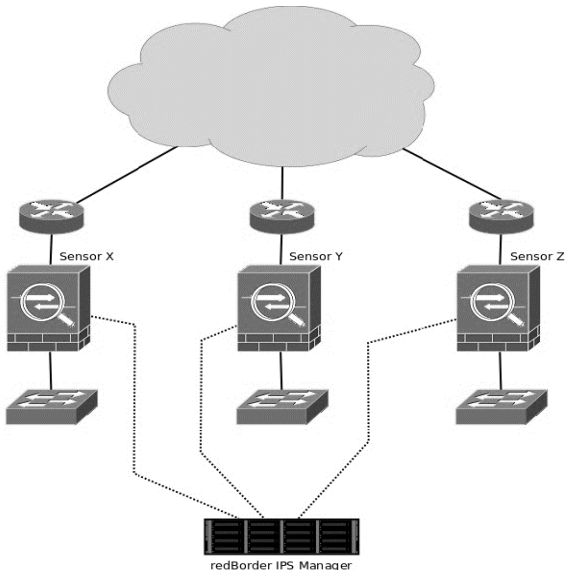


Fig. 15 Escenario básico de aplicación de RedBorder

Esta gran cantidad de datos que managerá el manager de RedBorder (es así como se denomina al que hemos llamado cerebro), serán enviados, recibidos y procesados aplicando técnicas de Big Data, tecnología totalmente desconocida para mí y tan presente en el mundo de las TIC.

Viendo la Fig. 11 nos hacemos una idea del funcionamiento general y del impacto en una red del producto de seguridad RedBorder.

Uno de los hitos pendientes en el desarrollo de RedBorder, es el de desarrollar un sensor SNMP que recopile datos y los incorpore al flujo de datos que reciba el manager.

El objetivo de este proyecto por tanto será el de adaptar, desarrollar e integrar una herramienta que tenga esta funcionalidad.

2.1.1 Planteamiento inicial

En una segunda reunión, Jaime Nebrera plantó las bases del proyecto. El objetivo era el de incoporar una sonda SNMP en diferentes puntos de la red. Esta sonda se encargará de consultar periódicamente determinados objetos indicados por el manager, e incorporar su valor al flujo de datos procesados por el manager.

La herramienta debe adaptarse a unas condiciones de trabajo impuestas por el objetivo final, que es el de trabajar conjuntamente con las diferentes herramientas proporcionadas por RedBorder[25]. Para conocer las características que se buscan con la herramienta, debemos conocer los puntos que "delimitan" nuestro trabajo. Esto es, conocer el entorno en el que se va a trabajar y conocer los objetivos que se desean conseguir, para así establecer las características deseadas.

Desde el punto de vista del entorno en el que se va a trabajar, tenemos lo siguiente:

- La sonda debe estar orientada a trabajar en redes de gran extensión.
- No se puede obviar la eficiencia del funcionamiento de la herramienta, ya que a pesar de que se implantará en equipos de gran rendimiento, estos alojarán gran cantidad de servicios y manejarán una cantidad muy grande de datos.
- El manager o cerebro del RedBorder debe controlar esta herramienta, tanto en su configuración general como en los objetos a monitorizar.
- La salida de datos debe estar adaptada a los protocolos del manager.

Estos puntos de trabajo hacen obvias las siguientes características de la herramienta:

- El trabajar con redes de gran extensión implica que, la catidad de dispositivos o hosts a monitorizar, además de la cantidad de objetos a monitorizar de cada host es muy elevada. Esto hace indispensable un funcionamiento asíncrono en el envío y recepción de los mensajes SNMP (PDU).
- La eficiencia en el funcionamiento de la herramienta nos obliga a un desarrollo multihilo de la misma. En puntos posteriores desarrollaremos los pasos seguidos y las consideraciones tomadas para ello.
- En cuanto al manejo de la configuración por parte del manager, se intuye como caso más simple el de manejarla a través de ficheros de configuración. La sonda tendrá definida la ruta por defecto del fichero

[25] Para entender esto, se debe comprender que RedBorder es un conjunto de herramientas, todas ellas controladas por un software central, el que ellos denominan el "manager". La herramienta de monitorización SNMP que aquí se trata, debe de tener presente el entorno donde va a funcionar y con el conjunto de herramientas con las que se va a integrar.

de configuración, en la cuál el manager colocará estos ficheros. Durante la ejecución, si el manager carga un nuevo fichero, sólo tendrá que indicar mediante una señal a la sonda que esta debe recargar la configuración. Se detallará en puntos posteriores.

- Por imposición del equipo de ENEO Tecnología, la salida de datos hacia el manager deberá ser bajo el protocolo Apache Kafka, usando para ellos la librería "librdkafka"[26] que será detallada en capítulos posteriores.

Tras esto, definimos la siguiente tabla como "plan de viaje" o como guía durante la investigación y desarrollo del proyecto.

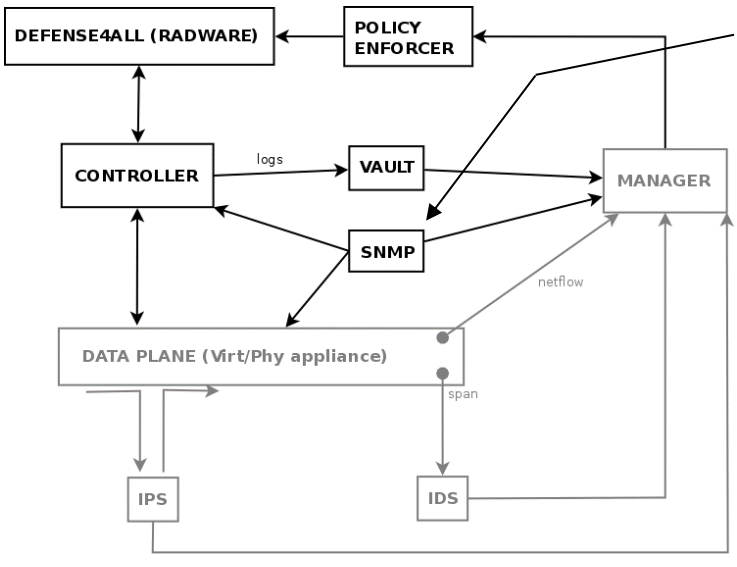
Condición de trabajo	Característica de la herramienta
Red de gran extensión	Asíncrona
Eficiencia de funcionamiento	Multihilo
Control al manager	Recarga en ejecución de los ficheros de configuracion
Salida adaptada al manager	Salida a través de Apache Kafka

Tabla 2 Objetios y características perseguidas

Esta hoja de ruta marca las características principales que planteamos para la búsqueda de una herramienta. Será en todo momento el horizonte del proyecto y lo mencionaremos durante todo el desarrollo.

2.1.2 Proyecto SIGMONA

Tras las primeras reuniones y tras plantar las bases del proyecto, surge una necesidad y a la vez una oportunidad. Desde ENEO Tecnología se ve la oportunidad de incorporar la recopilación de datos SNMP a su trabajo en un proyecto de nivel europeo, el proyecto SIGMONA.



La incorporación de nuestra herramienta al proyecto SIGMONA no modifica nuestra tabla de objetivos y características, ya que, el trabajo de ENEO Tecnología es el de incorporar el conjunto de sus herramientas de seguridad al despliegue de la red móvil SIGMONA.

Antes de continuar, vamos a conocer el proyecto.

Fig. 16 Esquema del desarrollo de ENEO Tecnología en SIGMONA

[26] Librería que implementa el protocolo de mensajería Apache Kafka. Más información en: https://github.com/edenhill/librdkafka

2.1.2.1 ¿Que és el proyecto SIGMONA[27]?

Información completa del proyecto por parte del equipo de Nextel, empresa matriz de ENEO Tecnología y participe del desarrollo de este proyecto.

Citando a Nextel [1]:

"El proyecto SIGMONA, "SDN Concept in Generalized Mobile Network Architectures", estudiará las arquitecturas de red y funciones para la evolución de las redes móviles LTE / EPC (3GPP). El foco principal está en la red, aunque también se incidirá en un enfoque del sistema end-to-end, incluyendo el sistema de radio LTE. El proyecto aplicará las últimas tecnologías de conexión de redes y de computación, y arquitecturas en la red móvil LTE / EPC móvil.

El concepto de Software Defined Networking (SDN) permite la separación del plano de reenvío de datos de los planos de control. Los interruptores SDN habilitados, routers y puertas de enlace son controlados a través de un sistema operativo SDN controlador / red, y son vistos como recursos virtuales. El plano de control de los elementos de conexiones de redes se puede desplegar en la nube por parte del operador. Las aplicaciones proporcionadas por un operador, como son las Redes de Entrega de Contenidos (CDN), Voz sobre IP (VoIP) e IP-TV también pueden ser implementadas en la nube.

Al introducir una Red Móvil Definida por Software, cambiaría la arquitectura de red de las actuales redes LTE (3GPP). Abrirá nuevas oportunidades para el tráfico, los recursos y la gestión de la movilidad, e impondrá nuevos retos en materia de seguridad en la red. Las soluciones relativas a la virtualización de red en las redes móviles de transporte, así como efectos sobre la supervisión de la red y las soluciones de gestión de red, también serán relevantes. Además se prevé un cambio en las inversiones en la red y los costes de funcionamiento. Por otra parte, puede cambiar la cadena de valor y surgir nuevos modelos de negocio. Todos estos aspectos serán estudiados en el proyecto SIGMONA.

Fig. 17 Logo proyecto SIGMONA

El objetivo principal del proyecto es la aplicación de las últimas tecnologías de computación y de redes y arquitecturas en la red LTE / EPC móvil de banda ancha (3GPP).

El proyecto se centrará principalmente en la evaluación, especificación y validación de un Concepto de red móvil Definido por Software el cual aplica SDN (del inglés Software Defined Networking), la virtualización de redes y el cloud computing o "computación en la nube" en las redes móviles LTE. El proyecto proporcionará un entendimiento de la viabilidad y las oportunidades de los conceptos de este tipo de red, así como unan evaluación de los límites del rendimiento y de la escalabilidad de las nuevas tecnologías aplicadas a las redes móviles de banda ancha. Serán considerados distintos casos de uso, como la distribución de contenidos. También se expondrá el impacto en costos de la red, la cadena de valor y los modelos de negocio. Además será definido el impacto en el modelo de interfaces abiertas y la función de los estándares y órganos relacionados."

Como conclusión y de una forma resumida, el proyecto SIGMONA se encarga de recoger todas las técnicas de routing, switching, seguridad… aplicadas en las redes modernas, y adaptarlas a un concepto de red móvil definido por software.

[27] Información aportada por Nextel, empresa "padre" de ENEO Tecnología. Fuente: http://www.nextel.es/idi2/sigmona

2.1.2.2 Este proyecto en SIGMONA

ENEO Tecnología aportara su producto RedBorder para ser adaptado a la red del Proyecto SIGMONA. Como hemos visto en el esquema, RedBorder desplegará su manager en la red, que actuará sobre esta a través del IPS[28], y del Policy Enforcer (no se explicará ya que no es motivo de trato en este proyecto), y que recopilará información a través del IDS[29], de una herramienta de recopilación, adaptación y normalización de logs, y de nuestro poller SNMP.

Es en ese punto de la estructura de seguridad de la red a través de RedBorder, donde se localiza nuestra herramienta. Esto no cambia los objetivos que perseguimos y la búsqueda de la herramienta permanece intacta.

[28] IPS: "Intrusion prevention systems" o Sistema de prevención de intrusos
[29] IDS: "Intrusion detection system" o Sistema de detección de intrusiones

3 INVESTIGACIÓN

Si supiese qué es lo que estoy haciendo, no le llamaría investigación, ¿verdad?

- Albert Einstein -

No fue fácil la fase incial de investigación. Teniendo los requisitos, faltaba la vía por la que llevar a cabo la herramienta que teníamos en mente. Nunca es fácil empezar, pero esta vez Jaime Nebrera supo dar el empujón incial a este proyecto. Vamos a recorrer en esta sección cada una de las ideas y herramientas que investigamos y trabajamos en la búsqueda de la solución más adecuada.

3.1 Requisitos

Qué queremos. Esa es la pregunta que todos debemos hacernos a la hora de empezar un proyecto. Gracias al trabajo que se realizó en esta dirección, pudimos cerrar bien los objetivos fundamentales, cosa que nos fue de gran ayuda a la hora de plantear y valorar las diferentes herramientas y soluciones que planteabamos.

3.1.1 Objetivos

Como se ha comentado en el punto anterior (vea **2. Planteamiento incial**), el primer planteamiento de éste proyecto dejó claro varios puntos fundamentales que aquí resumiremos. Podemos decir a rasgos generales y sin entrar en detalle que tres palabras definen perfectamente el objetivo que perseguiamos con la herramienta. Estas

son, **eficiencia, escalabilidad e integración**.

3.1.1.1 Eficiencia

Nuestra herramienta está pensada para funcionar en redes grandes, donde el volumen de datos que deberá manejar, debido al gran número de elementos a monitorizar, se convierte en un punto determinante durante el desarrollo.

La forma de realizer éste proceso de manera eficiente es asegurando que el uso de la herramienta aproveche el máximo rendimiento disponible, esto es, siendo una aplicación multi hilo.

Añadimos esta característica a nuestro filtro de búsqueda, siendo de vital importancia. Se deberá garantizar que correrá el mayor número de hilos posibles, encargandose cada uno de realizar las funciones de monitorización.

3.1.1.2 Escalabilidad

Hay una cuestión de la que derivan fundamentalmente todas las carecterísticas deseadas, y esta es la necesidad de trabajar en grandes redes de datos.

Ya se ha comentado la necesidad de que la aplicación sea eficiente en el uso de recursos, pero se añade también otro requisito fundamental, la escalabilidad.

El protocol SNMP (*Simple Network Management Protocol*[30]) es un protocolo de capa de aplicación que nace en 1990 con su primera versión SNMP[31] (su última nace en 2002, SNMPv3[32], pero, a pesar de las significantes mejoras en cuanto a la seguridad en la comunicación, la mayoría de redes siguen usando principalmente sus versiones anteriores).

Esto nos indica que, a pesar de ser un protocolo que en versiones posteriores soluciona aspectos de seguridad y escalabilidad no incluidos en la primera version, la gran mayoría de herramientas desarrolladas en este aspecto son soluciones privadas que aportan soluciones concretas a un determinado escenario, por lo general no siendo este de gran volumen de datos.

Un objetivo principal será el de buscar una herramienta que sea capaz de trabajar en este entorno, siendo escalable. Esto trae consigo una consecuencia, la herramienta tiene que trabajar con el protocol SNMP de forma asíncrona. No es aceptable que el programa (o uno de sus hilos) se bloquee mientras espera a la recepción de la respuesta.

[30] Traducción: Protocolo Simple de Administración en Red
[31] RFC 1157: https://tools.ietf.org/html/rfc1157
[32] RFC 3410: https://tools.ietf.org/html/rfc3410

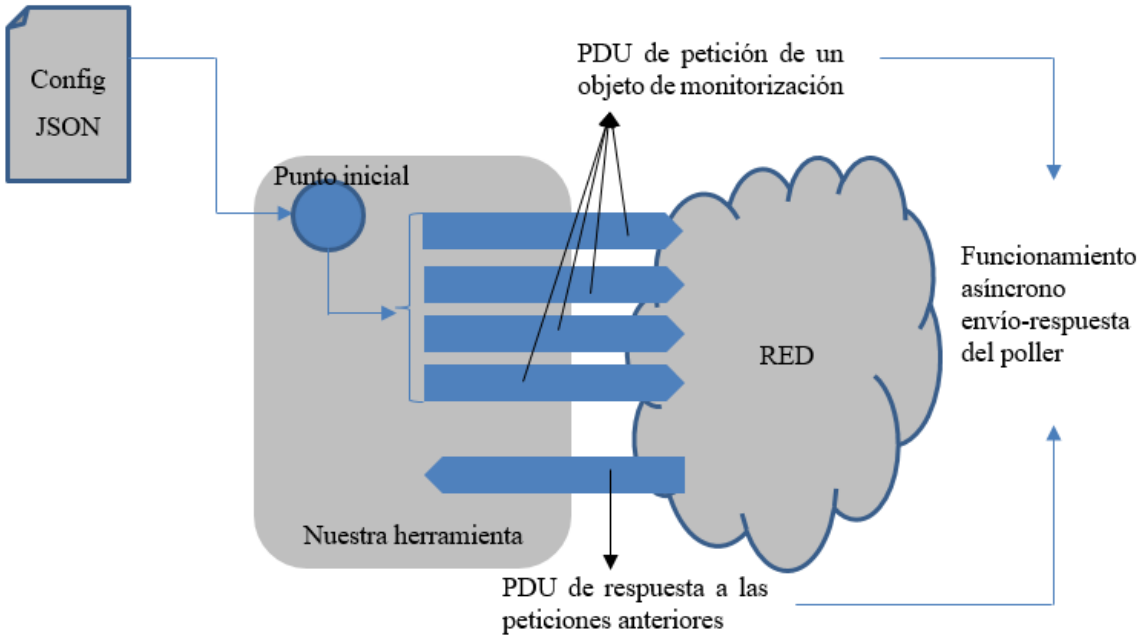


Fig. 18 Esquema del funcionamiento asíncrono con SNMP

Es decir, siguiendo la estructura de la argumentación dada hasta este momento, incorporamos al filtro de búsqueda de herramientas y soluciones, la escalabilidad. Escalabilidad dada en el sentido tanto del manejo de datos, como de la comunicación y proceso de petición-respuesta con los objetos a monitorizar.

3.1.1.3 Integración

Anotadas eficiencia y escalabilidad, vamos a describir y analizar el punto más restrictivo, la integración.

Tras todas las características descritas, y todas las condiciones fijadas, hacer que la solución adoptada funcione y se integre con RedBorder (solución completa de seguridad desarrollada principalmente por el equipo de ENEO Tecnología) se convierte en el eje de giro principal de éste proyecto.

¿Cómo o con qué ha de integrarse?

3.1.1.3.1 Control sobre la herramienta

El manager de RedBorder es el software central, el proceso encargado de la comunicación con el resto de herramienta, de la extracción de sus datos, de su proceso y de su aplicación en la red, además de controlar la interacción con el usuario a través de su frontend.

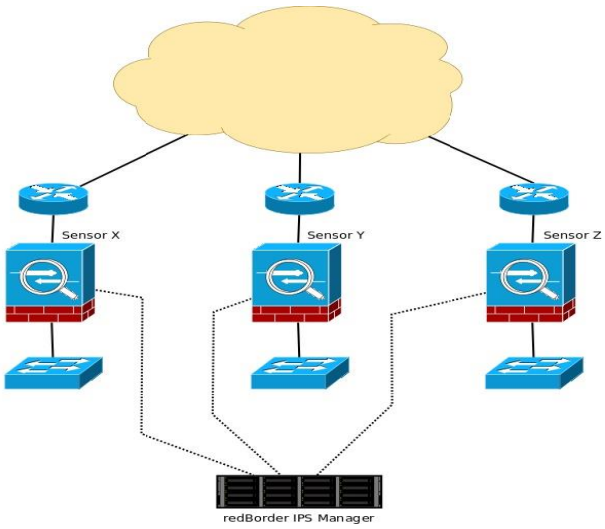


Fig. 19 Escenario básico de aplicación de RedBorder

Como podemos ver en esta imagen proporcionada por el equipo de desarrollo de RedBorder a modo de escenario básico[33], el manager es el software central encargado de controlar los diferentes sensores que recopilan datos sobre la red en la que vamos a trabajar.

De esta misma forma ha de funcionar nuestra solución de monitorización SNMP. Nuestro sensor debe integrarse en la red, recibiéndo su configuración por parte de éste y a su vez enviándole al manager el resultado de la monitorización.

3.1.1.3.2 Obtención del resultado

De igual manera y siguiendo la metodología manager-sensor aplicada por RedBorder y como necesidad impuesta por los puntos anteriores (escalabilidad y eficiencia), se hace necesario el uso de un protocolo de comunicación eficiente. En puntos posteriores hablaremos de Apache Kafka como la solución usada por RedBorder para la comunicacióne entre manager y sensor.

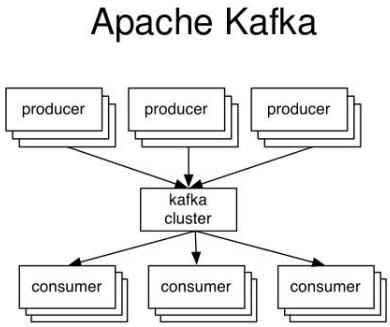


Fig. 20 Esquema del funcionamiento de Apache Kafka

Nuestra herramienta o sensor, ejecutará el productor de Apache Kafka mientras que en el manager se ejecutarán tantos consumidores como se necesiten para recopilar los datos de los diferentes sensores. Como hemos comentado se verá en puntos posteriores.

3.1.1.3.3 Software libre

El último punto en éste apartado, que además no es una limitación técnica, es la necesidad de que la herramienta sea de software libre. Esto viene impuesto por la filosofía de trabajo del equipo de ENEO Tecnología.

Debíamos incorporar a la búsqueda un filtro adicional, la herramienta debía de ser de software libre. O en todo caso, partir de una herramienta de software libre para realizar las modificaciones oportunas que nos condujeran al objetivo final.

[33] Fuente: http://support.redborder.net/hc/en-us/articles/201698231-1-3-Basic-scenario-of-the-redBorder-IPS-solution

3.1.2 Primera propuesta

Zabbix es una solución de monitorización presente en multitud de empresas. Incluye opciones de monitorización sobre diferentes plataformas y protocolos, incluyendo una version propia de agente de monitorización (vease https://www.zabbix.com/documentation/3.0/manual/concepts/agent).

Encontramos Zabbix como primera solución, y sin considerer más soluciones pasamos a trabajar con ella (error que posteriormente comentaremos).

Zabbix cumplía a priori todos los requisitos impuestos:

- Herramienta de software libre[34]
- Asíncrona en la monitorización
- Multihilo

Todos salvo uno, la integración. Este sería nuestro trabajo y nuestra principal piedra en el camino.

Fig. 21 Mapa simplificado de la presencia de Zabbix

3.2 Zabbix

Zabbix[35] fue la primera herramienta que contemplamos para dar solución a la monitorización. A priori Zabbix contaba con los siguientes puntos fuertes:

- Herramienta de software libre
- Solución presente en multitud de empresas del sector[36]
- Documentación extensa y equipo de desarrollo activo

Zabbix nos aportaba una solución de monitorización sencilla, la cual poder incorporar a Redborder sirviéndonos de base para el objetivo final.

ZABBIX
The Ultimate Open Source Monitoring Solution

Fig. 22 Logotipo de Zabbix

Una instalación simple desde la página oficial y comenzamos a trabajar con esta herramienta.

3.2.1 Primeros pasos con Zabbix

Zabbix es compatible con infinidad de plataformas[37], añadiendo esto un punto a favor a esta herramienta.

Podemos resumir el comportamiento de la herramienta

[34] Código disponible en: http://www.zabbix.com/download.php
[35] Página oficial Zabbix: http://www.zabbix.com/
[36] Empresas más importantes que usan Zabbix en su red: http://www.zabbix.com/users.php
[37] Plataformas compatibles con Zabbix: http://www.zabbix.com/download.php

con el siguiente esquema [2]:

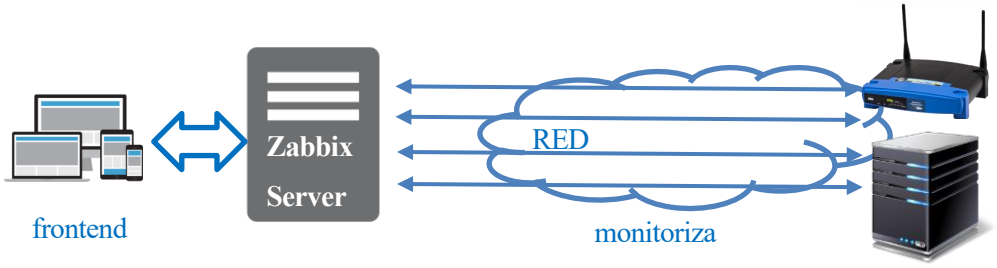


Fig. 23 Funcionamiento básico Zabbix

"Zabbix server"[38] es el proceso central del software de Zabbix. Este se encarga de la monitorización, almacenado temporal, calculo de umbrales, envío de notificaciones…

Realmente Zabbix server se puede dividir en tres componentes [2]:

- Zabbix Server
- Web Frontend
- Base de datos

Analizando este funcionamiento, nuestra idea era usar Zabbix incorporado al manager de Redborder como un proceso:

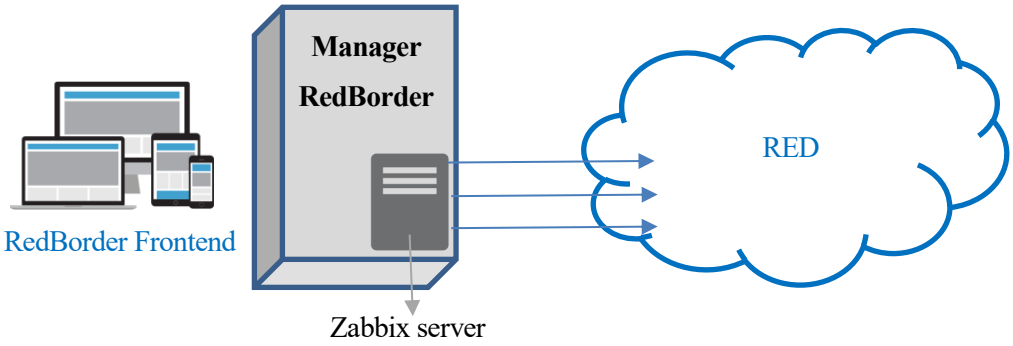


Fig. 24 Esquema intengración Zabbix con RedBorder

Esta idea nos trajo los primeros inconvenientes de trabajar con Zabbix, y estos eran:

- Zabbix server se basaba en la comunicación con su frontend. En nuestro caso, el manager de RedBorder debía simular esa comunicación, con objeto de configurar el server y de recibir los datos que resultan de la monitorización.
- Zabbix server, como se ha comentado, mantiene una base de datos en pro de la persistencia de los mismos. A pesar de que esto pueda parecer una ventaja, en nuestro caso no lo era. El objetivo de nuestra herramienta es trabajar con redes de gran dimensión y el uso de la base de datos añadía un punto de fallo no deseable.

[38] Documentación Zabbix server: https://www.zabbix.com/documentation/3.0/manual/concepts/server

3.2.1.1 Zabbix proxy

Continuando con el trabajo con Zabbix y con el de su documentación, observamos una herramienta llamada "Zabbix proxy" [3][39]. Zabbix ofrece una solución a aquellos clientes que tienen su red separada geográficamente. Para ello se implementa un proxy del Zabbix server en dicha red:

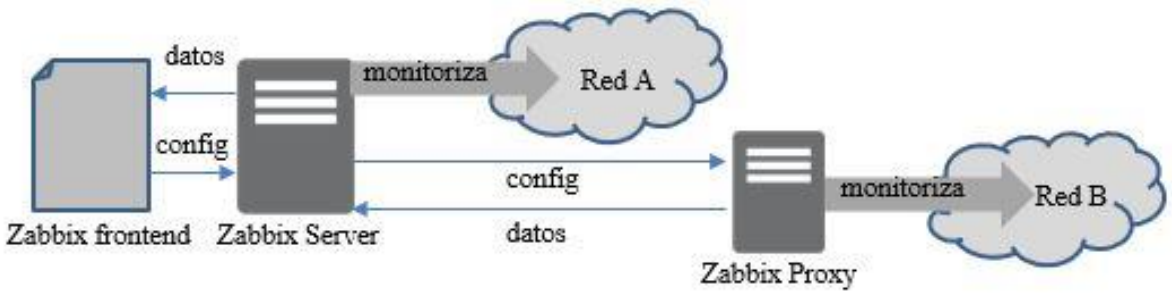


Fig. 25 Esquema Zabbix server y Zabbix proxy

Una herramienta sencilla encargada sólo de la recepción de la configuración y del envío del resultado de la monitorización de vuelta al server. Herramienta sencilla que encaja perfectamente con la idea inicial.

En ese caso, la adaptación con RedBorder sería diferente a lo planteado con Zabbix server. El manager sustituiría a Zabbix server y el Zabbix proxy se encargaría de la monitorización.

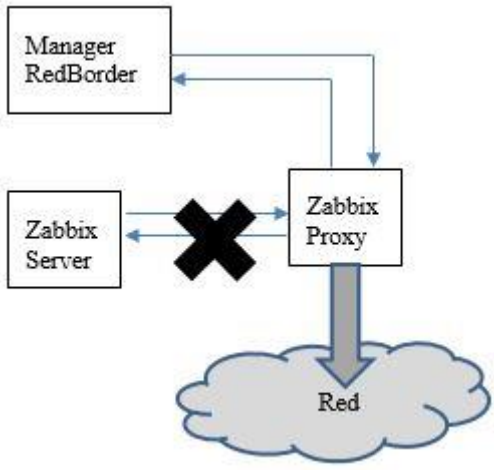


Fig. 26 Esquema adaptación Zabbix proxy con RedBorder

El objetivo sería tomar Zabbix proxy y sustituir su comunicación con Zabbix server por una comunicación con el manager de RedBorder, así se recibiría la comunicación y sólo deberíamos modificar el formato de salida del proxy para que se adapte al requerido por el manager.

Esto suponía una enorme ventaja frente a la solución inicial:

- Zabbix proxy es una herramienta pensaba solo para la monitorización. Podríamos decir que se encarga de monitorizar ciertos elementos, y los envía de vuelta una vez obtenidos. Esto encajaba perfectamente con nuestro objetivo.
- Zabbix proxy se ejecuta como un proceso y es mucho menos exigente en recursos que Zabbix server.
- Zabbix proxy, al igual que Zabbix server ofrece el servicio completo de monitorización SNMP. Monitoriza elementos en todas las versiones (1, 2c y 3), es capaz de establecer umbrales, capturar Traps y diseñar agentes[40] específicos para cada aplicación.

[39] Documentación de Zabbix proxy: https://www.zabbix.com/documentation/3.0/manual/concepts/proxy
[40] Agentes en Zabbix: https://www.zabbix.com/documentation/3.0/manual/concepts/agent

	Proxy
Lightweight	**Yes**
GUI	No
Works independently	**Yes**
Easy maintenance	**Yes**
Automatic DB creation[1]	**Yes**
Local administration	No
Ready for embedded hardware	**Yes**
One way TCP connections	**Yes**
Centralised configuration	**Yes**
Generates notifications	No

Tabla 3 Características de Zabbix proxy[41] [4]

Dicha tabla es proporcionada en la documentación para resolver una cuestión. Cuando un cliente usa Zabbix para monitorizar su red, bajo qué casos esta justificado el uso de Zabbix proxy. O dicho de otra forma, ¿cuando hay que plantearse el uso de Zabbix proxy?

Podemos observar varios detalles interesantes. El primero de ellos es que, la propia docmentación hace incapié en el menor peso de Zabbix proxy, es decir, que se presenta como una solución mucho menos intensiva en el uso de recursos. Esto, como ya se ha comentado, es muy importante si observamos el proyecto desde una visión más global. El manager de RedBorder es una herramienta que se encarga de controlar todos los aspectos de la red, de recopilar todos sus datos y de presentar un frontend actualizado al usuario. Esto pasa desde análisis de malware y detección de intrusos hasta el análisis, parseado y filtrado de logs[42].

Otro punto importante es que, según se nos proporciona, Zabbix proxy opera de manera independiente. Esto es cierto en cuanto a que Zabbix proxy es un proceso que corre sin dependencia de que exista un Zabbix server. Pero al trabajar con la herramienta apreciamos que el proxy no puede funcionar sin el server, ya que es de este de quien recibe la configuración y la lista de objetos a monitorizar. Discutiremos esto más adelante como una desventaja de esta solución.

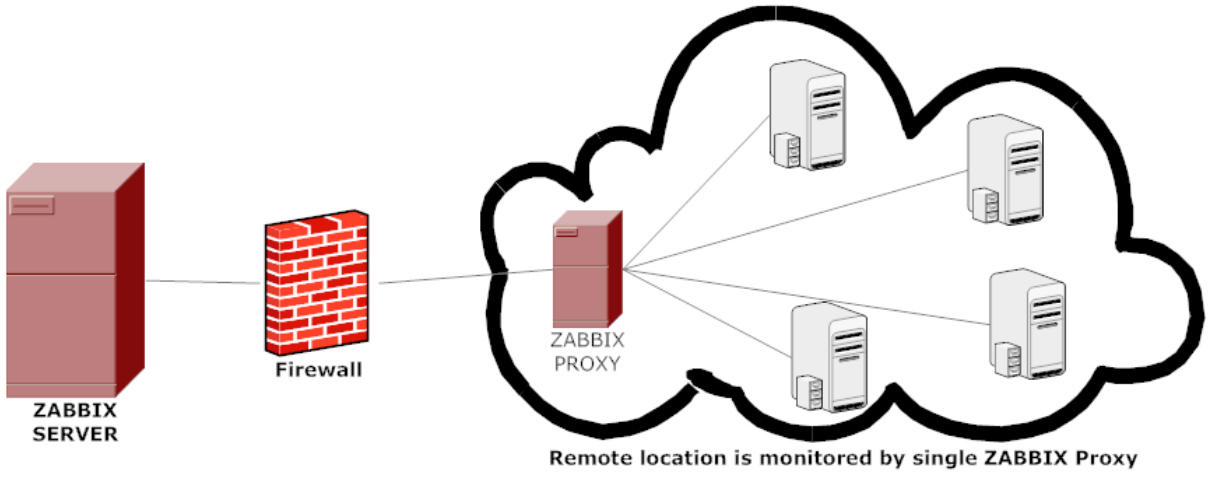


Fig. 27 Monitorización remota de Zabbix proxy[43]

Por último comentamos los puntos que se presentan como una piedra en el camino. Puntos que posteriormente

[41] Tabla comparativa proporcionada en la documentación de Zabbix: https://www.zabbix.com/documentation/3.0/manual/distributed_monitoring
[42] Más información sobre las funcionalidades de RedBorder: https://redborder.com/how-it-works
[43] Esquema proporcionado en la documentación de Zabbix: https://www.zabbix.com/documentation/3.0/manual/distributed_monitoring/proxies

estudiaremos y que son clave para replantear la continuación del proyecto con el uso de esta herramienta.

El primero de ellos es el ya comentado, y es el que podemos identificar en la tabla como "Local administration" (administración en local). Esto significa que hay un dependendcia con el server en cuanto a la configuración y administración.

El otro punto importante es "Automatic DB creation" (creación automática de una base de datos). Este es un punto que discutiremos en las desventajas de esta herramienta y que, fundamentalmente, introduce un punto de fallo a la herramienta no deseable.

Con todo esto se observa que Zabbix proxy es una solución razonable a nuestro problema, luego las tareas de desarrollo para una completa adaptación serían:

- Identificar el formato de envío de la configuración por parte de Zabbix server para que podamos replicarlo en nuestro manager.
- Identificar el formato de salida de datos para poder modificarlo y adaptarlo a la mensajería de alto rendimiento de Apache Kafka.

3.2.1.1.1 Formato de la configuración

```
connect_to_server(&sock, 600, CONFIG_PROXYCONFIG_RETRY); /* retry till have a connection */

if (SUCCEED != get_data_from_server(&sock, ZBX_PROTO_VALUE_PROXY_CONFIG))
    goto out;

if ('\0' == *sock.buffer)
{
    zabbix_log(LOG_LEVEL_WARNING, "cannot obtain configuration data from server: empty stri
    goto out;
}

if (SUCCEED != zbx_json_open(sock.buffer, &jp))
{
    zabbix_log(LOG_LEVEL_WARNING, "cannot obtain configuration data from server: %s", zbx_j
    goto out;
}
```

Code 1 Identificación del formato de configuración

Del código de Zabbix [5] proxy y centrándonos sobre la función encargada de la recepción de la configuración por parte del servidor, vemos que esta espera una configuración recibida en formato JSON[44] a través de una conexión TCP.

Es suficiente información de momento. Posteriormente analizaremos las ventajas y desventajas de la herramienta y hablaremos del formato de la configuración.

[44] JSON: http://www.json.org/

3.2.1.1.2 Salida de datos

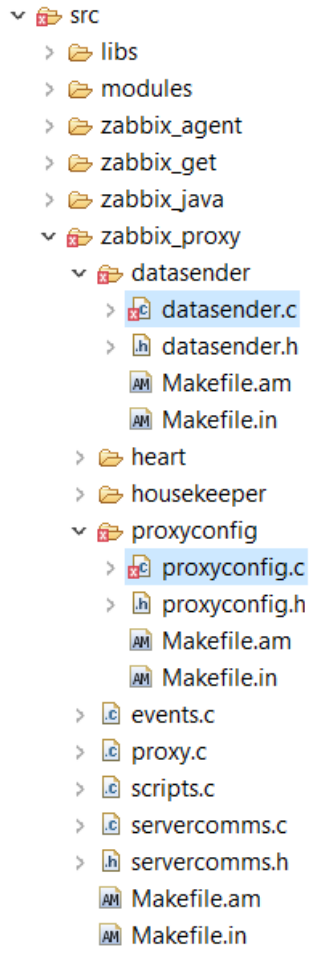


Fig. 28 Estructura de ficheros de Zabbix [5]

Como podemos ver, la estructura interna de la herramienta es bastante clara. Identificada en el proxy la función de configuración y detectado que este espera un archivo JSON a través de una conexión TCP con el server, vamos a analizar el envío de datos por parte del proxy.

Dentro del "datasender", que podemos traducir como el "emisor de datos" encontramos tres funciones:

- **static void history_sender (…)**

  Función que se conecta al server para enviar en formato JSON mediante una conexión TCP el conjunto de datos que se tengan almacenados

- **static void host_availability_sender (…)**

  Función que se conecta al server para enviar en formato JSON mediante una conexión TCP un listado de disponibilidad de los objetos que el proxy monitoriza

- **ZBX_THREAD_ENTRY (…)**

  Función que será ejecutada por un hilo. Bucle continuo encargado de la manipulación de los datos entrantes, de su introducción en la base de datos y de la llamada periódica a **history_sender ()**

Como podemos observar, el formato que se usa en la comunicación con el server es JSON, y es algo deseado desde un inicio debido a que es esta la forma de trabajar del manager de RedBorder.

A pesar de esto vemos que el envío de datos incluye de manera estricta un paso de estos por la base de datos local. Esto es un gran inconveniente que analizaremos a continuación.

3.2.2 Ventajas e inconvenientes de Zabbix

A lo largo del análisis de la herramienta, de localizar e identifcar todas sus características y de establecer el esquema de cómo se adaptará a nuestras necesidades y a la de los formatos específicos del manager de RedBorder, pudimos establecer una serie de ventajas e inconvenientes que hizo plantearse la continuación en el uso de Zabbix.

A continuación se listarán una serie de funcionales adicionales de Zabbix para posteriormente analizar la tabla de ventajas e inconvenientes.

3.2.2.1 Base de datos y función “housekeeper”

Los desarrolladores de Zabbix nombraron como “housekeeping” a todo el proceso de insertado, actualización y borrado de datos por parte del server o el proxy en la base de datos local.

Podemos decir que la dirección de los datos es:

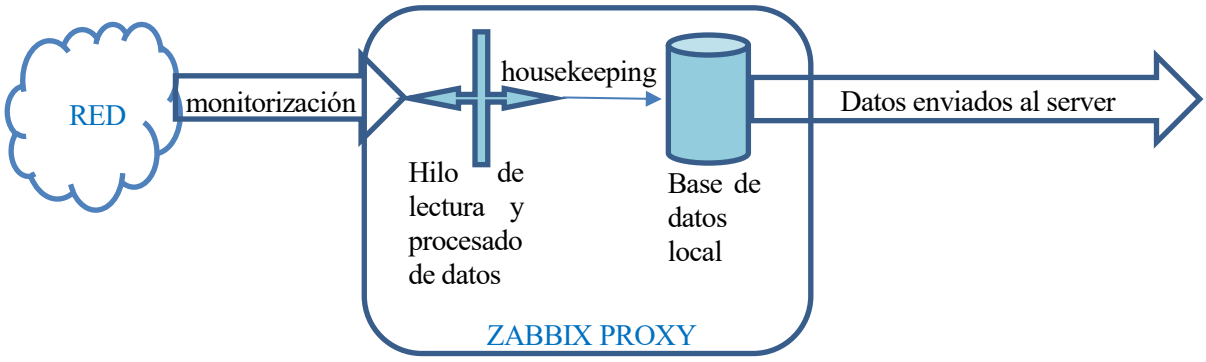


Fig. 29 Esquema de tránsito de datos dentro de Zabbix proxy

Como podemos ver, en el proceso de Zabbix proxy hay una constante interacción con la base de datos local. Las bases de datos que acepta Zabbix son relacionales [6], entre otras[45]:

- MySQL
- PostgreSQL
- Oracle
- IBM DB2
- SQLite

De entre todas estas opciones, se hace recomnedable el uso de SQLite. Durante el proceso de instalación de Zabbix, la base de datos será creada de manera transparente.

A pesar de esta comodidad que nos proporciona usar SQLite, el uso de una base de datos relacional es un punto a considerer debido a la gran extension de la red que vamos a manejar. El paso intermedio por una base de datos local es una desventaja que analizaremos en puntos posteriors.

3.2.2.2 Conectividad y función “heartbeat”

La funcionalidad llamada de “heartbeat”, de su traducción “latido de corazón”, se basa en una comunicación periódica del servidor con objeto de detector pérdida de conectividad con alguno de los hosts.

Su función principal:

static int send_heartbeat(void)

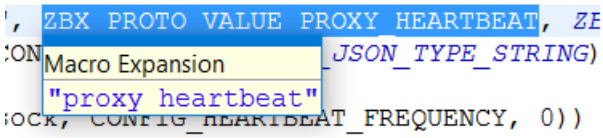
se encargará de enviar en formato JSON, a través de una conexión TCP, dos strings, “host” y “request”, como podemos ver a continuación:

[45] Creación de una base de datos para Zabbix: https://www.zabbix.com/documentation/3.0/manual/appendix/install/db_scripts

```c
static int	send_heartbeat(void)
{
	zbx_sock_t	sock;
	struct zbx_json j;
	int		ret = SUCCEED;
	char		*error = NULL;

	zabbix_log(LOG_LEVEL_DEBUG, "In send_heartbeat()");

	zbx_json_init(&j, 128);
	zbx_json_addstring(&j, "request", ZBX_PROTO_VALUE_PROXY_HEARTBEAT, ZBX_JSON_TYPE_STRING);
	zbx_json_addstring(&j, "host", CONFIG_HOSTNAME, ZBX_JSON_TYPE_STRING);

	if (FAIL == connect_to_server(&sock, CONFIG_HEARTBEAT_FREQUENCY, 0)) /* do not retry */
		return FAIL;

	if (SUCCEED != put_data_to_server(&sock, &j, &error))
	{
		zabbix_log(LOG_LEVEL_WARNING, "sending heartbeat message to server failed: %s", error);
		ret = FAIL;
	}

	zbx_free(error);
	disconnect_server(&sock);

	return ret;
}
```

Code 2 Función de heartbeat en zabbix proxy

Estos strings contendrán la información del host y del tipo de request realizada, que en este caso, será una definida como **ZBX_PROTO_VALUE_PROXY_HEARTBEAT** para indicar que es un request de "heartbeat".



Code 3 Macro que define la petición de "heartbeat"

3.2.2.3 Análisis

Analizadas las principales funciones de Zabbix y teniendo en mente la adaptación al proyecto para que cumpla nuestros objetivos inciales, vamos a analizar una serie de ventajas e inconvenientes que implican el usar esta herramienta.

Estos inconvenientes deberán de ser solventados con trabajo adicional sobre la herramienta. Al ver las conclusiones veremos el dilema planteado sobre si trabajar para solventar los inconvenientes o buscar otra herramienta.

La investigación pasa a una siguiente fase. Búsqueda de nuevas soluciones, análisis de estas incluyendo la ya planteada y la toma de decision.

3.3 Soluciones alternativas

La investigación nos llevó a las siguientes herramientas.

3.3.1 RFC [7] [8]

Esta herramienta[46] es en sí un modulo en Perl. Es una herramienta de monitorización basada en la librería **net-snmp** [9][47] para perl.

Como se puede observar ofrece una sencilla función de monitorización. Podemos decir que es una solución no profesional proporcionada en una comunidad creada en torno al lenguaje Perl (www.perlmonks.org).

A pesar de ofrecer una solución sencilla para la monitorización, siendo además muy efectiva, más de 7000 nodos en menos de 4 minutos, 1000 nodos en apenas 30 segundos usando un 25% de CPU… no cumple todos los objetivos iniciales.

Usando de base esta herramienta deberíamos implementar:

- Una comunicación con el manager de RedBorder, para que este pueda enviarle la configuración de monitorización, así como los objetos a monitorizar
- Añadir parámetros de configuración tales como el tiempo entre envíos de PDU
- Modificar el formato de salida. Actualmente esta herramienta imprime su salida por linea de commandos, luego se deberá añadir un modulo que envíe los datos al manager a través de Kafka
    - Este punto nos lleva a lo siguiente. Necesitamos una librería o modulo de Perl para hacer uso de dicho protocolo. Esto nos lleva a dos opciones:
        - API de Apache Kafka desarrollada por TrackingSoft y compartida de froma gratuita a través de GitHub (https://github.com/TrackingSoft/Kafka)
        - Productor[48] y consumidor[49] desarrollado por Sergey Gladkov compartido a través de **cpan.org** y de uso libre

      Ambas soluciones presentan una opción viable y a considerar, luego serán analizadas con el estudio de las difernetes soluciones.

Podemos considerer RFC como una solución válida para la monitorización, que sirviendonos de base y desarrollando ciertas funcionalidades necesarias e impuestas por los requisitos ya comentados, puede servirnos para alcanzar el objetivo final. Aun así, debemos de mencionar ciertos inconvenientes para poder hacer una posterior valoración entre todas las herramientas planteadas:

- A pesar de no ser un inconveniente técnico, la poca familiarización con el lenguaje Perl es un obstáculo que añade un tiempo adicional al desarrollo del proyecto, tiempo que debe ser invertido en la asimilación de conceptos y manejo de este lenguaje.
- Los puntos a desarrollar ya comentados añaden un trabajo extra sobre la herramienta. No es un obstáculo insalvable, pero es un trabajo extra que debemos valorar a la hora de realizar el análisis de las herramientas propuestas.
- No podemos asegurar que sea una herramienta testeada y evaluada en entornos reales. O al menos podemos decir que no esta tan desarrollada como Zabbix. Su última actualización data de Enero de 2008, luego, el no contar con actualizaciones es la mayor piedra en el camino de RFC.

[46] Herramienta: http://www.perlmonks.org/?node=664360 / http://www.perlmonks.org/?node_id=662528
[47] Net-snmp para Perl: http://net-snmp.sourceforge.net/docs/perl-SNMP-README.html
[48] Productor Kafka en Perl: http://search.cpan.org/~sgladkov/Kafka-0.9001/lib/Kafka/Producer.pm
[49] Consumidor Kafka en Perl: http://search.cpan.org/~sgladkov/Kafka-0.9001/lib/Kafka/Consumer.pm

3.3.2 Rated

Rated [10][50] es el proceso (daemon) de monitorización del proyecto RTG[51].

De su descripción podemos leer:

> **Rated is a high performance threaded SNMP poller written in C which stores the time series data in a SQL database. It uses the net-snmp library (http://www.net-snmp.org/) for retrieving the SNMP data and has a modular backend for connecting to various databases. At the moment it has drivers for MySQL, PostgreSQL and Oracle (experimental).**

Es decir, el proceso rated es de alta eficiencia, siendo multihilo. Almacena temporalmente los datos en una base de datos local al igual que Zabbix, luego incorporará este punto de fallo e inconveniente. Al igual que RFC, se basa en el uso de la librería net-snmp, pero en este caso, en lenguaje C.

Como vemos, de su descripción sacamos las principales carácterísticas de Rated, las cuales son extremadamente llamativas salvo una, el uso de la base de datos local.

Podiamos pensar que, el uso de esta herramienta es similar al del uso de Zabbix, pero mercee la pena recordar ciertas funcionalidades de Zabbix, como la de housekeeping, que Rated a priori no mantiene. Esto representa una ventaja respecto a Zabbix ya que la adaptación con RedBorder sería mucho menos costosa.

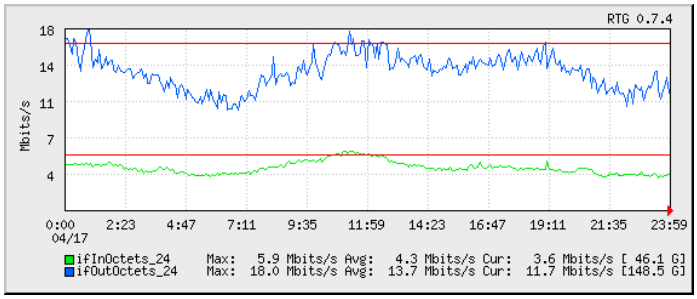


Fig. 30 Representación de datos en RTG. Similar a nuestro objetivo con el manager de RedBorder

A pesar de que el uso de la base de datos no representa tanta amenza como en Zabbix, eliminar esta funcionalidad y a la vez mantener las actualizaciones periódicas de esta herramienta no es algo trivial.

De Rated, podemos listar los siguientes inconvenientes:

- Mantiene una copia de los datos en una base de datos local. Esto se debe a que esta herramienta no esta pensada para trabajar de manera distribuida. En nuestro caso, es el manager de RedBorder el que mantendrá los datos y nuestra herramienta solo deberá encargarse de la monitorización.
- Modificar la salida de datos para poder usar Apache Kafka. El equipo de ENEO Tecnología usa en sus herramientas desarrolladas en C la librería **librdkafka**[52] la cual será estudiada en puntos posteriores.
- La recepción de la configuración debe ser dada por parte del manager, y a ser posible en formato JSON. Rated incopora la funcionalidad de cargar o recargar la configuración dada ante la recepción de la señal SIGHUP, luego, a pesar de ser un inconveniente, puede resultar una ventaja con un trabajo mínimo.

No profundizamos en estos aspectos y los dejamos para su posterior análisis.

Con todo Rated se presenta como una opción a considerar como solución alternativa a Zabbix.

[50] Rated en GitHub: https://github.com/mprovost/rated
[51] Proyecto RTG: http://rtg.sourceforge.net/
[52] Librdkafka: https://github.com/edenhill/librdkafka

3.3.3 Ruby-snmp

En la búsqueda de nuevas alternativas, no solo nos centramos en soluciones completas, sino también en librerías. SNMP es un protocolo que está asentado en internet, de uso mundial y de aparición en casi la totalidad de las redes. Es difícil decir por qué, pero el uso de SNMP en las diversas compañías ha sido muy personal y poco compartido. Cada red plantea un escenario diferente (basta ver las particularidades que impone nuestro escenario) y cada compañía desarrolla su solución.

Es por esto por lo que se nos plantea el uso de librerías que nos permitan adaptar nuestra herramienta a las necesidades concretas de nuestro escenario.

En un primer caso planteamos el uso de la librería ruby-snmp [11][53].

A pesar de no ser un lenguaje con el que estemos familiarizados, esta librería ofrece determinadas ventajas:

- Es una librería que implementa el protocol SNMP directamente, no usando otras librerías como net-snmp.
- Debido al punto anterior, ruby-snmp implementa ciertas funciones que otras librerias, como net-snmp, no.
- El manejo de lo los tipos de datos resultado de la monitorización se hace mucho más sencillo en ruby que en C.

Como podemos ver, esta librería implementa una serie de ventajas que hace que la consideremos como una opción de peso.

Aun así nos encontramos con el gran obstáculo de que, la herramienta, debe de ser desarrollada al completo. Esto nos proporciona la ventaja de adaptarla al entorno deseado, pero a su vez implica más horas de trabajo, y con total seguridad, de tener que acortar los objetivos del proyecto.

3.3.4 Net-snmp en C

Como última opción destacada (ya que descartamos otras opciones investigadas pero no tenidas en cuenta para una posterior valoración) nos encontramos con net-snmp[54].

Tanto Rated, como RFC implementan esta librería para la función de envío y recepción de PDU. Luego, el uso de esta librería en una herramienta propia desarrollada con las características deseadas es una opción de mucho peso, con la ventaja sobre ruby-snmp que en este caso podemos usar el lenguaje de programación C.

Net-snmp [12] nace en 1992 en la Universidad Estadounidense de Carneige-Melon (Pittsburgh). Y menciono éste dato para indicar que esta librería está ampliamente probada. Desde su nacimiento ha incorporado progresivamente las diferentes versiones de SNMP y actualmente la herramienta se encuentra en estado estable ofreciendo monitorización a todas las versions SNMP (ruby-snmp no soportaba la version 3). El equipo de desarrollo se encuentra desarrollando diversas funciones como APIS para otros lenguajes, ampliando funcionalidad de agentes, manejo de traps ampliado… En su web oficial se puede ver la lista de proyectos activos[55].

Usando esta librería contamos con una base sólida para la monitorización. Pero nos encontramos la misma objección que con ruby-snmp, el tiempo usado en la programación completa de la herramienta acortará los objetivos de nuestro proyecto con total seguridad.

[53] Librería ruby-snmp: https://github.com/hallidave/ruby-snmp
[54] Página oficial de net-snmp: http://www.net-snmp.org/
[55] Proyectos activos net-snmp: http://www.net-snmp.org/dev/projects.html

3.4 Análisis y comparativa

Habiendo hecho la selección de herramienta, se dedicó el tiempo necesario para analizar cada herramienta, con el objetivo de analizar todas las ventajas y desventajas antes de volver a cometer el mismo error que con Zabbix.

De esta forma formulamos la siguiente tabla comparativa:

Tabla 4 Análisis de ventajas e inconvenientes de las distintas soluciones

Herramienta	Ventajas	Inconvenientes
Zabbix	o Solución **completa y estable** de monitorización. Herramienta presente en gran número de empresas y con un equipo de desarrollo activo que mantiene la herramienta actualizada. o Incorpora todas las **funcionalidades deseadas**. Hace uso de todas las versiones de SNMP, monitoriza TRAPs, configuración desde fichero JSON… o Herramienta **desarrollada en lenguaje C** con un código ya trabajado debido a los primeros meses de trabajo en la herramienta. Funciones localizadas para proceder inmediatamente a su modificación. o **Documentación completa** y comunidad activa, dando solución a muchos de los problemas encontrados durante su instalación, uso y manipulación.	o La **incorporación de una base de datos local** en el proceso de recopilación y envío de datos por parte del proxy al server, se convierte en un inconveniente importante. La modificación de la herramienta en este aspecto implica que, las siguientes actualizaciones del equipo de Zabbix deberán ser incorporadas a mano, o simplemente no incorporadas. o Imitar desde el manager de RedBorder toda la **comunicación del Zabbix server con el proxy**. Esto pasa por imitar todas la funcionalidades descritas en el capítulo 3.2.2 o Un inconveniente sobre el que habría que trabajar, pero de mucha menos importancia que los anteriores, es el de la **modificación de la salida**. Debemos modificar la salida de datos para usar el protocolo Apache Kafka, como se nos demanda desde ENEO Tecnología.
RFC	o Cumple la característica fundamental que necesitamos en la monitorización. **Asíncronismo en el envío de petición-respuesta**. o Esta herramienta nos proprorciona un **bloque de código muy sencillo** que se encarga de la monitorización. Esto es un buen inicio para incorporar las funcionalidades extra deseadas	o El **lenguaje de programación Perl** es un lenguaje desconocido para mí como desarrollador. Esto es un inconveniente a considerer ya que debemos tener en cuenta el tiempo necesario para la familiarización con este. o No incorpora una de las características fundamentales que buscamos con la herramienta, que sea eficiente en el uso de recursos siendo programada para ser **multi hilo**.

		○ Las **características adicionales** como la de la recepción de la configuración en formato JSON, o la salida de datos usando el protocol Apache Kafka deberás ser implementadas.
Rated	○ Su presencia en el proyecto RTG nos indica que es una **herramienta solida y probada**. ○ Cumple con las características fundamentales. **Asíncronismo** en el envío de petición y recepción de la respuesta y eficiencia en el uso de recursos hacienda uso de la **programación multi hilo** ○ **Equipo de desarrollo en activo** incorporando progresivamente nuevas funcionalidades y añadiendo nuevos parches y actualizaciones. ○ Herramienta **desarrollada en C**	○ A pesar del parecido con Zabbix, esta herramienta está pensada no para funcionar de manera distribuida de la fromar server-proxy, sino para trabajar y almacenar los datos de manera local. Es indispensable en esta herramienta el uso de una **base de datos local** para almacenar los datos. ○ Debemos trabajar el **formato de salida** para que se envíen los resultados de la monitorización a través de Apache Kafka. ○ Otro punto de desarrollo es el de la **recepción de la configuración** por parte del server. Rated ya incorpora la lectura y parseado de la configuración, y en caso de ser necesario incorpora una recarga de esta ante la señal de SIGHUP
Ruby-snmp	○ Librería que nos permite el envío de PDUs de manera **asíncrona** de una manera sencilla ○ Esta librería en Ruby, gracias a la **simplicidad** de este lenguaje, incorpora un punto de sencillez en la creación y envío de PDUs superior a lo que ofrecen otras librerías en otros lenguajes	○ Al igual que Perl incorpora una desventaja cuando hablamos de RFC, Ruby es también un **lenguaje desconocido.** Esto implica un tiempo de familiarización ○ El uso de una librería implica que la herramienta ha de ser **desarrollada desde cero**. No es del todo una desventaja ya que podemos adaptarnos perfectamente al escenario, aunque requiera de más tiempo de desarrollo
Net-snmp	○ **Librería ampliamente usada**. La más popular para implementar el protocol SNMP en todas sus versiones. Ampliamente probada y presente en multitud de programas (Rated o RFC por ejemplo) ○ Nos permite de una manera relativamente sencilla llevar un **asincronismo en el envío-recepción** de PDU. Perfectamente	○ Al igual que con ruby-snmp, el uso de una librería implica **desarrollar la herramienta desde cero**, y consecuentemente, disminuir el alcance del proyecto

[redacted]	documentado y con multitude de ejemplos[56] ○ **Librería en C**. Supone una ventaja respecto a la familiaridad del lenguaje y la eficiencia.	

En este punto, y estudiando cada ventaja y cada inconveniente que vemos en la tabla, dispusimos de una reunión "final". Debiamos de decidir qué herramienta usar para el alcance de los objetivos, teniendo siempre en en mente los tiempo que implicaba el trabajo con cada solución.

3.5 Toma de decision

Después del tiempo invertido en Zabbix, decidimos que no era una buena opción seguir trabajando con la herramienta. El formato de salida podíamos llegar a cambiarlo, pudiendo pasar el código de esta modificación a los desarrolladores de Zabbix, para que fuera incluido en el programa y contara con soporte para las próximas actualizaciones. Podíamos trabajar para que el manager sustituyera a Zabbix server, implementando las funcionalidades de conectividad, ya que no supondría un gran esfuerzo en comparación con el que habría que hacer para el resto de herramientas.

El problema principal del uso de Zabbix venía dado por el mantenimiento de la base de datos local. Eliminar esta funcionalidad no era un asunto trivial, además de que, en el caso de que esta dependencia con la base de datos fuese eliminada, no podíamos pasar este código a los desarrolladores de Zabbix, ya que supondría un cambio radical en su herramienta.

Si optabamos por eliminar la base de datos, no contaríamos con las actualizaciones y parches de Zabbix. Es decir, o usabamos la base de datos e incorporabamos un punto de fallo indeseable para ENEO Tecnología, o modificabamos esto con un trabajo adicional importante y añadiamos las vulnerabilidades y fallos futuros de Zabbix, asunto obviamente descartado.

Fig. 31 Net-snmp como solución

Como conclusion, tanto Zabbix como Rated (esta última por el mismo motivo, el uso de una base de datos local) quedaban descartadas para la continuación de este proyecto.

De las soluciones propuestas, solo nos restaba RFC o el uso de una librería y como consecuencia, partir desde cero para el desarrollo de la herramienta. RFC solo nos aportaba una base, la de la monitorización. El uso de Perl y el hecho de tener que desarrollar la mayor parte de la herramienta en el caso de RFC, y el uso de ruby para ruby-snmp, nos hizo coincidir en que la solución más razonable era la del uso de net-snmp.

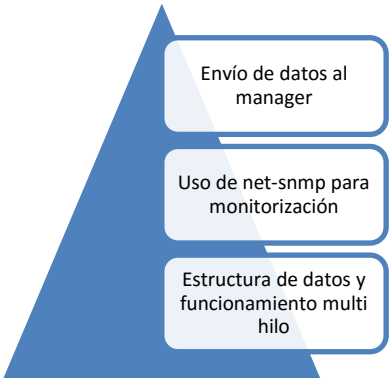


Fig. 32 Bases de la herramienta desarrollada usando net-snmp

Net-snmp, como librería para la monitorización, nos ofrecía una solución estable, ampliamente probada y en desarrollo continuo desde sus inicios en los 90. Es decir, no nos sería dificil la implementación del proceso asíncrono de petición-respuesta.

Además, desarrollar la herramienta en un lenguaje de programación como C suponía una ventaja en cuanto a la familiarización del lenguaje, como a la eficiencia, ya que el uso de memoria compartida y la programación multi hilo se hace mucho más eficiente en este lenguaje.

En el siguiente capítulo comentaremos el planteamiento inicial, la estructura de la herramienta y finalmente, tanto los objetivos alcanzados como los puntos de mejora.

[56] Funcionamiento asíncrono de net-snmp: http://www.net-snmp.org/wiki/index.php/TUT:Simple_Async_Application

4 HERRAMIENTA DESARROLLADA

Primero resuelve el problema. Entonces, escribe el código.

John Johnson

En este capítulo vamos a cubrir todo el proceso de desarrollo de la herramienta. Finalmente decididos por el uso de **net-snmp** como librería, el trabajo a realizar es el de un desarrollo completo y desde cero de una herramienta de monitorización SNMP en red.

Como ya se ha comentado en puntos posteriores, los puntos clave son la eficiencia, la escalabilidad y la integración. En los siguientes capítulos usaremos estos puntos como pilares en torno a los cuales iremos construyendo la herramienta.

Finalmente y debido a nuestra decisión, vamos a comentar cómo los objetivos del proyecto se ven acortados. El hecho de tener que desarrollar la herramienta desde cero, hace que los objetivos de integración inicialmente planteados queden como un planteamiento futuro que se sale del alcance de este proyecto.

4.1 Net-snmp como base para la monitorización

Como ya hemos comentado, la librería net-snmp nos va a servir como base para la monitorización SNMP. El motor principal de la herramienta, siguiendo el pilar de la eficiencia, será el de la realización de peticiones sobre determinados objetos snmp, y su recepción de las respuestas de manera independiente.

Comenzaremos, al igual que en el desarrollo de la herramienta, a detallar desde lo más básico, hasta las estructuras más complejas de nuestro programa.

4.1.1 Funcionamiento asíncrono

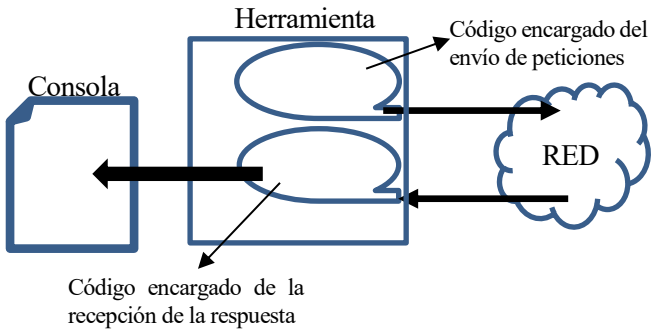


Fig. 33 Esquema de funcionamiento asíncrono

De una manera muy simple, podemos ver el esquema el funcionamiento del motor de monitorización de nuestra herramienta.

Tendremos:

- **Motor de envío de peticiones.** No se quedará a la espera de una respuesta, sino que enviará una petición a todos los objetos a monitorizar, una tras otra, hasta su finalización.
- **Motor de recepción de PDU**. Mientras se ejecuta el motor de envío de peticiones, tendremos un manejador, que se activará ante la llegada de una respuesta y que la procesará de manera independiente.

Teniendo en mente el esquema del funcionamiento asíncrono de la monitorización, vamos a detallar cómo net-snmp nos permite esto. Para el desarrollo de este bloque nos servimos de la documentación proporcionada por el equipo de desarrollo de net-snmp[57] y en particular de uno de sus ejemplos, el de una aplicación simple asíncrona[58].

4.1.1.1 Motor de envío de peticiones

Como ya hemos comentado, vamos a separar la monitorización en dos bloques. Este primer bloque será el encargado de enviar a la red las consultas o peticiones de los objetos que queramos monitorizar.

No es objeto de esta memoria detallar el funcionamiento del protocolo SNMP, por lo que detallaremos cómo hacemos uso de la librería net-snmp para realizar la monitorización.

La librería nos ofrece la libertad de trabajar de manera síncrona o asíncrona con el intercambio petición-respuesta. Evidentemente y siendo uno de nuestros pilares básicos la eficiencia, nuestro motor de envío de peticiones funcionará de manera asíncrona, esto quiere decir, la recepción y procesado de las respuestas a las peticiones enviadas se realizará de manera independiente.

Desde net-snmp, para el envío de peticiones debemos operar de la siguiente forma:

- **Inicio de sesión** con el host cuyos objetos queremos monitorizar
- **Envío de peticiones** de forma asíncrona [13]
- **Cierre de sesión**

[57] Documentación de net-snmp: http://www.net-snmp.org/wiki/
[58] Aplicaciíon asíncrona simple: http://www.net-snmp.org/wiki/index.php/TUT:Simple_Async_Application

A continuación profundizaremos en la librería para detallar cómo nuestra herramienta se basa en net-snmp para la consecución de los objetivos.

4.1.1.1.1 Inicio de sesión

El protocolo SNMP tal cual lo define la **RFC1157** [14][59] ofrece un servicio de petición y respuesta de datos de monitorización a través de la red, usando para ello una definición previa de los datos que son pedidos (a través de un fichero de definición de objetos o MIB). A estos objetos se les asigna una identificación numérica unequívoca (OID).

Aplicación	SNMP
Transporte	TCP
Red	IP

Tabla 5 Pila de protocolos de SNMP

De esta definición original no se extrae una necesidad de una comunicación previa entre equipo que solicita determinado objeto y host objeto de la monitorización. Por esto puede parecer extraño que la librería **net-snmp**, como paso previo al envío de peticiones, requiera de un inicio de sesión.

Esto tiene fácil explicación observando la torre de protocolos (ver **Tabla 5**). La librería ofrece una interfaz en el envío de peticiones, para ello creará la consulta o PDU, y se la pasará a la capa de transporte, la cual si requerirá de una sesión permanente para el intercambio de datos. El objetivo de esta librería es abstraer el detalle de la comunicación, por lo que ofrece un funcionamiento a través de sesiones.

Para mantener localizada toda la información sobre la comunicación entre el equipo que realiza las peticiones y los hosts que las reciben, net-snmp hace uso de estructuras de datos.

Para ello:

```
void snmp_sess_init(struct snmp_session *session)
```

Esta función acepta como parámetro una estructura, que nos servirá tanto para incluir la información necesaria para la conexión con el host a través de la red, como para almacenar infromación necesaria para la obtención de la respuesta.

En la **Tabla 6** vemos cada uno de los campos incluidos en esta estructura. Es fácil ver cómo a través de esta estructura se proporciona a **net-snmp** toda la información de conectividad necesaria, permitiéndonos así abstraernos de la comunicación TCP/IP que soporta al protocol SNMP.

Campo	Descripción
version	Versión del protocolo snmp que usará para las peticiones-repsuestas
localname	IP o nombre de la máquina que realizará las peticiones
peername	IP o nombre del host objetivo de las peticiones
local_port	Puerto de la máquina que realizará las peticiones
remote_port	Puerto del host objetivo de las peticiones
callback	Opcional. Nos permite indicar que función se activara ante la recepción de una respuesta. Usado para el funcionamiento asíncrono
Sessid	ID de sesión. Utilizado para relacionar la respuesta entrante con la sesión que realizó

[59] https://tools.ietf.org/html/rfc1157

	la petición
community	Comunidad SNMP para la conexión

Tabla 6 Descripión de algunos campos de la estructura de sesión de net-snmp

A modo de resumen, con cada host objeto de monitorización deberemos abrir una sesión. Usaremos la función **snmp_sess_init** asociándole una estructura **snmp_session,** la cual nos proprocionará una fuente de entrada y salida y una persistencia de datos para dicha sesión.

4.1.1.1.2 Envío de peticiones

Como ya hemos comentado, se hace una necesidad que el motor de envío de peticiones no se bloquee a la espera de la respuesta, esto es, su funcionamiento debe de ser asíncrono.

¿Cómo le damos solución desde net-snmp?

En la estructura de persistencia de datos de la sesión, **snmp_session**, aparece un parámetro antes mencionado, el parámetro de **callback**, el cual nos permite indicar una función para dicha sesión, que será ejecutada ante la recepción de una respuesta a una petición previa de dicha sesión. El concepto de "recepción de la respuesta" no es trivial y en posteriores capítulos veremos cómo se detecta esta recepción.

En el motor de peticiones solo nos ocuparemos de añadir correctamente una función de callback a las sesiones.

Una vez iniciada la sesión, podemos pasar a la creación y envío de PDUs. El motor de envío de peticiones lanzará a la red de manera periódica un conjunto de PDUs a modo de petición de los objetos que queremos monitorizar.

Para la creación de la PDU nos servimos de

```
struct snmp_pdu * snmp_pdu_create ( int command )
```

Esta función nos permité crear una PDU. Desde el protocolo SNMP se definen cinco tipos de PDU[60]:

- GetRequest-PDU
- GetNextRequest-PDU
- GetResponse-PDU
- SetRequest-PDU
- Trap-PDU

No es objeto de esta memoria profundizar y detallar el protocolo SNMP, luego no entraremos a detallar cada PDU. Nos basta saber que **net-snmp** nos proporciona con **snmp_pdu_create** un método genérico de creación de PDU, proporcionándonos para ello su único parámetro, un entero por el cuál, a través de las macros definidas, podremos indicar qué tipo de PDU deseamos crear.

En el motor de envío de peticiones, nuestro objetivo es generar las PDU de peticiones, es decir, **GetRequest-PDU**. En el uso de la función usaremos como parámetro la macro **SNMP_MSG_GET**.

Adicionalmente a la creación de la PDU de petición, necesitamos "rellenar" dicha PDU con los datos concretos de la petición. Para ello:

```
void snmp_add_null_var ( struct snmp_pdu * PDU, oid * OID, int OIDLen )
```

Esta función añade a la PDU una variable NULL con el OID especificado. No hay que interpretar que dicha

[60] RFC1157. Página 15. Apartado 4 Protocol Specification. https://www.ietf.org/rfc/rfc1157.txt

variable NULL tiene el mismo significado que el valor nulo en la mayoría de lenguajes. En SNMP, se define como variable NULL a un conjunto formado por OID-NULL. Cuando la respuesta llegue a sus destino, creará una PDU de respuesta con dicha variable NULL pero sustituyendo dicho campo por el valor apuntado por el OID.

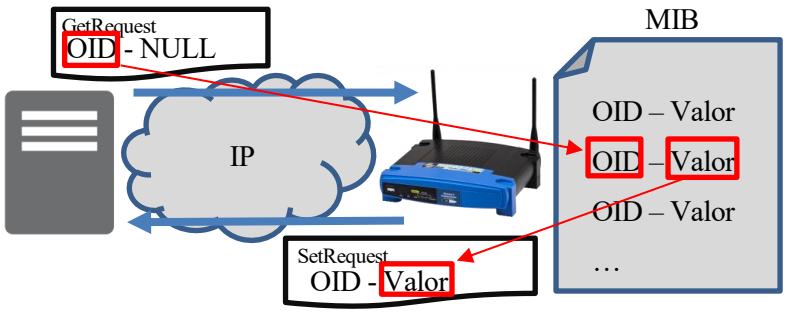


Fig. 34 Variables NULL en el proceso petición-respuesta

Para incluir el OID del objeto cuyo valor queremos solicitar, contamos con los dos últimos parámetros, en el cual indicamos OID y el tamaño del mismo.

Formada la PDU sólo resta su envío. Para ello:

int snmp_send (struct snmp_session *session, struct snmp_pdu *pdu)

Una vez correctamente incializada la sesión y en cuya estructura encontramos todos los datos necesarios para la conexión, y una vez creamos la PDU de petición, incluyendo el OID del objeto cuyo valor saolicitamos, a través de la función de send enviamos la PDU a la red.

El motor de peticiones básicamente estará compuesto por una fase de apertura de sesiones, un bucle de lanzado de peticiones y al cierre de la herramienta, un proceso encargado del cierre de sesión y liberación de la memoria que describiremos a continuación.

Este proceso de creación de PDU y envío de esta a la red se realizará de manera continua. Posteriormente veremos la estructura completa del programa ya que este motor de envío de peticiones deberá ser ejecutado en un numero concreto de hilos, debiendo así hacer un reparto de carga de monitorización entre cada motor de envío.

Adicionalmente y como consecuencia del funcionamiento periódico e indefinido de este motor, debemos de dotarlo de una capacidad de realizar una parada segura (segura en el sentido del uso de recursos, liberando correctamente todo lo reservado) a petición del usuario. También será detallado en otros capítulos.

4.1.1.1.3 Cierre de sesión

Debemos tener en cuenta que la sesión es un bloque de memoria reservado para la persistencia de los datos necesarios para la comunicación. Por ello, net-snmp no provee de una interfaz para su liberación:

int snmp_close (struct snmp_session *session)

Además, debemos liberar los bloques de memoria que hemos reservados para almacenar las PDU hasta que son enviadas. Al igual que con los datos de la sesión, la librería nos proporciona:

void snmp_free_pdu (struct snmp_pdu *pdu)

Como hemos comentado en el capítulo anterior, el motor de envío, que será ejecutado en varios hilos, estará provisto de un mecanismo de parada segura que esperará la finalización del bucle de monitorización, cerrará las sesiones y liberará toda memoria que haya sido reservada.

4.1.1.1.4 Resumen

Como visión global de éste apartado (**4.1.1.1 Motor de envío de peticiones**) nos debe de quedar lo siguiente.

El bloque principal de monitorización es lo que denominamos **motor de envío de peticiones**. Este tomará una información de monitorización (esto es la información de conectividad con un host junto que los objetos u OID que se desean monitorizar), y creará una PDU que enviará a la red para cada OID.

Como el funcionamiento de nuestro herramienta va a ser multihilo, ejecutaremos más de un motor de envío de peticiones, en concreto, uno en cada hilo. Se quedan algunas cuestiones en el aire que resolveremos en capítulos posteriores. Estas son:

- Cómo se almacenará la información para el envío de peticiones
- Cómo nuestra herramienta recibirá esta información por parte de RedBorder
- Cómo se hará el reparto de esta información entre cada motor (o hilo)
- Cómo se actualize la información de monitorización durante la ejecución de la herramienta

Manteniendo estas consultas en el aire, pasamos a describer el segundo element clave en la monitorización propiamente dicha. El motor de recepción de PDU de respuesta.

4.1.1.2 Motor de recepción de PDU

Una vez detallado el motor de envío de peticiones, el cuál se encargará de tomar una serie de hosts y una serie de objetos de cada host y enviar de manera periódica **GetRequest-PDU** para cada uno de dichos objetos, nuestra siguiente ocupación pasa al motor de recepción.

Una vez cada host objetivo reciba dicha PDU de petición, creará otra de tipo **SetRequest-PDU** rellenando la variable NULL con el valor del objeto que se indica en la petición previa (ver **Fig. 34 Variables NULL en el proceso petición-respuesta**).

Nuestro objetivo fundamental es que el funcionamiento de estos dos motores sea asíncrono, funcionalidad cubierta por net-snmp y mostrada como una aplicación simple en su documentación[61]. Podemos ver que el funcionamiento esquematizado es el mostrado en la **Fig. 35**.

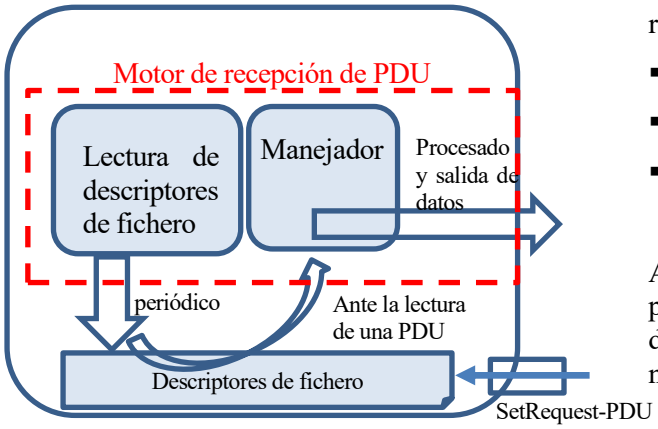


Fig. 35 Esquema del funcionamiento del motor de recepción de PDU

Podemos establecer tres procesos en el motor de recepción. Estos serían:

- Lectura de descriptores de ficheros
- Llamada al manejador de cada sesión
- Procesado de los datos de cada PDU

A continuación entraremos en detalle con cada proceso y en concreto se detallará por qué la lectura de los descriptores de fichero se hace un proceso necesario.

[61] Aplicación simple de monitorización asíncrona http://www.net-snmp.org/wiki/index.php/TUT:Simple_Async_Application

4.1.1.2.1 Lectura de los descriptores de fichero

La monitorización de los descriptores de ficheros es la forma que tienen los programas de esperar a que un descriptor se encuentre listo por la ocurrencia de algún determinado evento realcionado con las operaciones de entrada/salida.

Por tanto, nuestro motor de recpción deberá realizar una monitorización de estos descriptores, esperando que se reciba una PDU. Dicha PDU será la respuesta a una petición previa de una de las sesiones **snmp** activas. Cuando una PDU sea detectada en los descriptores de ficheros se llamará al manejador definido en la apertura de la sesión.

Desde net-snmp el procedimiento es el siguiente:

Desde un hilo independiente al resto de la ejecución de la herramienta (incluido el motor de envío de peticiones), tendremos un bucle encargado de la monitorización de los descriptores. Durante este bucle, ante la activación de alguno de los descriptores, intentaremos leer una **SetRequest-PDU**. En caso de que la lectura haya sido correcta se llamará al manejador, definido previamente en la sesión, que procesará los datos de dicha PDU.

Desde el lenguaje de programación C contamos con la siguiente librería:

<sys/select.h>

Esta nos proporciona una abstracción que nos permiten realizar la monitorización del descriptor de fichero.

typedef struct fd_set {}[62]

Dicha estructura nos permite agrupar un conjunto de sockets, con el objetivo de usarlo en posteriores funciones que nos permitan detectar ciertos eventos en dichos sockets. Usaremos esta estructura para elegir los sockets por los que esperamos la recpeción de las **SetRequest-PDU**.

void FD_ZERO(fd_set *set);

La librería nos proporciona 4 funciones macros para manipular las estructuras de sockets **fd_set**. Entre ellas encontramos esta, que nos permite limpiar o poner a cero el set (o conjunto de descriptores). Usaremos esta macro para "limpiar" la estructura antes de que, a través de net-snmp, podamos rellenarla con los sockets concretos a monitorizar a la espera de una SetRequest-PDU.

int snmp_select_info (int *numfds, fd_set *fdset, struct timeval *timeout, int *block)

Una vez a cero la estructura **fd_set,** usaremos net-snmp para introducir los datos de los sockets concretos necesarios para la detección de la recepción de una PDU. No profundizamos sobre estos datos concretos ya que net-snmp nos permite, gracias a esta función, una abstracción.

Esta función únicamente nos permite rellenar la estructura **fd_set** con la información completa para SNMP, es en cambio la función **select** proporcionada por **<sys/select.h>** la que nos permitirá obtener los datos concretos de los sockets que han sido activados.

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

[62] Descripción completa: https://msdn.microsoft.com/es-es/library/windows/desktop/ms737873(v=vs.85).aspx

Como hemos comentado, esta función será la que nos devuelva la información concreta de los sockets, para que mediante una función dada por net-snmp, llamada **snmp_read**, podamos realizar la lectura de la PDU entrante,

void snmp_read (fd_set *fdset)[63]

Por último será esta la función que realizará la lectura de la PDU y activará el manejador de la sesión, que se encargará del procesado de los datos.

Como hemos visto, **net-snmp** y **select.h** nos porporciona un conjunto de funciones para realizar una monitorización del descriptor de fichero en busca de PDU de respuesta entrantes. A continuación veremos una simplificación de código del funcionamiento principal de la monitorización del descriptor de ficheros por parte del motor de recepción de PDU.

```
while() {
  int fds = 0, block = 1;
  fd_set fdset;
  struct timeval timeout;
  FD_ZERO(&fdset);
  snmp_select_info(&fds,&fdset,&timeout, &block);
  fds=select(fds,&fdset,NULL,NULL, block?NULL: &timeout);
  if (fds)
    snmp_read(&fdset);
  else
    snmp_timeout();
}
```

Code 4 Bloque básico de monitorización del descriptor de fichero

En este bloque básico de código, el cuál es una simplificación de lo que finalmente se incluye en el motor de recepción de PDU, podemos ver la actuación conjunta de **net-snmp** y **select.h** para monitorizar el descriptor de fichero en busca de una PDU entrante.

Una vez se produzca la lectura de una PDU mediante **snmp_read** se llamará de forma automática al manejador de la sesión definido en la estructura de la misma. Este es el siguiente proceso del motor de recepción y lo veremos en el siguiente apartado

4.1.1.2.2 Manejador de la sesión

Este punto ha sido visto previamente en el apartado **4.1.1.1.1. Apertura de sesión.**

En la sesión que abrimos en el motor de envío de peticiones, a través de la estructura usada, podemos indicar mediante el parámetro callback, la función que queremos ejecutar ante la recepción de una PDU.

Para la recepción de una PDU se hace como paso indispensable la monitrorización de los descriptores de fichero concretos, tal como hemos visto en el capítulo anterior **4.1.1.2.2 Lectura de los descriptores de fichero.**

[63] Detallado del proceso compelto de monitorización del descriptor de ficheros con net-snmp http://linux.die.net/man/3/snmp_close

En concreto, en la estructura de la sesión tenemos:

callback	A través de este parámetro indicamos qué función se debe ejecutar para los paquetes SNMP entrantes
callback_magic	Indicamos qué parámetro deseamos pasar como argumento a la función anteriormente definida

Tabla 7 Parámetros en snmp_session para definir la función de callback

En nuestra herramienta, crearemos una función genérica para la extracción y procesado de los datos incluidos como respuesta en la **SetRequest-PDU**. Esta función será la que llamamos como **procesa_pdu** y es el objeto de descripción en el siguiente capítulo.

4.1.1.2.3 Extracción y procesado de datos

Sabemos que en SNMP existen muchos tipos de datos[64], además que este protocolo nos permite la flexibilidad de crear en nuestras MIBs tipos de datos compuestos por los primitivos definidos en la norma.

Podríamos pensar que cada sesión podría tener definido un manejador en función del tipo de datos que desea recibir, pero se entiende que no es una solución viable. Un host tiene definido en sus mibs una variedad de tipos de datos que podemos pedir, luego, se hace una necesidad que la función manejadora sea genérica e independiente al tipo de dato que va a recibir.

```
/*
 * Funcion: int procesa_pdu(int operation, struct snmp_session *sp,
 *  int reqid, struct snmp_pdu *pdu, void *magic)
 *
 *      Funcion que se definirá de callback en las sesiones snmp.
 *      Se llamara cada vez que se recibe una trama pdu.
 *      SU funcion principal sera enviar por kafka los
 *      datos de la PDU recibida
 *
 *      Parametros callback:
 *      1)int operation             --> the possible operations are RECEIVED_MESSAGE and TIMED_OUT
 *      2)struct snmp_session* session  --> the session that was authenticated using community
 *      3)int reqid                 --> the request ID identifying the transaction within this session. Use 0 for traps
 *      4)struct snmp_pdu* pdu          --> a pointer to PDU information. You must copy the information because it will be freed elsewh
 *      5)void* magic                   --> a pointer to the data for callback()
 */
int procesa_pdu(int operation, struct snmp_session *sp, int reqid, struct snmp_pdu *pdu, void *magic);
```

Fig. 36 Función manejadora de la sesión

El manejador es la función que será ejecutada ante el evento de recepción de PDU. Como hemos dicho, esta función es genérica, luego el primer paso es relacionar la PDU recibida con la consulta previa. Posteriormente se comentará cómo se almacenará la información de hosts y OIDs y veremos cómo cada uno de estos está identificado por un entero, que sirve de identificación unequívoca de cada elemento.

Como vemos en la **Fig. 36**, dentro de la función incluimos el parámetro **void* magic**, el cual, si recordamos la **Tabla 7** apunta hacia el parámetro que declaramos en la estructura de inicio de sesión. Incluimos este parámetro porque es indispensable para que nuestra función sea correctamente identificada como manejador por parete de net-snmp. En nuestra herramienta no le daremos uso, ya que toda la información necesaria será extraída de la PDU entrante, recibida en el parámetro **pdu.**

[64] Tipos de datos en SNMP: https://www.webnms.com/cagent/help/technology_used/c_snmp_overview.html

En posteriores capítulos detallaremos el procesado de la PDU, ya que necesitamos previamente conocer la estructura de datos interna de la aplicación y cómo procedemos con la comunicación con el manager mediante el protocolo Apache Kafka.

4.1.1.2.4 Resumen

Sin entrar en detalles sobre cómo nuestra herramienta envía los datos de la monitorización a RedBorder o cómo se almacenan los datos de monitorización para que, una vez llegue una PDU de respuesta, podamos identificarla con una consulta previa, el funcionamiento del motor de recepción de PDU queda bien detallado.

La monitorización de los descriptores de ficheros hace que la función definida como **callback** en la estructura de inicio de sesión, se ejecute ante un evento de recepción de PDU en uno de los sockets para SNMP. Es en esta función de callback donde se extraerán los datos concretos del resultado de la monitorización y se etiquetarán y enviarán a RedBorder.

4.1.1.3 Resumen del funcionamiento asíncrono

A modo de resumen el siguiente esquema ilustra el funcionamiento asíncrono dentro de la herramienta del motor de envío de peticiones y el motor de recepción.

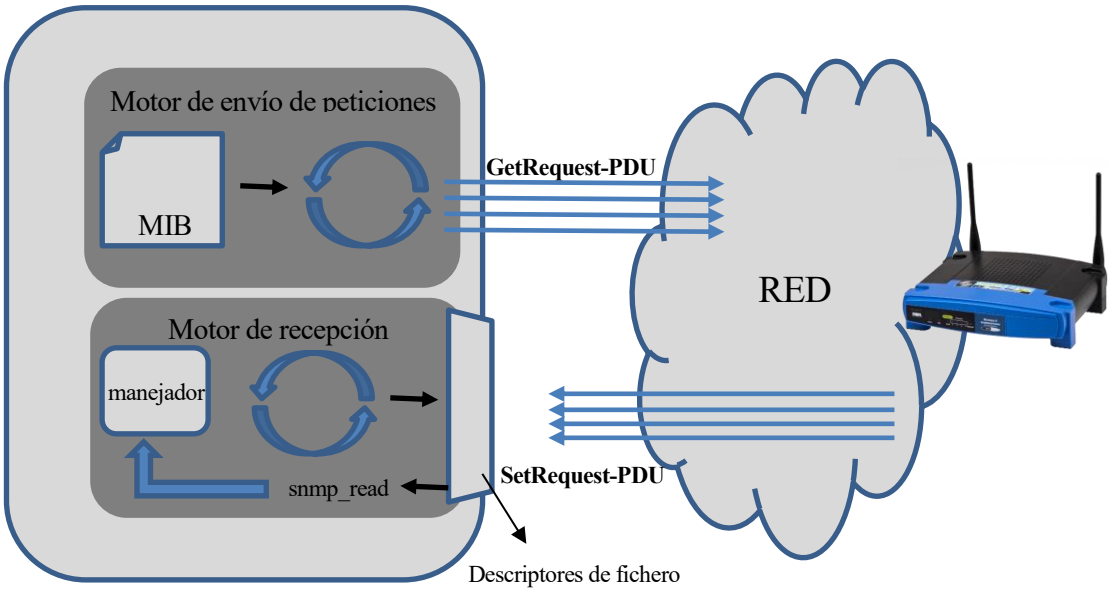


Fig. 37 Resumen del funcionamiento asíncrono

Vemos pues cómo ambos motores trabajan de manera independiente, separando el envío de peticiones de la recepción, extracción y procesado de datos de la respuesta. Quedan aún muchos aspectos que detallar, los cuales hacen de esta, una herramienta eficiente y completamente integrada con RedBorder.

4.2 Intercambio de datos con RedBorder

Ya comentamos en puntos posteriores que la adaptación con el sitema de RedBorder y con el funcionamiento de todas sus herramientas, era un pilar fundamental en el desarrollo de este proyecto. Adicionalmente, los problemas fundamentales que encontramos en las herramientas propuestas, y que finalmente fueron descartadas, estaban relacionados con problemas de adaptación.

Pero, **¿qué significa adaptación?**

Nuestro proyecto, y en concreto nuestra herramienta, está pensado para dar solución a un objetivo concreto dentro de un proyecto mucho mayor, que es el de la monitorización, análisis y actuación en una red grande mediante la solución desarrollada por ENEO Tecnología llamada RedBorder.

Por lo tanto, uno de nuestros objetivos es la integración de nuestra herramienta en el modo de trabajo de RedBorder. A modo de sensor, nuestra aplicación será controlada desde el manager central, actuará recopilando los datos snmp que se solicitan, y enviará estos de vuelta al manager en un formato específico.

Nuestra herramienta realizará una monitorización de la red de forma autónoma, en concreto mediante un funcionamiento asíncrono entre envío de peticiones y recepción de respuestas como hemos visto en el capítulo anterior. Par ello se require un flujo de información con el manager, que podemos identificar fundamentalmente como:

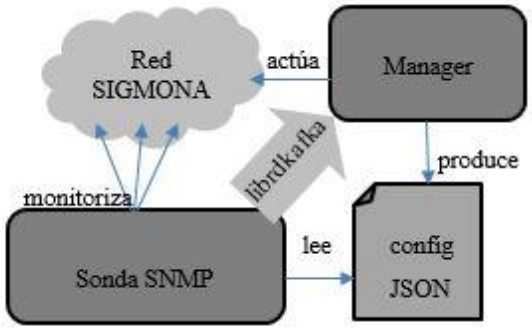


Fig. 38 Esquema de relaciones en el proyecto

- **Parámetros de configuración**

  El manager central de RedBorder debe de ser capaz de determinar ciertos parámetros de configuración de la herramienta. A continuación detallaremos qué parámetros concretos serán controlados desde el manager, como por ejemplo el número de hilos con los que se ejecutará la herramienta

- **Objetos a monitorizar**

  El flujo de comunicación debe ser constante cuando hablamos de los objetos a monitorizar. Nuestra herramienta debe de estar en constante comunicación con el manager, el cuál irá indicando los objetos que solicita. RedBorder implementa un frontend para la interacción con el usuario, el cuál puede actuar en cualquier momento para añadir datos sobre determinados objetos o eliminar objetos que ya no desea monitorizar. Este funcionamiento será detallado a continuación.

- **Salida de datos**

  Al igual que desde el manager se indican qué objetos monitorizar, nuestra herramienta debe de ser capaz de comunicar a éste el resultado de dicha monitorización. RedBorder funciona con un manager central que controla un conjunto de sensores. El número de sensores que envía datos al manager puede ser muy elevado en redes de gran extensión, luego desde ENEO Tecnología usan Apache Kafka como protocolo de mensajería de alto rendimiento, lo cual permite al manager una mayor organización en la lectura y procesado de los datos de entrada.

A continuación pasaremos a detallar cada una de las salidas y entradas de datos de nuestra herramienta.

4.2.1 Funcionamiento con jansson.h

Vamos a detallar el conjunto de funciones que nos ofrece esta librería [15] para extraer los datos de un archivo JSON. En nuestro fichero de configuración vamos a encontrar tres tipos de datos:

- JSON OBJECT
- STRING
- INTEGER

Existen más tipos definido, pero sólo nos serán útiles estos tres.

4.2.1.1 Formato JSON

El formato de un fichero JSON, será el de un objeto principal, que contendrá una lista de lementos en su interior. Vamos a servirnos de un ejemplo de un fichero de monitorización usado en nuestra herramienta:


```
[
{
    "sensor_id": 1,
    "sensor_name": "prueba",
    "sensor_ip": "195.218.195.228",
    "community" : "variation/virtualtable",
    "snmp_version": 2,
    "modif": 1,
    "monitors":
    [
        {
            "oid_id": 1,
            "name": "paqEnt",
            "oid": "IF-MIB::ifInOctets.1",
            "modif": 1,
            "t_monit": 2
        },
        {
            "oid_id": 2,
            "name": "paqUcast",
            "oid": "IF-MIB::ifInUcastPkts.1",
            "modif": 1,
            "t_monit": 2
        }
    ]
},
{
    "sensor_id": 2,
    "sensor_name": "prueba2",
    "sensor_ip": "195.218.195.228",
    "community" : "variation/virtualtable",
    "snmp_version": 2,
    "modif": 1,
    "monitors":
    [
        {
            "oid_id": 3,
            "name": "paqEnt",
            "oid": "IF-MIB::ifInOctets.1",
            "modif": 1,
            "t_monit": 2
        },
        {
            "oid_id": 4,
            "name": "paqUcast",
```

Code 5 Ejemplo de formato JSON

Como vemos en **Code 5**, identificamos una estructura clara, un fichero JSON es una lista de objetos.

Dentro de cada objeto encontraremos definidos una serie de parámetros, los cuales pueden ser de cualquiera de los tipos antes mencionados. Como vemos en el ejemplo, podremos encontrar INTEGER como "**sensor_id**", STRING como "**sensor_ip**" u otra lista de objetos como en el caso de "**monitors**".

En los siguientes capítulos entraremos en detalle con el contenido de cada fichero de configuración usado por esta herramienta.

A continuación veremos la interfaz que nos ofrece **jansson.h** para la extracción de estos datos y conversión a un tipo de dato manejable para el lenguaje de programación C.

4.2.1.2 Extracción de datos con jansson.h

Esta librería nos proprociona una interfaz sencilla para la extracción de datos. Como es evidente, no puede ser el mismo procedimiento el que extraiga un tipo de datos u otro, es por esto por lo que el formato del fichero JSON debe estar bien definido, y un error en este debe de ser tratado convenientemente.

json_load_file()

Esta función cargará en memoria el fichero que queremos procesar. Se exige como paso previo a la extracción de datos. Entre otros parámetros, acepta la ruta donde se encuentra dicho fichero. En nuestra herramienta, los ficheros de configuración serán pasados como argumentos en la ejecución.

Antes de seguir con la extracción de datos, es importante observar cómo se manejan los errores en este fichero. En este primer caso, la ocurrencia de un error en la carga del fichero, detendría y finalizaría el proceso de parseo.

Se observa en **Code 6** que la detección de un fallo en **sensores**, provoca una entrada en la condición de fallo, que imprimirá en el correspondiente log un mensaje de error y finalizará la función de parseo.

```
/* parse text into JSON structure */
sensores = json_load_file(rutaconf,0,&error);

fprintf(stdout,"\n\n----------Proceso de parseo de monitorizacion:\n");

if (!sensores) {
    LOG_PRINT("Error al abrir fichero de monitorizacion");
    LOG_PRINT("error: on line %d:",error.line);
    LOG_PRINT("error: %s",error.text);
    error_estdin = 1;
}
```

Code 6 Manejo de error en la carga del fichero JSON

¿Siempre finalizará esta función ante la ocurrencia de un error?

No siempre. A lo largo del fichero procuraremos solventar los errores. Esto significa que un error de procesado en un objeto, hará que la herramienta deje de procesar este objeto y pase al siguiente, avisando con un mensaje en el log del objeto concreto (identificado numéricamente) que no ha podidod ser parseado correctamente.

Una vez cargado el fichero, y como ya hemos comentado, debemos extraer la lista d eobjetos principal que compone el fichero. Para ello tendremos un bucle **for** que recorrerá todos los objetos de esta lista.

json_array_size()

Esta función nos servirá para establecer los límites del bucle **for** mencionado. Nos devuelve el número de elementos de la lista que le pasemos como argumento. Siguiendo con el ejemplo anterior:

```
/*
 * Bucle que recorre cada array/host del fichero
 * Cada iteracción es un host
 */
fprintf(stdout,"Numero de objetos hosts: %d\n",json_array_size(sensores));
for(i = 0; (i < json_array_size(sensores))&&(error_for); i++) {
```

Code 7 Bucle para la lista principal del fichero JSON

Dentro de este bucle, obtenedremos cada elemento de la lista de la siguiente forma:

json_array_get (lista, elemento)

Usando esta función, y usando el índice del bucle for, extraeremos en cada iteracción un elemento de esta lista. En nuestro caso, cada elemento de la lista estará compuesto de enteros, cadenas e incluso otras listas. Vamos a detallar las funciones de extracción. No se hace necesario detallar la estructura completa de la función de parseo, ya que usando las funciones aquí descritas podremos ir recorriendo la estructura JSON.

json_object_get (objeto, parámetro)

Esta función extrae un parámetro concreto del objeto JSON. Si recordamos ele ejemplo de **Code 5**, podemos servirnos del siguiente código para extraer los diferentes elementos que componen cada objeto de la lista:

```
sensor_id = json_object_get(sensor,"sensor_id");
sensor_name= json_object_get(sensor,"sensor_name");
sensor_ip= json_object_get(sensor,"sensor_ip");
sensor_community= json_object_get(sensor,"community");
sensor_snmp_version= json_object_get(sensor,"snmp_version");
sensor_modif = json_object_get(sensor,"modif");
```

Code 8 Extracción de parámetros de un objeto

```
json_t *sensores;
json_t *sensor;
json_t *sensor_id,*sensor_name,*sensor_ip,*sensor_community;
json_t *sensor_snmp_version, *sensor_modif;
```

Code 9 Variables para almacenar objetos JSON

Una vez tenemos en memoria estos objetos JSON, nuestro objetivo será comprobar que dichos objetos están definidos en el fichero de la forma esperada. Es decir, si establecemos que el parámetro **sensor_id** debe de ser un entero, antes de proseguir con la extracción de datos, deberemos de realizar la comprobación de que el parámetro obtenido con **json_object_get** es un entero.

Las funciones

json_is_string () , json_is_integer() y json_is_array()

nos servirán para la detección de errores en el formato esperado del fichero de configuración.

Finalmente, la obtención de estos datos en el tipo específico del lenguaje de programación C se realizará usando las siguientes funciones:

json_string_value () y json_integer_value ()

Siguiendo el ejemplo usado hasta ahora:

```
//¿Existe? --> Lo buscamos por id
id_sensor = json_integer_value(sensor_id);
nodo_encontrado = busca_nodo(id_sensor);

//Obtenemos resto de cabecera
nombre_sensor = json_string_value(sensor_name);
ip_sensor = json_string_value(sensor_ip);
comunidad_sensor = json_string_value(sensor_community);
version_sensor = json_integer_value(sensor_snmp_version);
```

Code 10 Extracción de parámetros a enteros o cadenas

Hemos definido las funciones principales de la extracción de datos. El parseo de los ficheros de configuración es uno de los puntos clave de la herramienta, contando con un fichero de funciones para ello de más de 500 líneas. Para ver cómo se trabaja directamente con estas funciones, siendo cuidadosos en la detección de errores, ver **Anexo 2: Ficheros de configuración y estructuras de datos**.

4.2.2 Recepción de parámetros de configuración general

Determinados parámetros de configuración deben ser aportados desde el manager de RedBorder. Parámetros relacionados con el funcionamiento de la herramienta y no con los detalles concretos de la monitorización. Para ello, desde nuestro poller SNMP, proporcionamos dicha funcionalidad.

El funcionamiento general de los sensores que se integran con RedBorder es que los parámetros de configuración sean recibidos por parte del manager en formato JSON, luego, a pesar de no ser una necesidad de primer orden, en pro de la integración de la herramienta, procedemos de esta forma.

Parámetros

En concreto, desde el manager de RedBorder se van a recibir los siguientes parámetros:

- **Threads**

  Hilos para el motor de envío de peticiones. Damos de esta forma libertad al manager para que controle el uso de recursos que desea. La decision será parte del manager y no forma parte de los objetivos de nuestro proyecto, pero dependerá de la máquina donde se desea ejecutar la herramienta y de la red a monitorizar. Se lanzarán tantos hilos como se indiquen en este parámetro, repartiendo la carga de monitorización de manera equitativa entre cada hilo. Como hemos visto, cada hilo ejecutará el motor de envío de peticiones, apareciendo así un punto de vital importancia, el del reparto equitativo de la carga de monitorización. Se verá más adelante.

- **Parámetros de Apache Kafka**

  En capítulos posteriores hablaremos de la salida de datos hacia el manager mediante el protocolo Apache Kafka. Para ello es necesario que el manager indique algunos parámetros necesarios para la conectividad. En concreto, en la configuración general se incluirá información para conectar con el broker, **kafka_broker**, además de el topic o tema al que se deberán incluir los paquetes de nuestra herramienta, **kafka_topic**.

- **Ruta de mibs**

  Adicionalmente se indicará desde el manager dónde se han cargado las mibs. Por defecto la herramienta buscará en las rutas por defecto, que son:

  - **/usr/local/share/snmp/mibs**
  - **/usr/local/share/snmp**

  Aun así se da la posibilidad de que, por características individuales de cada sistem, si se han definido nuevas mibs y se desean cargar en una ruta específica, se pueda indicar a nuestra herramienta.

Como vemos, la configuración general de la herramienta es muy reducida, y esto es algo lógico. La mayor parte de la información de configuración está relacionada con la monitorización, y, por motivos que más adelante explicaremos, vendrá dada en un fichero de configuración distinto.

Para que nuestra herramienta pueda entender o parsear este fichero nos serviremos de la librería **jansson.h**[65], que será detallada a continuación.

config.json

```
[
    {
        "threads": 10,
        "kafka_broker": "127.0.0.1:9092",
        "kafka_topic": "rb_monitor",
        "mibs":
        [
            {"ruta": "/usr/local/share/snmp/mibs"},
            {"ruta": "/usr/local/share/snmp"}
        ]
    }
]
```

Fig. 39 Ejemplo de fichero JSON de configuración

4.2.3 Recepción de parámetros de monitorización

Como ya hemos comentado, la decisión tomada respecto a la configuración fue la de separar en dos ficheros estos parámetros, encontrándo el fichero de configuración general antes descrito, y de manera independiente, el fichero que nos ocupa, el de la configuración específica de la monitorización.

¿Por qué dos ficheros de configuración?

Antes de nada, hay que tener en cuenta que hay parámetros de configuración que son estáticos, que se configuran una vez se lance la herramienta y no se modfican. Este es el caso de los parámetros de configuración general antes descritos. El número de hilos puede servir de ejemplo para ilustrar que estos parámetros no van a ser modificados durante la ejecución del programa.

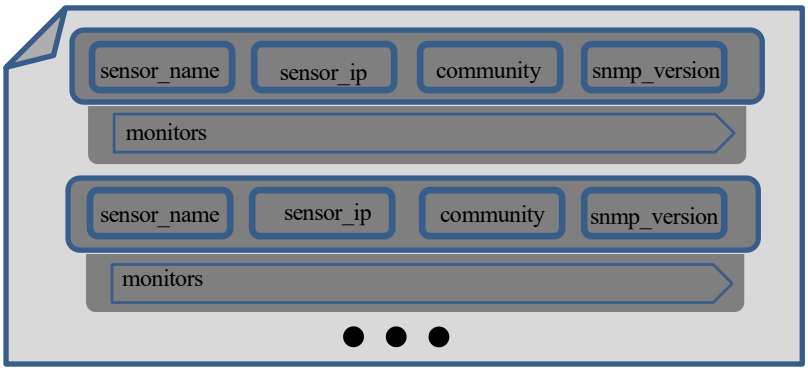
En cambio, los parámetros relacionados con la monitorización van a ser objeto de continuo cambio. RedBorder ofrece un frontend a los usuarios y/o administradores de la red para que puedan modificar, entre otras muchas cosas pero contrándonos en lo que nos ocupa, qué parámetros desea monitorizar, el intervalo de petición de estos, ver gráficas, etc. Es por esto que la lista de hosts y objetos a monitorizar va a estar en continuo cambio.

Nuestra herramienta estará dotada de una funcionalidad que le permitirá recargar la configuración de monitorización durante la ejecución. El manager enviará la configuración de monitorización en formato JSON y avisará mediante una señal a la herramienta para que proceda con la recarga de esta. A continuación en este mismo capítulo detallaremos este proceso.

[65] Página oficial: http://www.digip.org/jansson/

4.2.3.1 Parámetros

Vamos a describir cómo se organiza el fichero con la información de monitorización.



Éste será un fichero en formato JSON, que incluirá una lista de objetos (**JSON Object**). Posteriormente detallaremos la sintaxis JSON y cómo nos servimos de la libresía **jansson** para extraer la información.

Cada objeto de esta lista incluye toda la información necesaria para la conexión con un host.

Fig. 40 Configuración de monitorización

Cada uno de estos objetos contará con los siguientes parámetros:

- **sensor_name**

  Es el identificador del host. Un string que sirve para identificar al host en la comunicación manager-poller. Esta información no es útil para alcanzar al host objetivo en la red.

- **sensor_ip**

  Dirección IP de nuestro host objetivo. Es la información que posteriormente se incluirá en la sesión snmp abierta con cada host (ver **4.1.1.1.1 Inicio de sesión**)

- **community**

  Comunidad SNMP utilizada para acceder a los valores de los OID que solicitamos. La consulta debe incluir la comunidad en la que se encuentra definido el objeto que se solicita

- **snmp_version**

  Versión SNMP usada en la petición-respuesta. En nuestra implementación solo aceptamos version 1 y 2 de SNMP por motivos que adelante discutiremos.

Además de estos parámetros, se incluyen dos parámetros no relacionados con la conectivida pero que son de vital importancia para el funcionamiento interno de la herramienta:

- **sensor_id**

  Identificador numérico para este host dentro de nuestra herramienta. Posteriormente veremos cómo este parámetro es necesario a la hora de gestionar la memoria compartida

- **modif**

  Bandera (0 o 1) que nos indicará si queremos crear, modificar o eliminar este host. Se detallará en capítulos posteriores.

La información relativa a un host no se limita a la de conectividad, sino que dentro de la información de un host debemos de encontrar qué objetos o OIDs concretos queremos monitorizar de dicho host. Luego adicionalmente a estos parámetros de conectividad, en este fichero de monitorización vamos a encontrar toda la información de cada objeto SNMP que queremos monitorizar. Para ellovamos a incluir como ultimo parámetro una lista de objetos. Cada uno de estos nuevos objetos JSON contendrá la información de un OID.

Este ultimo parámetro, denominado **monitors** es por tanto una lista de objetos con la siguiente información:

- **name**

  Al igual que sensor_name, nos sirve para identificar a este objeto en la comunicación entre manager de RedBorder y nuestra herramienta

- **oid**

  Identificador de objeto SNMP. Podremos incluir tanto su string identificador, como su lista de enteros. Para esto, nuestra herramienta tomará el oid y realizará una traducción desde la mib, obteniendo así el oid en forma de dígitos, incluyéndolo de forma correcta en cada GetRequest-PDU.

- **oid_id**

  Identificador numerico para este OID

- **modif**

  Bandera para creación, modificación o eliminación de este OID

- **t_monitor**

  Tiempo deseado entre cada envío de petición a este OID

De esta forma, el manager de RedBorder será capaz de proporcionar a través de éste fichero todos los parámetros relativos a la monitorización. Nuestra herramienta tomará este fichero, parseándolo y almacenándolo en memoria.

```
"monitors":
[
    {
        "oid_id": 1,
        "name": "paqEnt",
        "oid": "IF-MIB::ifInOctets.1",
        "modif": 1,
        "t_monit": 2
    },
```

Fig. 41 Ejemplo de lista monitors

De su parámetro **modif** presente en cada objeto, vemos cómo se proporciona la funcionalidad de que, mediante el envío de nuevos ficheros de configuración, se añade o se eliminen nuevos objetos para la monitorización.

Cabe recordar que este proyecto nace de la necesidad de una herramienta eficiente y que trabajara en redes de gran extension, es por esto que el siguiente apartado detallaremos cómo se crea y se gestiona la memoria que será compartida por todos los hilos del motor de envío de peticiones.

4.2.3.2 Organización en memoria

Una vez recibido el fichero de monitorización, habrá que realizar un parseo para obtener los datos desde JSON (realizado gracias a la librería **jansson,** que será detallada en capitulos posteriores). Teniendo los datos en memoria, debemos estructurarlos de manera eficiente, ya que todos los hilos del motor de envío de peticiones van a tener que acceder a estos datos para poder generar las **GetRequest-PDU**.

A continuación vamos a detallar el funcionamiento de la organización de la memoria.

Almacenado de datos

Para almacenar los datos de la monitorización, vamos a “dibujar” en la memoria la misma estructura que encontramos en el fichero JSON.

Como ya hemos visto, en el fichero encontramos una lista de objetos, donde, cada uno de estos contiene la información relativa a un host. Dentro de estos, encontraremos información de conectividad hacia el host y otra lista de objetos. Estos nuevos objetos contendrán los OIDs de los objetos SNMP que queremos solicitar.

Para organizar esto en memoria, nos vamos a servir de dos estructuras de datos.

La primera es la que se define como **st_host**. Usaremos esta estructura para almacenar la información de cada host. En ella encontramos los siguientes parámetros:

```
typedef struct host {
    int id;
    char * name;
    char * ip;
    char * community;
    int version;

    int corte;
    int fallo_sesion;

    struct snmp_session sesion;
    struct snmp_session *punt_sesion;

    st_oid *oids;

    struct host *next;
}st_host;
```

Code 11 Estructura de host

id	Identificador numérico del host
name	Identificador de host, obtenido de sensor_name en el fichero de monitorización.
ip	Dirección IP del host, obtenido de **sensor_ip** en el fichero de monitorización
community	Comunidad SNMP usada para las peticiones, obtenido de **community**
version	Versión del protocolo SNMP usada con el host para las peticiones-respuestas, obtenido de **snmp_version**
corte	Variable auxiliar que usaremos para el reparto de hosts entre los diferentes hilos del motor de envío de peticiones. Se detallará en capítulos posteriores
fallo_sesion	Variable auxiliar para indicar que la apertura de sesión con este host ha sido fallida. Su uso será detallado en capítulos posteriores
sesion	Estructura usada para la apertura de sesión con este host
punt_sesion	Puntero a la estructura anterior. Se alamcena este puntero por facilidad de uso en la herramienta
oids	Puntero a una lista de estructuras adicional que contendrá la información de los objetos SNMP que se desean monitorizar

Tabla 8 Parámetros de st_host

Al igual que en nuestro fichero de monitorización, en memoria tendremos una lista enlazada de nodos que contendrán toda la información relativa a cada host cuyos objetos deseamos monitorizar. Por cada objeto de la lista del fichero JSON, crearemos un nodo **st_host**, incluyendo en sus variables toda la información de conectividad que encontramos en el fichero.

¿Cómo se almacena la información de los objetos a monitorizar?

La filosofía por la que se organiza el fichero de monitorización, en la cuál dentro de cada objeto, continente de la información de cada host, encontramos una lista de otro tipos de objeto, los cuales contienen cada oid de cada objeto SNMP que se desea monitorizar.

Siguiendo este procedimiento, añadimos el parámetro **oids** a la estructura **st_host** con la intención de enlazar una lista de nodos con los oids de los objetos a monitorizar. En concreto **oids** es un puntero a una estructura llamada **st_oid**, que definimos dentro de nuestra herramienta.

Usaremos una lista enlzada de esta estructura para almacenar la información de los objetos que se desean monitorizar del host. En **st_oid** encontramos los siguientes parámetros:

```
typedef struct oid {
  int id;
  char * name;
  char * oid_name;

  oid Oid[MAX_OID_LEN];
  size_t OidLen;

  int enviado_id;
  int ultimo_reqid;

  long int t_monitor;
  long int t_actual;

  struct oid *next;
}st_oid;
```

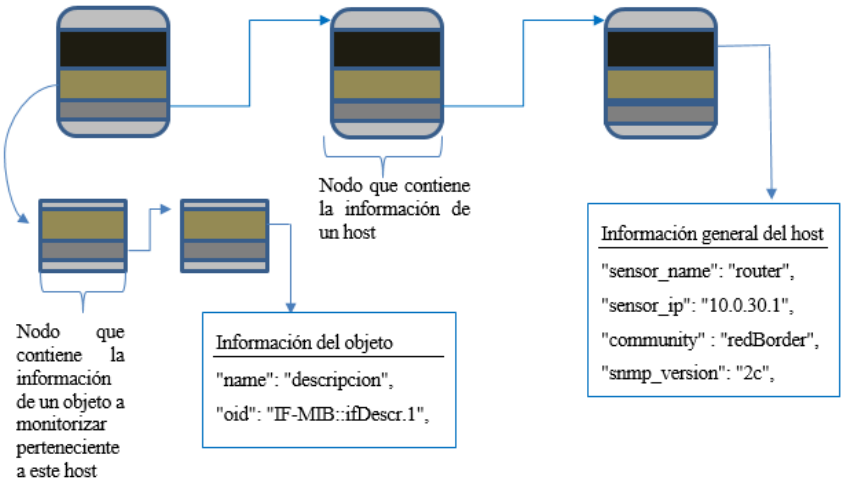
Code 12 Estructura de OID

id	Identificador numérico de este OID
name	Nombre dado al oid para su identificación con el manager, contenida en el parámetro **name** dentro de el fichero de monitorización
oid_name	Identificador de OID que posteriormente será traducido con la MIB para obtener la cadena de enteros que conforman este OID. Se obtiene del parámetro **oid** del fichero de monitorización
Oid	Tabla que se usará para almacenar el resultado de la traducción del parámetro **oid_name**
OidLen	Tamaño efectivo de la tabla **Oid**
enviado_id	Identificador de la ultima PDU enviada
ultimo_reqid	Identificador de la ultima PDU recibida y enviada a RedBorder
t_monitr	Tiempo entra el envío de cada petición
t_actual	Variable tiempo auxiliar

Tabla 9 Parámetros de st_oid

Cada nodo de la lista enlazada contendrá la información particular de cada oid. Nuestro motor de envío de peticiones recorrerá esta lista formando las **GetRequest-PDU** como ya se vió en capítulos anteriores (ver **4.1.1.1.2 Envío de peticiones**).

Importante destacar la presencia de los parámetros **enviado_id** y **ultimo_reqid** se usará para detectar duplicidad en la respuesta. Al monitorizar el descriptor de ficheros, se pueden detectar **SetRequest-PDU** duplicadas en respuesta a la misma consulta y por lo tanto llamar al manejador varias veces para la misma **SetRequest-PDU**. Nuestro interés es filtrar estas PDU y enviar hacia el manager de RedBorder sólo un resultado. Si almacenamos el ultimo id enviado hacia RedBorder nos aseguramos que no repetimos el envío de los resultados de monitorización



Como resultado final del parseo del fichero de monitorización, obtendremos una lista enlazada de nodos que contendrán la información de cada host que se desea monitorizar.

Adicionalemnte, enlazaremos a cada nodo una lista enlazada con la información de los oid que se desean solicitar.

Fig. 42 Esquema de la organización en memoria

Pero llegados a este punto nos surge una cuestión, **¿cómo accederán los hilos del motor de envío de peticiones a esta estructura?**

En posteriores capítulos detallaremos cómo se reparte esta estructura entre los hilos del motor de envío de peticiones. Además se detallará el funcionamiento para la recarga del fichero de monitorización, que requerirá un trato especial.

4.2.3.3 Funcionamiento con ficheros de configuración

Una vez descrita la estructura interna de los ficheros de configuración, y comprendidos los parámetros que estos aportan, pasamos a analizar cómo será el funcionamiento de la herramienta con estos ficheros.

El fichero de configuración general

Como hemos comentado, los parámetros de configuración general están relacionados directamente con el funcionamiento de la herramiente. De este fichero se obtendrán sus datos solo al inicio de la herramienta y se almacenarán estos datos en variables globales. Se entiende que modificar este tipo de parámetros no es algo común ni relacionado con decisiones del usuario final de RedBorder, luego para realizar una modificación en la herramienta de este calibre se deberá rearrancar la herramienta.

El fichero de monitorización

En cambio y a diferencia del anterior, el fichero de monitorización incluye una información susceptible de estar en continuo cambio. La idea final es ofrecer desde RedBorder una interfaz gráfica al usuario que le permita modificar, añadir y eliminar objetos que monitorizar, al igual que modificar la forma en que estos le son representados.

Para ello debemos de dotar a nuestra herramienta de un mecanismo que le permita recibir continuamente nuevos ficheros de monitorización que añadan, modifican o eliminen los hosts y sus objetos.

En concreto y si recordamos el apartado **4.2.2.1 Parámetros**, incluimos un parámetro especial en cada objeto que encontramos dentro del fichero JSON. Este parámetro nos indicará si al procesor el nuevo fichero de configuración, dicho objeto debe de ser añadido, modificado o eliminado.

Inicialmente el fichero de monitorización será parseado creando la estructura enlazada que hemos visto en el punto anterior. Pero,

¿cómo se recarga el fichero de monitorización durante la ejecución de la herramienta?

4.2.3.3.1 SIGHUP para la recarga de la configuración

No es objeto de esta memoria comprender el funcionamiento de Apache Zookeeper[66], pero es una funcionalidad de esta herramienta manejar los ficheros de configuración dentro de un cluster[67]. RedBorder se sirve de esto para su comunicación con los sensores.

[66] Página oficial de Apache Zookeeper: https://zookeeper.apache.org/
[67] Definición de cluster: https://es.wikipedia.org/wiki/Cl%C3%BAster_(inform%C3%A1tica)

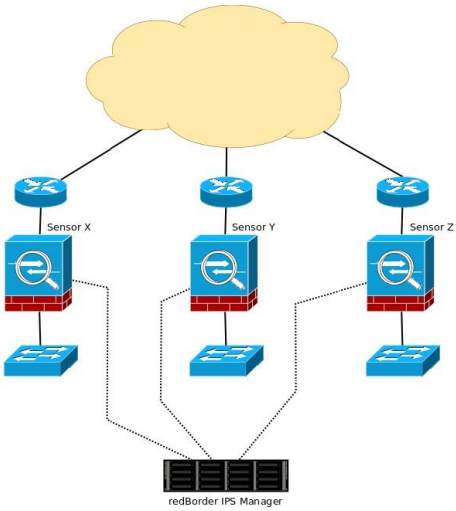


Fig. 43 Esquema de Redborder

Como vemos en la **Fig 41 Esquema de RedBorder**, la información recopilada por los sensores, uno de los cuales será nuestra herramienta, será enviada al manager. El manager no se compone de un único proceso, sino que se dispone en un cluster de servidores organizados a través de Apache Zookeeper. Adicionalmente, la mensajería de alto rendimiento es Apache Kafka, la cual debe ser implementada a través de Zookeeper.

Por lo tanto, el funcionamiento desde el manager será el siguiente:

Cada vez que se desee recargar un fichero de configuración, se colocará este en la ruta preestablecida (en nuestro caso, indicada a través de línea de comandos en el momento de la ejecución), y se enviará al proceso concreto una señal de SIGHUP.

Sabiendo este funcionamiento procedemos de la siguiente forma desde nuestra herramienta:

- **Registramos un manejador para SIGHUP**

  Usaremos una función llamada recargaConfig, que será la encargada del proceso de recargar la configuración de monitorización.

```
/*
 * Registro manejadores:
 * - SIGINT -> Detecta cierre e inicia un cierre seguro
 * - SIGHUP -> Señal para la recarga de la configuración
 */
signal(SIGINT, handlerCierre);
signal (SIGHUP, recargaConfig);
```

Code 13 Registro de manejadores

- **RecargaConfig()**

  Esta función será la ejecutada ante la recepción de SIGHUP. Como ya comentamos, esto se producirá cuando un nuevo fichero de monitorización se encuentre ya localizado en el directorio indicado por línea de comandos en el momento de lanzar nuestra herramienta.

```
void recargaConfig(int dummy) {

    * Accedemos a la recarga usando una variable
   pthread_mutex_lock(&lock);
   carga_config();
   pthread_mutex_unlock(&lock);
```

Code 14 Manejador de SIGHUP

Esta función a su vez llamará a la función encargada de parsear el nuevo fichero de monitorización e interactuar con la estructura según proceda. Pero se debe prestar especial atención a las variables de mutex que protegen el acceso a la estructura dinámica.

- **Protección del acceso a la estructura**

  Usaremos una variable de mutex denominada **lock**. Esta variable protegerá el acceso a la estructura de hosts mientras se está realizando una recarga del fichero de monitorización. Esto es debido a que las modificaciones que puedan producirse pueden originar conflictos si los hilos están recorriendo la lista en el momento en el que se producen. Para ello, el bucle de monitorización del motor de envío de peticiones comprobará el valor de **lock** antes de cada iteracción, deteniéndose si esta variable se encuentra bloqueada.

- **Modificaciones a la estructura**

Como hemos comentado, en el apartado **4.2.2.1 Parámetros** vimos como, tanto en la información proporcionada del host como del oid dentro del fichero de monitorización, se incluye un parámetro denominado **modif**.

Este parámetro nos indicará la forma en la que se ha de actuar con cada objeto, tanto host como oid, que extraemos del fichero de monitorización (**ver Fig 42**).

Se contemplan las tres acciones básicas sobre la estructura, que son crear, eliminar y modificar la información de un nodo.

En el proceso de recarga de configuración procederemos de la siguiente forma:

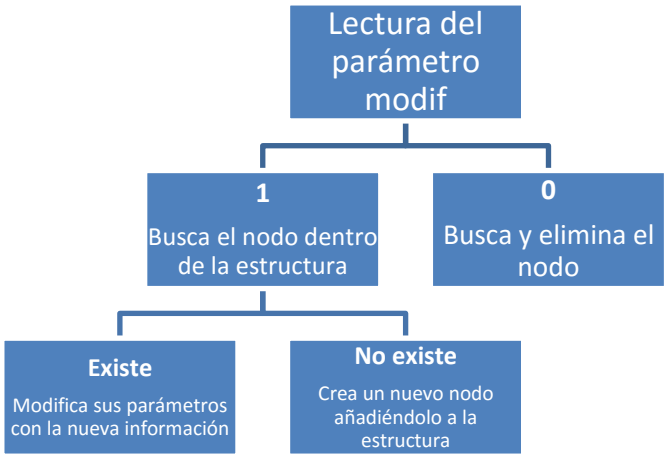


Fig. 44 Parámetro modif en el fichero de monitorización

- Extración de los datos del fichero JSON
- Lectura del parámetro modif
- Creación, eliminación o modificación del nodo extraido en función del parámetro anterior

Para la búsqueda de un nodo dentro de la estrcutura, nos serviremos de las siguientes funciones, que buscaran basándose en el id del nodo que queremos buscar.

```
/*
 * Funcion busca_nodo()
 * Busca un nodo en la lista enlazada en base a su id
 */
st_host * busca_nodo(int id_sensor);


/*
 * Funcion busca_oid()
 * Busca un oid en la lista enlazada en base a su id
 */
st_oid * busca_oid(int id_oid, st_oid* lista);
```

Code 15 Funciones auxiliares de busqueda dentro de la estructura

De esta forma se provee la funcionalidad al manager de RedBorder de modificar la estructura sobre la que trabajará el motor de envío de peticiones, añadiendo así nuevos objetos o eliminando los que, a petición del usuario, ya no se deseen monitorizar.

Posteriormente detallaremos la forma en la que los nuevos nodos entran en la estructura para mantener la carga balanceada entre hilos.

4.2.4 Envío de datos a través de Apache Kafka

Apache Kafka [16] es un software desarrollado por la Apache Software Foundation[68], cuya misión es proveer de un servicio de mensajería de alto rendimiento a través de un cluster de servidores.

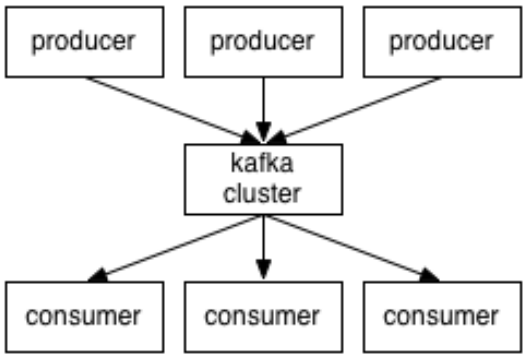


Fig. 45 Esquema de funcionamiento de Apache Kafka

Como vemos en la **Fig. 43**, la mecánica de funcionamiento es la de la existencia de **producers** que serán fuente de mensajes y **consumers** que serán los encargados de consumir y procesar dichos mensajes.

La comunicación con los servidores del cluster se realiza mediante **TCP** pero con algunas características adicionales que lo dotan de una mayor eficiencia[69], que no son objeto de estudio de este proyecto.

El proceso por el cuál se organizan y clasifican los mensajes del cluster y a través del cuál se pueden dividir los mensajes producidos, es el de la existencias de **topics** o etiquetas de categoría. Un **topic** es un identificador o etiqueta que se le asigna a los mensajes generados por los **producers** y que servirán para poder identificar el tipo de mensaje que los **consumers** van a ir procesando.

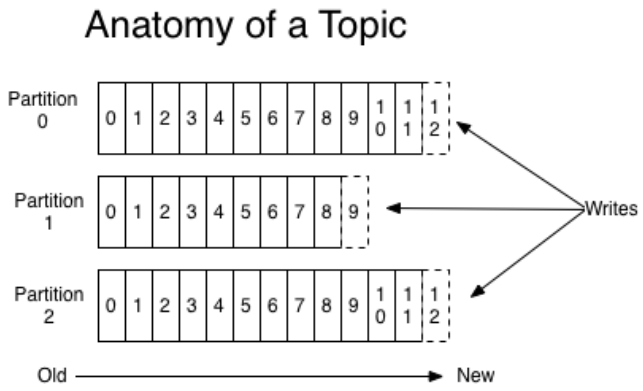


Fig. 46 Topic en el cluster de Kafka

Los mensajes producidos en un determinado **topic** serán almacenados en el cluster en diferentes particiones como observamos en la **Fig. 44**.

Como resultado final encontraremos un cluster organizado a través de Zookeeper sobre el que se implementa el servicio de mensajería Apache Kafka. En dicho cluster tendremos particiones de los diferentes topics que contendrán los mensajes que los consumidores irán procesando.

A grandes rasgos podemos obtener el esquema del funcionamiento de RedBorder. El manager constará de un conjunto de consumidores, que procesará los datos consumidos, mostrándolos en el frontend de RedBorder, procesándolos o analizándolos. Estos mensajes serán producidos por los sensores, los cuales generarán mensajes en un topic concreto que identificará el tipo de mensajes que genera.

Con esta idea en mente vamos a analizar el trabajo de nuestra herramienta para incorporar los mensajes a un topic de Kafka.

4.2.4.1 Librdkafka

Para implementar éste protocolo haremos uso de la librería, para el lenguaje de programación C, **librdkafka**[70]. Apache nos proprociona una librería para lenguaje Java, pero la comunidad ha desarrollado librerías para una gran variedad de lenguajes[71]. Para el lenguaje de programación C, la propia documentación de Apache Kafka nos dirije a un proyecto de GitHub llamado **librdkafka** [17]**.**

Vamos a entrar en detalles sobre cómo es usada por nuestra herramienta.

---

[68] Apache Software Foundation: http://www.apache.org/
[69] TCP en Apache: https://kafka.apache.org/protocol.html
[70] Librdkafka: https://github.com/edenhill/librdkafka
[71] Librerías de Apache Kafka para diferentes lenguajes: https://cwiki.apache.org/confluence/display/KAFKA/Clients

Nuestra herramienta solo tiene que implemetar el proceso de un **producer**, ya que los **consumer** serán objetivo del manager de RedBorder.

Nuestro objetivo es pues iniciar la comunicación con el broker (nombre que se le da al proceso que controla al cluster) y añadir nuestros mensajes a un topic en concreto.

4.2.4.1.1 Inicio

Inicialmente debemos de realizar un paso de configuración previo antes de añadir mensajes a un topic. Para ello:

rd_kafka_conf_t * rd_kafka_conf_new (void)

Esta función crea un objeto de configuración que es lo que será devuelto. Esto crea un objeto de configuración vacío, el cuál tendremos que ir rellenando con datos más concretos antes de empezar el proceso de envío de mensajes. Para añadir parámetros de configuración:

rd_kafka_conf_res_t rd_kafka_conf_set(rd_kafka_conf_t *conf, const char *name, const char *value, char *errstr, size_t errstr_size);

Con esta función añadiremos parámetros de configuración identificados por un string pasado en el parámetro **name**. Por ejemplo, vemos en **Code 10** como añadimos al proceso una señal de terminación a través de esta función.

```
/* Quick termination */
snprintf(tmp, sizeof(tmp), "%i", SIGIO);
rd_kafka_conf_set(conf, "internal.termination.signal", tmp, NULL, 0);
```

Code 16 Uso de rd_kafka_conf_set

Este mismo proceso debemos realizar para la creación y uso de un determinado topic. Usaremos las siguientes funciones de similar sintaxis a las anteriores.

rd_kafka_topic_conf_t * rd_kafka_topic_conf_new (void);

rd_kafka_topic_t *rd_kafka_topic_new(rd_kafka_t *rk, const char *topic, rd_kafka_topic_conf_t *conf);

Con estas dos funciones **librdkafka** nos permite inicar una estructura de configuración de un topic en primer lugar y posteriormente crear dicho topic identificado por un string, pasado como parámetro en el arumento **topic**.

Vemos el funcionamiento en un ejemplo simple en **Code 11**.

```
static rd_kafka_t *rk;
rd_kafka_topic_conf_t *topic_conf;
char *topic = "prueba";

/* Topic configuration */
topic_conf = rd_kafka_topic_conf_new();

/* Create topic */
rkt = rd_kafka_topic_new(rk, topic, topic_conf);
topic_conf = NULL; /* Now owned by topic */
```

Code 17 Creación y configuración de un topic

Un último paso previo al envío de mensajes a determinado topic es el de añadir el broker o los brokers a la configuración. Esto se permite mediante:

```
int rd_kafka_brokers_add(rd_kafka_t *rk, const char *brokerlist);
```

Indicamos una lista de brokers a modo de string separados por el carácter coma ",".

Estas funciones nos permiten un primer paso de configuración de porductor. Veamos cómo producer mensajes a determinado topic y el funcionamiento interno de **librdkafka**.

4.2.4.1.2 Productor

La función principal de los productores es:

```
int rd_kafka_produce(rd_kafka_topic_t *rkt, int32_t partition, int msgflags, void *payload, size_t len, const void *key, size_t keylen, void *msg_opaque);
```

Vamos a analizar los parámetros:

- **rkt**

  El topic creado con **rd_kafka_topic_new**

- **partition**

  Entero que selecciona la partición en la que incluir nuestro mensaje (ver **Fig. 44**). Puede ser un entero o un flag que indique asignación automática. Este flag es **RD_KAFKA_PARTITION_UA**

- **msgflags**

  En este flag existen dos posibilidades:

  - **RD_KAFKA_MSG_F_FREE**

    La librería liberará el mensaje una vez termine con él

  - **RD_KAFKA_MSG_F_COPY**

    El mensaje será copiado, luego en el puntero al mensaje, una vez se haya llamado a **rd_kafka_produce**, lo encontramos aún por lo que tendremos que tener en cuenta que debe ser eliminado después

  Posteriormente detallaramos qué opción es la elegida por nuestra herramienta y por qué.

- **payload** y **len**

  El mensaje propiamente dicho y su longitud. Importante destacar la flexibilidad de **librdkafka** y de **Apache Kafka** dando a este parámetro un tipo **void**, pudiéndo así considerar cualquier tipo de datos como un mensaje.

- **key** y **keylen**

  Con este parámetro se nos da la opción de añadir un mensaje adicional. Si se incluye un valor no nulo, se incluirá este mensaje en la partición indicada junto con el emnsaje principal indicado en payload. Podemos considerarlo como un mensaje auxiliary.

- **msg_opaque**

  Parámetro que permite añadir datos al callback definido para esta función. Actúa de la misma forma que **callback_magic** en net-snmp (ver apartado **4.1.1.2.2 Manejador de la sesión**).

Podemos entender el funcionamiento de envío de mensajes. Una vez realizado el proceso de incio por el cual creamos una configuración y un topic y añadimos los brokers necesarios, podemos proceder con la función de producción. Pero,

¿qué es la función de callback en librdkafka?

Librdkafka nos permite la opción de callback que se ejecutará cada vez que se envíe un mensaje. Podemos definir el manejador gracias a la siguiente función:

```
void rd_kafka_conf_set_dr_msg_cb(rd_kafka_conf_t *conf, void (*dr_msg_cb)
(rd_kafka_t *rk, const rd_kafka_message_t * rkmessage, void *opaque));
```

A través del segundo parámetro podemos indicar la función deseada como callback para cada mensaje enviado.

Al igual que en **net-snmp** no bastaba con definir y registrar el manejador, sino que teníamos que realizar una monitorización de determinados sockets en los descriptores de ficheros, en **librdkafka** debemos de realizar un proceso periódico para que los callbacks sean ejecutados. Esto es:

```
int rd_kafka_poll(rd_kafka_t *rk, int timeout_ms);
```

Con esto definimos las herramientas necesarias para producir los resultados de la monitorización a un determinado topic que, posteriormente, serán consumidos por el manager de RedBorder.

Antes de pasar a ver cómo implementa nuestra herramienta el mecanismo de producción de Apache Kafka, vamos a ver las funciones de cierre en **librdkafka**.

4.2.4.1.3 Cierre

Las siguientes funciones son las necesarias para finalizar correctamente la producción y liberar los recursos reservados:

```
void rd_kafka_topic_destroy(rd_kafka_topic_t *rkt)

void rd_kafka_destroy(rd_kafka_t *rk)

void rd_kafka_topic_conf_destroy(rd_kafka_topic_conf_t *topic_conf);
```

Se puede observar fácilmente que cada función elimina y libera cada recurso reservado en la fase de incio, lo que require de una explicación más detallada. Para ver al detalle cada función aquí mencionada basta observer la documentación de **librdkafka**[72].

A continuación veremos cómo incorporar las tres fases de inicio, producción y cierre dentro de nuestra herramienta, funcionando conjuntamente con el motor de recepción de respuestas.

[72] Documentación de librdkafka: https://github.com/edenhill/librdkafka/blob/master/src/rdkafka.h

4.3 Organización de datos

Llegados a este punto, hemos descrito el funcionamiento básico de la monitorización (ver **4.1 Net-snmp como base para la monitorización**), la recepción y recarga de parámetros de configuración por parte de RedBorder (ver apartados **4.2.1 Recepción de parámetros de configuración general** y **4.2.2 Recepción de parámetros de monitorización**) al igual que el envío por parte de nuestra herramienta de los resultados de la monitorización a través de Apache Kafka (ver **4.2.3 Apache Kafka**).

Comprendidas estas funcionalidades vamos a describir cómo nos servimos de las estructuras de datos internas para el intercambio de datos y balanceo de carga entre todas las partes de la herramienta.

4.3.1 Motor de envío de peticiones multihilo

En el apartado **4.1.1.1 Motor de envío de peticiones** vimos cómo usábamos de base la librería **net-snmp** para programar un motor de envío de **SetRequest-PDU** a la red. El funcionamiento visto en ese apartado únicamente detalla cómo funciona el manjeo de la sesión SNMP y cómo se creaba la PDU para ser enviada al host destino. Pero, **¿cómo se organiza éste motor en la herramienta final?**

Ya comentamos en la introducción que uno de los puntos principales de la herramienta debería de ser la eficiencia, y vimos obvio que el funcionamiento debía de ser multihilo. Vamos a detallar cómo implementamos esto.

```
while ((numero_hilo < num_hilos)) {
    p_hilo_prov = &hilos[numero_hilo];
    error_hilo=pthread_create(p_hilo_prov,NULL,poller,(void *)numero_hilo);
    if (error_hilo!=0) {
        fprintf(stdout,"Error de lanzado en el hilo nº%d\n",numero_hilo);
    }
    else {
        numero_hilo++;
        hilos_lanzados++;
    }
}
```

Code 18 Bucle lanzado de hilos

En **Code 12** tenemos el bucle encargado de lanzar los hilos del motor de peticiones.

Cada hilo ejecutará la función **poller**, recibiendo esta como argumento un entero que identifica a este hilo. Este parámetro será luego detallado y se comprenderá su utilidad cuando detallemos el reparto de carga.

Como observamos, la función **poller** es la que ejecutará todo el proceso de inicio de session, envío de peticiones y cierre visto en el apartado **4.1.1.1**.

Hay dos aspectos importantes a tener en cuenta llegados a este punto:

- **¿Cómo se reparte la información de monitorización entre hilos?**

  Es decir, todos los hilos trabajarán sobre la misma estructura de datos, y debemos de evitar que dos hilos accedan a la misma información, ya que no deseamos que se hagan dos consultas para el mismo OID. Por ello debemos de planear un mecanismo de reparto de carga. Éste será detallado en el siguiente capítulo

- **¿Cómo funciona el bucle de monitorización?**

  Hay varios aspectos que hay que tener en el bucle de monitorización, y que hace que éste no sea trivial. El primero de ellos es el del incio de sesión. **¿Qué hacemos si no se abre correctamente una sesión?**

  Otro aspecto importante que observamos si tenemos en cuenta la información que recibimos por parte del manager, es el de la gestión de los tiempos de monitorización. **¿Cada cuántos segundos se produce una iteracción del bucle?**

4.3.1.1 Reintentos en la apertura de sesiones

La clave de esto la vimos en el apartado **4.2.2.2 Organización en memoria** donde vimos que en la estructura que recoge la información de un host, la llamada **st_host**, incluye un parámetro denominado **fallo_sesion.** Éste parámetro será un entero que usaremos para marcar que, dentro del bucle de envío de peticiones, al llegar a este host, debemos realizer un reintento de apertura de sesión.

Cuando la apertura de una de las sesiones falla, usamos éste parámetro para indicar un valor, definido en una macro, que indica e número de iteracciones hasta el siguiente reintento de apertura de la sesión.

En cada iteracción del hilo en su bucle de monitorización, reducirá en **1** éste parámetro hasta que llegue a cero, que reintentará la apertura.

```
//Comprobamos que se haya abierto correctamente
if (!(lista_host_prov->punt_sesion)) {
    //Marcamos este host --> contador de reintentos
    lista_host_prov->fallo_sesion = ESPERA_REINTENTOS;
}
else {
    //fprintf(stdout,"Sesion abierta correctamente\n");
    lista_host_prov->fallo_sesion = 0;
}
```

Code 19 Reintento de apertura de sesión

De esta forma los hilos realizarán una apertura inicial y un reintento progresivo a lo largo del bucle de monitorización.

4.3.1.2 Gestión del tiempo de monitorización

Si recordamos los datos que recibimos por parte de RedBorder, uno de los parámetros que acompañaban a la información de los OIDs es el del tiempo que se desa entre envío de peticiones a ese OID.

Se entiende que el tiempo no va a ser uniforme entre todos los OIDs que monitoriza el hilo, luego, **¿cómo gestionamos esto?**

Tenemos que recorder dós parámetros en la estructua de almacenamiento de los datos de OIDs, **st_oid**:

- **t_monitor**: Almacena el tiempo en segundos entre cada monitorización. Es el dato proporcionado por el manager
- **t_actual:** Almacena el tiempo que ha pasado desde la última petición a este OID

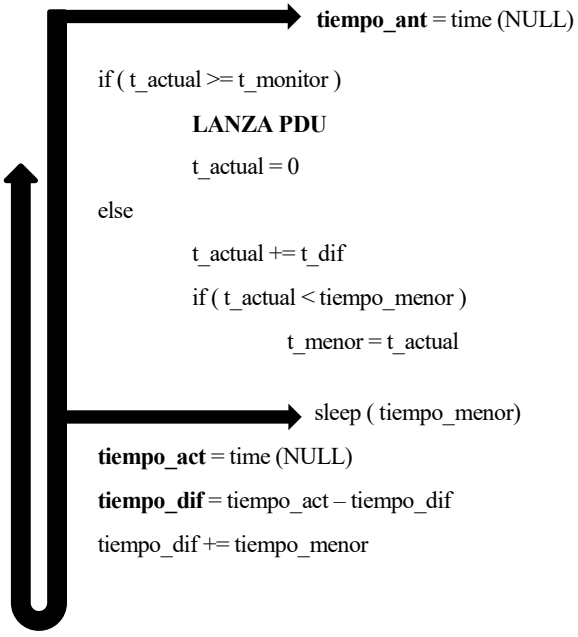


Fig. 47 Esquema gestión del tiempo de monitorización

El proceso de gestión del tiempo de monitorización se basará en las dos variables antes comentadas, incluidas en la estructura **st_oid** y adicionalemnte en cuatro variables actualizadas durante la ejecución del bucle de monitorización. Estas son:

- tiempo_ant
- tiempo_act
- tiempo_dif
- tiempo_menor

En la **Fig 48** vemos el esquema de funcionamiento, actualizando en cada bucle el **t_actual**, monitorizando cuando llegue al valor de **t_monitor**, y esperando entre cada iteracción el menor tiempo hasta la siguiente monitorización.

Como vemos, de esta forma el bucle de monitorización iteracciona entre tiempos, durmiendo al final del bucle hasta que sea necesite enviar la siguiente PDU en alguno de los OID.

4.3.2 Carga de monitorización

En capítulos anteriores hemos descrito cómo el funcionamiento del motor de envío de peticiones es multihilo, además de haber definido cómo se organiza la estructura de datos que contiene los datos necesarios para la monitorización. Pero,

¿cómo sabe cada hilo qué tiene que monitorizar y qué no?

La idea adoptada es la siguiente:

4.3.2.1 Reparto de carga en el motor de envío de peticiones

Una vez tenemos el "mapa" de memoria, en la que tenemos la información de los hosts y sus objetos en una lista enlazada, debemos programar el acceso parcial que hará cada hilo del motor de envío de peticiones. Esto es, cada hilo recorrerá la lista de manera parcial, debiendo tener en cuenta que dos hilos no deben de monitorizar el mismo host.

Si recordamos del apartado **4.2.2.2 Organización en memoria** (ver **Tabla 8**), a la estructura **st_host** añadimos una variable llamada **corte**, definida incialmente como una variable auxiliar para el reparto de memoria.

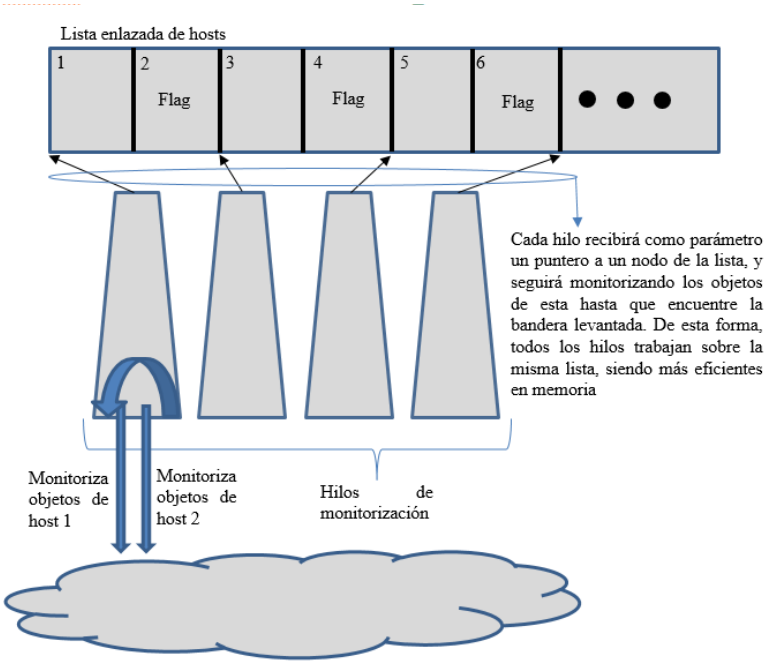


Fig. 48 Esquema de acceso a la estructura

La idea de reparto de carga se basa en controlar el inicio y fin de la monitorización de cada hilo. Para ello debemos tener control sobre el aaceso al primer host a monitorizar y debemos poder indicar al hilo cuál es el último nodo que monitoriza en su recorrido lineal por la lista.

Como vemos en la **Fig. 45**, cada hilo accederá a un punto diferente de la lista, recorriendo esta hasta la aparición de la variable corte. Luego, nuestra principal misión será la de controlar el acceso a los hilos y la de activar las banderas en los puntos concretos de la estructura que nos interese.

4.3.2.1.1 Reparto de cortes

En el reparto de cortes recae el peso del balanceo de carga. El primer paso es establecer los cortes adecuados sobre la estructura para posteriormente repartir el punto de acceso entre los hilos.

Para este proceso nos servimos de la función descrita en **Code 13**.

```c
/*
 * Funcion: lista_cortes()
 * Funcion que establece las variables de corte
 * en la estructura de hosts
 *
 */
void lista_cortes (st_host * lista, int host, int resto, int hilos) {

    int llenado = 0;

    if(host!=0)
    {
        //CASO A: Mas hosts que hilos
        while (lista != NULL)
        {
            llenado++;
            if (llenado == host) {
                llenado = 0;
                if (resto > 0) {
                    resto--;
                    lista = lista->next;
                }
                lista->corte=1;
            }
        lista = lista->next;
        }
    }
    else {
        //Caso B: Mas hilos que hosts
        //Establezco corte en todos los hosts
        while (lista!=NULL) {
            if (lista->next != NULL)
                lista->corte=1;
            lista=lista->next;
        }
    }
}
```

Code 21 Función de reparto de cortes

Esta función denominada **lista_cortes()** tomará los siguientes argumentos:

- **Lista**: Lista enlazada del programa
- **Host:** Numero de hosts que va a monitorizar cada hilo (reparto equitativo que veremos a continuación)
- **Resto:** Hosts sobrantes del reparto equitativo
- **Hilo**: Numero de hilos

```c
resto = num_hosts%num_hilos;
host_hilo = num_hosts/num_hilos;

//Establecemos las variables corte
lista_cortes (lista_hosts,host_hilo,resto,num_hilos);
```

Code 22 Llamada a lista_cortes()

Como vemos en **Code 14**, el reparto de hosts que va a monitorizar cada hilo es equitativo. Se divide el número total de hosts entre el número de hilos y se redondea a la baja, almacenando los hosts restantes como la variable **resto**, que será pasada como tercer parámetro a la función **lista_cortes.**

Se dan dos casos claramente diferenciados dentro de **lista_cortes**:

Caso A: A la hora del reparto hay más hosts a monitorizar que hilos lanzados. En este caso se hace un recorrido por la lista, contando hasta llegar al número de hosts que vamos a monitorizar en cada hilo. Llegados hasta este punto, si hay resto disponible, cuento uno más y decremento el resto. Al terminar este recorrido pongo la variable corte a 1 y empieza a contar de nuevo.

De esta forma se repartirá la carga de forma equitativa. Un ejemplo sería:

Disponemos de 10 hilos. El número total de hosts a monitorizar es de 23.

Host/hilo: 2

Resto: 3

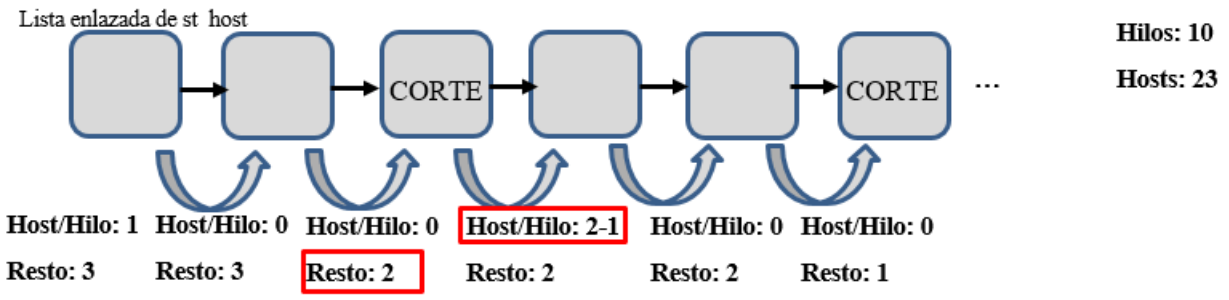


Fig. 49 Esquema de reparto de cortes

Caso B: Por el contrario, si hay menos hosts que hilos, establecemos a 1 la variable corte de todos los hosts presentes en la estructura, ya que cada hilo sólo monitorizará un host.

4.3.2.1.2 Acceso de los hilos a la estructura

Una vez tenemos los cortes a la estructura, debemos indicar a cada hilo desde qué punto debe acceder a esta para que monitorice hasta que encuentre la variable corte a 1. Llegados a este punto reiniciará el bucle de monitorización.

Para ello nos servimos de la tabla que denominamos **acceso_hilos**.

```
//      -->Memoria compartida<--

//Lista enlazada con la informacion de monitorizacion
st_host * lista_hosts;

//Tabla de punteros a los hilos
pthread_t * hilos;

//Tabla almacenar los accesos de los diferentes hilos
st_host ** acceso_hilos;
```

Code 23 Memoria compartida

Será una tabla de punteros a la estructura **st_host**, lo que es trivial ya que cada posición de la tabla contendrá un puntero hacia un nodo concreto de la estructura dinámica.

Esta tabla, con presencia global al igual que la estructura dinámica de hosts como vemos en **Code 15**, deberá ser rellenada justo después del cálculo de cortes y cada vez que se cargue o recargue la configuración de monitorización.

La mecánica es la siguiente:

Lo primero es realizar una reserva dinámica para una tabla de tantas posiciones como hilos tengamos disponibles

```
acceso_hilos = (st_host **) malloc (num_hilos * sizeof(st_host *))
```

Una vez creada, pasamos al proceso de llenado. Para ello, nos basaremos en los cortes ya establecidos en la estructura. Como es el proceso **lista_cortes** el encargado de repartir la carga entre los diferentes hilos, el proceso de llenado de la tabla de reparto se soluciona con un recorrido simple por la tabla de hosts. Almacenamos las posiciones siguientes a la aparición de un corte. Es decir, en la posición 0 almacenamos el primer host. Recorremos la lista hasta la aparición de un corte y almacenamos en la siguiente posición de la tabla el puntero al nodo contiguo al que tiene la variable corte activada. Procedemos de esta forma hasta el final de la lista.

Si recordamos el caso B del apartado anterior, caso en el que existen más hilos disponibles que nodos en la estructura, es lógico pensar que esta tabla no tiene por qué estar rellena al completo. Por este motivo, y sólo en éste caso, almacenamos NULL en aquellas posiciones sobrantes.

4.3.2.1.3 Implicaciones de la recarga del fichero de monitorización

¿Qué sucede con la tabla de reparto y las variables corte de la estructura si se recarga la configuración? ¿Qué ocurre si al recargar la configuración se elimina un nodo con la variable a corte activada? ¿O un nodo que aparece en la tabla de reparto?

El aspecto más importante, y que hace al reparto de carga un punto de especial mención, es el de la adaptación del reparto de carga a las fluctuaciones de nuestra herramienta en cuanto a la carga de monitorización.

En el apartado **4.3.2 Recarga y reestructuración de datos** veremos todas las implicaciones que tiene la modificación de la estructura de datos, incluida las modificaciones y adaptación que se ha de hacer para mantener un reparto de carga equitativo a pesar de los cambios.

4.3.2.1.4 Resumen del reparto de carga

Finalmente, la idea clave en el reparto de carga proviene de la interacción con la tabla de acceso y las variables cortes activas presente en la estructura.

Si vemos la **Fig 45**, vemos que cada hilo accede a una posición diferente de la estructura, monitorizando hasta que aparece activada la variable corte. El siguiente hilo comoenzarña su monitorización justo en el nodo siguiente. Por este motivo y viendo que en ningún momento dos hilos acceden a la misma posición de memoria, no se hace necesario proteger la memoria dinámica por el acceso de los diferentes hilos.

```
{
    "sensor_id": 1,
    "sensor_name": "prueba",
    "sensor_ip": "195.218.195.228",
    "community" : "variation/virtualtable",
    "snmp_version": 2,
    "modif": 1,
    "monitors":
    [
        {
            "oid_id": 1,
            "name": "paqEnt",
            "oid": "IF-MIB::ifInOctets.1",
            "modif": 1
        },
        {
            "oid_id": 2,
            "name": "paqUcast",
            "oid": "IF-MIB::ifInUcastPkts.1",
            "modif": 1
        }
    ]
},
{
    "sensor_id": 2,
    "sensor_name": "prueba2",
    "sensor_ip": "195.218.195.228",
    "community" : "variation/virtualtable",
    "snmp_version": 2,
    "modif": 1,
    "monitors":
    [
        {
            "oid_id": 3,
            "name": "paqEnt",
            "oid": "IF-MIB::ifInOctets.1",
            "modif": 1
        },
        {
            "oid_id": 4,
            "name": "paqUcast",
            "oid": "IF-MIB::ifInUcastPkts.1",
            "modif": 1
        }
    ]
}
```

Code 24 Ejemplo simple - monitorización

Si realizamos una ejecución simple de nuestra herramienta, en la que disponemos de dos hosts con dos OIDs cada uno, y dos hilos disponibles para el motor de envío de peticiones, el resultado del reparto de carga mostrado por pantalla es el siguiente:

```
{
    "threads": 2,
    "kafka_broker": "127.0.0.1:9092",
    "kafka_topic": "rb_monitor"
}
```

Code 25 Ejemplo simple – configuración general

```
----------Lista enlazada:
[prueba2]->[IF-MIB::ifInUcastPkts.1]->[IF-MIB::ifInOctets.1]->|NULL
[prueba]->[IF-MIB::ifInUcastPkts.1]->[IF-MIB::ifInOctets.1]->|NULL

----------Tabla de reparto:
 [0]prueba2     [1]prueba

----------Lista de cortes:
[prueba2]->corte->[prueba]->corte->NULL
```

Code 26 Ejemplo simple – resultado reparto de carga

Este ejemplo es sencillo pero el resultado es evidente. La tabla de reparto contiene, para el primer hilo, el acceso al primer host denominado **prueba2**. A su vez, este host tiene la variable corte activada, luego, el primer hilo monitorizará únicamente el host **prueba2**. El acceso del segundo hilo es **prueba** y sólo monitorizará este host porque igualmente tiene la variable corte activada.

Como conclusión podemos decir que el reparto de carga se realiza en base al número de hosts a monitorizar y no teniendo en cuenta otros factores, como por ejemplo, el número de OIDs que monitoriza cada host. En el apartado de mejoras propondremos una solución a esta simplicidad en el reparto de carga.

4.3.3 Recarga y reestructuración de datos

En este apartado vamos a detallar todas los aspectos que se tratan derivados de la recarga, y por lo tanto modificación, de la estructura de datos de monitorización.

En el apartado **4.2.2.3.1 SIGHUP para la recarga de la configuración** analizamos la funcionalidad de recarga de los datos de monitorización de la que se dota esta herramienta. El resultado de esta recarga es que se podía añadir, eliminar o modificar los nodos creados en la estructura dinámica. Pero, **¿qué detalles debemos proteger ante posibles cambios en la estructura?**

4.3.3.1 Recarga de datos

Como hemos comentado, el primer asunto a tratar es el de la recarga de datos. Si bien es cierto que en otros apartados ya comentamos el mecanismo por el cuál el manager de RedBorder comunica cuando y cómo modificarlos, no se detalla cómo nuestra herramienta realiza internamente este proceso.

Como ya se comentó, el parámetro **modif,** dentro de cada elemento del fichero de monitorización, determina si la nueva información ha de ser creada, modificada o eliminada (ver **Fig. 42**).

4.3.3.1.1 Modificación

El proceso de modificación de los datos es trivial ya que únicamente consiste en modificar el valor de los parámetros dentro de la estructura.

Aun así cabe especial mención al proceso de incio de sesión con net-snmp. En el caso de modificación de los parámetros en la estructura **st_host**, la cual tenía los que vemos en **Code 19**, debemos modificar la sesión.

```
typedef struct host {
    int id;
    char * name;
    char * ip;
    char * community;
    int version;

    int corte;
    int fallo_sesion;

    struct snmp_session sesion;
    struct snmp_session *punt_sesion;

    st_oid *oids;

    struct host *next;
}st_host;
```

Code 27 Estructura st_host

Estos parámetros están relacionados con la sesión net-snmp abierta incialmente. Ante esta ocurrencia la solución adoptada pasa por cerrar la sesión abierta incialmente.

Un vez cerrada la sesión debemos de indicar al hilo encargado de la monitorización de este host que debe de realizar de nuevo el proceso de apertura de sesión con los nuevos datos.

Para ello está incorporada a la estructura la variable **fallo_sesion**.

Esta tendrá fundamentalmente dos usos. El primero de ellos será el de indicar a los hilos que el incio de sesión ha fallado, y que debe de reintentarlo en cada iteracción del bucle de monitorización. Este mismo uso nos sirve para el caso que nos ocupa. Al cerrar la sesión por modificación de sus parámetros, activamos esta variable y los hilos del motor de envío de peticiones, en cada iteracción del bucle de monitorización, intentará abrir la sesión, ahora ya con los nuevos parámetros.

4.3.3.1.2 Insercción

El aspecto más importante a tratar desde el punto de vista de la inserción es, **¿en qué punto añadimos los nuevos nodos?**

Vamos a detallar cómo se realiza el proceso de inserción tanto para un host como para un oid nuevo.

4.3.3.1.2.1 Inserción de un nuevo host

En el momento del parseo del fichero de monitorización, si el parámetro **modif** se encuentra a 1 y no encontramos un nodo con el **id** igual al del que estamos leyendo, debemos d ecrear un nuevo nodo con los parámetros leído e insertarlo a la lista enlazada de hosts.

Para la inserción de este nuevo host, programamos la función

void anida_host (st_host * nodo)

Esta función distingue dos momentos del programa. En el inicio esta herramienta no tomará ningún criterio en la inserción de los nodos, sino que los irá enlazando en primera posición. En cambio, una vez tengamos la estructura en memoria se realiza un proceso previo a la inserción.

En **Code 20** vemos que se realiza una "cuenta" antes de la inserción. Se cuenta el número de nodos antes de cada corte, lo que podemos traducir como la carga en el número de hosts de cada hilo.

Tener en cuenta de que, en el caso comentado de que dispusiéramos de más hilos que hosts en la estructura, al acceder a la tabla de **acceso_hilos** encontraría NULL para hilos sin carga, y tomaría estos como los hilos menos cargados.

Una vez localizado el hilo al cuál le queremos añadir el nuevo nodo, lo insertamos al principio. Esto implica que debemos modificar la tabla **acceso_hilos**, poniendo este nuevo nodo en la posición del hilo que le corresponde.

De esta forma realizamos una inserción de nuevos nodos respetando el balanceo de carga entre hilos, y sin romper el funcionamiento de este mediante las variables corte y la tabla de acceso de los hilos.

```
//Si la estructura ya esta en memroia
if(estruct_memoria==1) {
    //Calculamos cual es el hilo con menos carga
    //-->i=0
    id_menos_cargado = 0;
    while (host_prov!=NULL && host_prov->corte!=1) {
        hilo_menos_cargado++;
    }

    //-->i=1...num_hilos
    for (i=1; i<num_hilos; i++){
        prov = acceso_hilos[i];
        while (prov!=NULL && prov->corte!=1) {
            carga_hilo_actual++;
        }
        if (carga_hilo_actual<hilo_menos_cargado) {
            hilo_menos_cargado = carga_hilo_actual;
            id_menos_cargado = i;
        }
    }

    //Lo insertamos al inicio
    auxiliar = acceso_hilos[id_menos_cargado];
    acceso_hilos[id_menos_cargado]=nodo;
    nodo->next = auxiliar;
}
```

Code 28 Insercción de host

4.3.3.1.2.2 Inserción de un nuevo OID

Este caso es más sencillo que el anterior. Hay que recordar que la lista de OIDs no viene dividida por necesidades de balancear la carga, luego no incluye variables de corte ni tabla de acceso para los hilos. El hilo que monitorice un host, monitorizará al completo su lista de OIDs.

Su función análoga al caso de los host es

void anida_oid (st_host * nodo, st_oid * oid)

En este caso la inserción se realiza al inicio de la lista, debiéndo modificar únicamente el parámetro que apunta a esta en la estructura del host.

```
void anida_oid (st_host * nodo, st_oid * oid) {
    st_oid * auxiliar = nodo->oids;
    nodo->oids = oid;
    oid->next = auxiliar;
}
```

Code 29 Inserción de OID

4.3.3.1.3 Eliminación

El caso de la eliminación de un nodo tiene implícitas ciertas precauciones de obligado cumplimiento.

En el caso de un host, la función diseñada para ello es

void elimina_nodo_completo(st_host * elimina)

y las precauciones son:

- **El host tiene la variable a corte activada**

  Esto quiere decir que este nodo es el último en el bucle de monitorización de un hilo, luego si eliminamos este nodo sin tener en cuenta esto, es posible que los mecanismos de reparto de carga planteados a través de las variables corte deje de funcionar y varios nodos lancen a la red la misma petición.

  En este caso la solución es sencilla. Recorremos la tabla buscando el nodo, almacenando siempre el

puntero al nodo anterior. Si al encontrarlo, este nodo tiene la variable corte a 1, colocamos a 1 la variable cote del nodo anterior, pudiénso ahora sí eliminar este nodo sin riesgo de perder la partición de la estructura.

Ocurre un situación particular. **¿Y si el nodo anterior también tiene la variable de corte activada?**

Si entendemos este caso, vemos claro que esto sólo se da cuando hay un hilo que sólo monitoriza un host y es precisamente este host el queremos monitorizar. Se puede entender fácilmente como en este caso, el nodo que queremos eliminar aparecerá también en la tabla de accesos. Es por esto por lo que resolveremos esta cuestión en el siguiente punto.

- **El host se encuentra en la tabla de accesos**

  Si en la búsqueda del host vemos que pertenece a una posición de la tabla de acceso, la solución es similar al caso anterior, se almacena en la tabla de acceso el siguiente nodo de la estructura.

  Aquí ocurre una situación particular que enlaza con la mencionada en el caso anterior. Si estamos en el caso que queremos eliminar el único host que monitoriza un hilo, ocurre lo siguiente:

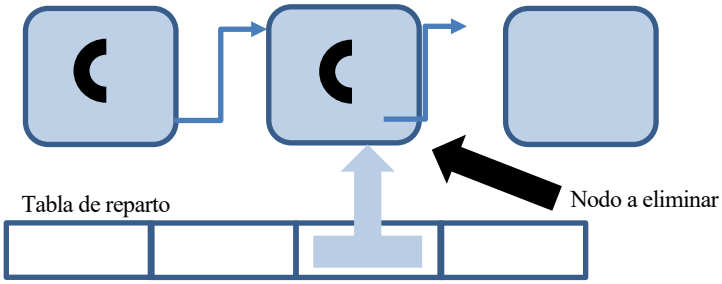


Fig. 50 Caso particular de eliminación

En este caso esquematizado en la **Fig. 47** vemos lo que comnetábamos en el punto anterior. Si aparecen dos cortes consecutivos es porque el segundo corte pertenece a un nodo único en un hilo.

En este caso especial de eliminación basta con poner a NULL la tabla de reparto en la posición en la que aparece el nodo y elimar a este de manera normal, sin modificar el corte de ningún nodo vecino. Este hilo por tanto quedará vacío, pero tendrá más prioridad a la hora de la inserción de nuevos nodos como vimos en el apartado **4.3.2.1.2.21 Insercción de un host**.

- **Es el primer nodo de la estructura dinámica**

  Adicionalmente a las precauciones que puede llevar implícitas el primer nodo de la lista enlazada (como ener la variable corte a 1 o aparecer en la tabla de accesos), el único aspecto de especial mención es el de modificar el putero global a esta lista antes de eliminar el nodo. En el caso de OID modificar el puntero a la estructura en el nodo del host.

4.3.3.2 Balanceo de carga ante cambios

Como hemos comentado en el punto anterior, el proceso de recarga de la configuración tiene presente el balanceo de carga propuesto por la tabla de acceso a hilos y las variables corte en la estructura. Tanto la insercción de nuevos nodos tiene en cuenta el hilo más descargado, como la eliminación, que adopta las precauciones para no corromper la memoria. No hace necesario añadir ningún punto adicional ante la recarga de la configuración.

4.4 Integración

En los puntos anteriores hemos habado del uso de **net-snmp** para la monitorización, del intercambio de datos con RedBorder, tanto de configuración a través de archivos **JSON**, como de envío de datos resultantes de la monitorización a través de **Apache Kafka** y de todo el proceso interno para el **manejo de datos**.

Una vez detallada cada parte de la herramienta vamos a proceder a aportar una vision global de esta, aclarándo cómo los diferentes módulos se relacionan entre sí.

Para ello nos vamos a servir de un esquema general, al cuál iremos "hacienda zoom" en aquellas partes que nos interesa prestar especial atención.

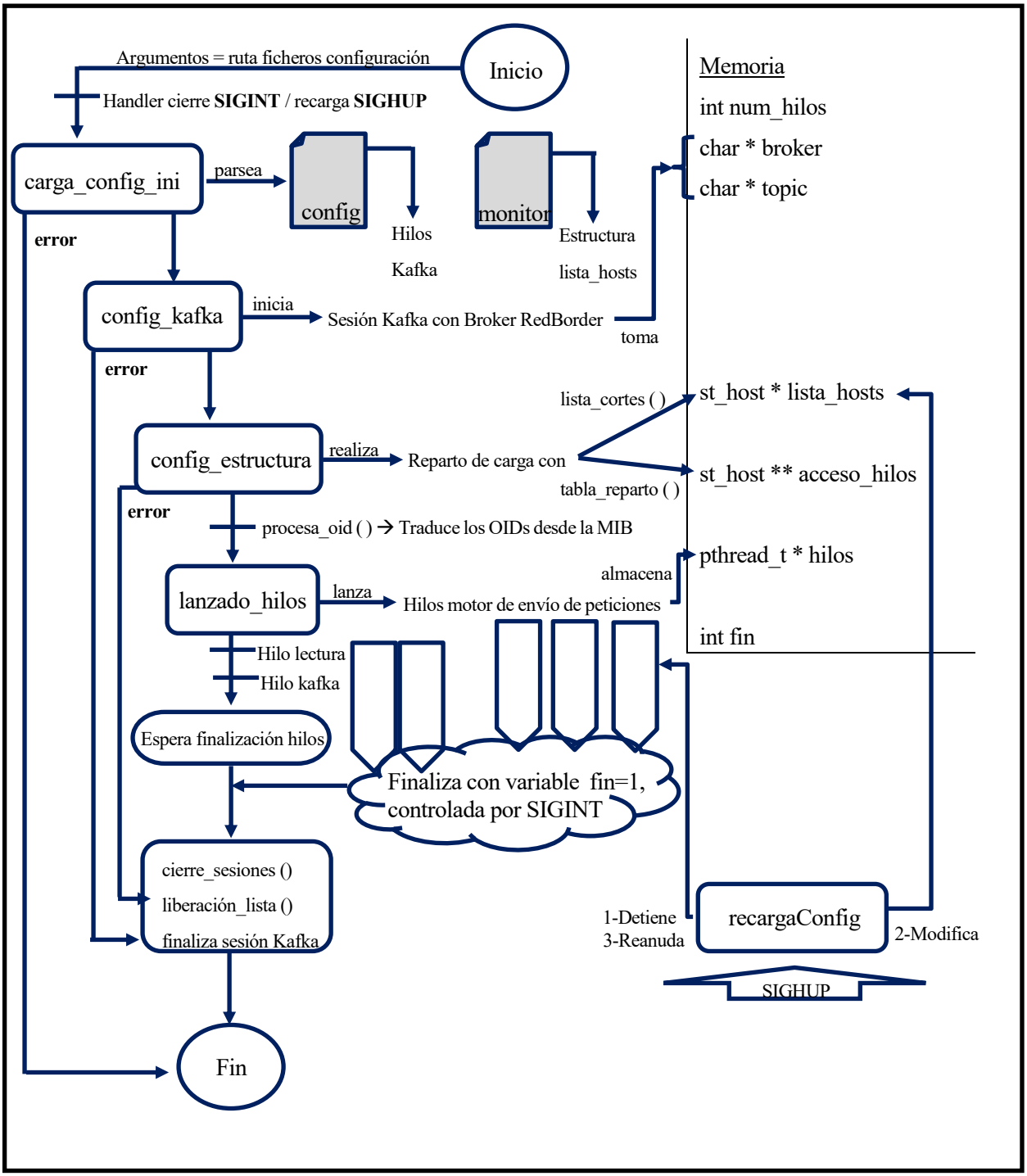


Fig. 51 Esquema general de la herramienta

En la **Fig. 48** tenemos el esquema general del funcionamiento de la herramienta. A continuación vamos a hacer mención a determinados aspectos de integración entre partes que necesitan ser detallados.

Podemos diferenciar tres fases dentro de la ejecución de la herramienta:

1- Configuración incial

2- Ejecución de los hilos

3- Cierre

4.4.1 Configuración inicial

El proceso de configuración incial se realiza en el momento de la ejecución. Se toman los dos argumentos, que son los ficheros de configuración, y con la llamada a la función **carga_confi_ini** se realiza una primera carga de estos ficheros.

En el apartado **4.2.1 Recepción de parámetros de configuración general** y **4.2.2 Recepción de parámetros de monitorización** se detalla éste proceso de parseo y obtención de datos, al igual que su organización en memoria.

Como ya vimos, una vez realizado éste proceso de parseo de los ficheros de configuración, tendremos como resultado una estructura enlazada en memoria con todos los datos para la monitorización proporcionados por RedBorder.

Adicionalemente, en esta primera fase de la herramienta se "abrirá" la conexión con el broker de Apache Kafka indicado por RedBorder. En el apartado **4.2.3 Apache Kafka** se detalla todos los pasos necesarios para incializar las estructuras usadas para el envío de mensajes a determinado topic en el broker de Kafka.

Por ultimo y antes de ejecutar los motores de monitorización, debemos de realizar un reparto equitativo de la carga, ya que el funcionamiento del motor de envío de peticiones es multi hilo, y funcionará sobre la misma estructura enlazada. Para ello nos servimos de un mecanismo de reparto y corte sobre la estructura como se detalla en el apartado **4.3.1 Reparto de carga**.

4.4.2 Ejecución de hilos

Una vez realizado el proceso de configuración incial, debemos lanzar los motores. Lanzamos los hilos que ejecutarán el motor de envío de peticiones, programados a través de la función **poller**.

```
 * Funcion: void * poller (void * thread_args)
void * poller (void * thread_args)
{

    //ID numerico usado para acceder a la tabla de reparto
    int id hilo = (int)thread args;
```

Code 30 Función poller

Esta función **poller**, que será la ejecutada por cada hilo recibirá un valor numérico como argumento, que servirará para identificar a este hilo de cara al reparto de carga. Estos hilos tomarán los datos de monitorización de la estructura enlazada para realizar el envío de peticiones usando de base **net-snmp** como se ha visto en el apartado **4.1.1.1 Motor de envío de peticiones**.

El reparto de la carga se hará controlando el punto de incio y fin donde cada hilo ha de monitorizar. El inicio será controlado por la tabla **acceso_hilos,** y el fin mediante variables cortes en la estructura. Todo ello queda detallado en el apartado **4.3.1 Reparto de carga**.

El motor de recepcicón de respuestas monitorizará el descriptor de ficheros haciendo que se ejecuté la función de callback de la sesión (ver apartado **4.1.1.1.1 Inicio de sesión**). Esto constará de un único hilo que realizará las funciones detalladas en el apartado **4.1.1.2 Motor de recepción de PDU**.

Por último, se hace necesario un hilo de Kafka que haga sucesivos **poll** como ya vimos en el apartado **4.2.3 Apache Kafka**. La ejecución de este hilo es trivial:

```
void * poll_kafka (void * thread_args) {
    while (fin) {
        /**
        * Returns the current out queue length:
        * messages waiting to be sent to, or acknowledged by, the broker
        */
        if (rd_kafka_outq_len(rk) > 0)
            rd_kafka_poll(rk, 0);
    }
    /*
     * Cuando llegamos a fin, tenemos que terminar de
     * hacer poll a los que quedan
     */
    while (rd_kafka_outq_len(rk) > 0) {
        rd_kafka_poll(rk, 0);
    }
    return 0;
}
```

Code 31 Hilo de polls de Kafka

4.4.3 Cierre

A través del manejador definido para la señal SIGINT, que es la enviada al proceso principal cuando desde la terminal se detecta **Ctrl+C**, indicamos a los hilos lanzados que deben de realizar una finalización segura.

Los hilos del motor de envío de peticiones finalizarán el bucle de monitorización, es decir, en cada iteracción comprobará que la variable **fin** no esté activada. El hilo de lectura y de Kafka realizará el mismo proceso, esperando a finalizar la iteracción en marcha para terminar.

Una vez terminado los hilos y antes de terminar el programa, se debe de cerrar las seisones snmp mediante la función **cierre_sesiones**, y posteriormente eliminar todos los elementos reservados dinámicamente. Estos son:

- Lista enlazada de hosts y OIDs
- Tabla de reparto
- Tabla que almacena los punteros a los hilos

Por último, lo restante será cerrar y liberar los recursos reservados para la sesión de Apache Kafka.

Este proceso es el que llamamos cierre seguro de la herramienta y esta descrito en el apartado **4.3.3 Cierre y liberación de datos**.

5 PUNTOS DE MEJORA

Iniciamos nuestra solución particular, que fue la de desarrollar desde la base una herramienta de monitorización, sabiendo que los objetivos del proyecto de verían acortado con respecto a las expectativas iniciales.

En el incio contábamos con que **Zabbix** nos proporcionara la solución de monitorización, pasando así a aspectos como la integración con RedBorder o la adaptación a un trabajo distribuido con Zookeeper. El tener que partir desde una posición de cero trabajando desde los aspectos más fundamentales de la monitorización hasta el procesado eficiente de datos, hace evidentemente que el tiempo pensado para el proyecto deje poco margen para trabajar en todos los aspectos planteados.

El hecho de dedicar tiempo a partes fundamentales de la herramienta, como puede ser la base de la monitorización o el manejo interno eficiente de los datos, nos hizo recortar en otras. A continuación vamos a detallar los diferentes puntos de mejora de la herramienta, la mayoría de ellos no llevados a cabo por falta de tiempo.

5.1 Monitorización SNMP

5.1.1 SNMPv3

Como hemos comentado, el motor de envío de peticiones trabaja con SNMP y SNMPv2. Por ello se deja pendiente el desarrollo de un motor de peticiones capaz de enviar consultas SNMPv3.

SNMPv3 incorpora principalmente un cifrado extremo a extremo en la conexión SNMP. El trabajo principal para incorporar esta funcionalidad es el de añadir la información necesaria para el cifrado en la estructura de inicio de sesión.

Tabla 10 Parámetros adicionales para **snmpv3**

Parameter	Command Line Flag	snmp.conf token
securityName	-u *NAME*	defSecurityName NAME
authProtocol	-a *(MD5|SHA)*	defAuthType (MD5|SHA)
privProtocol	-x *(AES|DES)*	defPrivType DES
authKey	-A *PASSPHRASE*	defAuthPassphrase PASSPHRASE
privKey	-X *PASSPHRASE*	defPrivPassphrase PASSPHRASE
securityLevel	-l *(noAuthNoPriv|authNoPriv|authPriv)*	defSecurityLevel (noAuthNoPriv|authNoPriv|authPriv)
context	-n *CONTEXTNAME*	defContext CONTEXTNAME

Estos parámetros deberán ser proporcionados por el manager, luego los efectos en la herramienta serían:

- Modificación del motor de envío de peticiones, en concreto de la función encargada de la apertura de sesión **SNMP**, dando la posibilidad de poder abrir con el host destino una sesión SNMPv1, SNMPv2c o SNMPv3
- El punto más importante afecta a la recepción de datos por parte del manager. La información que se recibe por parte del manager viene en formato JSON como ya se ha comentado. Se hace necesario que la lectura del fichero de monitorización sea flexible, ya que se necesita incorporar información adicional si queremos SNMPv3.

Como observamos, añadir SNMPv3 a nuestra herramienta no es trivial, pero tampoco un punto complejo. Aun así realizando una gestión del tiempo y observando los objetivos del proyecto, se dejó este punto como una funcionalida prescindible ya que, tras la reunión con los encargados del proyecto se indicó que en la red del proyecto **SIGMONA** no íbamos a encontrar **SNMPv3.**

5.1.2 Traps SNMP

Además, una posible funcionalidad para la monitorización SNMP podría ser la de habilitar a nuestra herramienta de la capacidad de recibir y procesar **TRAPs SNMP**.

Una trap es generado por el agente para reportar ciertas condiciones y cambios de estado a un proceso de administración[73]

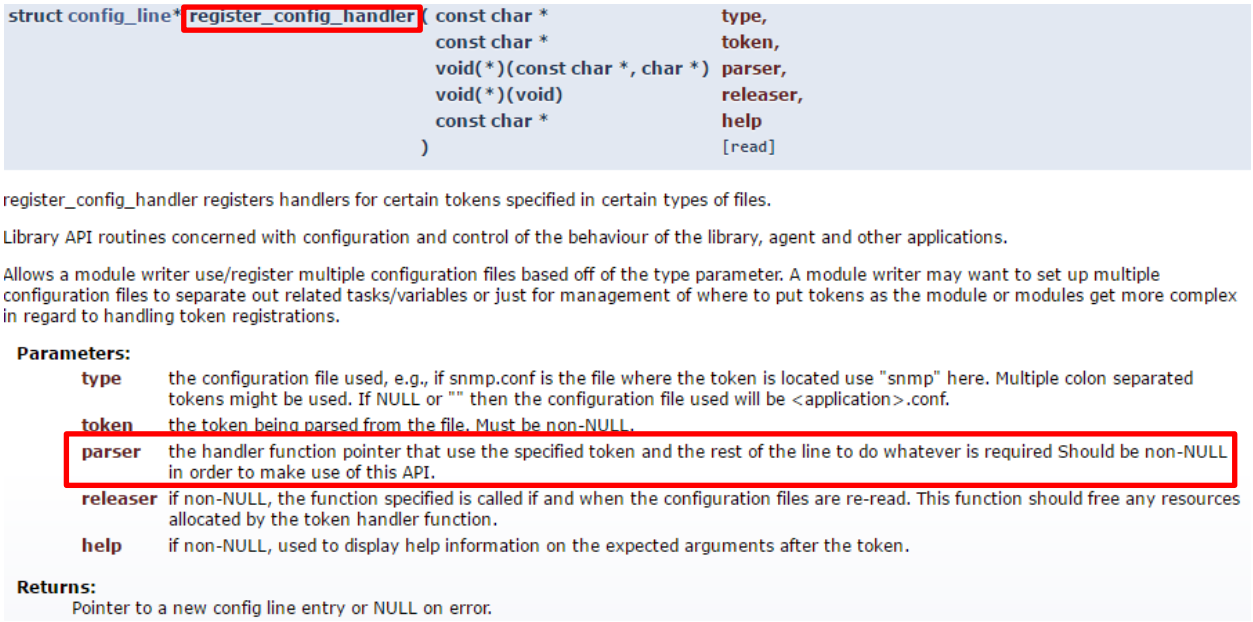
Si entendemos el proceso por el cual nuestra herramienta realiza el proceso de envío/recepción de PDUs podemos entender cómo añadir la funcionalidad de la recpción de TRAPs supone un desarrollo mucho mayor que en el caso de permitir SNMPv3.

[73] Fuente: https://es.wikipedia.org/wiki/Simple_Network_Management_Protocol#Trap

Para añadir esta funcionalidad, desde **net-snmp** se nos proporciona lo siguiente:

register_config_handler ( )

Como podemos ver desde la documentación de **net-snmp**[74]



struct config_line* **register_config_handler** (const char * **type**,
const char * **token**,
void(*)(const char *, char *) **parser**,
void(*)(void) **releaser**,
const char * **help**
) [read]

register_config_handler registers handlers for certain tokens specified in certain types of files.

Library API routines concerned with configuration and control of the behaviour of the library, agent and other applications.

Allows a module writer use/register multiple configuration files based off of the type parameter. A module writer may want to set up multiple configuration files to separate out related tasks/variables or just for management of where to put tokens as the module or modules get more complex in regard to handling token registrations.

Parameters:

type	the configuration file used, e.g., if snmp.conf is the file where the token is located use "snmp" here. Multiple colon separated tokens might be used. If NULL or "" then the configuration file used will be <application>.conf.
token	the token being parsed from the file. Must be non-NULL.
parser	the handler function pointer that use the specified token and the rest of the line to do whatever is required Should be non-NULL in order to make use of this API.
releaser	if non-NULL, the function specified is called if and when the configuration files are re-read. This function should free any resources allocated by the token handler function.
help	if non-NULL, used to display help information on the expected arguments after the token.

Returns:
Pointer to a new config line entry or NULL on error.

Fig. 52 Documentación de net-snmp sobre register_config_handler

A través de esta función somos capaces de registrar un "handler" o manjeador para determinado tipo de evento. En concreto:

register_config_handler ("snmptrapd", "traphandle", snmptrapd_traphandle, NULL, "oid|\"default\" program [args ...] ")

A partir de éste registro y añadiéndo la función **snmptrapd_traphandle ()** podemos recibir los traps, procesar la PDU y aplicar el mismo procesado, extracción e incorporaciónal topic de Kafka que ya hacemos en nuestra herramienta.

A pesar de haber registrado un manejador, no hemos indicado la sesión a la que corresponde dicho TRAP. Para ello debemos de realizar una apertura con el host origen de los traps diferente a lo visto en el apartado **4.1.1.1.1 Inicio de sesión**.

```
void * snmptrap (void *argp)
{
  Config               *conf = (Config *) argp;
  netsnmp_transport    *transport = NULL;
  netsnmp_session      *ss = NULL;
  char                 listen[32];

  snprintf(listen,sizeof(listen),"%s:%u", get_ipaddr(conf->listen_addr),conf->snmptrap_port);
  transport = netsnmp_tdomain_transport(listen, 1, "udp");
  if (transport == NULL) {
    syslog(LOG_ERR, "SNMP trap: couldn't open %s -- errno %d (\"%m\")\n",
                    listen, errno);
    SOCK_CLEANUP;
    exit(1);
  }
  ss = snmptrapd_add_session(transport);
```

dirección / puerto

Code 32 Inicio de sesión para la recepción de TRAPs

Como vemos en **Code 23**, el inicio de session se hace de una manera ligeramente diferente a como hemos hecho en el motor de envoi de peticiones. La idea principal es la de ir añadiendo sesiones iniciadas a partir de la dirección y puerto de donde se reciben los traps.

[74] http://net-snmp.sourceforge.net/dev/agent/group__read__config.html#ga9a3e481d8eb7d3ef08efa25b9a186c3b

El string con la dirección y puerto será pasado a la función

netsnmp_tdomain_transport ()

Las cual devuelve un puntero a la estructura ya inicializada, estructura de tipo **netsnmp_transport**. No es objeto de este apartado profundizar en la API de **net-snmp** sino ofrecer una solución a los puntos que han quedado pendientes, dejándolos así documentados para futuras modificaciones.

Como vemos en **Code 23** y **Code 24**, esta estructura define una sesión de recepción de TRAPs, que será abierta usando la función **snmptrapd_add_session**, la cual si usará la apertura de sesión como estudiamos en el apartado **4.1.1.1.1 Inicio de session.**

```
static netsnmp_session * snmptrapd_add_session(netsnmp_transport *t)
{
    netsnmp_session sess, *session = &sess, *rc = NULL;

    snmp_sess_init(session);
    session->peername = SNMP_DEFAULT_PEERNAME;  /* Original code had NULL here */
    session->version = SNMP_DEFAULT_VERSION;
    session->community_len = SNMP_DEFAULT_COMMUNITY_LEN;
    session->retries = SNMP_DEFAULT_RETRIES;
    session->timeout = SNMP_DEFAULT_TIMEOUT;
    session->callback = snmp_input;
    session->callback_magic = (void *) t;
    session->authenticator = NULL;
    sess.isAuthoritative = SNMP_SESS_UNKNOWNAUTH;
```

Code 33 Inicio de sesión a partir de la estructura netsnmp_transport

De esta forma, y registrando un handler, iniciando la sesión y monitorizando el descriptor de ficheros a la espera de PDus entrantes damos solución a la recepción de los TRAPs.

Al igual que en el caso anterior el punto que mayor modificación require es el de aportar flexibilidad al fichero de monitorización, que debe indicarnos qué traps desea recibir.

La herramienta incialmente está pensada para funcionar en cualquier ámbito, ya que toda la información proviene de otro proceso central. Por esto se deja este apartado como posible implementación para futuros proyectos.

5.2 Trabajo con Zookeeper

Como aplicación distribuida que es, una idea de este proyecto era la de adaptarla al uso con Zookeeper[75], el software de **Apache** para la coordinación de servicios distribuidos.

De esta manera podráimos tener varias herramientas monitorizando sobre la misma red, haciéndo así nuestra solución algo más escalable.

Como ya hemos comentado, los objetivos del proyecto se acortaron debido al desarrollo de la herramienta desde los pilares más básicos, pero, **¿es realmente necesaria esta integración?**

A pesar de ser un objetivo inicial del proyecto, el no integrar el uso de Zookeeper no impide que se ejecute la

[75] Web oficial de Apache Zookeeper: https://zookeeper.apache.org/

misma herramienta en varios servidores sobre la misma red.

El haber diseñado una herramienta con una total dependencia de los datos de configuración por parte del manager, a pesar de complicar muchos aspectos en el desarrolllo, una vez finalizado nos aporta una gran flexibilidad.

Desde el manager de RedBorder se puede realizar un reparto de carga entre las diferentes herramientas que se ejecutan en la red. Para ello nos serviremos de los ficheros de configuración diseñados y detallados en **4.2.2 Recepción de parámetros de monitorización**.

No es objetivo de éste proyecto el trabajo en el manager, sino el de diseñar un sensor SNMP. La flexibilidad aportada gracias a los ficheros de configuración hace que desde el manager se pueda controlar la carga de datos a monitorizar al igual que el uso de recursos por parte de la herramienta. Esto significa que no hay necesidad implícita en el uso de Apache Zookeeper.

Aun así, es metodología de RedBorder el trabajo con esta herramienta para las aplicaciones distribuidas y queda como posible mejora a este proyecto. Ya en **4.2.2.3.1 SIGHUP para la recarga de la configuración**, detallamos que la programación de la recarga de la configuración a tarvés de la recepción de la señal de SIGHUP se realize teniendo en cuenta una daptación con Zookeeper.

6 PRESUPUESTO

Presupuesto			
Concepto	Coste / ud	Total unidades	Total
Ingeniero Junior Telecomunicaciones ▪ **Investigación** ▪ **Desarrollo** ▪ **Integración** ▪ **Tests y mejora continua** ▪ **Documentación**	 13€/hora 13€/hora 13€/hora 13€/hora 13€/hora	 65 horas 200 horas 125 horas 50 horas 150 horas	 845€ 2.600€ 1.625€ 650€ 1.950€
Puesto trabajo ▪ **Equipo portátil**[76] ▪ **Monitor** [77] ▪ **Otros**	 700€ 150€ 100€	 1 2 1	 700€ 300€ 100€
Total:			**8.770€**

[76] Referencia: http://www.pccomponentes.com/asus_f556uj_xo010t_intel_core_i7_6500u_8gb_1tb_gt_920m_15_6_.html
[77] Referencia: http://www.pccomponentes.com/benq_gl2460hm_24__led_multimedia.html

7 CONCLUSIONES

Volviendo la vista atrás, observando los errores, los puntos dónde el tiempo no se gestionó adecuadamente, las decisiones tomadas… con lo que más me quedo es con lo aprendido.

La solución incial propuesta, la adaptación de Zabbix, plantaba en el proyecto una base sólida de monitorización sobre la que trabajar. Empezar un proyecto de estas dimensiones era algo nuevo para mí, luego Zabbix tenía que funcionar.

Tras un tiempo de instalación, comprensión del funcionamiento, trabajo con la API y hasta con el código fuente de la herramienta, se empezó a considerar que quizás, la solución planteada, no era la más adecuada y que antes de seguir con el proyecto debíamos hacer un trabajo de investigación previo.

El hecho de desmontar el trabajo realizado con Zabbix no supuso mucho, ya que el trabajo fue principalmente teórico, planteando diferentes vías de integración sin llegar a realizar nada. El problema llegados a este punto fue el de pérdida de confianza. Cómo iba a ser capaz de llevar este proyecto adelante si después de más de un mes volvíamos al punto de partida.

El trabajo de investigación y la toma de decisiones es una de las mejores experiencias profesionales que he

tenido durante la carrera. Aplicar los conocimientos, el método lógico adquirido durante el transcurso de la carrera aplicado a la toma de una decisión para un proyecto de este calibre, que aún no siendo un gran proyecto, en esos momentos yo lo consideraba así.

Finalmente, el desarollar una herramienta desde cero asienta los conocimientos de programación adquiridos durante la carrera. Dista mucho la realización de una práctica guiada, de trabajar en un proyecto software de este nivel. Abrir un documento blanco y empezar a pensar y a modular el proyecto, estableciendo pasos, objetivos… sin lugar a dudas es una gran experiencia a nivel técnico.

Y todo esto, además de los conocimentos adquiridos, queda la satisfacción al finalizar el proyecto. A pesar de no cumplir los objetivos inicialmente planteados, el haber superado los momentos de desconfianza tras la marcha fallida de Zabbix, tener la herramienta finalizada con los objetivos generales cumplidos es sin duda una gran satisfacción.

Como punto final decir que este proyecto ha reafirmado mi posición, quiero dedicar mi vida profesional a la seguridad. Lo vivido en ENEO Tecnología durante la realización del proyecto ha asentado mi pasión por la profesión. Es cierto que este proyecto se basa más en desarrollo software que en seguridad de la información, pero es esto precisamente lo que afirma mi posición. Entiendo el desarrollo software como una parte fundamental del trabajo, pero el objetivo en mi vida profsional va a ser la formación continua para poder tomar parte de las decisiones de éste.

8 REFERENCIAS

[1] N. SA, «Nextel,» 2012. [En línea]. Available: http://www.nextel.es/idi2/sigmona.

[2] Z. Group, «Zabbix Documentation - Zabbix Concepts,» [En línea]. Available: https://www.zabbix.com/documentation/3.0/manual/concepts.

[3] Z. Group, «Zabbix Documentation - Zabbix proxy,» [En línea]. Available: https://www.zabbix.com/documentation/3.0/manual/concepts/proxy.

[4] Z. Group, «Zabbix Documentation - Comparacion Server - Proxy,» [En línea]. Available: https://www.zabbix.com/documentation/3.0/manual/distributed_monitoring .

[5] Z. Group, «Zabbix Code,» [En línea]. Available: http://www.zabbix.com/download.php.

[6] Z. Group, «Zabbix Documentation - Cracion de base de datos,» [En línea]. Available: https://www.zabbix.com/documentation/3.0/manual/appendix/install/db_scripts .

[7] H. -. P. user, «Perlmonks - RFC for SNMP monitoring,» [En línea]. Available: http://www.perlmonks.org/?node_id=662528.

[8] H. -. P. user, «Update of RFC,» [En línea]. Available: http://www.perlmonks.org/?node=664360.

[9] N.-s. Group, «Net-SNMP en Perl,» 26 Mayo 2011. [En línea]. Available: http://net-snmp.sourceforge.net/docs/perl-SNMP-README.html .

[10] m. -. G. user, «GitHub - Rated,» [En línea]. Available: https://github.com/mprovost/rated.

[11] H. -. G. user, «GitHub - Ruby-snmp,» [En línea]. Available: https://github.com/hallidave/ruby-snmp.

[12] N.-s. Group, «Pagina oficial,» [En línea]. Available: http://www.net-snmp.org/.

[13] N.-s. Group, «Aplicación asíncrona,» [En línea]. Available: http://www.net-snmp.org/wiki/index.php/TUT:Simple_Async_Application.

[14] IETF, «RFC 1157,» [En línea]. Available: https://tools.ietf.org/html/rfc1157.

[15] a. -. G. user, «Jannson library,» [En línea]. Available: http://www.digip.org/jansson/.

[16] A. Foundation, «Apache Kafka,» [En línea]. Available: http://kafka.apache.org/.

[17] E. -. G. user, «Librdkafka library,» [En línea]. Available: https://github.com/edenhill/librdkafka.

9 ANEXOS

A continuación vamos a hacer un detallado del código final de la herramienta donde se encontrarán todo lo descrito a lo largo de esta memoria. Veremos desde la estructura de ficheros hasta cada función programada para los objetivos del proyecto.

El código se encuentra subido como proyecto en la plataforma GitHub[78] de forma que todas las mejoras planteadas en el punto **5** de esta memoria, puedan ser implementadas de forma sencilla para futuros usos de esta herramienta por parte de terceros.

A continuación insertamos el código estable de la herramienta:

[78] Proyecto en GitHub: https://github.com/albjodbor/kafkaSNMPpoller

9.1 Anexo 1: Estructura del proyecto

A continuación indicamos las funciones del proyecto, que se clasifican en los diferentes ficheros:

- carga.c y carga.h
  - carga_config()
  - carga_config_ini()
- config.c y config.h
  - cierre_sesiones()
  - config_estructura()
  - config_kafka()
  - lanzado_hilos()
  - lista_cortes()
  - msg_delivered()
  - procesa_oid()
  - table_reparto()
- datos.h
  - st_host
  - st_oid
- estructura.c y estructura.h
  - anida_host ()
  - anida_oid()
  - busca_nodo()
  - busca_oid()
  - copia_cadena()
  - elimina_nodo_completo()
  - elimina_oid_completo()
  - imprime_cortes()
  - imprime_lista()
  - libera_lista_oid()
  - libera_nodo()
  - libera_oid()
  - liberacion_lista()
  - LongitudCadena()
- hilos.c y hilos.h
  - hilo_lectura()
  - poll_kafka()
  - poller()
  - print_result()

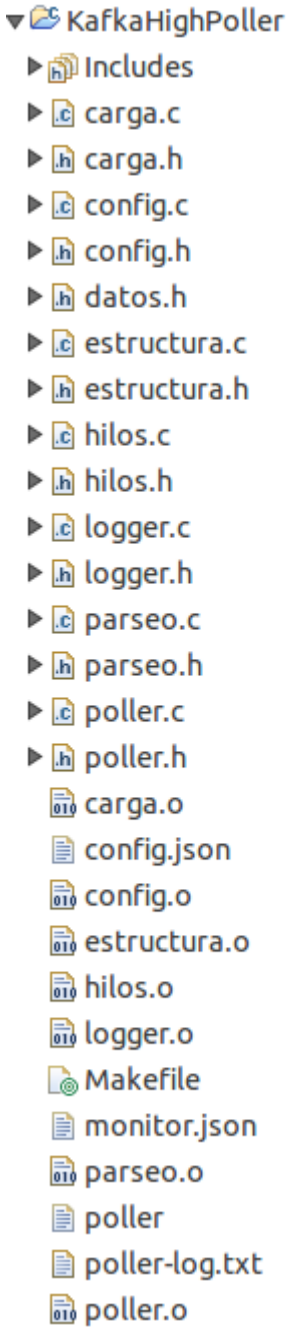


Fig. 53 Esquema de ficheros del proyecto

    - procesa_pdu()
    - produce_msg()
    - productor_kafka()
- logger.c y logger.h
    - LOG_PRINT()
- parseo.c y parseo.h
    - carga_conf()
    - carga_monitor()
- poller.c y poller.h
    - **main()**
    - handlerCierre()
    - recargaConfig()

Makefile

```
CC = gcc
CFLAGS= -g
PROG = poller
HEADERS = poller.h datos.h estructura.h logger.h hilos.h parseo.h config.h carga.h
PROGO = poller.o estructura.o logger.o hilos.o parseo.o config.o carga.o
LIB = -lpthread -lrdkafka -lz -lrt -ljansson
BUILDLIBS=`net-snmp-config --libs`

default: $(PROG)

$(PROG): $(PROGO)
	$(CC) $(CFLAGS) -o $(PROG) $(PROGO) $(LIB) $(BUILDLIBS)

$(PROG).o: $(PROG).c $(HEADERS)
	$(CC) $(CFLAGS) -c poller.c $(LIB) $(BUILDLIBS)

estructura.o: estructura.c estructura.h datos.h
	$(CC) $(CFLAGS) -c estructura.c $(LIB) $(BUILDLIBS)

logger.o: logger.c logger.h datos.h
	$(CC) $(CFLAGS) -c logger.c $(LIB) $(BUILDLIBS)

hilos.o: hilos.c hilos.h datos.h
	$(CC) $(CFLAGS) -c hilos.c $(LIB) $(BUILDLIBS)

parseo.o: parseo.c parseo.h datos.h
	$(CC) $(CFLAGS) -c parseo.c $(LIB) $(BUILDLIBS)

config.o: config.c config.h datos.h
	$(CC) $(CFLAGS) -c config.c $(LIB) $(BUILDLIBS)

carga.o: carga.c carga.h datos.h
	$(CC) $(CFLAGS) -c carga.c $(LIB) $(BUILDLIBS)

clean:
	$(RM) *.o $(PROG) *~
```

9.2 Anexo 2: Ficheros de configuración y estructura de datos

carga.c

```
#include "carga.h"
#include "logger.h"

/*
 * Funcion: int carga_config_ini()
 *          Carga configuracion al inicio.
 *          Funcion que llamara a los parseadores de los
 *          ficheros de configuracion e intentara cargar la
 *          configuracion hasta que sea correcta
 *          NO CONTEMPLA la lista vacia
 */
int carga_config_ini()  {
     //Valor devuelto
     int error = 0;
     int reintentos = 0;

     //--------------CONFIGURACION--------------
     LOG_PRINT("Intento cargar configuracion general");
     //Bucle funcion que lee la configuracion
     do {
            LOG_PRINT("Carga general n° %d",reintentos);
            error_confic = 0;
            //Carga la configuracion general -> Guarda en variables poller.h
            carga_conf();
            reintentos++;
     } while ((fin==0)||(reintentos >= 5));

     if (error_confic != 1)
            LOG_PRINT("Configuracion general cargada correctamente");
     else {
            LOG_PRINT("Fallo en carga de configuracion general");
            error=1;
     }

     //Si ha dado fallo, error = 1 y finalizo
     if (error != 1) {
            LOG_PRINT("Configuracion general:");
            LOG_PRINT("Numero maximo de hilos: %d",num_hilos);
            LOG_PRINT("Broker de kafka: %s",brokers);
            LOG_PRINT("Topic de kafka: %s",topics);

            LOG_PRINT("Intento cargar configuracion monitorizacion");
            reintentos = 0;
            //Llamamos a la funcion que crea la estructura dinamica
            do {
                   LOG_PRINT("Carga monitorizacion n° %d",reintentos);
                   error_estdin = 0;
                   //Carga la configuracion hosts -> Guarda en lista_hosts en poller.h
                   carga_monitor();
                   reintentos ++;
            } while ((fin==0)||(reintentos>=5));
            if (error_estdin != 1)
                   LOG_PRINT("Configuracion hosts cargada correctamente");
            else {
                   LOG_PRINT("Fallo en carga de configuracion hosts");
                   error=1;
            }
     }

     return error;
}


/*
```

```
   * Funcion: int carga_config()
   *          Funcion que llamara a los parseadores de los
   *          ficheros de configuracion e intentara cargar la
   *          configuracion hasta que sea correcta
   *          NO CONTEMPLA la lista vacia
   */
  int carga_config()  {
       //Valor devuelto
       int error = 0;

       LOG_PRINT("Intento cargar nueva configuracion hosts");
       //Llamamos a la funcion que crea la estructura dinamica
       do {
              //Variable activada desde carga_monitor
              error_estdin = 0;
              /*
               * carga_monitor:
               * - parsea fichero de monitorizacion
               * - crea/modifica/elimina de la estructura
               */
              carga_monitor();
       } while ((error_estdin==1)||(fin==0));
       if (error_estdin != 1)
              LOG_PRINT("Nueva configuracion hosts cargada correctamente");
       else {
              LOG_PRINT("Fallo en carga de nueva configuracion hosts");
              error=1;
       }
       return error;

  }
```

carga.h

```
  #ifndef DATOS_H
  #include "datos.h"
  #endif

  #include "poller.h"


  /*
   * Funcion: int carga_config_ini()
   *          Carga configuracion al inicio.
   *          Funcion que llamara a los parseadores de los
   *          ficheros de configuracion e intentara cargar la
   *          configuracion hasta que sea correcta
   *          NO CONTEMPLA la lista vacia
   */
  int carga_config_ini();


  /*
   * Funcion: int carga_config()
   *          Funcion que llamara a los parseadores de los
   *          ficheros de configuracion e intentara cargar la configuracion hasta que
sea correcta
   *          NO CONTEMPLA la lista vacia
   */
  int carga_config();
```

config.c

```
#include "config.h"
#include "hilos.h"
#include "logger.h"

/*
 * Funcion: config_kafka()
 *   Funcion que inicia la configuración para permitir en
 *   envío de mensajes de kafka
 */
void config_kafka () {

     rd_kafka_conf_t *conf;
     rd_kafka_topic_conf_t *topic_conf;

     char errstr[512];

     error_kafka=0;

     /* Kafka configuration */
     conf = rd_kafka_conf_new();

     /* Topic configuration */
     topic_conf = rd_kafka_topic_conf_new();

     //Establecemos funcion de callback
     rd_kafka_conf_set_dr_cb(conf, msg_delivered);

     /* Create Kafka handle */
     if (!(rk = rd_kafka_new(RD_KAFKA_PRODUCER, conf, errstr, sizeof(errstr)))) {
            fprintf(stderr, "%% Failed to create new consumer: %s\n",  errstr);
            error_kafka=1;
            exit(1);
     }

     /* Add brokers */
     if (rd_kafka_brokers_add(rk, brokers) == 0) {
            fprintf(stderr, "%% No valid brokers specified\n");
            error_kafka=1;
            exit(1);
     }

     /* Create topic */
     rkt = rd_kafka_topic_new(rk, topics, topic_conf);
     topic_conf = NULL; /* Now owned by topic */
}

/**
 * Message delivery report callback.
 * Called once for each message.
 */
void msg_delivered (rd_kafka_t *rk, void *payload, size_t len, int error_code,
                                    void *opaque, void *msg_opaque) {

     if (error_code)
            LOG_PRINT("Message delivery failed: %s",rd_kafka_err2str(error_code));
     else
            LOG_PRINT("Message delivered: %s",(char *)payload);
}




/*
 * Funcion: void lanzado_hilos()
 *          Funcion que se encarga de lanzar los hilos
 *          del motor de envío de peticiones.
 *          Almacenará sus punteros en la tabla hilos
```

```
 */
void lanzado_hilos() {

     int numero_hilo=0;
     int error_hilo;

     //Almacenara provisionalmente el puntero al hilo
     pthread_t * p_hilo_prov;

     while ((numero_hilo < num_hilos)) {
            p_hilo_prov = &hilos[numero_hilo];
            error_hilo=pthread_create(p_hilo_prov,NULL,poller,(void *)numero_hilo);
            if (error_hilo!=0) {
                   fprintf(stdout,"Error de lanzado en el hilo n°%d\n",numero_hilo);
            }
            else {
                   numero_hilo++;
                   hilos_lanzados++;
            }
     }
}



/*
 * Funcion: config_estructura ()
 *          1)Calculo de llenado de hilos
 *          2)Establece cortes
 *          3)Inicializa la tabla de reparto
 */
void config_estructura ()  {

     st_host * lista_host_prov;

     num_hosts=0;

     //------------------HILOS------------------

     /*
      * Llenado de hilos:
      * 1) De la configuracion tenemos el numero de hilos que podemos lanzar
      * 2) Calculamos el numero de host a monitorizar
      */
     lista_host_prov = lista_hosts;
     while (lista_host_prov!=NULL){
            num_hosts++;
            lista_host_prov = lista_host_prov->next;
     }
     LOG_PRINT("Vamos a monitorizar %d hosts",num_hosts);

     /*
      * CALCULAR CUANTOS HOSTS VAN EN CADA HILO:
      *
      * Num_host % Num_hilos -> Nos da el resto
      * Num_host / Num_hilos -> Hosts que van en cada hilo
      * Tenemos que ver que hacer con el resto
      * --> Añadimos uno adicional en cada hilo
      */

     resto = num_hosts%num_hilos;
     host_hilo = num_hosts/num_hilos;

     //Establecemos las variables corte
     lista_cortes (lista_hosts,host_hilo,resto,num_hilos);
     LOG_PRINT("Tenemos disponibles %d hilos",num_hilos);
     LOG_PRINT("Vamos a lanzar %d hosts por hilo",host_hilo);
     LOG_PRINT("Con un resto a repartir de %d",resto);
```

```
	//Inicializamos la tabla de reparto
	tabla_reparto(num_hilos);

}

/*
 * Funcion: lista_cortes()
 * Funcion que establece las variables de corte
 * en la estructura de hosts
 *
 */
void lista_cortes (st_host * lista, int host, int resto, int hilos) {

	int llenado = 0;

	if(host!=0)
	{
		//CASO A: Mas hosts que hilos
		while (lista != NULL)
		{
			llenado++;
			if (llenado == host) {
				llenado = 0;
				if (resto > 0) {
					resto--;
					lista = lista->next;
				}
				lista->corte=1;
			}
		lista = lista->next;
		}
	}
	else {
		//Caso B: Mas hilos que hosts
		//Establezco corte en todos los hosts
		while (lista!=NULL) {
			if (lista->next != NULL)
				lista->corte=1;
			lista=lista->next;
		}
	}
}




/*
 * Funcion: tabla_reparto()
 * Se encargara de inicializar la tabla que
 * contendra en cada posicion el puntero al
 * nodo por el cual empezara a monitorizar
 * cada hilo
 */
void tabla_reparto (int hilos) {
	int i;
	int fin_for = 0;
	int enc_corte = 0;
	int enc_null = 0;
	st_host * lista_host_prov = lista_hosts;

	//Reserva de memoria para la tabla
	acceso_hilos = (st_host **)malloc(hilos * sizeof(st_host *));

	if (acceso_hilos == NULL) {
		fprintf(stdout,"Fallo Memoria de acceso_hilos\n");
	}

	//Tabla que almacenará el puntero a la estructura a la que accederá cada hilo
```

```
      for (i=0; (i<num_hilos)&&(fin_for==0); i++) {
            //Almaceno en la tabla
            acceso_hilos[i] = lista_host_prov;

            //Continuamos hasta corte o NULL
            enc_corte=0;
            while (enc_corte==0) {
                  if (lista_host_prov == NULL) {
                        fprintf(stdout,"Null encontrado\n");
                        enc_null=1;
                        fin_for=1;
                        enc_corte=1;
                  }
                  else {
                        if (lista_host_prov->corte == 1) {
                              enc_corte=1;
                        }
                        lista_host_prov=lista_host_prov->next;
                  }
            }
      }
      //Relleno resto tabla a NULL
      if (enc_null == 1){
            while (i<num_hilos) {
                  fprintf(stdout,"Almaceno en pos %d NULL\n",i);
                  acceso_hilos[i] = NULL;
                  i++;
            }
      }

      //Imprimimos la tabla de reparto
      fprintf(stdout,"\n----------Tabla de reparto:\n ");
      for (i=0; i<num_hilos; i++) {
            if(acceso_hilos[i]!=NULL)
                  fprintf(stdout,"[%d]%s\t",i,acceso_hilos[i]->name);
            else
                  fprintf(stdout,"[%d]NULL\t",i);
      }
      fprintf(stdout,"\n");

  }

  /*
   * Funcion: procesa_oid()
   * Realizara las traducciones de los OID
   * desde las MIBs cargadas anteriormente
   */
  void procesa_oid()
  {
      st_host * lista_host = lista_hosts;
      st_oid * lista_oid;
      while (lista_host != NULL) {
            lista_oid = lista_host->oids;
            while (lista_oid != NULL) {
                  lista_oid->OidLen    =    sizeof(lista_oid->Oid)/sizeof(lista_oid-
>Oid[0]);
                  if  (!read_objid(lista_oid->oid_name,  lista_oid->Oid,  &lista_oid-
>OidLen)) {
                        snmp_perror("read_objid");
                        //exit(1);
                     }
                  lista_oid = lista_oid->next;
            }
            lista_host = lista_host->next;
      }
  }

  /*
```

```
 * Funcion: cierre_sesiones()
 * Se encarga de cerrar todas las sesiones
 * abiertas en la estructura dinamica
 */
void cierre_sesiones() {
     st_host * lista_host_prov = lista_hosts;
     while (lista_host_prov!= NULL) {
            if (lista_host_prov->fallo_sesion==0) {
              snmp_close(lista_host_prov->punt_sesion);
            }
            lista_host_prov = lista_host_prov->next;
     }
}
```

config.h

```
#ifndef DATOS_H
#include "datos.h"
#endif

#include "poller.h"


/*
 * Funcion: config_kafka()
 *   Funcion que inicia la configuración para permitir en
 *   envío de mensajes de kafka
 */
void config_kafka ();

/**
 * Message delivery report callback.
 * Called once for each message.
 */
void msg_delivered (rd_kafka_t *rk, void *payload, size_t len, int error_code,
                                    void *opaque, void *msg_opaque);

/*
 * Funcion: void lanzado_hilos()
 *          Funcion que se encarga de lanzar los hilos
 *          del motor de envío de peticiones.
 *          Almacenará sus punteros en la tabla hilos
 */
void lanzado_hilos();


/*
 * Funcion: config_estructura ()
 *          1)Calculo de llenado de hilos
 *          2)Establece cortes
 *          3)Inicializa la tabla de reparto
 */
void config_estructura ();


/*
 * Funcion: lista_cortes()
 * Funcion que establece las variables de corte
 * en la estructura de hosts
 *
 */
void lista_cortes (st_host * lista, int host, int resto, int hilos);


/*
 * Funcion: tabla_reparto()
```

```
 * Se encargara de inicializar la tabla que
 * contendra en cada posicion el puntero al
 * nodo por el cual empezara a monitorizar
 * cada hilo
 */
void tabla_reparto (int hilos);


/*
 * Funcion: procesa_oid()
 * Realizara las traducciones de los OID
 * desde las MIBs cargadas anteriormente
 */
void procesa_oid();


/*
 * Funcion: cierre_sesiones()
 * Se encarga de cerrar todas las sesiones
 * abiertas en la estructura dinamica
 */
void cierre_sesiones();
```

estructura.c

```
#include "estructura.h"

/*
 * Funcion: imprime_lista()
 * Funcion auxiliar que imprime la lista enlazada por pantalla
 */
void imprime_lista () {
     st_oid * prov;
     st_host * lista_prov = lista_hosts;
     fprintf(stdout,"\n----------Lista enlazada:\n");
     while (lista_prov!= NULL) {
            fprintf(stdout,"[%s]->",lista_prov->name);
            prov=lista_prov->oids;
            while (prov!=NULL) {
                   fprintf(stdout,"[%s]->",prov->oid_name);
                   prov = prov->next;
            }
            fprintf(stdout,"|NULL\n");
            lista_prov=lista_prov->next;
     }
}


/*
 * Funcion: imprime_cortes()
 * Funcion auxiliar que imprime la lista enlazada
 * por pantalla pero con una estructura
 * que permite la visualizacion de los cortes
 */
void imprime_cortes () {
     st_host * lista_prov = lista_hosts;
     fprintf(stdout,"\n----------Lista de cortes:\n");
     while (lista_prov!=NULL) {
            fprintf(stdout,"[%s]->",lista_prov->name);
            if (lista_prov->corte == 1)
                   fprintf(stdout,"corte->");
            lista_prov = lista_prov->next;
     }
     fprintf(stdout,"NULL\n");
}
```

```
/*
 * Funcion: elimina_oid_completo
 * Elimina de la estructura el oid indicado,
 * realizando una acomodacion segura en la
 * estructura dinamica y en la tabla de reparto
 */
void elimina_oid_completo(st_oid * elimina) {

     int encontrado_oid = 0;
     int final_oid = 0;

     st_host* lista_host_prov = lista_hosts;
     st_oid* lista_oid_prov;
     st_oid * auxiliar;

     //A) buscamos OID anterior
     while(!encontrado_oid) {
            lista_oid_prov = lista_host_prov->oids;
            final_oid=0;
            while ((lista_oid_prov->next != elimina) && (!final_oid)) {
                  lista_oid_prov = lista_oid_prov->next;
                  if (lista_oid_prov->next == NULL)
                        final_oid=1;
            }
            if (lista_oid_prov->next == elimina)
                  encontrado_oid=1;
            else
                  lista_host_prov = lista_host_prov->next;
     }

     //B) Eliminacion segura
     if (encontrado_oid) {
            //B.1)Desenlazado de la estructura
            auxiliar = lista_oid_prov->next;
            lista_oid_prov->next = auxiliar->next;
            //B.2)¿Es el primero?
            if (lista_host_prov->oids==auxiliar)
                  lista_host_prov->oids=auxiliar->next;
            //B.3)Eliminacion
            libera_oid(auxiliar);
     }

}


/*
 * Funcion: elimina_nodo_completo
 * Elimina de la estructura el nodo indicado,
 * realizando una acomodacion segura en la
 * estructura dinamica y en la tabla de reparto
 */
void elimina_nodo_completo(st_host * elimina) {
     int final_est = 0;
     int i = 0;
     st_host * lista_host_prov =lista_hosts;
     st_host * auxiliar;

     //Comprobamos que no sea el primero
     if (lista_host_prov == elimina) {
            //Desenlazado
            lista_hosts = lista_host_prov->next;
            //Si esta en la tabla de reparto, estar el primero
            acceso_hilos[0]= lista_hosts;
            //Si el primero es corte,no pasa nada
            //Elimino
            libera_nodo(lista_host_prov);
     }
     else {
            //A) Buscamos el host anterior
```

```
            while ((lista_host_prov->next != elimina)&&(!final_est)) {
                  lista_host_prov = lista_host_prov->next;
                  if (lista_host_prov->next == NULL)
                        final_est = 1;
            }
            //Si no lo encontramos, no realizamos eliminacion
            if (!final_est) {
                  //B) Eliminacion segura
                  //B.1) Desenlazado de la estructura
                  auxiliar = lista_host_prov->next;
                  lista_host_prov->next = auxiliar->next;
                  //B.2) Comprobacion tabla reparto
                  for (i=0; i<num_hilos; i++) {
                        if (auxiliar==acceso_hilos[i]) {
                              acceso_hilos[i] = auxiliar->next;
                        }
                  }
                  //B.3) Comprobacion corte
                  if (auxiliar->corte ==1) {
                        lista_host_prov->corte = 1;
                  }
                  //B.4) Eliminacion
                  libera_nodo(auxiliar);
            }
      }
}


/*
 * Funcion: libera_oid()
 * Libera la memoria que ocupa el oid indicado
 */
void libera_oid(st_oid * elimina) {
      free(elimina->name);
      free(elimina->oid_name);
      free(elimina);
}

/*
 * Funcion: libera_oid()
 * Libera la memoria que ocupa el host indicado,
 * incluyendo su lista de oids
 */
void libera_nodo(st_host * elimina) {
      free(elimina->community);
      free(elimina->ip);
      free(elimina->name);

      //Cierre sesion
      snmp_close(elimina->punt_sesion);

      //Lista oid
      libera_lista_oid(elimina->oids);

      free(elimina);

}

/*
 * Funcion: libera_lista_oid()
 * Libera completamente la lista indicada
 */
void libera_lista_oid(st_oid * lista) {
      st_oid * auxiliar;

      while (lista!=NULL) {
            auxiliar = lista;
            lista = lista->next;
```

```
            libera_oid(auxiliar);
      }


}

/*
 * Funcion: liberacion_lista()
 * Libera completamente la estructura de hosts y oids
 */
void liberacion_lista() {
      st_host * auxiliar;

      while(lista_hosts!=NULL) {
            auxiliar = lista_hosts;
            lista_hosts = lista_hosts->next;
            libera_nodo(auxiliar);
      }

}



/*
 * Funcion busca_nodo()
 * Busca un nodo en la lista enlazada en base a su id
 */
st_host * busca_nodo(int id_sensor) {
      //Lista en lista_hosts
      int encontrado = 0;
      st_host * lista_host_prov = lista_hosts;
      while(lista_host_prov != NULL && encontrado==0) {
            if (lista_host_prov->id == id_sensor)
                  encontrado = 1;
            else
                  lista_host_prov = lista_host_prov->next;
      }

      if (encontrado == 1)
            return lista_host_prov;
      else
            return NULL;
}


/*
 * Funcion busca_oid()
 * Busca un oid en la lista enlazada en base a su id
 */
st_oid * busca_oid(int id_oid, st_oid* lista) {
      st_oid * lista_oid_prov = lista;
      int encontrado = 0;
      while(lista_oid_prov != NULL && encontrado==0) {
            if (lista_oid_prov->id == id_oid)
                  encontrado = 1;
            else
                  lista_oid_prov = lista_oid_prov->next;
      }

      if (encontrado == 1)
            return lista_oid_prov;
      else
            return NULL;

}


/*
 * Funcion: anida_host()
 * Anida un host en la estructura balanceando la carga
```

```
 */
void anida_host (st_host * nodo) {
     st_host * host_prov;
     st_host * prov;
     st_host * auxiliar;
     host_prov = lista_hosts;
     int i;
     int hilo_menos_cargado = 0;
     int carga_hilo_actual = 0;
     int id_menos_cargado;

     //Si la estructura ya esta en memroia
     if(estruct_memoria==1) {
            //Calculamos cual es el hilo con menos carga
            //-->i=0
            id_menos_cargado = 0;
            while (host_prov!=NULL && host_prov->corte!=1) {
                  hilo_menos_cargado++;
            }

            //-->i=1...num_hilos
            for (i=1; i<num_hilos; i++){
                  prov = acceso_hilos[i];
                  while (prov!=NULL && prov->corte!=1) {
                        carga_hilo_actual++;
                  }
                  if (carga_hilo_actual<hilo_menos_cargado) {
                        hilo_menos_cargado = carga_hilo_actual;
                        id_menos_cargado = i;
                  }
            }

            //Lo insertamos al inicio
            auxiliar = acceso_hilos[id_menos_cargado];
            acceso_hilos[id_menos_cargado]=nodo;
            nodo->next = auxiliar;
     }
     //Si la estructura no esta en memoria, inicio del programa
     else {
            //Enlazo consecutivamente al inicio
            auxiliar = lista_hosts;
            lista_hosts=nodo;
            nodo->next = auxiliar;
     }

}


/*
 * Funcion: anida_oid()
 * Anida un oid en la estructura
 * Siempre al principio
 */
void anida_oid (st_host * nodo, st_oid * oid) {
     st_oid * auxiliar = nodo->oids;
     nodo->oids = oid;
     oid->next = auxiliar;
}



//-----------Funciones auxiliares-----------

int LongitudCadena (const char *cadena) {
     int j=0;
     while (cadena[j] != '\0') {
            j++;
     }
```

```
     return j;
}


/*
 * Funcion auxiliar para copia de cadenas.
 * Necesaria para incluir el \0 que no incluye
 * la libreria jansson por defecto
 */
void copia_cadena(const char * origen, char * destino, int tam) {
     int ind;

     for (ind=0; ind<tam; ind++) {
            destino[ind] = origen[ind];
     }
     ind++;
     destino[ind]='\0';
}
```

estructura.h

```
#ifndef DATOS_H
#include "datos.h"
#endif

#include "poller.h"


/*
 * Funcion: imprime_lista()
 * Funcion auxiliar que imprime la lista enlazada por pantalla
 */
void imprime_lista ();


/*
 * Funcion: imprime_cortes()
 * Funcion auxiliar que imprime la lista enlazada
 * por pantalla pero con una estructura
 * que permite la visualizacion de los cortes
 */
void imprime_cortes ();


/*
 * Funcion: elimina_oid_completo
 * Elimina de la estructura el oid indicado,
 * realizando una acomodacion segura en la
 * estructura dinamica y en la tabla de reparto
 */
void elimina_oid_completo(st_oid * elimina);


/*
 * Funcion: elimina_nodo_completo
 * Elimina de la estructura el nodo indicado,
 * realizando una acomodacion segura en la
 * estructura dinamica y en la tabla de reparto
 */
void elimina_nodo_completo(st_host * elimina);

/*
 * Funcion: libera_oid()
 * Libera la memoria que ocupa el oid indicado
 */
void libera_oid(st_oid * elimina);
```

```
/*
 * Funcion: libera_oid()
 * Libera la memoria que ocupa el host indicado,
 * incluyendo su lista de oids
 */
void libera_nodo(st_host * elimina);


/*
 * Funcion: libera_lista_oid()
 * Libera completamente la lista indicada
 */
void libera_lista_oid(st_oid * lista);


/*
 * Funcion: liberacion_lista()
 * Libera completamente la estructura de hosts y oids
 */
void liberacion_lista();


/*
 * Funcion busca_nodo()
 * Busca un nodo en la lista enlazada en base a su id
 */
st_host * busca_nodo(int id_sensor);


/*
 * Funcion busca_oid()
 * Busca un oid en la lista enlazada en base a su id
 */
st_oid * busca_oid(int id_oid, st_oid* lista);


/*
 * Funcion: anida_host()
 * Anida un host en la estructura balanceando la carga
 */
void anida_host (st_host * nodo);

/*
 * Funcion: anida_oid()
 * Anida un oid en la estructura
 * Siempre al principio
 */
void anida_oid (st_host * nodo, st_oid * oid);



//-----------Funciones auxiliares-----------

int LongitudCadena (const char *cadena);


/*
 * Funcion auxiliar para copia de cadenas.
 * Necesaria para incluir el \0 que no incluye
 * la libreria jansson por defecto
 */
void copia_cadena(const char * origen, char * destino, int tam);
```

parseo.c

```
  #include "parseo.h"
  #include "logger.h"
  #include "estructura.h"




  /*
  * Funcion: void carga_monitor ()
  * Parseara y procesara el fichero de monitorizacion
  * creando y modificando la estructura dinamica
  */
  void carga_monitor () {

  //Puntero a la lista -> puntero lista_hosts GLOBAL

  char * rutaconf = fich_monit;
  json_error_t error;

  //Configuracion sensores
  st_host * nodo_encontrado;
  json_t *sensores;
  json_t *sensor;
  json_t *sensor_id,*sensor_name,*sensor_ip,*sensor_community,*sensor_snmp_version,
*sensor_modif;
  const char *nombre_sensor;
  const char *ip_sensor;
  const char *comunidad_sensor;
  int version_sensor;
  int modif_sensor;
  int id_sensor;

  int tam_nombre_sensor;
  int tam_ip_sensor;
  int tam_comunidad_sensor;

  //Configuracion OID
  st_oid * oid_encontrado;
  json_t *oids;
  json_t *oid;
  json_t *oid_name, *oid_oid, *oid_id, *oid_modif, *oid_time;
  const char *nombre_oid;
  const char *cid_oid;
  int id_oid;
  int modif_oid;
  int t_monit;

  int tam_nombre_oid;
  int tam_id_oid;

  //Variables auxiliares
  int i;
  int j;
  int error_for = 1;
  int error_for2 = 1;

  int error_host=0;
  int error_oid=0;

  st_host * nodo_prov=NULL;
  st_oid * oid_prov=NULL;
  st_oid * lista_oid=NULL;


  /* parse text into JSON structure */
  sensores = json_load_file(rutaconf,0,&error);
```

```
  fprintf(stdout,"\n\n----------Proceso de parseo de monitorizacion:\n");

  if (!sensores) {
  LOG_PRINT("Error al abrir fichero de monitorizacion");
  LOG_PRINT("error: on line %d:",error.line);
  LOG_PRINT("error: %s",error.text);
  error_estdin = 1;
  }
  //Salgo
  else {
  LOG_PRINT("Fichero de monitorizacion abierto correctamente");
  //Primero comprobamos que es un array
  if (!(json_is_array(sensores))) {
  LOG_PRINT("error: en fich_monitor: El documento no es un array");
  error_estdin = 1;
  }
  //Salgo
  else {
  /*
  * error_host define un error en la reserva de memoria, con lo que el nodo no sera
almacenado
  * error_for define un error en el fichero, con lo que salimos e informamos del error
  */

  /*
  * Bucle que recorre cada array/host del fichero
  * Cada iteracción es un host
  */
  fprintf(stdout,"Numero de objetos hosts: %d\n",json_array_size(sensores));
  for(i = 0; (i < json_array_size(sensores))&&(error_for); i++) {

  fprintf(stdout,"Objeto host n° %d\n",i+1);

  sensor = json_array_get(sensores,i);
  if (!json_is_object(sensor)) {
  LOG_PRINT("error: en fich_monitor: Datos de sensor no son un objeto");
  error_for = 0;
  }
  else {
  //Si es correcto sensor... obtengo sus datos de cabecera
  sensor_id = json_object_get(sensor,"sensor_id");
  sensor_name= json_object_get(sensor,"sensor_name");
  sensor_ip= json_object_get(sensor,"sensor_ip");
  sensor_community= json_object_get(sensor,"community");
  sensor_snmp_version= json_object_get(sensor,"snmp_version");
  sensor_modif = json_object_get(sensor,"modif");

  if (!json_is_integer(sensor_id)) {
  LOG_PRINT("error: en fich_monitor: sensor_id no es un integer");
  }
  else if(!json_is_string(sensor_name)) {
  LOG_PRINT("error: en fich_monitor: sensor_name no es un string");
  }
  else if(!json_is_string(sensor_ip)) {
  LOG_PRINT("error: en fich_monitor: sensor_ip no es un string");
  }
  else if(!json_is_string(sensor_community)) {
  LOG_PRINT("error: en fich_monitor: sensor_community no es un string");
  }
  else if(!json_is_integer(sensor_snmp_version)) {
  LOG_PRINT("error: en fich_monitor: sensor_snmp_version no es un entero");
  }
  else if(!json_is_integer(sensor_modif)) {
  LOG_PRINT("error: en fich_monitor: sensor_modif no es un entero");
  }
  else {
  //No ha habido error en el parseo de la cabecera
  /*
```

```
* Modif indica que hacer con este host
* Si vale 1: Crear este host si no existe o modificarlo si existe
* Si vale 0: Eliminar este host y sus oids
*/
modif_sensor = json_integer_value(sensor_modif);
if (modif_sensor==1) {


//¿Existe? --> Lo buscamos por id
id_sensor = json_integer_value(sensor_id);
nodo_encontrado = busca_nodo(id_sensor);

//Obtenemos resto de cabecera
nombre_sensor = json_string_value(sensor_name);
ip_sensor = json_string_value(sensor_ip);
comunidad_sensor = json_string_value(sensor_community);
version_sensor = json_integer_value(sensor_snmp_version);

//Obtenemos los tamanos
tam_nombre_sensor = strlen (nombre_sensor);
tam_ip_sensor = strlen (ip_sensor);
tam_comunidad_sensor = strlen (comunidad_sensor);

//--->EXISTE EL NODO
if (nodo_encontrado!= NULL) {

//fprintf(stdout,"El nodo %d existe con ID = %d\n",i+1,id_sensor);

//NAME
free(nodo_encontrado->name);
nodo_encontrado->name = (char *)calloc(tam_nombre_sensor,sizeof(char));
if (nodo_encontrado->name!=NULL)
strcpy(nodo_encontrado->name,nombre_sensor);
else {
//TODO Fallo memoria
fprintf(stdout,"Fallo Memoria\n");
}

//IP
free(nodo_encontrado->ip);
nodo_encontrado->ip = (char *)calloc(tam_ip_sensor,sizeof(char));
if (nodo_encontrado->ip!=NULL)
strcpy(nodo_encontrado->ip,ip_sensor);
else {
//TODO Fallo memoria
fprintf(stdout,"Fallo Memoria\n");
}

//COMUNIDAD
free(nodo_encontrado->community);
nodo_encontrado->community = (char *)calloc(tam_comunidad_sensor,sizeof(char));
if (nodo_encontrado->community!=NULL) {
strcpy(nodo_encontrado->community,comunidad_sensor);
}
else {
//TODO Fallo memoria
fprintf(stdout,"Fallo Memoria\n");
}

//VERSION
nodo_encontrado->version = version_sensor;

//Marcamos para reabrir sesion
nodo_encontrado->fallo_sesion=1;
}
//--->NO EXISTE EL NODO
/*
* Si el nodo no existe, lo creamos y
* lo anidamos al final de la estructura
*/
```

```
else {
//fprintf(stdout,"El nodo %d no existe, lo creo con con ID = %d\n",i+1,id_sensor);
//Creamos un nodo de host
nodo_prov = (st_host *) malloc (sizeof(st_host));
//ID
nodo_prov->id=id_sensor;


//NAME
nodo_prov->name = (char *)calloc(tam_nombre_sensor,sizeof(char));
if (nodo_prov->name!=NULL) {
strcpy(nodo_prov->name,nombre_sensor);
}

//IP
nodo_prov->ip = (char *)calloc(tam_ip_sensor,sizeof(char));
if (nodo_prov->ip != NULL) {
strcpy(nodo_prov->ip,ip_sensor);
}

//COMUNIDAD
nodo_prov->community = (char *)calloc(tam_comunidad_sensor,sizeof(char));
if (nodo_prov->community!=NULL) {
strcpy(nodo_prov->community,comunidad_sensor);
}

//VERSION
nodo_prov->version = version_sensor;

//OID

//Inicializaciones
//Fallo_sesion a 1 para que abra la sesion
nodo_prov->oids = NULL;
nodo_prov->fallo_sesion = 1;
nodo_prov->next = NULL;
nodo_prov->corte = 0;

//Anidamos
anida_host(nodo_prov);

//De momento nodo_encontrado esta a NULL
nodo_encontrado = nodo_prov;
}

/*
* LLegados a este punto, tenemos un host dentro de la
* estructura, ya sea porque ha sido modificado o creado.
* Vamos a pasar a modificar, crear o eliminar sus OIDs
*/
//Obtenemos la lista de OID
oids = json_object_get(sensor,"monitors");
if(!json_is_array(oids)) {
error_for2 = 0;
LOG_PRINT("error: en fich_monitor: monitors no es un array");
}
else {
/*
* Bucle que recorre cada array/oid del host
* Cada iteracción es un oid
*/
fprintf(stdout,"El nodo %d tiene %d OIDS\n",i+1,json_array_size(oids));
for (j = 0; (j < json_array_size(oids))&&(error_for2); j++) {

fprintf(stdout,"Nodo %d Oid %d\n",i+1,j+1);

oid_prov = NULL;
```

```
oid = json_array_get(oids,j);
if (!json_is_object(oid)) {
LOG_PRINT("error: en fich_monitor: Datos de OID no es un objeto");
}
else {

//Datos de OID
oid_id = json_object_get(oid,"oid_id");
oid_name= json_object_get(oid,"name");
oid_oid= json_object_get(oid,"oid");
oid_modif = json_object_get(oid,"modif");
oid_time = json_object_get(oid,"t_monit");

//Comprobamos que los datos son correctos
if(!json_is_string(oid_name)) {
LOG_PRINT("error: en fich_monitor: oid_name no es un string");
}
else if(!json_is_string(oid_oid)) {
LOG_PRINT("error: en fich_monitor: oid_oid no es un string");
}
else if (!json_is_integer(oid_id)) {
LOG_PRINT("error: en fich_monitor: id_oid no es un integer");
}
else if (!json_is_integer(oid_modif)) {
LOG_PRINT("error: en fich_monitor: modif no es un integer");
}
else if (!json_is_integer(oid_time)) {
LOG_PRINT("error: en fich_monitor: t_monit no es un integer");
}
else {

//Obtenemos parametro modif
modif_oid = json_integer_value(oid_modif);

if (modif_oid == 1) {
//Modifico o creo
id_oid = json_integer_value(oid_id);
oid_encontrado = busca_oid(id_oid, nodo_encontrado->oids);

t_monit = json_integer_value(oid_time);

nombre_oid = json_string_value(oid_name);
cid_oid = json_string_value(oid_oid);
tam_nombre_oid = strlen (nombre_oid);
tam_id_oid = strlen (cid_oid);


if (oid_encontrado != NULL) {

//fprintf(stdout,"OID con id %d encontrado, lo modifico\n",id_oid);

//Modificar
//NAME
oid_encontrado->name = (char *)calloc(tam_nombre_oid+1,sizeof(char));
if (oid_prov->name!=NULL) {
copia_cadena(nombre_oid,oid_prov->name,tam_nombre_oid);

}
else {
//TODO Fallo Memoria
fprintf(stdout,"Fallo Memoria\n");
}

//OID
oid_encontrado->oid_name = (char *)calloc(tam_id_oid+1,sizeof(char));
if (oid_encontrado->oid_name!=NULL) {
copia_cadena(cid_oid,oid_prov->oid_name,tam_id_oid+1);

}
else {
```

```
//TODO Fallo Memoria
fprintf(stdout,"Fallo Memoria\n");
}

//TIME
oid_encontrado->t_monitor = t_monit;

oid_prov->t_actual=0;

//Al llegar  aqui, informacion de OID modificada
}
else {

//fprintf(stdout,"OID con id %d no encontrado, lo creo\n",id_oid);

//Crear y anidar
//Creamos un nodo
oid_prov = NULL;
oid_prov = (st_oid *) malloc (sizeof(st_oid));
if (oid_prov == NULL) {
error_oid = 1;
//TODO
}
else {
//Almacenamos informacion
//NAME
oid_prov->name = (char *)calloc(tam_nombre_oid+1,sizeof(char));
if (oid_prov->name!=NULL) {
copia_cadena(nombre_oid,oid_prov->name,tam_nombre_oid);
//fprintf(stdout,"nombre %s con tamanio %d\n",oid_prov->name,tam_nombre_oid);
}
else {
//TODO Fallo Memoria
fprintf(stdout,"Fallo Memoria\n");
}

//OID
oid_prov->oid_name = (char *)calloc(tam_id_oid+1,sizeof(char));
if (oid_prov->oid_name!=NULL) {
copia_cadena(cid_oid,oid_prov->oid_name,tam_id_oid);
//fprintf(stdout,"oid %s con tamanio %d\n",oid_prov->oid_name,tam_id_oid);
}
else {
//TODO Fallo Memoria
fprintf(stdout,"Fallo Memoria\n");
}

//TIME
oid_prov->t_monitor = t_monit;

//Inicializaciones
oid_prov->next=NULL;
oid_prov->t_actual=0;


//Anidamos
anida_oid(nodo_encontrado,oid_prov);

}

}

}
else {
//Elimino
//TODO
if (oid_encontrado != NULL) {
elimina_oid_completo(oid_encontrado);
```

```
}

}
}
}
}//Fin for OIDs
}//Lista de OIDs no es un array, la salto
}//Fin modif = 1
else {
//Si modif es cero, lo eliminamos
//¿Existe?
if (nodo_encontrado != NULL) {
//TODO
elimina_nodo_completo(nodo_encontrado);
}
//Si no existe no hacemos nada
}

}//Si error en cabecera, sig host
}//El sensor no es un objeto, sig host
}//Fin for de hosts
}//Documento no es array
}//Error de apertura
//SALIDA
if (error_estdin == 0) {
LOG_PRINT("Fichero de monitorizacion parseado correctamente");
}
else {
LOG_PRINT("ERROR: Fichero de monitorizacion no parseado");
}
json_decref(sensores);

}




/*
* Funcion: void carga_conf ()
* Parseara el fichero de configuracion
*/
void carga_conf () {

/*
* Inicializa los valores de configuracion general:
* - broker
* - topic
* - max_hilos
* - ruta mibs
*/
char * rutaconf = fich_conf;
json_error_t error;


//Configuracion general
json_t *conf;
json_t *numer_hilos,*broker,*topic;

const char *kafka_broker_json;
const char *kafka_topic_json;

int tam_broker;
int tam_topic;

/* parse text into JSON structure */
json_t *root = json_load_file(rutaconf,0,&error);

if (!root) {
error_confic = 1;
```

```
LOG_PRINT("Error al abrir fichero de configuracion");
LOG_PRINT("error: on line %d:",error.line);
LOG_PRINT("error: %s",error.text);
}
else
{
LOG_PRINT("Fichero de configuración abierto correctamente");
//Primero comprobamos que es un objeto
if (!(json_is_array(root))) {
error_confic = 1;
LOG_PRINT("error: en fich_config: El documento no es un array");
}
else {
//Si es un array, continuamos
conf = json_array_get(root,0);
if (!json_is_object(conf)) {
error_confic = 1;
LOG_PRINT("error: en fich_config: Datos conf no es un objeto");
}
else {
//Si es un objeto, continuamos
numer_hilos = json_object_get(conf,"threads");
broker = json_object_get(conf,"kafka_broker");
topic = json_object_get(conf,"kafka_topic");


if(!json_is_integer(numer_hilos))
{
error_confic = 1;
LOG_PRINT("error: en fich_config: thread no es un numero");
}
else if(!json_is_string(broker)) {
error_confic = 1;
LOG_PRINT("error: en fich_config: broker no es un string");
}
else if(!json_is_string(topic)) {
error_confic = 1;
LOG_PRINT("error: en fich_config: topic no es un string");
}
else {

//----------HILOS----------
num_hilos = json_integer_value(numer_hilos);
//----------BROKER----------
kafka_broker_json= json_string_value(broker);
tam_broker = strlen (kafka_broker_json);
brokers = (char *)calloc(tam_broker,sizeof(char));
//Copia kafka_broker_json en broker
strcpy(brokers,kafka_broker_json);
//----------TOPIC----------
kafka_topic_json= json_string_value(topic);
tam_topic = strlen(kafka_topic_json);
topics = (char *)calloc(tam_topic,sizeof(char));
//Copia kafka_topic_json en broker
strcpy(topics,kafka_topic_json);

//TODO: carga rutas de las mibs


LOG_PRINT("Fichero de configuración parseado correctamente");
}
}
}
}
json_decref(root);
}
```

parseo.h

```
#ifndef DATOS_H
#include "datos.h"
#endif

#include "poller.h"

/*
 * Funcion: void carga_monitor ()
 * Parseara y procesara el fichero de monitorizacion
 * creando y modificando la estructura dinamica
 */
void carga_monitor ();

/*
 * Funcion: void carga_conf ()
 * Parseara el fichero de configuracion
 */
void carga_conf ();
```

9.3 Anexo 3: Hilos

hilos.c

```
#include "hilos.h"
#include "estructura.h"
#include "logger.h"



/*
* Funcion: void * poller (void * thread_args)
*
*          Funcion ejecutada por hilos. Se encarga del envio de pdu continuo.
*          Recibira una lista enlazada de los hosts que tiene que monitorizar
*/
void * poller (void * thread_args)
{

//ID numerico usado para acceder a la tabla de reparto
int id_hilo = (int)thread_args;

st_oid * lista_oid_prov=NULL;
struct snmp_pdu *pregunta=NULL;

//Tiempo
long int tiempo_ejec;
long int tiempo_ant;
long int tiempo_dif;
long int tiempo_menor;

//Banderas
int fin_lista = 1;
int corte = 1;


//Punteros a la estructura
st_host * lista_host_prov = acceso_hilos[id_hilo];
fprintf(stdout,"\n\nHablamos desde el hilo nº %d con su primer acceso a [%s]\n y
monitoriza:\n",id_hilo,lista_host_prov->name);
fprintf(stdout,"->[%s]",lista_host_prov->name);
while ((lista_host_prov->corte != 1) && (lista_host_prov->next != NULL)) {
fprintf(stdout,"->[%s]\t ",lista_host_prov->name);
lista_host_prov = lista_host_prov->next;
```

```
}
fprintf(stdout,"\n");
lista_host_prov = acceso_hilos[id_hilo];


/*
* Abrimos una sesion para cada host, almacenando los punteros
* en la estructura dinamica
* Levantamos la bandera fallo_sesion para las sesiones no abiertas.
* Seran reintentadas posteriormente
*/
while (corte) {

pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock);

//fprintf(stdout,"Abro sesion de host %s\n",lista_host_prov->name);
//Almacenamos el puntero en la estructura dinamica
snmp_sess_init(&(lista_host_prov->sesion));


//SNMPv2c
if (lista_host_prov->version == 2) {

lista_host_prov->sesion.version = SNMP_VERSION_2c;

lista_host_prov->sesion.peername = strdup(lista_host_prov->ip);

lista_host_prov->sesion.community = (u_char*)strdup(lista_host_prov->community);
lista_host_prov->sesion.community_len = strlen(lista_host_prov->community);


lista_host_prov->sesion.callback = procesa_pdu;
lista_host_prov->sesion.callback_magic = lista_host_prov->oids;

lista_host_prov->punt_sesion = snmp_open(&(lista_host_prov->sesion));
}
//SNMPv1
else if(lista_host_prov->version == 1) {

lista_host_prov->sesion.version = SNMP_VERSION_1;

lista_host_prov->sesion.peername = strdup(lista_host_prov->ip);

lista_host_prov->sesion.community = (u_char*)strdup(lista_host_prov->community);
lista_host_prov->sesion.community_len = strlen(lista_host_prov->community);


lista_host_prov->sesion.callback = procesa_pdu;
lista_host_prov->sesion.callback_magic = lista_host_prov->oids;

lista_host_prov->punt_sesion = snmp_open(&(lista_host_prov->sesion));
}



//Comprobamos que se haya abierto correctamente
if (!(lista_host_prov->punt_sesion)) {
//Marcamos este host --> contador de reintentos
lista_host_prov->fallo_sesion = ESPERA_REINTENTOS;
}
else {
//fprintf(stdout,"Sesion abierta correctamente\n");
lista_host_prov->fallo_sesion = 0;
}


//Llegamos a corte o a NULL -> paramos el bucle
```

```
  if ((lista_host_prov->corte == 1)||(lista_host_prov->next==NULL)) {
  corte = 0;
  //fprintf(stdout,"Fin apertura sesiones por llegada a corte o NULL\n");
  }
  else
  lista_host_prov = lista_host_prov->next;


  }



  //Todas las sesiones abiertas o marcadas para reintento


  //BUCLE MONITORIZACION -> Finaliza solo ante Ctrl C
  /*
  * 1) Intento abrir sesion
  * 2) Monitoriza sus oids
  */
  tiempo_menor = 1000;
  tiempo_dif = 0;
  tiempo_ejec = (long int) time(NULL);
  while (fin) {



  //pthread_mutex_lock(&lock);
  //pthread_mutex_unlock(&lock);

  tiempo_ant= (long int) time(NULL);
  fprintf (stdout,"Inicio bucle con tiempo_ant: %ld y tiempo_dif:
%ld\n",tiempo_ejec,tiempo_dif);



  lista_host_prov = acceso_hilos[id_hilo];
  fin_lista = 1;


  do {
  //Si hemos llegado al ultimo host, levantamos la bandera
  //fprintf(stdout,"Empiezo con host: %s\n",lista_host_prov->name);
  if ((lista_host_prov->corte == 1) || (lista_host_prov->next == NULL))
  fin_lista = 0;



  //SI SESION ABIERTA -> ENVIO
  if(lista_host_prov->fallo_sesion == 0) {
  //fprintf(stdout,"Su sesion esta abierta correctamente\n");
  lista_oid_prov = lista_host_prov->oids;


  //Lanzamos una pregunta por cada OID
  while (lista_oid_prov != NULL) {

  //Comprobacion tiempo
  if (lista_oid_prov->t_actual >= lista_oid_prov->t_monitor) {
  lista_oid_prov->t_actual = 0;
  //Monitorizo
  fprintf(stdout,"Monitorizo [%s] -> OID %s\n",lista_host_prov->name,lista_oid_prov-
>oid_name);
  pregunta=NULL;
  pregunta = snmp_pdu_create(SNMP_MSG_GET);
  snmp_add_null_var(pregunta, lista_oid_prov->Oid, (int)lista_oid_prov->OidLen);
  if (!(snmp_send(lista_host_prov->punt_sesion, pregunta))) {
  snmp_perror("snmp_send");
  snmp_free_pdu(pregunta);
```

```
}
//Una vez enviada la pregunta, almaceno el id
lista_oid_prov->enviado_id = pregunta->reqid;
}
else {
//Actualizamos tiempo sumando diferencia
lista_oid_prov->t_actual += tiempo_dif;
if (lista_oid_prov->t_actual<tiempo_menor)
tiempo_menor = lista_oid_prov->t_actual;
}

lista_oid_prov = lista_oid_prov->next;
}
//OID lanzados
}

//SI SESION NO ABIERTA -> Intento abrir
else {
/*
* Solo vamos a reintentar cuando se cumplan tantas interacciones
* como nos marque el numero de reintentos. Así no saturamos con
* un numero excesivo de reintentos
*
*/
if (lista_host_prov->fallo_sesion > 1) {
lista_host_prov->fallo_sesion--;
}
else {
//Reintentamos abrir la sesion cuando fallo_sesion sea 1
snmp_sess_init(&(lista_host_prov->sesion));

//SNMPv2c
if (lista_host_prov->version == 2) {
lista_host_prov->sesion.version = SNMP_VERSION_2c;

lista_host_prov->sesion.peername = strdup(lista_host_prov->ip);

lista_host_prov->sesion.community = (u_char*)strdup(lista_host_prov->community);
lista_host_prov->sesion.community_len = strlen(lista_host_prov->community);


lista_host_prov->sesion.callback = procesa_pdu;
lista_host_prov->sesion.callback_magic = lista_host_prov->oids;

lista_host_prov->punt_sesion = snmp_open(&(lista_host_prov->sesion));
}
//SNMPv1
else if(lista_host_prov->version == 1) {
lista_host_prov->sesion.version = SNMP_VERSION_1;

lista_host_prov->sesion.peername = strdup(lista_host_prov->ip);

lista_host_prov->sesion.community = (u_char*)strdup(lista_host_prov->community);
lista_host_prov->sesion.community_len = strlen(lista_host_prov->community);


lista_host_prov->sesion.callback = procesa_pdu;
lista_host_prov->sesion.callback_magic = lista_host_prov->oids;

lista_host_prov->punt_sesion = snmp_open(&(lista_host_prov->sesion));
}

if (!(lista_host_prov->punt_sesion)) {
fprintf(stdout,"fallo sesion\n");
lista_host_prov->fallo_sesion=ESPERA_REINTENTOS;
}
else {
lista_host_prov->fallo_sesion=0;
```

```
  }
  }
  }

  lista_host_prov = lista_host_prov->next;

  //Antes de terminar, desbloqueo mutex
  } while (fin_lista);
  sleep((int)tiempo_menor);
  //Acabamos de terminar la lista de host
  //Esperamos el tiempo indicado -> INTERVALO MONITORIZACION
  tiempo_ejec = (long int) time (NULL);
  tiempo_dif = tiempo_ejec-tiempo_ant;

  //Minimo espero 1 seg
  if (tiempo_dif == 0) {
  tiempo_dif = 1;
  sleep(1);
  }

  fprintf(stdout,"Fin del bucle con tiempo_ejec: %ld tiempo_dif: %ld y tiempo_menor:
%ld\n",tiempo_ejec,tiempo_dif,tiempo_menor);

  //TODO Tiempo entre monitorizacion
  fprintf (stdout,"\n");

  }

  /*
  * Antes de empezar a cerrar las sesiones, comprobamos que el
  * hilo de lectura ha finalizado
  */


  fin_lista=1;
  lista_host_prov = acceso_hilos[id_hilo];
  while (fin_lista) {
  sleep(1);
  if (fin_lectura == 0)
  fin_lista = 0;
  }

  return 0;
  }



  /*
  * Funcion: int procesa_pdu(int operation, struct snmp_session *sp,
  *    int reqid, struct snmp_pdu *pdu, void *magic)
  *
  *          Funcion que se definirá de callback en las sesiones snmp.
  *          Se llamara cada vez que se recibe una trama pdu.
  *          SU funcion principal sera enviar por kafka los
  *          datos de la PDU recibida
  *
  *          Parametros callback:
  *          1)int operation                         --> the possible operations are
RECEIVED_MESSAGE and TIMED_OUT
  *          2)struct snmp_session* session   --> the session that was authenticated
using community
  *          3)int reqid                             --> the request ID identifying the
transaction within this session. Use 0 for traps
  *          4)struct snmp_pdu* pdu                  --> a pointer to PDU information.
You must copy the information because it will be freed elsewhere
  *          5)void* magic                           --> a pointer to the data for
callback()
  */
  int procesa_pdu(int operation, struct snmp_session *sp, int reqid, struct snmp_pdu
*pdu, void *magic) {
```

```
  int fin_busca = 1;

  st_oid * lista_oid_prov = (st_oid *)magic;

  //netsnmp_variable_list * vars;

  //2) Buscamos el oid por el id de la pregunta
  while ((lista_oid_prov->enviado_id != reqid)||fin_busca==0) {
  if(lista_oid_prov->next==NULL)
  fin_busca=0;
  lista_oid_prov=lista_oid_prov->next;
  }
  if (fin_busca) {
  //Hemos encontrado el OID
  if (!(lista_oid_prov->enviado_id == lista_oid_prov->ultimo_reqid)) {
  //Si no son iguales, lo guardo y lo imprimo
  lista_oid_prov->ultimo_reqid=reqid;
  if (operation == NETSNMP_CALLBACK_OP_RECEIVED_MESSAGE) {


  //Mandar mensaje por kafka
  productor_kafka(pdu, sp);


  //Imprimimos por linea de comandos
  print_result(STAT_SUCCESS, sp, pdu);
  }
  }
  //Si son iguales no hago nada
  }
  return 0;
  }



  /*
  * Funcion: productor_kafka()
  *          Funcion que acepta una PDU, extrae sus datos
  *          y lo incluye al broker y al topic ya iniciado
  */
  void productor_kafka (struct snmp_pdu *pdu, struct snmp_session *sp) {
  struct timeval now;
  struct timezone tz;
  struct tm *tm;

  struct variable_list *vp;
  char buf[1024];
  char auxmensaj[2048];

  int ok=1;

  while (ok) {
  if (mensaj == NULL) {
  mensaj = (char *) malloc (4096*sizeof(char));

  gettimeofday(&now, &tz);
  tm = localtime(&now.tv_sec);

  //Creamos el mensaje a enviar
  sprintf(mensaj, "%.2d:%.2d:%.2d.%.6d ", tm->tm_hour, tm->tm_min, tm->tm_sec,
(int)now.tv_usec);


  vp = pdu->variables;
  if (pdu->errstat == SNMP_ERR_NOERROR) {
  while (vp) {
  snprint_variable(buf, sizeof(buf), vp->name, vp->name_length, vp);
```

```
sprintf(auxmensaj, "%s: %s\n", sp->peername, buf);

strcat(mensaj,auxmensaj);

vp = vp->next_variable;
}
}

produce_msg (mensaj);
ok=0;
}
}
}



/*
* Funcion: void * hilo_lectura (void * thread_args)
*
*           Funcion ejecutada en un hilo. Se encargará de recibir las
*           respuestas a las pdu enviadas por los poller a traves de la
*           funcion send
*/
void * hilo_lectura (void * thread_args) {

//Variables auxiliares
int fds = 0;
int block = 1;
struct timeval timeout;

//Leo mientras no reciba Ctrl_C
while(fin) {
snmp_select_info(&fds, &fdset, &timeout, &block);
fds = select(fds, &fdset, NULL, NULL, block ? NULL : &timeout);
if (fds){
snmp_read(&fdset);
}
else {
/*
* Si se produce un timeout es que no hay nada que leer de
* un determinado descriptor de fichero
*/
snmp_timeout();
}
}

/*
* Activo variable global, para saber que se ha terminado leer
* y poder cerrar las sesiones
*/
fin_lectura = 0;
return 0;

}



/*
* Polls the provided kafka handle for events.
*
* Events will cause application provided callbacks to be called.
*
* The 'timeout_ms' argument specifies the minimum amount of time
* (in milliseconds) that the call will block waiting for events.
* For non-blocking calls, provide 0 as 'timeout_ms'.
* To wait indefinately for an event, provide -1.
*
* Events:
* - delivery report callbacks (if dr_cb is configured) [producer]
* - error callbacks (if error_cb is configured) [producer & consumer]
```

```
  * - stats callbacks (if stats_cb is configured) [producer & consumer]
  *
  * Returns the number of events served.
  */
  void * poll_kafka (void * thread_args) {
  while (fin) {
  /**
  * Returns the current out queue length:
  * messages waiting to be sent to, or acknowledged by, the broker.
  */
  if (rd_kafka_outq_len(rk) > 0)
  rd_kafka_poll(rk, 0);
  }
  /*
  * Cuando llegamos a fin, tenemos que terminar de
  * hacer poll a los que quedan
  */
  while (rd_kafka_outq_len(rk) > 0) {
  rd_kafka_poll(rk, 0);
  }
  return 0;
  }



  /*
  * produce_msg
  * Función que, con la configuracion definida en el main
  * hara de productor, y enviara la cadena pasada como parametro
  */
  void produce_msg (char * cadena) {
  /*
  * Argumentos:
  * Topic: rd_kafka_topic_t *rkt              --> GLOBAL
  * particion: int32_t partitition            --> RD_KAFKA_PARTITION_UA
  *                                                                    O un numero de
particiones (0..N)
  * ------------------------------------------------------------------------------------
--
  * ANOTACION--> If partition is RD_KAFKA_PARTITION_UA the configured partitioner will
  * be run for each message (slower), otherwise the messages will be enqueued
  * to the specified partition directly (faster).
  * ------------------------------------------------------------------------------------
--
  * flags: int msgflags                              --> RD_KAFKA_MSG_F_FREE
  * ------------------------------------------------------------------------------------
--
  * ANOTACION--> RD_KAFKA_MSG_F_FREE - rdkafka will free(3) 'payload' when it is done
  *                       with it.
  * ------------------------------------------------------------------------------------
--
  * payload: void *payload                    --> datos a enviar --> como cadena ¿?
  * longitud: size_t len                            -->    longitud de los datos
  * key: const void *key                            --> DE MOMENTO NO
  * keylen: size_t keylen                    -->    ||
  * ------------------------------------------------------------------------------------
--
  * ANOTACION--> 'key' is an optional message key of size 'keylen' bytes, if non-NULL
it
  * will be passed to the topic partitioner as well as be sent with the
  * message to the broker and passed on to the consumer.
  * ------------------------------------------------------------------------------------
--
  * msg_opaque: void *msg_opaque             --> DE MOMENTO NO
  * ------------------------------------------------------------------------------------
--
  * ANOTACION--> 'msg_opaque' is an optional application-provided per-message opaque
  * pointer that will provided in the delivery report callback (`dr_cb`) for
```

```
  * referencing this message.
  * ------------------------------------------------------------------------------
--
  */

  //RD_KAFKA_PARTITION_UA -> asignado automatico de la particion
  //RD_KAFKA_MSG_F_FREE -> Liberara los datos cuando termine
  if (rd_kafka_produce(rkt, RD_KAFKA_PARTITION_UA, RD_KAFKA_MSG_F_FREE,
  cadena, sizeof(cadena), NULL, 0, NULL) == -1) {
  LOG_PRINT("Failed to produce");
  }
  mensaj=NULL;
  }



  int print_result (int status, struct snmp_session *sp, struct snmp_pdu *pdu)
  {
  char buf[1024];
  struct variable_list *vp;
  int ix;
  struct timeval now;
  struct timezone tz;
  struct tm *tm;

  gettimeofday(&now, &tz);
  tm = localtime(&now.tv_sec);
  fprintf(stdout, "%.2d:%.2d:%.2d.%.6d ", tm->tm_hour, tm->tm_min, tm->tm_sec,
(int)now.tv_usec);
  switch (status) {
  case STAT_SUCCESS:
  vp = pdu->variables;
  if (pdu->errstat == SNMP_ERR_NOERROR) {
  while (vp) {
  snprint_variable(buf, sizeof(buf), vp->name, vp->name_length, vp);
  fprintf(stdout, "%s: %s\n", sp->peername, buf);
  vp = vp->next_variable;
  }
  }
  else {
  for (ix = 1; vp && ix != pdu->errindex; vp = vp->next_variable, ix++)
  ;
  if (vp) snprint_objid(buf, sizeof(buf), vp->name, vp->name_length);
  else strcpy(buf, "(none)");
  fprintf(stdout, "%s: %s: %s\n",
  sp->peername, buf, snmp_errstring(pdu->errstat));
  }
  return 1;
  case STAT_TIMEOUT:
  fprintf(stdout, "%s: Timeout\n", sp->peername);
  return 0;
  case STAT_ERROR:
  snmp_perror(sp->peername);
  return 0;
  }
  return 0;
  }
```

hilos.h

```
  #ifndef DATOS_H
  #include "datos.h"
  #endif

  #include "poller.h"

  #define ESPERA_REINTENTOS 4;


  /*
   * Funcion: void * poller (void * thread_args)
   *
   *         Funcion ejecutada por hilos. Se encarga del envio de pdu continuo.
   *         Recibira una lista enlazada de los hosts que tiene que monitorizar
   */
  void * poller (void * thread_args);

  /*
   * Funcion: int procesa_pdu(int operation, struct snmp_session *sp,
   *   int reqid, struct snmp_pdu *pdu, void *magic)
   *
   *         Funcion que se definirá de callback en las sesiones snmp.
   *         Se llamara cada vez que se recibe una trama pdu.
   *         SU funcion principal sera enviar por kafka los
   *         datos de la PDU recibida
   *
   *         Parametros callback:
   *         1)int operation                          --> the possible operations are
RECEIVED_MESSAGE and TIMED_OUT
   *         2)struct snmp_session* session   --> the session that was authenticated
using community
   *         3)int reqid                              --> the request ID identifying the
transaction within this session. Use 0 for traps
   *         4)struct snmp_pdu* pdu                   --> a pointer to PDU information.
You must copy the information because it will be freed elsewhere
   *         5)void* magic                            --> a pointer to the data for
callback()
   */
  int procesa_pdu(int operation, struct snmp_session *sp, int reqid, struct snmp_pdu
*pdu, void *magic);

  /*
   * Funcion: productor_kafka()
   *         Funcion que acepta una PDU, extrae sus datos
   *         y lo incluye al broker y al topic ya iniciado
   */
  void productor_kafka (struct snmp_pdu *pdu, struct snmp_session *sp);

  /*
   * Funcion: void * hilo_lectura (void * thread_args)
   *
   *         Funcion ejecutada en un hilo. Se encargará de recibir las
   *         respuestas a las pdu enviadas por los poller a traves de la
   *         funcion send
   */
  void * hilo_lectura (void * thread_args);

  /*
  * Polls the provided kafka handle for events.
  *
  * Events will cause application provided callbacks to be called.
  *
  * The 'timeout_ms' argument specifies the minimum amount of time
  * (in milliseconds) that the call will block waiting for events.
  * For non-blocking calls, provide 0 as 'timeout_ms'.
  * To wait indefinately for an event, provide -1.
```

```
  *
  * Events:
  * - delivery report callbacks (if dr_cb is configured) [producer]
  * - error callbacks (if error_cb is configured) [producer & consumer]
  * - stats callbacks (if stats_cb is configured) [producer & consumer]
  *
  * Returns the number of events served.
  */
  void * poll_kafka (void * thread_args);

  /*
   * produce_msg
   * Función que, con la configuracion definida en el main
   * hara de productor, y enviara la cadena pasada como parametro
   */
  void produce_msg (char * cadena);

  int print_result (int status, struct snmp_session *sp, struct snmp_pdu *pdu);
```

9.4 Anexo 4: Función principal

poller.c

```
  #include "poller.h"

  #include "logger.h"
  #include "estructura.h"
  #include "hilos.h"
  #include "parseo.h"
  #include "config.h"
  #include "carga.h"

  int main (int argc, char * argv[]) {


  //VARIABLES
  int i;
  int k;
  size_t l;
  unsigned char *ptc;

  //ARGUMENTOS -> 1) Fichero configuracion 2) Fichero monitorizacion
  if (argc != 3){
  fprintf(stdout,"Se necesitan dos argumentos\n");
  fprintf(stdout,"Uso: poller [ruta fichero configuracion] [ruta fichero
monitorizacion]\n");
  }
  else {
  LOG_PRINT("Iniciando el programa....");

  //Puntero global al fichero de configuracion
  fich_conf = argv[1];
  fich_monit = argv[2];

  //Inicializacion de variables globales
  fin = 1;
  lista_hosts = NULL;
  estruct_memoria=0;
  hilos_lanzados = 0;
  mensaj=NULL;

  p_lectura = &lectura;
  p_kafka = &kafka;

  //Inicializamos el descriptor de fichero
  FD_ZERO(&fdset);
```

```
  /*
  * Registro manejadores:
  * - SIGINT -> Detecta cierre e inicia un cierre seguro
  * - SIGHUP -> Señal para la recarga de la configuración
  */
  signal(SIGINT, handlerCierre);
  signal (SIGHUP, recargaConfig);

  //Inicializacion MIBs
  netsnmp_init_mib();
  //TODO desde fichero configuracion
  add_mibdir("/usr/local/share/snmp/mibs");

  //------------------------------CONFIGURACION------------------------------
  if(!carga_config_ini()) {
  estruct_memoria=1;
  imprime_lista();

  //-------------------------------KAFKA-------------------------------
  /*do {
  fprintf(stdout,"Intento de cargar configuración de Kafka\n");
  config_kafka();
  } while(config_kafka==1 || fin==0);*/

  if (error_kafka==0) {

  //-------------------------------ESTRUCTURA-------------------------------
  if (fin != 0)
  {
  config_estructura ();
  procesa_oid();
  imprime_cortes(lista_hosts);

  //-------------------------------HILOS-------------------------------
  /*
  * Tenemos la lista marcada en las variables corte
  * A cada hilo le pasaremos el puntero a un nodo, y
  * monitorizara hasta que encuentre corte = 1 o NULL
  */
  if (fin != 0) {

  //Creacion de tabla de almacenamiento de punteros de hilos
  hilos = (pthread_t *) malloc(num_hilos*sizeof(pthread_t));

  //-----------------HILO POLLER-----------------
  lanzado_hilos();
  fprintf(stdout,"\n----------Tabla de hilos:\n");
  for (k=0;k<num_hilos;k++) {
  fprintf(stdout,"[%d]%u\t",k,hilos[k]);
  }
  fprintf(stdout,"\n");

  sleep(5);
  //-----------------HILO LECTURA-----------------
  //Lanzamos un hilo para la lectura
  pthread_create(p_lectura,NULL,hilo_lectura,NULL);

  //-----------------HILO POLL KAFKA-----------------
  //pthread_create(p_kafka,NULL,poll_kafka,NULL);


  //--------------------------------------------------------------------------------
--------------------
  //-----------------------------------------------CTRL C-----------------------------
--------------------
  //--------------------------------------------------------------------------------
--------------------
```

```
sleep(5);


if (hilos_lanzados > 0) {
/*
* Tenemos que esperar la finalizacion de cada hilo
* usando la tabla de hilos
*/
for (i=0; i<num_hilos; i++) {

pthread_join(hilos[i],NULL);
fprintf(stdout,"Espero hilo n° %d\n",i);
}

//Esperamos el hilo de lectura
pthread_join(lectura,NULL);
fprintf(stdout,"Espero hilo lectura\n");
pthread_join(kafka,NULL);
fprintf(stdout,"Espero hilo Kafka\n");

cierre_sesiones();

//Finalizamos el productor KAFKA
// Destroy topic
rd_kafka_topic_destroy(rkt);
// Destroy the handle
rd_kafka_destroy(rk);
// Let background threads clean up and terminate cleanly.
rd_kafka_wait_destroyed(2000);

//Durante la eliminacion se cierran las sesiones
liberacion_lista();
free(acceso_hilos);
acceso_hilos=NULL;
free(hilos);
hilos=NULL;

//Comprobacion liberacion
if (lista_hosts == NULL)
fprintf(stdout,"Lista liberada correctamente\n");

}
else {
//Error en lanzado de hilos
LOG_PRINT("Error Fatal en lanzado de hilos");
/*//Finalizamos el productor KAFKA
// Destroy topic
rd_kafka_topic_destroy(rkt);
// Destroy the handle
rd_kafka_destroy(rk);
// Let background threads clean up and terminate cleanly.
rd_kafka_wait_destroyed(2000);*/
liberacion_lista();
free(acceso_hilos);
free(hilos);
}
}
}
}//FIN KAFKA
else {
LOG_PRINT("Error Fatal en carga de configuracion de Kafka");
}
}
else {
//Se ha producido un error, SALIR
LOG_PRINT("Error Fatal en carga de configuracion");
}
```

```
} //Error de argumentos

return 0;

}



/*
* Funcion: void handlerCierre (int dummy)
*          Funcion que detectara la señal de cierre.
*          Este handler, ante la recepcion de Ctrl-C (SIGINT)
*          cerrara todas las sesiones y finalizara el programa
*/
void handlerCierre (int dummy) {
fin = 0;
}

/*
* Funcion: void recargaConfig(int dummy)
*          Funcion que recarga la configuracion y creara la estructura dinamica
*          Borrara y reconstruira cortes
*/
void recargaConfig(int dummy) {

/*
* Accedemos a la recarga usando una variable
* mutex que impedira acceder a la memoria compartida
* a los hilos del motor de envio de peticiones
*/
pthread_mutex_lock(&lock);
sleep(5);
carga_config();
procesa_oid();
pthread_mutex_unlock(&lock);

imprime_lista(lista_hosts);
imprime_cortes(lista_hosts);

}
```

poller.h

```
#ifndef DATOS_H
#include "datos.h"
#endif


/*
 * Funcion: void handlerCierre (int dummy)
 *          Funcion que detectara la señal de cierre.
 *          Este handler, ante la recepcion de Ctrl-C (SIGINT)
 *          cerrara todas las sesiones y finalizara el programa
 */
void handlerCierre (int dummy);


/*
 * Funcion: void recargaConfig(int dummy)
 *          Funcion que recarga la configuracion de monitorizacion
 *          Modificara la estructura, luego se requiere un acceso
 *          seguro mediante variables de mutex
 */
void recargaConfig(int dummy);
```

```
//         -->Datos generales de configuración<--

//Ruta a ficheros de configuracion
char * fich_conf;
char * fich_monit;

//Kafka
char *brokers;
char *topics;



//         -->Variables globales<--

//Estadisticas
int num_hilos;
int num_hosts;
int resto;
int host_hilo;

//Flag que indica que ya hay una estructura en memoria;
int estruct_memoria;

//Contador de hilos que se han lanzado
int hilos_lanzados;

//Flag que indica el fin del programa
int fin;

//Flag que indica que el hilo de lectura ha finalizado
int fin_lectura;

//Mutex para acceso a memoria compartida
pthread_mutex_t lock;

//Descriptor de fichero
fd_set fdset;

//Configuracion de kafka
static rd_kafka_t *rk;
rd_kafka_topic_t *rkt;
char * mensaj;

//Errores en ficheros
int error_estdin;
int error_confic;
int error_kafka;


//         -->Hilos<--

//Hilo de lectura
pthread_t lectura;
pthread_t * p_lectura;

//Hilos de Kafka
pthread_t kafka;
pthread_t * p_kafka;




//         -->Memoria compartida<--

//Lista enlazada con la informacion de monitorizacion
st_host * lista_hosts;

//Tabla de punteros a los hilos
pthread_t * hilos;
```

```
//Tabla almacenar los accesos de los diferentes hilos
st_host ** acceso_hilos;
```

datos.h

```
#ifndef DATOS_H
#define DATOS_H
#endif


#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <signal.h>
#include <string.h>
#include <stdarg.h>
#include <stdlib.h>
#include <unistd.h>

//Libreria de NET_SNMP
#include <net-snmp/net-snmp-config.h>
#include <net-snmp/net-snmp-includes.h>

//Libreria para parsear fichero JSON
#include <jansson.h>

//Libreria para producir mansajes KAFKA
#include <librdkafka/rdkafka.h>


typedef struct oid {
  int id;
  char * name;
  char * oid_name;

  oid Oid[MAX_OID_LEN];
  size_t OidLen;

  int enviado_id;
  int ultimo_reqid;

  long int t_monitor;
  long int t_actual;

  struct oid *next;
}st_oid;

typedef struct host {
     int id;
     char * name;
     char * ip;
     char * community;
     int version;

     int corte;
     int fallo_sesion;

     struct snmp_session sesion;
     struct snmp_session *punt_sesion;

     st_oid *oids;

     struct host *next;
}st_host;
```

9.5 Anexo 5: Otras funciones

logger.c

```
#include "logger.h"

FILE *fp ;
static int inicio = 0;

char* print_time()
{
    time_t t;
    char *buf;
    time(&t);
    const char * timechar =ctime(&t);
    size_t tam_time = strlen(timechar);
    buf = (char*)malloc(tam_time+ 1);
    snprintf(buf,strlen(timechar),"%s ", timechar);
    return buf;
}

void log_print(char* filename, int line, char *fmt,...)
{
    va_list         list;
    char            *p, *r;
    int             e;

    if(inicio > 0)
      fp = fopen ("poller-log.txt","a");
    else
      fp = fopen ("poller-log.txt","w");

    char * ptime = print_time();
    fprintf(fp,"%s ",ptime);
    free(ptime);
    va_start( list, fmt );

    for ( p = fmt ; *p ; ++p )
    {
        if ( *p != '%' )
            fputc( *p,fp );
        else
        {
            switch ( *++p ) {
            case 's':  {
                r = va_arg( list, char * );
                fprintf(fp,"%s", r);
                continue;
            }
            case 'd': {
                e = va_arg( list, int );
                fprintf(fp,"%d", e);
                continue;
            }
            default:
                fputc( *p, fp );
            }
        }
    }
    va_end( list );
    fprintf(fp," [%s][line: %d] ",filename,line);
    fputc( '\n', fp );
    inicio++;
    fclose(fp);
}
```

logger.h

```
//logger.h
/*
 * Uso:
 * LOG_PRINT("Hello World ");
 * LOG_PRINT("Zing is back !!! %s %d",s,x++);
 */
#ifndef DATOS_H
#include "datos.h"
#endif

#include "poller.h"

#define LOG_PRINT(...) log_print(__FILE__, __LINE__, __VA_ARGS__ )
```