**Gestión de Redes**

**PRÁCTICA 1:**

**IMPLEMENTACIÓN DE UN AGENTE SNMPv1**

Alejandro Antonio Moreno Sancho

Pablo de Juan Fidalgo

Ingeniería Telemática

Curso 2020/2021

**Índice**

[**Introducción** 3](#_Toc60859182)

[**Funciones implementadas** 4](#_Toc60859183)

[**Esquema de la MIB** 7](#_Toc60859184)

[**Funcionalidades Implementadas** 8](#_Toc60859185)

[**Conclusión** 9](#_Toc60859186)

# **Introducción**

El siguiente documento contiene una breve síntesis del agente SNMPv1 implementado durante la práctica.

En el primer punto se explica el uso de cada una de las funciones implementadas incluidas dentro del archivo ag\_PabloAlejandro.cpp.

Posteriormente se muestra un esquema de la MIB, tanto en un archivo de texto en formato *.mib* para que pueda ser interpretada por *ManageEngine MibBrowser* como en el archivo *ag\_PabloAlejandro.cpp* ya que en nuestro caso ésta se carga de forma estática.

En el siguiente punto se detallan las funcionalidades que se han implementado en el agente.

Para finalizar se realiza una corta opinión y mención a los programas usados.

# **Funciones implementadas**

void print\_hex(const char\* buff, unsigned int l);

Se pasa como parámetro un array de bytes y la longitud de este e imprime por pantalla cada byte en decimal. Se ha utilizado principalmente para debuggear la recepción de paquetes y creación de estos para ser enviados.

void read\_integer(const char\* buff, uint8\_t L, nvalor\* V);

Dados un array de bytes y una longitud introduce en el parámetro *V* el valor integer equivalente a ese array de bytes.

void read\_octetstring(const char\* buff, uint8\_t L, nvalor\* V);

Crea espacio en memoria en el puntero V e introduce la cadena que contiene *buff* donde *L* es su longitud.

void read\_ipaddress(const char\* buff, uint8\_t L, nvalor\* V);

Crea espacio en memoria y copia el array de bytes contenido en *buff* que representa una dirección IP a *V* donde *L* es la longitud del array de bytes *buff*.

void read\_oid(const char\* buff, uint8\_t L, nvalor\* V);

Crea espacio en memoria y copia el array de bytes contenido en *buff* que representa un ObjectIDentifier en *V* donde *L* es la longitud del array de bytes *buff*.

void final\_oid(char\* oid, const uint8\_t\* cad, uint8\_t L);

Dado el puntero a un array de bytes oid que representa un ObjectIDentifier se transforma a formato de cadena añadiendo puntos y preparado para ser mostrado por pantalla, *L* es la longitud del array de bytes oid.

uint16\_t read\_tlv(const char\* buff, uint8\_t\* T, uint8\_t\* L, nvalor\* V);

Dado el puntero a los datos recibidos por el agente en *buff*, se lee el siguiente TLV (Type Length Value).

Dentro de esta función se utilizan las funciones anteriores para leer el valor dependiendo del tipo que sea. Los tipos que puede haber son: Integer, Octet String, Null, OID, IpAddress, Sequence Of, Get, GetNext, Set. Si el tipo es Sequence Of vuelve a leer el siguiente TLV recursivamente hasta que no sea de este tipo antes de salir de la función. Devuelve el número de bytes leídos del buffer hasta el próximo TLV.

nodo\* buscarOID(nodo\* MIB, char\* oid);

Dado el nombre de una instancia contenida en el puntero oid devuelve un puntero hacia el nodo correspondiente a dicha instancia, si no existe devuelve NULL.

MIB es un puntero al primer nodo de la MIB.

nodo\* buscarNextOID(nodo\* MIB, char\* oid);

Dado el nombre de una instancia contenida en el puntero oid, encuentra el nodo correspondiente a dicha instancia y devuelve un puntero al nodo siguiente a este. Siempre y cuando este nodo exista o el nodo encontrado no es el último, en caso contrario devuelve NULL.

size\_t oidToBytes(char\* oid, uint8\_t\* bytesoid);

Dada una cadena que contiene el nombre de instancia separado por puntos listo para ser mostrado por pantalla la transforma en un array de bytes para ser colocado en un paquete. Devuelve el tamaño del array de bytes *bytesoid* que contiene dicho nombre de instancia en bytes.

size\_t create\_response(nodo \* MIB, int requestid, uint8\_t operation, const char\* oid, char\* buff, size\_t l, uint16\_t VBL, uint8\_t T, nvalor V, int error, uint16\_t\* ain, SOCKET s);

Una vez leídos todos los campos del mensaje request identifica el tipo de operación que se requiere y crea la respuesta con ayuda de las funciones anteriormente explicadas. Sus parámetros son los siguientes:

nodo \* MIB 🡪 Puntero que apunta al primer nodo de la MIB.

int requestid 🡪 id de la petición recibida.

uint8\_t operation 🡪 Operación que se pide.

const char\* oid 🡪 Nombre de la instancia sobre la que se quiere realiza la operación.

char\* buff 🡪 Buffer que contiene los bytes del mensaje request y que se modificará para crear el mensaje response listo para ser enviado.

size\_t l 🡪 Longitud del buffer recibido.

uint16\_t VBL 🡪 Guarda la posición del primer valor de la VarBindList para poder acceder a él fácilmente.

uint8\_t T 🡪 Puntero que apunta a una variable de la función main que se utiliza para almacenar el tipo de un TLV leído.

nvalor V 🡪 Puntero que apunta a una variable de la función main que se utiliza para almacenar el valor de un TLV leído.

int error 🡪 variable en la que almacenamos el error que se va a enviar si se encuentra alguno. Se tienen encuentra cualquier tipo de los errores posibles dentro de las funciones Get, GetNext y Set tanto sobre tipos escalares como tablas.

uint16\_t\* ain 🡪 Array que almacena las posiciones de los bytes de tipo de: SNMP Message, SNMP PDU, VarBindList y VarBind para que se pueda acceder fácilmente a las longitudes y ser modificadas para no tener que crear una respuesta de cero y hacerla modificando el request recibido.

SOCKET s 🡪 Variable socket por el cual se envían las respuestas, aunque la respuesta se envíe en la función main fuera de la función actual es necesario por si se envían mensajes del tipo trap cuando ocurre que en una operación set se intenta escribir un valor que no está dentro de los límites establecidos dentro de la MIB.

# **Esquema de la MIB**

El esquema de la MIB que se ha creado tanto en el archivo *MIB.mib* como el que se carga de forma estática en el programa es el siguiente:

Diagrama

Descripción generada automáticamente

Dentro del archivo *ag\_PabloAlejandro.cpp* se define la estructura *nodo* que tiene los siguientes campos:

int tipo\_obj; 🡪 Almacena el tipo de objeto: escalar (0), nodo tabla (1), nodo fila (2), nodo columna (3).

int tipo\_de\_dato; 🡪 Almacena el tipo de dato: *Integer* (0), *Octet String* (1), *IpAddress (2)*.

int acceso; 🡪 Define el nivel de accesibilidad de la instancia: *non-accessible* (0), *read-only* (1), *read-write* (2).

char oid[2048]; 🡪 Almacena el valor del *Object IDentifier*.

char instancia[2048]; 🡪 Almacena el identificador de la instancia.

nvalor tipo\_valor; 🡪 Almacena el valor independientemente del tipo que sea.

int max\_bound; 🡪 Define el valor máximo que puede escribirse en un objeto de tipo *Integer* que tenga límite superior.

int min\_bound; 🡪 Define el valor mínimo que puede escribirse en un objeto de tipo *Integer* que tenga límite inferior.

struct nodo \*sig; 🡪 Apunta al siguiente nodo de la MIB.

struct nodo \*indice; 🡪 Apunta al nodo columna índice de esta tabla, solo se utiliza si es un nodo fila.

# **Funcionalidades Implementadas**

La MIB se crea y compila correctamente, se incluyen objetos a la MIB que mejoran la gestión del dispositivo (tablaRevisiones). Este objeto añade el nombre de la persona que realizó una inspección de seguridad y utiliza como índice el día en el que se realizó.

Se implementa un servidor UDP con el nombre *ag\_PabloAlejandro.cpp* donde el puerto es configurable dentro del programa mediante el define *UDPPORT*.

La MIB se carga estáticamente con objetos escalares (*INTEGER, OCTET STRING, OBJECT IDENTIFIER, IpAddress*).

Se distinguen entre objetos que pueden escribirse (*read-write*) y los que no (*read-only*). También se distinguen los objetos *non-accessible* (nodos de la tabla)

La MIB se carga estáticamente con objetos tabulares donde solo existe un índice y las columnas pueden ser *read-only* o *read-write*.

Se procesan mensajes que contienen un solo objeto (par objeto valor) correspondientes a un tipo escalar o a un tipo tabular.

Se tratan correctamente los códigos de error, tanto de la comunidad si el valor no es “public” avisando de que la comunidad no es válida y esperando el siguiente mensaje, como de errores de las operaciones (set de un valor *read-only*; set fuera de los límites; get, getnext o set de una instancia no existente o *non-accessible*)

Las operaciones de lectura funcionan correctamente en todos los tipos (*Integer*, *Octet* *String*, *IpAddress*, *ObjectIdentifier*) y dentro de tablas con índices de tipo *Integer* o *IpAddress*.

Con identificación de errores y generando el código de error específico en la respuesta.

Las operaciones de escritura funcionan correctamente en todos los tipos y dentro de tablas con índices de tipo *Integer* o *IpAddress*.

Las operaciones de escritura también distinguen los distintos tipos de error y son enviados en la respuesta.

En cuanto al recorrido de la MIB (operación *GetNext*) funciona correctamente con todos los tipos de objeto escalar y tabular identificando los errores de la operación y siendo enviados en el mensaje de respuesta, también para ambos tipos de tablas, con índice *Integer* y con índice *IpAddress*.

En cuanto a la gestión de notificaciones el agente detecta errores en las operaciones set cuando el valor que se pretende escribir es superior o inferior a los límites establecidos dentro de la MIB, y crea un paquete TRAP y es enviado correctamente, pudiendo comprobar esto con aplicaciones como Wireshark para visualizar el paquete de forma correcta.

# **Conclusión**

Comparando nuestro conocimiento actual sobre el protocolo de administración de red SNMP con el conocimiento previo a realizar la práctica podemos concluir que es de gran utilidad para afinar conceptos que no se terminaran de adquirir sobre la teoría explicada en clase. Consideramos no solo útil para asentar conceptos entendidos si no necesaria para aprender otros conceptos a los cuales es posible que no se le prestase tanta atención durante las clases teóricas o durante el estudio de esta.

Los principales problemas que nos encontramos podían solucionarse perfectamente mediante el uso de la información obtenida de clase o mediante la búsqueda en Internet.

También consideramos especialmente útil el trabajar en grupos de 2 personas, ya que es una forma de aprender a distribuir el trabajo con otro compañero. Esto ha hecho en nuestro caso que sea necesario una planificación antes de comenzar, lo cual ha sido clave para tener una idea clara y concisa de lo que queremos lograr y cómo hacerlo antes de empezar a teclear sin fundamento. El compartir el trabajo nos ha ayudado a la buena práctica de asignación de tareas independientes para el avance rápido del proyecto a la vez que ambos compartíamos el conocimiento aprendido para la comprensión total de programa.

Además del uso de aplicaciones externas como G*itHub* y *git*, aunque hayan consumido tiempo para aprender su uso, han sido de gran beneficio para la organización y trabajo simultáneo en la práctica. Otras herramientas como Wireshark han sido fundamentales. Su uso hasta ahora en el resto de las asignaturas del grado había sido con ejemplos muy concretos, pero ha sido beneficioso ver que tiene un uso bastante real cuando nos encontrábamos recibiendo el mensaje desde *ManageEngine MibBrowser* a nuestra aplicación y a la hora de crear y enviar las tramas de respuesta.

Finalmente consideramos que la práctica ha sido muy satisfactoria para ambos.