UNIVERSIDAD POLITÉCNICA DE VALENCIA

ESCUELA POLITÉCNICA SUPERIOR DE GANDIA

Grado en Ing. Sist. de Telecom., Sonido e Imagen

"Diseño y desarrollo de una aplicación web de gestión de dispositivos de una red mediante protocolo SNMP"

TRABAJO FINAL DE GRADO

Autor/a:
González Roselló, Héctor

Tutor/a:
Boronat Seguí, Fernando

Empresa:
Main Informática, SL

GANDIA, 2015

Índice

1. RESUMEN

En el Trabajo de Fin de Grado que se presenta, se ha realizado un programa integrado como módulo de un ERP de código abierto llamado Odoo, que funciona con una arquitectura de red cliente-servidor. Dicho módulo permite a un PC cliente localizar dispositivos conectados a la red, encontrar las tablas MIB de cualquiera de ellos y configurarlos para que realice un seguimiento del valor de un determinado OID. Todo ello mediante el uso del protocolo de red SNMP.

Además se han programado diferentes funciones para gestionar los valores de dichos OID:

- Gráfica de los valores registrados en las últimas 24h.
- Aviso mediante e-mail si el valor del OID cumple unas condiciones establecidas por el usuario.
- Cambio del valor de un OID de otro dispositivo de la red, si se cumplen las condiciones antes mencionadas.
- Introducción de comandos en la consola del servidor, si se cumplen las condiciones establecidas.

Palabras clave: **SNMP, Gestión de red**

SUMMARY

In this Final Degree Project, I've made an integrated program as a module of an open-code ERP named Odoo, that works with a client-server network architecture. This module allows a PC to locate network connected devices, find the MIB chart of any of them and configure them to make the monitoring of the value of a certain OID. All of it is realised through the use of the SNMP network protocol.

Furthermore, I've programmed some functions to manage the above mentioned OID:

- A graph that contains the last 24h registered values.
- An e-mail warning that notices if the OID value accomplishes some predefined conditions by the user.
- A function that allows to change the value of an OID, in other different device, if the current OID accomplishes the conditions.
- A function that automatically enters a command in the server terminal, if it accomplishes the conditions.

Key words: **SNMP, Network management**

2. IDEA GENERAL DEL PROYECTO Y OBJETIVOS

2.1. Idea general

Este proyecto trata de ofrecer una serie de servicios informáticos que tratan de cubrir unas necesidades muy concretas demandadas por una empresa. En concreto, lo que se necesitaba era desarrollar un módulo integrado en un ERP (Enterprise Resource Planning), concretamente en Odoo, anteriormente llamado OpenERP. Dicho software estará instalado en un servidor con un sistema operativo Ubuntu (Debian), versión 14.04 o superior.

Dicho módulo tenía que integrar en el ERP unas funciones que permitieran localizar los MIBS de cualquier dispositivo conectado a una red, seleccionar uno o más OIDS con la información que interesa al usuario y procesarla para guardar registros periódicamente, mostrar gráficos de los datos y avisos al usuario en caso de que el valor del OID cumpla unas condiciones pre-establecidas por el propio usuario.

2.2. Objetivos

- Visualizar un gráfico con el valor de un OID a lo largo del tiempo, de manera que el usuario pueda observar toda la información fácilmente y detectar posibles problemas, en caso de que los haya.

- Crear una tabla donde el usuario introduzca unas condiciones (mayor, menor, igual que un valor, etc.) que, en caso de cumplirse, ejecutarán unas funciones. Dichas condiciones se evaluarán de forma jerárquica, desde la parte superior de la tabla hacia la inferior.

- Función que mandará un aviso al e-mail que determine el usuario en caso de cumplirse alguna de las condiciones establecidas en la tabla anteriormente mencionada. Dicho e-mail deberá configurarse mediante un servidor SMTP de correo electrónico. Así mismo el e-mail estará asociado a un usuario propio del Odoo, para localizarlo más fácilmente.

- Función que cambiará un valor de un OID de cualquier dispositivo de la red automáticamente si se cumple alguna de las condiciones establecidas en la tabla de condiciones antes mencionada. Esta función tiene varias aplicaciones, aunque lo que se pretende es que funcione como un sistema de aviso al usuario mediante la activación de algún tipo de emisor externo de luz o sonido. Por ejemplo cambiar un valor de un OID que corresponda a un LED de un dispositivo de la red, de manera que una persona pueda visualizarlo y actuar en caso necesario.

- Función que permitirá introducir comandos en la terminal de Ubuntu del servidor automáticamente en caso de cumplirse alguna condición de la tabla de condiciones. El usuario podrá establecer el comando, que se introducirá en una terminal que se abrirá automáticamente en el PC servidor.

Cabe destacar que las funciones anteriormente mencionadas podrán ejecutarse tanto individualmente como conjuntamente a la vez, según las preferencias del usuario.

3. DESARROLLO DEL PROYECTO

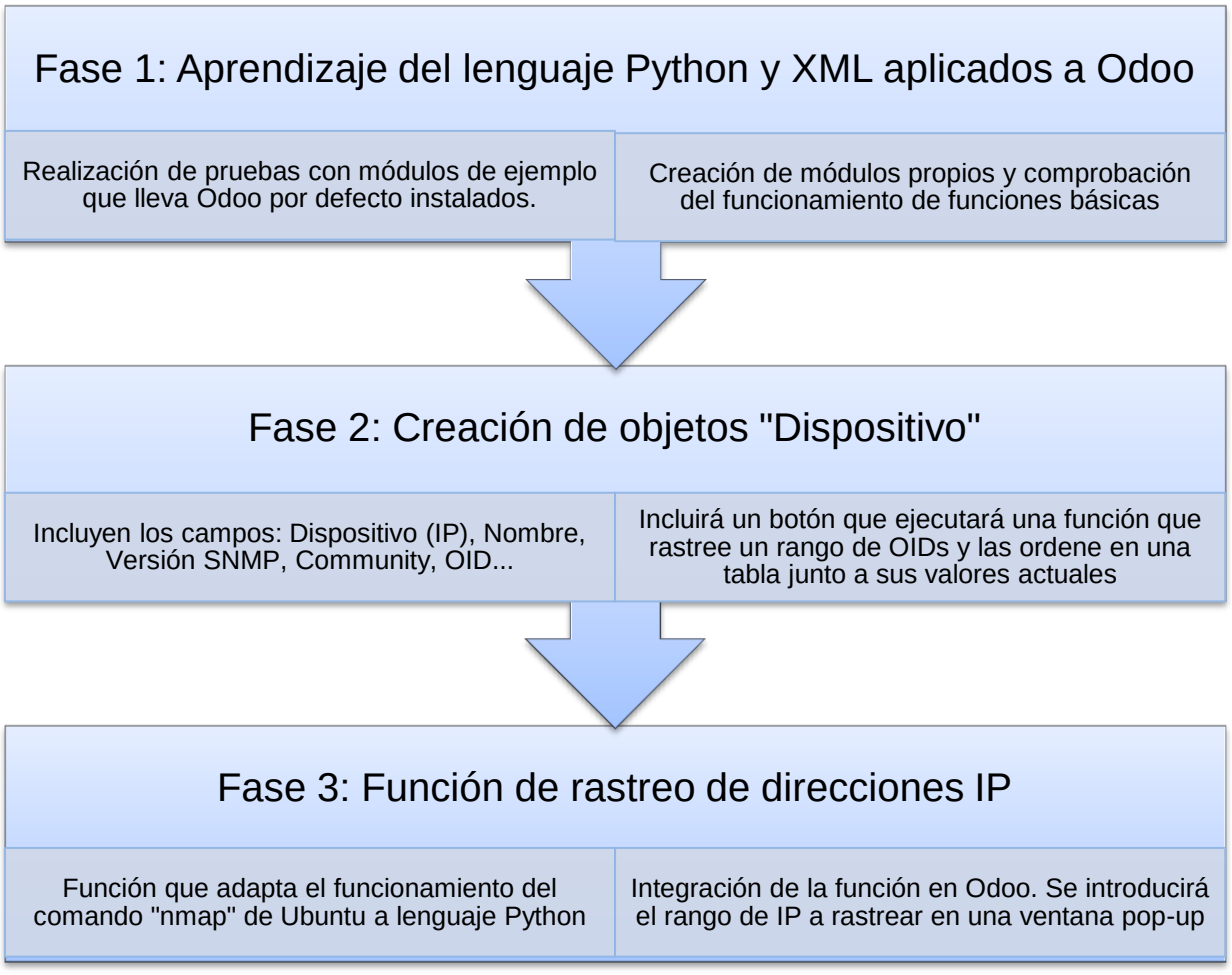
El módulo que se ha desarrollado consta de varios archivos que se instalarán dentro de una carpeta en el directorio de instalación del software Odoo, en Ubuntu.

Estos archivos, que contienen todo el código fuente, están en su mayoría programados mediante el lenguaje de programación Python, que es el que utiliza Odoo para ejecutar funciones e interactuar con la base de datos. La base de datos la crea el usuario en el momento de la instalación y configuración de Odoo, para integrarla en la plataforma, y está basada en el sistema de gestión de bases de datos PostgreSQL.

El lenguaje python nos viene muy bien, ya que se pueden incluir paquetes que nos permitirán incorporar las funciones necesarias para muchas de las operaciones que se llevarán a cabo.

3.1. Fases y partes del desarrollo

Al iniciar el proyecto, éste se dividió en una serie de fases, las cuales a medida que se completaban daban paso a otras nuevas. A continuación podremos observar un esquema con todas las fases que se realizaron, en conjunto, para una visualización más fácil y un resumen muy corto de cada una de sus partes:




Fase 4: Creación de objetos OID y complementación con Dispositivos
Creación de objetos "OID". Incluyen los campos: OID, Nombre, Valor (actual), Dispositivo, Usuario...
Creación automática de objetos OID al buscarlos en el formulario de Dispositivos, estableciendo un OID y valor automáticamente
Fase 5: Creación del temporizador de registros
Función incluida dentro de objetos "MIB" que al activarla iniciará un registro de valores del OID que tengamos seleccionado
Los registros se guardarán en una tabla aparte donde podrán ser consultados en todo momento
Fase 6: Visualización de la gráfica de valores de un OID
Función que se ejecutará al pulsar un botón en el formulario MIB y que mostrará una gráfico de los valores de los registros de un OID
Este gráfico podrá visualizarse tanto en forma de gráfico de barras como en forma de gráfico lineal
Fase 7: Creación de tabla de condiciones
Tabla que contendrá todas las condiciones que se comprobarán en caso de configurar una alerta
Las condiciones se evaluarán de forma escalonada, de arriba hacia abajo.
Fase 8: Creación de funciones de alerta
Se crearán 3 funciones que avisarán al usuario en caso de cumplirse alguna condición de la tabla de condiciones
Las funciones serán: Envío de e-mail, cambio de valor de OID en cualquier dispositivo, introducción de comando de terminal ubuntu
Fase 9: Comprobación de errores y añadido de detalles
Se comprueban posibles errores a la hora de enviar las alertas, y otras funciones
Se añaden algunas funciones extra como una tabla de Hosts para e-mails y un clear del log de registros

4. PROGRAMACIÓN Y REALIZACIÓN DE CADA FASE

4.1. Aprendizaje del lenguaje Python y XML aplicados a Odoo

En esta parte se han consultado ejemplos que Odoo nos proporciona para el aprendizaje del desarrollo de un módulo. La principal fuente de información ha sido la página web oficial de los desarrolladores de Odoo:
https://www.odoo.com/documentation/8.0/howtos/backend.html

En esta web nos enseñan muy básicamente a crear nuestros módulos desde cero. Se nos van detallando los pasos a seguir para poco a poco montar un módulo en Odoo. Cabe destacar que aunque es un tutorial muy básico hay algunas partes en las que no se termina de entender qué es lo que están haciendo y la información de algunos temas escasea. Muchos de los problemas que han surgido a la hora de crear el módulo los hemos tenido que consultar en otras webs especializadas en resolver dudas de programación (www.stackoverflow.com) o en la misma web de Odoo, en el apartado de foro/ayuda (https://www.odoo.com/es_ES/forum/help-1).

Obviamente además de toda esta información hemos tenido en todo momento guías y anotaciones sobre el uso básico de Python y XML tales como declaración de variables, uso de bucles… tanto en formato digital como en forma de apuntes escritos a mano.

Para empezar el desarrollo del primer módulo es imprescindible crear una serie de archivos que iniciarán y permitirán que Odoo ejecute dicho módulo. Los desarrolladores nos proporcionan para ello un comando de Ubuntu que facilitará esta labor, creando unos archivos por defecto que luego modificaremos nosotros mismos según nuestras necesidades. El comando es el siguiente:

```
$ odoo.py scaffold <nombre del módulo> <dirección directorio>
```

El comando automáticamente creará muchos archivos, de los cuales los más importantes para nosotros (y los que vamos a modificar) van a ser:

- **__openerp__.py:** Archivo python donde se guarda la información referente al módulo. Importante incluir aquellos directorios con archivos que se vayan a utilizar que no estén ubicados en el directorio raíz del módulo.
- **__init__.py:** Archivo python en el que se importan en el servidor aquellos archivos (python) que ejecutarán clases y funciones.
- **models.py:** Principal archivo python donde vamos a tener prácticamente todas las clases y funciones que se ejecutarán en el módulo. Cada clase corresponderá a un teórico objeto, el cual podrá ejecutar funciones.

Para representar toda la información programada en estos archivos python, debemos crear un archivo XML que actuará como una vista. Es decir, mediante el XML se representarán visualmente todos los datos de python en formato web, de manera que al entrar al servidor y seleccionar el módulo aparezcan los formularios, listas y demás personalizados como nosotros queremos. En nuestro módulo para un mayor orden, el archivo XML lo situaremos en un directorio llamado *views* que no viene creado por defecto.

A continuación vamos a ver cómo han quedado nuestros archivos **__openerp__.py** y **__init__.py**, ya que a partir de ahora sólo nos vamos a centrar en el archivo **models.py** que es donde está programado lo esencial del módulo.

```
{
  'name': "snmp_network",

  'summary': """
      SNMP network""",

  'description': """
      Aplicación para gestionar los dispositivos de una red
  """,

  'author': "Main Informática",
  'website': "http://www.main-informatica.com",


  # for the full list
  'category': 'Uncategorized',
  'version': '0.1',

  # any module necessary for this one to work correctly
  'depends': ['base'],

  # always loaded
  'data': [
      # 'security/ir.model.access.csv',

      'views/walk.xml',

  ],
  # only loaded in demonstration mode
  'demo': [
      'demo.xml',
  ],
}
```

Imagen 1: __openerp__.py

Como se puede observar en este archivo realmente tenemos información muy básica del módulo: nombre, descripción, versión, autor, website, etc.

Es imprescindible declarar en el apartado 'data' los archivos XML que no estén en el directorio raíz del módulo, en caso contrario no se va a ejecutar el módulo ya que no detectará donde está el XML que al fin y al cabo es el que representa visualmente la información. En este caso como se puede observar hemos declarado nuestro XML (walk.xml) que tenemos en el directorio *views* en nuestro módulo.

Imagen 2: __init__.py

En este archivo como hemos dicho anteriormente se importan los archivos python que usaremos en el módulo. El archivo *controllers(.py)* contiene información del sistema por defecto que no debemos modificar para nada, *models(.py)* como ya hemos dicho va a contener todas las clases de los objetos y sus funciones que detallaremos más adelante, y *wizards(.py)* que se explicará más adelante, contendrá objetos que se representarán en ventanas pop-up aparte (llamadas wizards).

4.2. Objeto Dispositivo

Para empezar a registrar valores, primero tenemos que indicar de alguna manera en qué dispositivo de la red se encuentra el MIB. Para ello se ha optado por dos métodos:

- Introducir la IP manualmente, creando un objeto *Dispositivo* e indicando la IP. Esta dirección la tendrá que conocer el usuario previamente y haber comprobado que está correctamente configurada en la red.
- Introducir un rango de IP en formato “xxx.xxx.xx.xx/aa” que nos almacene automáticamente todas las direcciones de los dispositivos conectados a la red. De esta forma podremos comprobar que el dispositivo que nos interesa está configurado correctamente y la función lo detecta.

Los objetos *Dispositivo* podrán ser visualizados de tres formas diferentes: en vista de lista, en vista de formulario o en vista kanban (iconos + texto). Mediante el XML nosotros elegiremos qué campos del objeto se van a visualizar en cada tipo de vista, programando cada una por separado.

A continuación vamos a ver en el archivo **models.py** donde se han declarado los campos y demás variables que contendrá cada objeto *Dispositivo*.

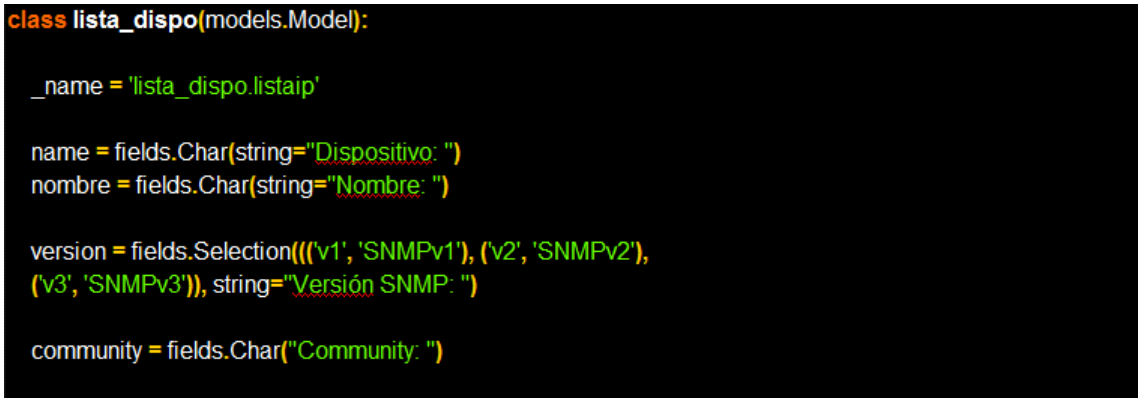


Imagen 3: Campos y variables Dispositivos (1/2)

```
rangomib = fields.Char("OID: ", default = "1.3.6")

userv3 = fields.Char("Usuario SNMPv3: ")
passv3 = fields.Char("Pass SNMPv3: ")

dispomib_id = fields.One2many('lista_mib.listamib', 'dispo_id',
string="Lista MIBs")

imagen = fields.Binary("Imagen", help="Seleccionar imagen")
```

Imagen 4: Campos y variables Dispositivos (2/2)

Podemos ver que se ha creado una clase, que representará a los objetos *Dispositivo*. En este caso el nombre de la clase es *lista_dispo* para facilitar la tarea de programación. Dentro de la clase, encontramos:

- **_name = 'lista_dispo.listaip'**: Este comando indica el nombre por el que el sistema va a reconocer al objeto a la hora de programar, es indispensable introducirlo en todas las clases (cada una el suyo único). Más adelante veremos como lo utilizaremos para apuntar a un objeto en el XML.

- **name = fields.Char(string='"Dispositivo:")**: Campo tipo Char (texto) en el que se registran las IP de los dispositivos, el "string" es una opción para personalizar el texto de la etiqueta (label). La función que veremos en el apartado siguiente rellenará automáticamente este campo.

- **nombre = fields.Char(string="Nombre: ")**: Campo tipo Char en el que el usuario introduce un nombre personalizado para el *Dispositivo*.

- **version = fields.Selection((('v1', 'SNMPv1'), ('v2', 'SNMPv2'), ('v3', 'SNMPv3')), string="Versión SNMP: ")**: Este campo es de tipo Selection, permitirá al usuario elegir entre los 3 tipos de versión de SNMP, dependiendo de la que tenga configurada el dispositivo. En las opciones dentro de los paréntesis el primer texto es la variable interna que representa a una opción y el segundo es el texto que el usuario visualizará (label). Este campo visualmente funcionará como un desplegable en el que el usuario verá las 3 opciones y elegirá la que desee.

- **community = fields.Char("Community: ")**: Campo tipo Char en el que se introducirá el community del dispositivo SNMP. El community es el nombre de un grupo que tendrá asignados unos privilegios dependiendo de cada dispositivo. Por lo general una community "public" nos dejará leer información del MIB pero no modificarla, mientras que una community "private" nos dejará tanto leer como modificar. También cabe destacar que generalmente los dispositivos no disponen de permisos "private" o lo que es lo mismo, de modificar información, por lo que habrá que configurarlo en caso de querer utilizar esta propiedad.

- **rangomib = fields.Char("OID: ", default = "1.3.6")**: Campo que se va a utilizar para la función que localizará los OID del MIB del dispositivo, que se detallará más adelante. En este campo se introducirá el rango de OID que el usuario quiere consultar.

- **userv3 = fields.Char("Usuario SNMPv3: ")**: Usuario en caso de seleccionar SNMP versión 3, que incluye estos parámetros para añadir seguridad.

- **passv3 = fields.Char("Pass SNMPv3: ")**: Contraseña en caso de seleccionar SNMP versión 3, para añadir seguridad.

- **dispomib_id = fields.One2many('lista_mib.listamib', 'dispo_id', string="Lista MIBs")**: Campo a destacar ya que va a ser muy importante a la hora de relacionar los objetos/clases *Dispositivo* y *OID*. Básicamente One2many es una relación entre objetos de Odoo, que asigna para el objeto actual muchos objetos de otro tipo. En este caso en la variable **dispomib_id** se crea una lista de valores (ids) que indicarán todos los objetos tipo *OID* que estén ligados a un dispositivo. Un *Dispositivo* tiene muchos *OID* asignados, obviamente. (*OID* hacen referencia a los OID+variable de cada tabla MIB, pero cada uno es un objeto individual).

- **imagen = fields.Binary("Imagen", help="Seleccionar imagen")**: Campo tipo Binary que permite al usuario subir al servidor un archivo de icono de imagen personalizado. La imagen servirá para distinguir los dispositivos en la vista kanban y en la vista de formulario.

Dentro de la clase *lista_dispo* se declarn las funciones que se van a ejecutar en el formulario de éste. La función que contiene la clase *lista_dispo* se encargará de rastrear el MIB de un dispositivo almacenando los resultados en otra clase que se explicará más adelante.

A continuación veremos la función y se explicará su funcionamiento:

4.2.1. Función Walk

```
def walk_mib(self, cr, uid, ids, context=None):
#Funcion GETNEXT que se ejecuta al pulsar el botón Buscar, en el Form
    ipversion = self.read(cr, uid, ids, ['version'], context=context)
    ipversion = ipversion[0]['version']

    #Para SNMPv2
    if(str(ipversion) == 'v2'):

        cmdGen = cmdgen.CommandGenerator()

        ipdis = self.read(cr, uid, ids, ['name'], context=context)
        ips = ipdis[0]['name']

        oidis = self.read(cr, uid, ids, ['rangomib'], context=context)
        dispoids = oidis[0]['rangomib']

        comundis = self.read(cr, uid, ids, ['community'], context=context)
        comun = comundis[0]['community']
```

Imagen 5: Función Walk v2 (1/2)

```
        cmdgen.CommunityData(comun),
        cmdgen.UdpTransportTarget((ips, 161)),
        str(dispoids),
        lookupNames=False, lookupValues=False
    )

    if errorIndication:
        print(errorIndication)
    else:
        if errorStatus:
            print('%s at %s' % (
                errorStatus.prettyPrint(),
                errorIndex and varBindTable[-1][int(errorIndex) - 1] or '?'
                )
            )
        else:
            tabla = varBindTable
            obj = self.pool.get('lista_mib.listamib')
            for record in self.browse(cr, uid, ids, context=context):
                for fila in tabla:
                    for oids, valors in fila:

                        oid = oids.prettyPrint()
                        val = valors.prettyPrint()
                        obj.create(cr, uid, {'mib_oid': oid, 'value': val,
                        'dispo_id': record.id, 'community': comun},
                        context=None)
```

Imagen 6: Función Walk v2 (2/2)

Para declarar la función se ha declarado como tal de la siguiente forma:

```
def walk_mib(self, cr, uid, ids, context=None):
```

Simplemente declaramos que es una función mediante **def**, a continuación el nombre de la función (*walk_mib* en este caso) y los parámetros que contendrá la función. Este procedimiento lo aplicaremos a la mayor parte de las funciones que programemos en Odoo.

A continuación tenemos:

```
ipversion = self.read(cr, uid, ids, ['version'], context=context)
ipversion = ipversion[0]['version']
```

La variable ipversion se encarga de leer la versión SNMP que tenga asignada el objeto actual. Lo hace en 2 instancias, la primera leerá directamente del objeto *Dispositivo* y almacenará el resultado en forma de lista/array. La segunda convertirá la variable de tipo lista a tipo Char.

```
else:
        tabla = varBindTable
        obj = self.pool.get('lista_mib.listamib')
        for record in self.browse(cr, uid, ids, context=context):
           for fila in tabla:
              for oids, valors in fila:

                 oid = oids.prettyPrint()
                 val = valors.prettyPrint()
                 obj.create(cr, uid, {'mib_oid': oid, 'value': val,
                 'dispo_id': record.id, 'community': comun},
                 context=None)
```

Esta parte del código se encarga de crear un nuevo objeto de tipo *OID* que explicaremos más adelante. Lo importante es observar los pasos que siguen en la función: podemos ver como creamos una variable **obj** que indica que estamos apuntando a un objeto tipo *OID* (lista_mib), recorremos cada fila de OIDs y valores que hemos rastreado y se los asignamos cada pareja de ellos a un nuevo objeto *OID*. Importante también la parte **'dispo_id': record.id** puesto que es el código que nos va a permitir crear la relación del actual *Dispositivo* con cada uno de los *OID* que se van creando y almacenando en la base de datos.

Seguidamente se han programado 3 condiciones, una para cada versión, ya que cada versión tiene una forma diferente de realizar el escaneo de MIBs. En las imágenes anteriores podemos observar la versión 2 SNMP.

En este caso se puede observar cómo se asignan los campos del formulario necesarios en variables de manera muy similar a la anterior variable *ipversion*.

De aquí debemos destacar las siguientes partes de la función:

```
errorIndication, errorStatus, errorIndex, varBindTable = cmdGen.nextCmd(
cmdgen.CommunityData(comun),
 cmdgen.UdpTransportTarget((ips, 161)),
 str(dispoids),
 lookupNames=False, lookupValues=False
 )
```

Este comando usará funciones del paquete cmdGen que hemos importado previamente. Lo importante de esta parte del código es indicar las variables que se han asignado. **común** en cuanto al community, **ips** corresponde a la ip del dispositivo, **161** es el puerto UDP que utiliza SNMP, **dispoids** corresponde al rango de OID que el usuario ha indicado en el formulario que se debe rastrear.

A continuación se observarán las condiciones de la función para la versión 1 y 3 de SNMP. La versión 1 es muy similar a la 2 porque el paquete de funciones *cmdGen* adapta la versión 1 a la 2, ya que necesitan el mismo tipo de información y realmente los cambios entre ambas no son perceptibles para el usuario. En cuanto a la versión SNMP v3 sí que tenemos dos campos nuevos a introducir: usuario y contraseña, que explicaremos a continuación de las imágenes.

```
elif (str(ipversion) == 'v1'):

    cmdGen = cmdgen.CommandGenerator()

    ipdis = self.read(cr, uid, ids, ['name'], context=context)
    ips = ipdis[0]['name']

    oidis = self.read(cr, uid, ids, ['rangomib'], context=context)
    dispoids = oidis[0]['rangomib']

    comundis = self.read(cr, uid, ids, ['community'], context=context)
    comun = comundis[0]['community']

    errorIndication, errorStatus, errorIndex, varBindTable = cmdGen.nextCmd(
        cmdgen.CommunityData(comun),
        cmdgen.UdpTransportTarget((ips, 161)),
        str(dispoids),
        lookupNames=False, lookupValues=False
    )

    if errorIndication:
        print(errorIndication)
    else:
        if errorStatus:
            print('%s at %s' % (
                errorStatus.prettyPrint(),
                errorIndex and varBindTable[-1][int(errorIndex) - 1] or '?'
                )
            )
        else:
            tabla = varBindTable
            obj = self.pool.get('lista_mib.listamib')
            for record in self.browse(cr, uid, ids, context=context):
                for fila in tabla:
                    for oids, valors in fila:

                        oid = oids.prettyPrint()
                        val = valors.prettyPrint()
                        obj.create(cr, uid, {'mib_oid': oid, 'value': val,
                        'dispo_id': record.id, 'community': comun},
                        context=None)
```

Imagen 7: Función Walk v1

```
elif (str(ipversion) == 'v3'):

    ipdis = self.read(cr, uid, ids, ['name'], context=context)
    ips = ipdis[0]['name']

    comundis = self.read(cr, uid, ids, ['community'], context=context)
    comun = comundis[0]['community']

    oidis = self.read(cr, uid, ids, ['rangomib'], context=context)
    dispoids = oidis[0]['rangomib']

    userdis = self.read(cr, uid, ids, ['userv3'], context=context)
    user = userdis[0]['userv3']

    passwdis = self.read(cr, uid, ids, ['passv3'], context=context)
    passw = passwdis[0]['passv3']

    cmdGen = cmdgen.CommandGenerator()

    errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
        cmdgen.CommunityData(comun),
        cmdgen.UsmUserData(str(user), str(passw)),
        cmdgen.UdpTransportTarget((ips, 161)),
        str(dispoids)
        #cmdgen.MibVariable('IP-MIB', 'ipAdEntAddr', ips)
    )

    if errorIndication:
        print(errorIndication)
    else:
        if errorStatus:
            print('%s at %s' % (
                errorStatus.prettyPrint(),
                errorIndex and varBinds[int(errorIndex)-1] or '?'
                )
            )
        else:
```

Imagen 8: Función Walk v3 (1/2)

```
            tabla = varBinds
            obj = self.pool.get('lista_mib.listamib')
            for record in self.browse(cr, uid, ids, context=context):
                for oids, vals in tabla:

                    oid = oids.prettyPrint()
                    val = vals.prettyPrint()

                    obj.create(cr, uid, {'mib_oid': oid, 'value': val,
                    'dispo_id': record.id, 'community': comun},
                    context=None)
lista_dispo()
```

Imagen 9: Función Walk v3 (2/2)

```
        cmdGen = cmdgen.CommandGenerator()

        errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
            cmdgen.CommunityData(comun),
            cmdgen.UsmUserData(str(user), str(passw)),
            cmdgen.UdpTransportTarget((ips, 161)),
            str(dispoids)
            #cmdgen.MibVariable('IP-MIB', 'ipAdEntAddr', ips)
        )
```

En esta parte del código se introducen las variables que hemos introducido en ambas versiones anteriores, pero además se deben añadir en **UsmUserData** el usuario (**user**) y contraseña(**passw**), que es la principal característica que añade esta versión para robustecer la seguridad del protocolo.

4.2.2. Vistas en XML

Para representar todos los campos e introducir los botones que ejecutarán las funciones del formulario se editará el archivo XML que se ha mencionado anteriormente (walk.xml).

La vista en forma de lista (tree view) quedará de la siguiente forma:

```
<!--TREE LISTA DISPOSITIVO-->
<record model="ir.ui.view" id="lista_dispo_tree">
  <field name="name">listaip.tree</field>
  <field name="model">lista_dispo.listaip</field>
  <field name="type">tree</field>
  <field name="arch" type="xml">
  <tree string="Resultado de la búsqueda">
    <field name="name"/>
    <field name="nombre"/>
  </tree>
  </field>
</record>
```

Imagen 10: Vista de lista de Dispositivos

Se puede observar cómo se llama al modelo (clase) **lista_dispo.listaip.** Cabe destacar que en la opción **id** se debe introducir un identificador de texto exclusivo para cada *record* de cada vista. En la vista de lista se nos mostrará por tanto los campos "name" que corresponde con la dirección IP de los objetos y "nombre" que corresponde al nombre personalizado que ha introducido el usuario en cada *Dispositivo*.

La vista de formulario es la más completa y por tanto más compleja ya que hay que distribuir todos los espacios para que cada elemento esté correctamente ubicado. Cabe destacar que se han empleado muchos campos de separación de espacios de columnas <**field colspan**> para este propósito.

Veamos como ha quedado la vista de formulario:

```
<record model="ir.ui.view" id="lista_dispo_form_view2">
  <field name="name">listaip.form</field>
  <field name="model">lista_dispo.listaip</field>
  <field name="arch" type="xml">
    <form string="Dispositivo">
      <sheet string="Dispositivos">
      <group col="4" string="Datos del dispositivo: ">
        <field colspan="2" name = "name"/><separator colspan="2"/>
        <field colspan="2" name = "nombre"/><separator colspan="2"/>
        <field colspan="2" name = "version"/><separator colspan="2"/>
        <field colspan="2" name = "community"/><separator colspan="2"/>
        <field colspan="2" name="imagen" widget="image"/>
        <separator colspan="4"/>
        <field colspan="2" name = "userv3" attrs="{'invisible': [('version', '!=', 'v3')]}"/><separator colspan="2"/>
        <field colspan="2" name = "passv3" attrs="{'invisible': [('version', '!=', 'v3')]}"/><separator colspan="2"/>

      </group>
```

Imagen 11: Vista de Formulario de Dispositivos (1/2)

En esta primera parte de la vista se han configurado los parámetros que asocian el <record> con el modelo **lista_dispo.listaip**. Se han distribuido los campos que

permiten introducir y visualizar la información del Dispositivo: IP, nombre, versión SNMP, Community, imagen personalizada, usuario y contraseña SNMPv3.

El usuario y contraseña v3 como se puede apreciar permanecen con el atributo 'invisible' con la excepción de que si se selecciona la versión de SNMP v3 automáticamente cambian su estado a 'visible' y aparecen en el formulario. De esta forma se evita que ocupen sitio innecesariamente cuando no se van a utilizar por haber seleccionado cualquiera de las otras 2 versiones.

Veamos la otra mitad de la vista:

```
<group string="Rastrear MIBS: " col="4">
  <separator colspan="4"/>
  <button colspan="1" name="walk_mib" string="Buscar" icon="fa-search" type="object" class="oe_inline oe_stat_button"/>
  <field colspan="2" name = "rangomib" string="Rango OID: "/><separator colspan="1"/>
</group>

    <field name="dispomib_id" widget="one2many_list">
      <tree editable="top">
        <field name="mib_oid"/>
        <field name="value"/>
      </tree>
    </field>
  </sheet>
</form>
</field>
</record>
```

Imagen 12: Vista de formulario de Dispositivos (2/2)

En la segunda parte de la vista se encuentra la parte que encontrará los OID dependiendo del valor introducido en el campo "rangomib". La función corresponde a lo explicado anteriormente en el apartado del archivo python. Se ejecutará al presionar el botón <**button**> que lleva el mismo nombre (walk_mib). Al terminar la operación los resultados se mostrarán en una cuadricula, en formato de lista. En dicha cuadricula el usuario podrá eliminar aquellos objetos *OID* que no quiera que sean guardados en la base de datos. Para mayor comodidad en la vista de lista de dichos objetos se podrán seleccionar varios elementos a la vez para no tener que eliminarlos uno a uno como sí ocurre en esta cuadrícula, que es más limitada en cuanto a este tipo de operaciones.

La vista kanban, como podremos apreciar en el ejemplo de demostración, simplemente nos facilita el acceso a la información de cada dispositivo. Se ha programado para que se muestre la imagen personalizada de cada *Dispositivo*, el nombre personalizado introducido por el usuario y la IP asignada.

Código perteneciente a la vista kanban:

```
<!-- KANBAN VIEW DISPOSITIVOS -->
<record model="ir.ui.view" id="dispositivos_kanban_view">
    <field name="name">listaip.kanban</field>
    <field name="model">lista_dispo.listaip</field>
    <field name="type">kanban</field>
    <field name="arch" type="xml">
        <kanban>
            <!--list of field to be loaded -->
            <field name="name"/>
            <field name="nombre"/>
            <field name="imagen"/>
            <templates>
            <t t-name="kanban-box">
                    <div class="oe_module_vignette">
                        <a type="open">
                            <img class="oe_kanban_image"
                                t-att-src="kanban_image('lista_dispo.listaip', 'imagen', record.id.value)" />
                        </a>
                        <div class="oe_module_desc">
                            <h4>

                            <field name="nombre"></field>

                            </h4>
                            <ul>
                                <li><field name="name"/></li>
                            </ul>
                        </div>
                    </div>
                </t>
            </templates>
        </kanban>
    </field>
</record>
```

Imagen 13: Vista Kanban de Dispositivos

En la vista kanban hay que destacar, además de su particular forma de distribución de los parámetros, que los campos que se vayan a utilizar para su representación deben cargarse al inicio del *<record>* o en caso contrario no se encontrarán.

Una vez se han programado todas las vistas, hay que indicar que éstas se deben cargar al presionar el menú *Dispositivos* que se encuentra en la parte izquierda del módulo en todo momento. La configuración de este menú es muy sencilla, la veremos posteriormente, lo único que se ha de indicar es un nombre para el menú y su cadena de texto que se visualizará. Por tanto para que el menú cargue las vistas habremos de configurar un **action** que indique el nombre del modelo y las vistas que se van a mostrar al usuario.

Código del **action** que hemos programado para objetos *Dispositivo*:

```
<!-- ACTION LISTA DISPOSITIVOS -->
<record model="ir.actions.act_window" id="dispo_list_action2">
    <field name="name">Dispositivos</field>
    <field name="res_model">lista_dispo.listaip</field>
    <field name="view_type">form</field>
    <field name="view_mode">kanban,tree,form</field>
    <field name="help" type="html">
      <p class="oe_view_nocontent_create">Crear dispositivo manualmente. Para buscar dispositivos disponibles, selecciona la opción Descubrir del menú.
      </p>
    </field>

</record>
```

Imagen 14: Action de Dispositivos

Visualización final del objeto **Dispositivo** por el usuario:

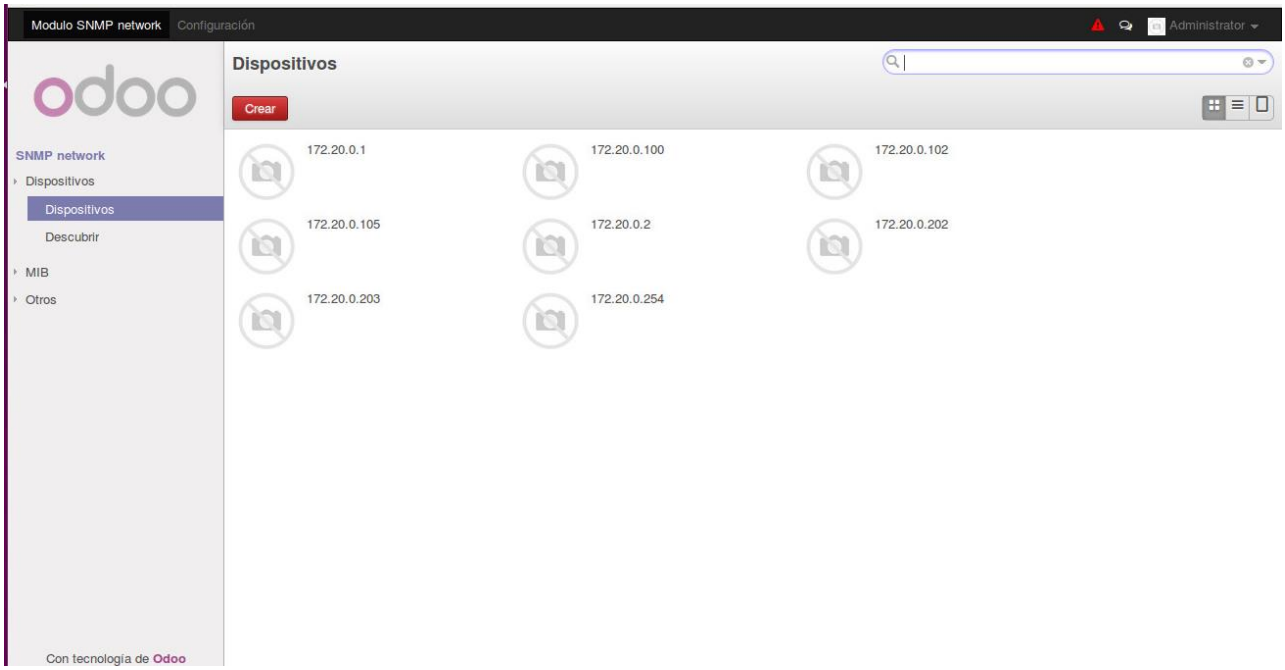


Imagen 15: Visualización Dispositivos (registros de ejemplo)

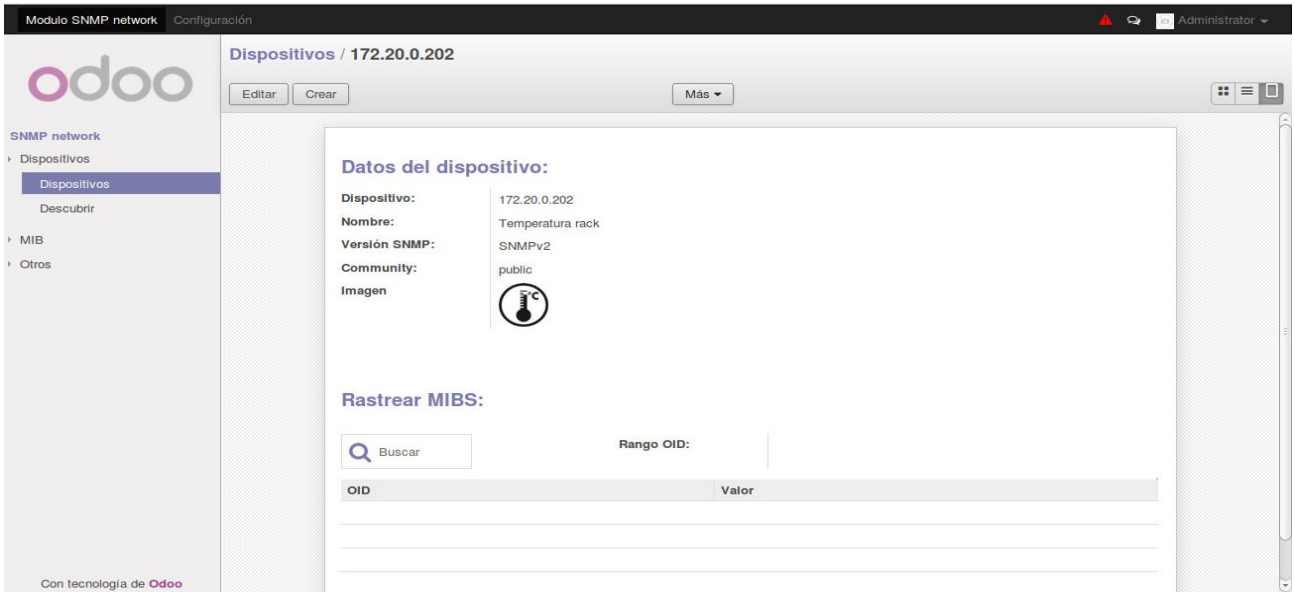


Imagen 16: Formulario de Dispositivos

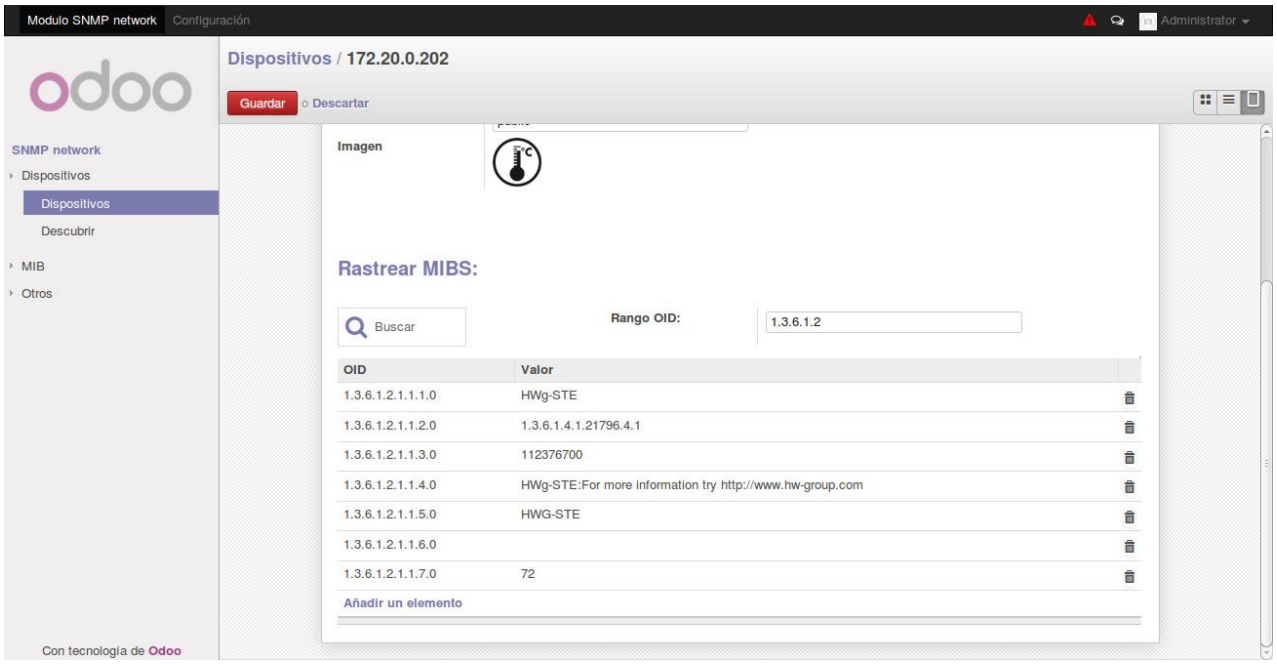


Imagen 17: Formulario Dispositivos después de ejecutar función de rastrear

4.3. Función de rastreo de direcciones IP

Una vez ya se ha programado la creación de objetos de tipo *Dispositivo* se puede proceder al diseño y programación de la función que permite rastrear las direcciones IP de los dispositivos conectados en la red del servidor. Esta función se va a ejecutar en una ventana de menú diferente llamado **Descubrir**, dicha ventana se visualizará como un formulario pop-up, no tendrá otro tipo de vistas.

La función que rastreará las direcciones IP de la red, está basada en la función para terminales **nmap** (Linux, MAC OS, Windows) que es de código abierto. Por tanto es imprescindible que el servidor tenga instalada la última versión para Linux, que se puede encontrar instalando directamente desde la terminal o descargando los paquetes correspondientes desde su web oficial https://nmap.org. Al terminar de ejecutarse creará tantos objetos *Dispositivo* como direcciones IP haya encontrado en el rango introducido por el usuario. Si no ha encontrado ninguna dirección en el rango introducido, no creará ningún objeto.

El código de la función se muestra a continuación, lo dividiremos en dos partes para analizarlo con más detalle:

```
# -*- coding: utf-8 -*-
from openerp.osv import osv, fields
import nmap
import warnings


class busca_dispo_wizard(osv.osv_memory):

    _name = 'busca_dispo_wizard.dispo'

    _columns = {

    'direccionip': fields.char('Rango de IPs:'),

    }

    def buscaips(self, cr, uid, ids, context=None):
```

Imagen 18: Función descubrir IP (1/2)

La función se ha creado en otro archivo diferente al *models.py* llamado **wizards.py**, en la misma ubicación que el primero. Un wizard es el nombre que reciben las ventanas externas al archivo donde se ubican los modelos y se muestran en ventanas pop-up. En este caso hemos importado algunos paquetes necesarios para su ejecución, como el mencionado nmap.

El nombre que recibe la clase correspondiente a la función es **busca_dispo_wizard** y solo contiene un campo: '*direccionip*'. Este campo es de tipo Char y contendrá el rango que el usuario introduzca para el rastreo de direcciones. A continuación ya se define la función en sí, que recibe el nombre de **buscaips**:

```
    def buscaips(self, cr, uid, ids, context=None):

        nm = nmap.PortScanner()

        rango = self.read(cr, uid, ids, ['direccionip'], context=context)
        rango = rango[0]['direccionip']

        nm.scan(hosts=rango, arguments='-n -sP -PE -PA161')
        hosts_list = [(x, nm[x]['status']['state']) for x in nm.all_hosts()]

        cont = 0
        host_list = []
        status_list = []
        obj = self.pool.get('lista_dispo.listaip')

        for host, status in hosts_list:
            host_list.append(host)
            status_list.append(status)
            obj.create(cr, uid, {'name': host_list[cont]}, context=None)
            cont = cont + 1

busca_dispo_wizard()
```

Imagen 19: Función descubrir IP (2/2)

Se puede observar como creamos un objeto *nm* que iniciará la función de rastreo *PortScanner()*.

```
nm = nmap.PortScanner()
```

Debemos crear una variable que contenga el contenido del campo del rango de direcciones que se pide al usuario, mediante los comandos:

```
rango = self.read(cr, uid, ids, ['direccionip'], context=context)

rango = rango[0]['direccionip']
```

Ahora ya se definen los argumentos como los introduciríamos normalmente con la función nmap en la terminal. Se creará una lista de hosts **hosts_list**, que contendrá tanto el host (IP) como el estado (ON/OFF), a nosotros solo nos interesan los hosts por tanto vamos a obviar el estado de los dispositivos.

```
for host, status in hosts_list:
        host_list.append(host)
        status_list.append(status)
        obj.create(cr, uid, {'name': host_list[cont]}, context=None)
        cont = cont + 1
```

El bucle *for* se encargará de recorrer todos los hosts encontrados (y su estado) y los iremos almacenando en otra lista por separado. Creamos un objeto *Dispositivo* por cada host+estado que se encuentre, que incluirán únicamente la dirección IP en el campo que le corresponde.

De esta manera al finalizar el proceso nos van a quedar una serie de objetos *Dispositivo* almacenados en la base de datos con su IP listos para su personalización y para la búsqueda de MIBs. Con esta función el usuario se asegura de que el dispositivo está bien configurado en la red, es rastreable y ya tiene el formulario preparado para rellenarlo con todos los datos.

Visualización del pop-up de la función:

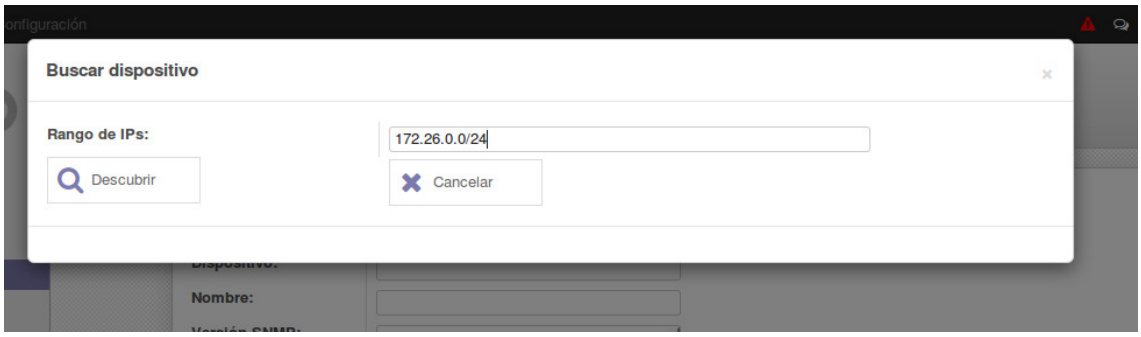


Imagen 20: Función Buscar dispositivos

4.4. Objetos OID y complementación con Dispositivos

Como hemos visto anteriormente, los objetos OID se crean en la función de rastrear MIB del formulario de *Dispositivos*. Estos objetos también dispondrán de las tres vistas (formulario, lista, kanban) para su representación, aunque será en la parte de formulario donde el usuario podrá configurar diferentes funciones de seguimiento del OID y su valor, además de visualizar gráficas y todas las funciones mencionadas al inicio de la memoria.

Cada objeto *OID* tiene ya asociados por defecto su dirección OID y el valor que se encontró en la primera búsqueda. Se implementará además en el formulario un botón que permitirá al usuario consultar el valor, para una consulta fácil de la información.

Se implementarán a su vez en dicho formulario las funciones objetivo del proyecto, que como hemos mencionado antes serán:

- Temporizador de seguimiento del valor del OID
- Tabla de condiciones para avisos
- Avisos vía e-mail, cambio de valor de OID, comando de terminal

A continuación se muestran los campos del objeto *OID*. Hay que tener en cuenta que se han declarado todos los campos, incluidos los que se utilizan en las funciones atrás mencionadas.

```
class lista_mib(models.Model):

    _name = 'lista_mib.listamib'

    mib_oid = fields.Char(string="OID")
    name = fields.Char(string="Nombre MIB: ")
    value = fields.Char(string="Valor")
    refresh_time = fields.Float(string="Tiempo comprobacion")
    community = fields.Char(string="commu")

    alerta_mail = fields.Boolean()
    alerta_consola = fields.Boolean()
    alerta_set = fields.Boolean()
    user = fields.Many2one('res.users', 'Usuario', select=True)

    valor_set = fields.Char()

    temp_activo = fields.Boolean()
```

```
dispo_id = fields.Many2one('lista_dispo.listaip', ondelete='cascade',
string="Dispositivo")

host_id = fields.Many2one('hosts.host', ondelete='cascade')
mib_id = fields.Many2one('lista_mib.listamib', ondelete='cascade')

condiciones_id = fields.One2many('condiciones.condicion', 'condicion_id',
string="Condiciones")

body = fields.Text()

imagen = fields.Binary("Imagen", help="Seleccionar imagen")

email = fields.Boolean()
consola = fields.Boolean()
set_val = fields.Boolean()
```

Imagen 21: Variables de la clase *OID*

Hemos creado otra clase/modelo llamada **lista_mib** que corresponde a los objetos tipo *OID.* Veamos detalladamente cada campo y su uso:

- **_name = 'lista_mib.listamib'**: Comando que indica el nombre por el que el sistema va a reconocer al objeto.
- **mib_oid = fields.Char(string="OID")**: Campo Char donde por defecto tendremos la dirección OID ya introducida.
- **name = fields.Char(string="Nombre MIB: ")**: Campo Char en el que el usuario elegirá un nombre personalizado para este objeto *OID*.
- **value = fields.Char(string="Valor")**: Valor del OID. Por defecto se mostrará el valor del primer rastreo pero podremos consultar el valor actual mediante un botón.
- **refresh_time = fields.Float(string="Tiempo comprobacion")**: Variable que introduce el usuario para establecer el número de minutos que se debe esperar entre comprobaciones automáticas del valor del OID. Campo utilizado en la función de comprobación.
- **community = fields.Char(string="commu")**: Variable Char que almacenará el community del OID, explicado anteriormente en *Dispositivo*.
- **alerta_mail = fields.Boolean()**: Variable Boolean que servirá para mostrar en el formulario los campos de configuración de la función de alerta por e-mail.
- **alerta_consola = fields.Boolean()**: Variable Boolean que servirá para mostrar en el formulario los campos de configuración de la función de aviso mediante comando en terminal.
- **alerta_set = fields.Boolean()**: Variable Boolean que servirá para mostrar en el formulario los campos de configuración de la función de cambio de valor de un OID.
- **user = fields.Many2one('res.users', 'Usuario', select=True)**: Esta variable contendrá la identificación de un usuario interno de Odoo que

hayamos creado (también enlaza al formulario de creación), se abre un menú con varias opciones a seleccionar por el usuario. Este usuario se empleará en la función de aviso por e-mail, ya que se enviarán los correos a la dirección asociada a ese usuario Odoo.

- **valor_set = fields.Char()**: Campo Char utilizado en la función de cambio de valor de un OID. En este campo el usuario introduce el nuevo valor que reemplazará al anterior.
- **temp_activo = fields.Boolean()**: Variable Boolean que activará el seguimiento temporizado del valor del OID.
- **dispo_id = fields.Many2one('lista_dispo.listaip', ondelete='cascade', string="Dispositivo")**: Variable tipo Many2one que relaciona el objeto *OID* actual con el objeto *Dispositivo* del que se ha obtenido. Esta variable, como se ha mencionado, se asignará automáticamente al rastrear las OID a cada uno de los objetos resultantes. El término Many2one viene de "varios para uno" lo cual indica que muchos objetos *OID* podrán relacionarse con el mismo objeto *Dispositivo*.
- **host_id = fields.Many2one('hosts.host', ondelete='cascade')**: Variable tipo Many2one como la anterior. Se utilizará en la función de aviso por e-mail, ya que requerirá seleccionar un host que el usuario habrá introducido previamente en otro formulario que veremos más adelante. También se da la posibilidad de rellenar el antes citado formulario en el momento mediante la opción de la lista "Crear Host".
- **mib_id = fields.Many2one('lista_mib.listamib', ondelete='cascade')**: Variable que sólo tendrá un uso auxiliar para identificar al presente objeto en caso de que el comando *self* no pueda ser utilizado para leer la información de algún campo.
- **condiciones_id = fields.One2many('condiciones.condicion', 'condicion_id', string="Condiciones")**: Campo que relacionará los objetos tipo *condiciones* con el actual *OID*. Los objetos tipo condiciones se crean en un formulario "invisible", explicado más adelante.
- **body = fields.Text()**: Campo de texto simple que contendrá el texto personalizado que el usuario quiera enviar en el e-mail de la función de aviso por e-mail. Aparte, como veremos, se enviará de forma predeterminada en el cuerpo del mensaje el valor actual del OID.
- **imagen = fields.Binary("Imagen", help="Seleccionar imagen")**: Campo tipo Binary que permite al usuario subir al servidor un archivo de icono de imagen personalizado. La imagen servirá para distinguir los valores que representan a cada OID en la vista kanban y en la vista de formulario.
- **email = fields.Boolean()**: Campo Boolean que activará la visualización de los campos y otros elementos de la función de aviso por e-mail en el formulario.
- **consola = fields.Boolean()**:Campo Boolean que activará la visualización de los campos y otros elementos de la función de comando de consola en el formulario.
- **set_val = fields.Boolean()**:Campo Boolean que activará la visualización de los campos y otros elementos de la función de cambio de valor de un OID en el formulario.

- **comando_terminal = fields.Text()**: Campo de texto simple que contendrá el comando que se quiera introducir en la terminal Linux del servidor.

Para proceder con el estudio de las funciones incluidas dentro del modelo *OID* se han separado en diferentes puntos ya que algunas de ellas, como el registro de valores que veremos inmediatamente después de este punto, necesitan de objetos y modelos adicionales que merecen una detallada explicació

.

4.5. OID: Temporizador de registros

El temporizador de registros consiste en el uso de la herramienta del software Odoo de automatizar acciones temporalmente para que se almacene el valor de un OID cada X minutos en una lista de un modelo que se creará para tal cometido. El modelo en sí será sencillo, para permitir al usuario consultar la lista de valores recogidos de todos los *OID* si se requiere. También dispone de una vista de formulario que solo contendrá los campos con la información para una visión más específica de un registro.

Por otra parte la principal encargada de que el registro de valores funcione correctamente será la función integrada en el modelo *OID*. La función está subdividida en dos funciones, debido a la complejidad de integrar todo el procedimiento en una función.

La primera función llamada **copiar_log** se encarga de crear el objeto crono (ir.cron) que viene por defecto en Odoo, que es el temporizador en sí. Una vez se ha creado correctamente el temporizador, se ejecutará automáticamente la otra función llamada **get_mibs** que será la que copiará los campos y obtendrá el valor a registrar. Pero no solo eso, las funciones de alertas mencionadas anteriormente también se ejecutarán dentro de ésta para simplificar operaciones y evitar cargas excesivas de memoria.

A continuación se muestra el código correspondiente a la función **copiar_log**:

```
def copiar_log(self, cr, uid, ids, context=None):

    reftimedis = self.read(cr, uid, ids, ['refresh_time'], context=context)
    reftime = reftimedis[0]['refresh_time']

    nombremib = self.read(cr, uid, ids, ['name'], context=context)
    nombremib = nombremib[0]['name']

    self.pool.get('ir.cron').create(cr, uid, {
        'name': str(nombremib),
        'interval_number': reftime,
        'interval_type': 'minutes',
        'numbercall': '-1',
        'doall': True,
        'user_id': uid,
        'model': 'lista_mib.listamib',
        'function': 'get_mibs',
        'args': [ids]
        }, context=context)

    cronos = self.pool.get('ir.cron')

    crono_id = cronos.search(cr, uid, [('name', '=', nombremib)],
    context=context)

    crono_activo = cronos.read(cr, uid, crono_id, ['active'],
    context=context)
    crono_activo = crono_activo[0]['active']

    self.write(cr, uid, ids, {'temp_activo': bool(crono_activo)},
    context=None)
```

Imagen 22: Función copiar_log

Se puede observar como se ha declarado la función mediante def, e inmediatamente vienen todas las variables y operaciones.

```
reftimedis = self.read(cr, uid, ids, ['refresh_time'], context=context)

reftime = reftimedis[0]['refresh_time']
```

A la variable reftime (de refresh time) se le asigna el valor del tiempo de comprobación que ha introducido el usuario en el campo del formulario. Se utilizará para crear y configurar el temporizador.

```
nombremib = self.read(cr, uid, ids, ['name'], context=context)

 nombremib = nombremib[0]['name']
```

La variable nombremib tendrá asignado el nombre personalizado que el usuario ha elegido para el OID desde el que se activará el temporizador/crono.

```
self.pool.get('ir.cron').create(cr, uid, {
```

```
'name': str(nombremib),
'interval_number': reftime,
'interval_type': 'minutes',
'numbercall': '-1',
'doall': True,
'user_id': uid,
'model': 'lista_mib.listamib',
'function': 'get_mibs',
'args': [ids]
}, context=context)
```

Se crea un objeto tipo **cron**, que viene por defecto incluido en Odoo e introducimos los datos requeridos. El nombre e intervalo corresponderán a las variables *nombremib* y *reftime* respectivamente, mencionadas anteriormente en este punto. Se especifica a su vez otros datos para la configuración del crono como 'numbercall' que lo situamos en valor -1 para indicar que no debe haber un límite de comprobaciones, el tipo de unidades de tiempo que será de minutos, el modelo *lista_mib.listamib* y la función que va a ejecutar al crearse el crono: *get_mibs*

```
cronos = self.pool.get('ir.cron')
crono_id = cronos.search(cr, uid, [('name', '=', nombremib)],
context=context)
crono_activo = cronos.read(cr, uid, crono_id, ['active'],
context=context)
crono_activo = crono_activo[0]['active']
self.write(cr, uid, ids, {'temp_activo': bool(crono_activo)},
context=None)
```

En conjunto este código nos servirá para indicar el estado del crono, para ello buscamos y seleccionamos el **cron** que acabamos de crear y leemos su campo 'active' que es un campo propio de estos objetos e indica si un crono está activado o no. Por defecto estará activado. Mediante *self.write()* introducimos en la variable 'temp_activo' el valor que hemos leído del campo 'active' del **cron**, que será True.

Una vez se activa el temporizador, se indicará en el formulario que está activo y se ejecutará la otra función cada X minutos que el usuario haya introducido.

Para analizar la segunda función (**get_mibs**) debemos tener en cuenta que se han incluido todas las funciones objetivo del módulo, por tanto el análisis se distribuirá a lo largo de varios puntos en esta memoria.

Seguidamente trataremos la primera parte de la función, que incluye la declaración de variables que tendrán asignados los valores de los campos del formulario y la creación de registros en el módulo de registros.

```
def get_mibs(self, cr, uid, ids, context=None):

    dispo = self.read(cr, uid, ids, ['dispo_id'], context=context)
    dispo = dispo[0]['dispo_id']
    dispo = dispo[1]

    old_dispo = self.read(cr, uid, ids, ['mib_old'], context=context)
    old_dispo = old_dispo[0]['mib_old']

    comu = self.read(cr, uid, ids, ['community'], context=context)
    comu = comu[0]['community']

    nombre = self.read(cr, uid, ids, ['name'], context=context)
    nombre = nombre[0]['name']

    body = self.read(cr, uid, ids, ['body'], context=context)
    body = body[0]['body']

    alerta_mail = self.read(cr, uid, ids, ['alerta_mail'], context=context)
    alerta_mail = alerta_mail[0]['alerta_mail']

    alerta_consola = self.read(cr, uid, ids, ['alerta_consola'],
    context=context)
    alerta_consola = alerta_consola[0]['alerta_consola']

    alerta_set = self.read(cr, uid, ids, ['alerta_set'], context=context)
    alerta_set = alerta_set[0]['alerta_set']

    usermail = self.read(cr, uid, ids, ['user'], context=context)
    usermail = usermail[0]['user']
    user_name = usermail[1]

    usuarios = self.pool.get('res.users')
    usuario_id = usuarios.search(cr, uid, [('name', '=', user_name)],
    context=context)
    usuario_mail = usuarios.read(cr, uid, usuario_id, ['login'],
    context=context)
    usuario_mail = usuario_mail[0]['login']

    comando = self.read(cr, uid, ids, ['comando_terminal'], context=context)
    comando = comando[0]['comando_terminal']

    obj_log = self.pool.get('log_mib.logmib')
```

Imagen 23: Función get_mibs. Variables.

Asignamos una variable a los campos que se van a utilizar, para operar con éstos internamente en la función. Cada valor se debe buscar y del array que obtenemos por defecto, seleccionar el elemento que corresponde con el valor en todos los casos.

A continuación se procede con la creación de objetos de registro, que serán guardados en el modelo correspondiente:

```
cmdGen = cmdgen.CommandGenerator()

errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
   cmdgen.CommunityData(comu),
   cmdgen.UdpTransportTarget((dispo, 161)),
   str(oid_dispo),
   lookupNames=False, lookupValues=False
   )

tabla = varBinds
for oids, valors in tabla:

      oid = oids.prettyPrint()
      val = valors.prettyPrint()

obj_log.create(cr, uid, {'oid': oid,
                  'value': val,
                  'dispoid': dispo,
                  'nombre': nombre,
                  'nombredispo': datetime.utcnow()},
                  context=None)

aviso_def = False
```

Imagen 24: Creación de registros

```
cmdGen = cmdgen.CommandGenerator()


errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
cmdgen.CommunityData(comu),
cmdgen.UdpTransportTarget((dispo, 161)),
str(oid_dispo),
lookupNames=False, lookupValues=False
)
```

Introduciremos como siempre las variables en las partes de la operación que se vayan a usar. *Comu* corresponde al community, *dispo* corresponde a la IP del dispositivo, el puerto SNMP 161 y por último el OID. De esta forma obtenemos el valor en el momento en el que se ejecuta esta parte del código de la función.

```
tabla = varBinds
```

```
for oids, valors in tabla:


oid = oids.prettyPrint()

val = valors.prettyPrint()


obj_log.create(cr, uid, {'oid': oid,

        'value': val,

        'dispoid': dispo,

        'nombre': nombre,

        'nombredispo': datetime.utcnow()},

        context=None)
```

Este código permite obtener el valor del OID y acto seguido se procede a crear un objeto de tipo *registro* con los campos del valor, IP del dispositivo, OID, nombre del dispositivo y el nombre asignado al registro (**datetime.utcnow()**) que corresponde a la fecha y hora del PC del servidor en forma de cadena de texto.

Ejemplo visual de la lista de registros:

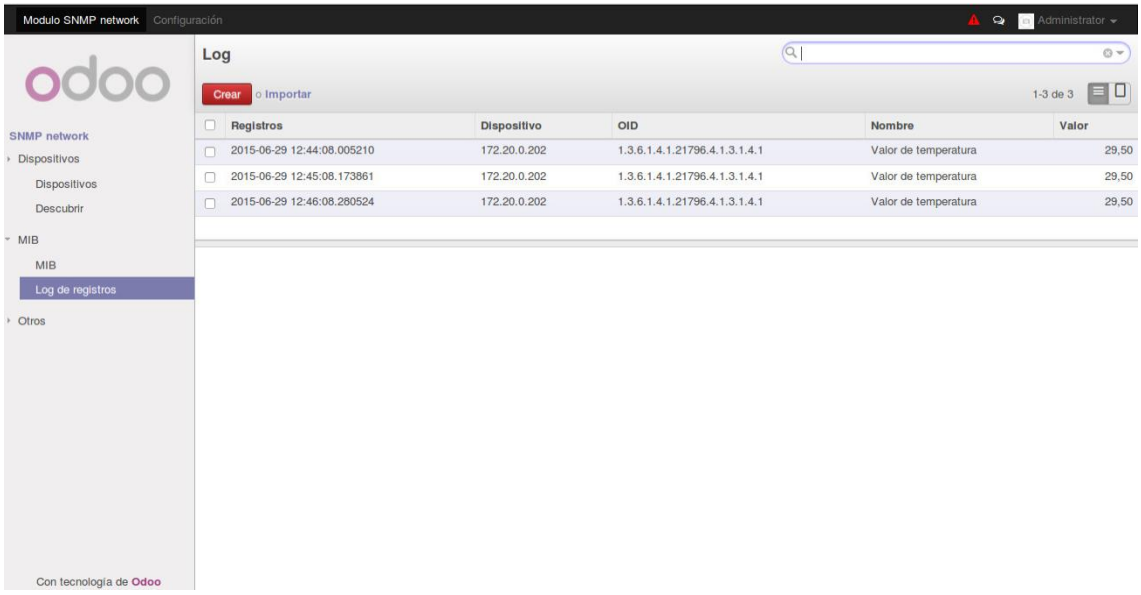


Imagen 25: Lista de registros

4.6. OID: Gráfica de valores.

Una vez se tienen varios registros de valores en su correspondiente lista del modelo de registros, se puede visualizar una gráfica que genera Odoo a partir de los campos que se le indiquen. En este caso la gráfica dibujará los valores respecto al tiempo, con un tope de 24 horas para no saturarla con excesivos valores, ya que se perdería parte de su utilidad, que es facilitar visualmente la consulta de información y la localización de valores anómalos o no esperados por el usuario.

```
def crea_grafico(self, cr, uid, ids, context=None):

    records_usados = self.pool.get('wizard_grafico.oid')
    ids_usadas = records_usados.search(cr, uid, [])

    records_usados.unlink(cr, uid, ids_usadas, context=None)

    dispo_actual = self.read(cr, uid, ids, ['dispo_id'], context=context)
    dispo_actual = dispo_actual[0]['dispo_id']

    oid_dispo = self.read(cr, uid, ids, ['mib_oid'], context=context)
    oid_dispo = oid_dispo[0]['mib_oid']
```

```
    name_dispo = self.read(cr, uid, ids, ['name'], context=context)
    name_dispo = name_dispo[0]['name']

    obj_log = self.pool.get('log_mib.logmib')

    ids = obj_log.search(cr, uid, [('oid', '=', oid_dispo),
    ('dispoid', '=', str(dispo_actual[1])), ('nombre', '=', str(name_dispo))],
    context=context)

    obj_wizard = self.pool.get('wizard_grafico.oid')
```

Imagen 26: Variables función crea_gráfico

La función de crear gráficos nos muestra una ventana externa con el gráfico, el cual recoge todos los campos en un modelo 'invisible' por el usuario. Al iniciar la función limpiaremos los registros del modelo *wizard_grafico* para que cada vez que se pulse el botón muestre una gráfica actualizada con todos los registros y valores almacenados hasta el momento.

```
records_usados = self.pool.get('wizard_grafico.oid')

ids_usadas = records_usados.search(cr, uid, [])



records_usados.unlink(cr, uid, ids_usadas, context=None)
```

Estas líneas de código se encargan de encontrar los registros almacenados anteriormente en la memoria del modelo del gráfico, seleccionarlos y eliminarlos para introducir los actuales a continuación.

El resto de variables solo tendrán asignados valores de los campos que vamos a usar a la hora de representar el gráfico, es decir el nombre del *OID*, la dirección OID y la dirección IP del dispositivo.

```
if len(ids) <= 1440:
    for record in obj_log.browse(cr, uid, ids, context=context):

        obj_wizard.create(cr, uid, {'nombredispo': record.nombredispo,
                                    'value': record.value,
                                    'dispoid': record.dispoid},
                                    context=None)
else:
    i = len(ids) - 1440
    n = 0
    for record in obj_log.browse(cr, uid, ids, context=context):
        if n < i:
            n = n + 1
```

```
        else:
            obj_wizard.create(cr, uid, {'nombredispo': record.nombredispo,
                                        'value': record.value,
                                        'dispoid': record.dispoid},
                                        context=None)

return {
 'name': 'Grafico',
 'view_type': 'graph',
 'view_mode': 'graph',
 'res_model': 'wizard_grafico.oid',
 'type': 'ir.actions.act_window',
 'target': 'new'
 }
```

Imagen 27: Creación de registros e invocación de la ventana externa

La segunda parte de la función se encarga de crear los nuevos registros en el modelo del wizard del gráfico que se ha mencionado y a continuación se indica que el gráfico se representará en forma de ventana externa/wizard.

if len(ids) <= 1440:

Esta condición indicará que sólo se crearán hasta 1440 registros, que si se selecciona la unidad mínima de tiempo de comprobación (1 minuto) corresponderá a 24 horas. De esta forma se evita la saturación de información en el gráfico.

```
if len(ids) <= 1440:
        for record in obj_log.browse(cr, uid, ids, context=context):
            obj_wizard.create(cr, uid, {'nombredispo': record.nombredispo,
                                        'value': record.value,
                                        'dispoid': record.dispoid},
                                        context=None)
```

En conjunto la primera condición creará tantos registros en el modelo de gráfico como registros de valores se hayan almacenado. Una vez se sobrepasan estos registros se avanza a la segunda condición.

```
else:
        i = len(ids) - 1440
        n = 0
        for record in obj_log.browse(cr, uid, ids, context=context):
            if n < i:
                n = n + 1
            else:
                obj_wizard.create(cr, uid, {'nombredispo': record.nombredispo,
                                            'value': record.value,
                                            'dispoid': record.dispoid},
                                            context=None)
```

En esta segunda condición se modifica el código para que se elimine el primer registro y se cree uno nuevo, que represente al valor más actual. De esta forma tenemos siempre los mismos registros pero se van sustituyendo los más antiguos por los más recientes.

```
return {
        'name': 'Grafico',
        'view_type': 'graph',
        'view_mode': 'graph',
        'res_model': 'wizard_grafico.oid',
        'type': 'ir.actions.act_window',
```

```
        'target': 'new'

        }
```

Con este código se indica que se debe abrir una nueva ventana tipo gráfico (graph), utilizando los datos del modelo *wizard_grafico* donde estarán los registros guardados.

Ejemplo de visualización de la ventana de gráficos:

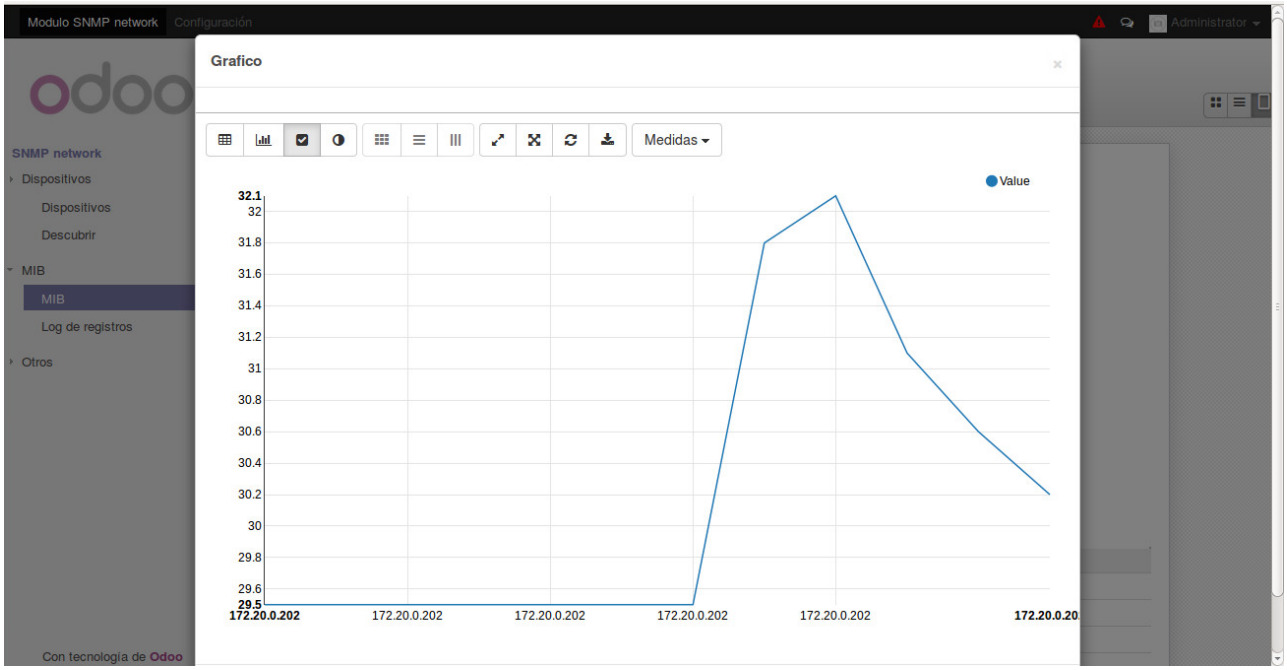


Imagen 28: Gráfico lineal

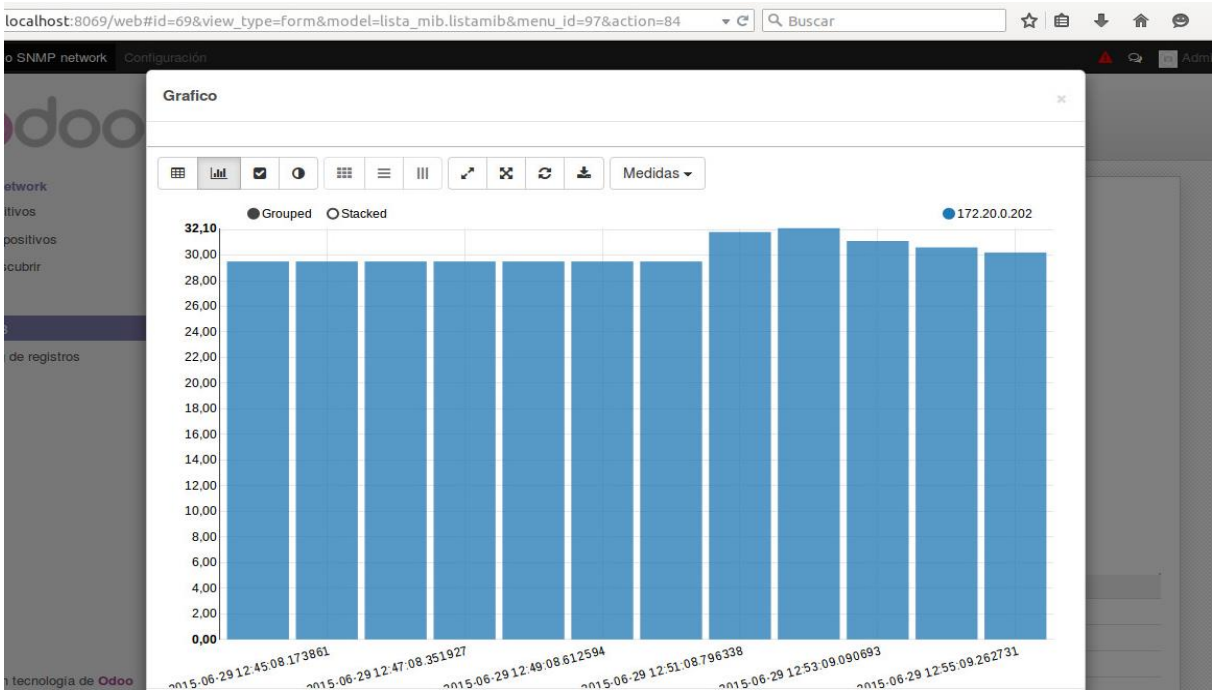


Imagen 29: Gráfico de barras

4.7. Modelo OID: Tabla de condiciones

La tabla de condiciones permite al usuario configurar las condiciones que deben de cumplir los valores para activar cualquiera de los avisos. Estas condiciones se evaluarán en la función *get_mibs* a la vez que se registra un valor nuevo, de forma que el nuevo valor se comparará con los que haya introducido el usuario y si se cumplen una o más condiciones se disparará el aviso (si se ha configurado previamente).

Las comparaciones se realizarán de forma jerárquica permitiendo que dos líneas se relacionen también mediante lógica proposicional. Al final de la línea se podrá elegir entre AND (y) o OR (o).

Los comparadores configurables en la tabla son:

- **Mayor**
- **Menor**
- **Mayor o igual**
- **Menor o igual**
- **Igual**
- **Distinto**

Veamos el código de la tabla:

```
lista_condiciones = self.read(cr, uid, ids,
['condiciones_id'], context=context)
lista_condiciones = lista_condiciones[0]['condiciones_id']

obj_condiciones = self.pool.get('condiciones.condicion')

condicio = []
marca = []
avisar = []

for cond in obj_condiciones.browse(cr, uid, lista_condiciones,
context=context):
    condicio.append(cond)
    marca.append(False)
    avisar.append(False)

n = 0
```

Imagen 30: Variables utilizadas

Declaración de las variables que se van a utilizar para las condiciones, avisos, etc.

En este punto lo más importante son los arrays **condicio, marca, avisar**. El array *condicio* mediante el for almacenará todas los AND y OR de la tabla, mientras que los arrays *marca* y *avisar* se rellenarán de False para crear tantos como registros haya en

la tabla. La variable **n** la utilizamos para separar el primer registro de los siguientes, ya que el primer registro no puede ser relacionado con otro anterior y debe ser codificado aparte.

Vamos a analizar el código de las comparaciones:

```
for condicion in obj_condiciones.browse(cr, uid, lista_condiciones,
context=context):

    if n == 0:
        if condicion.compara == 'mayor':
            if float(val) > condicion.valor:
                marca[n] = True
        elif condicion.compara == 'menor':
            if float(val) < condicion.valor:
                marca[n] = True
        elif condicion.compara == 'mayorigual':
            if float(val) >= condicion.valor:
                marca[n] = True
        elif condicion.compara == 'menorigual':
            if float(val) <= condicion.valor:
                marca[n] = True
        elif condicion.compara == 'igual':
            if float(val) == condicion.valor:
                marca[n] = True
        elif condicion.compara == 'distinto':
            if float(val) != condicion.valor:
                marca[n] = True
        if condicio[n].o_log != 'y' and condicio[n].o_log != 'o':
            avisar[n] = marca[n]
        if n == len(condicio) - 1:
            avisar[n] = marca[n]
        else:
            n = n + 1
```

Imagen 31: Comparaciones primera línea

En la primera parte de las comparaciones solo se tendrá en cuenta el primer registro (**n=0**) ya que como anteriormente es imposible que tenga ninguna otra línea no necesita “conocer” ninguno de sus AND o OR.

Se encontrará cual es la condición que el usuario ha seleccionado, y una vez localizada se iniciará la comprobación de la condición. En caso de cumplirse, se cambiará el valor del elemento actual del array de **marca** (que será el primero obviamente) a True. De esta forma al finalizar las comprobaciones si alguna marca del arrray tiene un True activado se comprobará si está relacionada la línea con un AND a la línea siguiente, y en caso de que la siguiente línea tenga una marca activada también con True y ningún AND introducido se iniciarán automáticamente las funciones de aviso seleccionadas por el usuario.

Una vez se realizan todas las comprobaciones se cambia el valor de **n** para avanzar a la siguiente línea/registro de la tabla.

```
else:
    if condicion.compara == 'mayor':
        if float(val) > condicion.valor:
            marca[n] = True
    elif condicion.compara == 'menor':
        if float(val) < condicion.valor:
            marca[n] = True
    elif condicion.compara == 'mayorigual':
        if float(val) >= condicion.valor:
            marca[n] = True
    elif condicion.compara == 'menorigual':
        if float(val) <= condicion.valor:
            marca[n] = True
    elif condicion.compara == 'igual':
        if float(val) == condicion.valor:
            marca[n] = True
    elif condicion.compara == 'distinto':
        if float(val) != condicion.valor:
            marca[n] = True
    if condicio[n - 1].o_log == 'o':
        if marca[n - 1] is True:
            avisar[n] = True
    elif condicio[n - 1].o_log == 'y':
        if marca[n - 1] is False and marca[n] is True:
            marca[n] = False
        elif marca[n - 1] is True and marca[n] is True:
            avisar[n] = True
        else:
            avisar[n] = False
    elif condicio[n - 1].o_log != 'y' and condicio[n - 1].o_log != 'o' and condicio[n].o_log != 'y' and condicio[n].o_log != 'o':
        avisar[n] = marca[n]

    n = n + 1
```

Imagen 32: Comprobaciones de la segunda línea hacia adelante

De la segunda línea hacia adelante las comprobaciones se evaluarán de la misma forma, pero se añade un código al finalizar, que comprobará si la condición de la línea anterior tenía un AND o OR para tener en cuenta esta operación lógica a la hora de activar los avisos. Si en una de las comprobaciones del 'for' encuentra la marca activada y la línea anterior tenía una marca activada con un AND o OR automáticamente se activará la variable de avisar.

```
for i in avisar:
    if i is True:
        aviso_def = True
if(alerta_mail):
```

Imagen 33: Aviso definitivo

Al finalizar, se recorre el array de *avisar* y si encuentra algún True se cambia la variable **aviso_def** a True. Esta variable se evaluará en las funciones de aviso para que cuando esté activada se ejecuten.

4.8. Funciones de alerta

4.8.1. Función de alerta por E-mail

El aviso por e-mail consiste en enviar un texto personalizado a una dirección de correo electrónico. La dirección de correo electrónico será la que se haya configurado en el usuario Odoo que se debe introducir en el formulario, en la parte superior.

Veamos el código de la función:

```
if(alerta_mail):

    id_host = self.read(cr, uid, ids, ['host_id'], context=context)
    id_host = id_host[0]['host_id']
    id_host = id_host[1]

    hosts = self.pool.get('hosts.host')
    hostid = hosts.search(cr, uid, [('name', '=', id_host)],
    context=context)

    dirhost = hosts.read(cr, uid, hostid, ['host_smtp'], context=context)
    dirhost = dirhost[0]['host_smtp']

    porthost = hosts.read(cr, uid, hostid, ['puerto_smtp'], context=context)
    porthost = porthost[0]['puerto_smtp']

    usersmtp = hosts.read(cr, uid, hostid, ['usuario_smtp'], context=context)
    usersmtp = usersmtp[0]['usuario_smtp']

    contrasmtp = hosts.read(cr, uid, hostid, ['contra_smtp'], context=context)
    contrasmtp = contrasmtp[0]['contra_smtp']
```

Imagen 34: Variables alerta e-mail

Si **alerta_mail** que es un campo boolean está en True, se ejecuta el código de la función. Declaramos las variables que contendrán los valores introducidos en los campos del host, que es un formulario simple externo al de *OID.* En caso de que no se haya introducido ningún registro de Host se permite al usuario abrir una instancia del otro formulario para crearlo de forma rápida.

```
if(aviso_def):
    smtpObj = smtplib.SMTP(host=dirhost,
    port=str(porthost))
    smtpObj.ehlo()
    smtpObj.starttls()
    smtpObj.ehlo()
    #raise osv.except_osv(('Error'), usersmtp+contrasmtp)
    smtpObj.login(user=str(usersmtp),
    password=str(contrasmtp))
    mensaje = 'Subject: %s\n\n%s' % ('ALERTA', body)
    smtpObj.sendmail(usersmtp, usuario_mail,
    mensaje + "\nValor:" + val)
```

Imagen 35: Envío de E-mail

Se comprueba que el aviso (*aviso_def*) está activado y en caso afirmativo se procede a rellenar los campos necesarios de la función con las variables creadas con este fin. Por defecto al final del menaje personalizado por el usuario que corresponde al campo **body** se introduce una cadena de texto con el valor actual del OID que se está evaluando, de manera que queda: **Valor:X** donde X corresponde al valor actual.

4.8.2. Función de alerta por cambio de valor de un OID

La función de cambio de valor de un OID consiste en cambiar el valor de un OID ya creado y configurado en la lista de registros de los objetos *OID*.

```
cmdGen = cmdgen.CommandGenerator()

errorIndication, errorStatus, errorIndex, varBinds = cmdGen.setCmd(
cmdgen.CommunityData(comundis, mpModel=0),
cmdgen.UdpTransportTarget((dispoip, 161)),
(str(oidis), valor_nuevo)
)
# Check for errors and print out results
if errorIndication:
    print(errorIndication)
else:
    if errorStatus:
        print('%s at %s' % (
        errorStatus.prettyPrint(),
        errorIndex and varBinds[int(errorIndex)-1] or '?'
    )
)
```

Imagen 36: Función cambio de valor OID

Se introducen las variables en la función que sustituye el valor de un OID, que funciona de manera similar a la función de rastrear las direcciones, pero en lugar de

mostrar valores y OID encontradas, selecciona una dirección y sustituye el valor por el que el usuario desea.

4.8.3. Función de alerta por comando de terminal

La última de las tres funciones de alerta se trata de la introducción de un comando en el terminal del servidor para ejecutar comandos del sistema Linux/Ubuntu. Esta función es muy simple ya que el servidor se estará ejecutando en un terminal ya abierto de manera que solo se debe introducir el comando mediante el código python que nos permite hacerlo.

```
if(alerta_consola):

    comando = self.read(cr, uid, ids, ['comando_terminal'], context=context)
    comando = comando[0]['comando_terminal']

    proc=Popen(comando, stdin=PIPE, shell=True)
    #proc.communicate('password')
    #proc.communicate('123')
```

Imagen 37: Función alerta de terminal

La función simplemente almacena el campo que representa el texto que se introducirá en la terminal y mediante la instrucción **Popen** para python, introducimos el comando en la consola y ejecutamos. Actualmente esta función no funciona correctamente ya que tiene un límite de introducción de comandos (solo hay una cadena de texto a introducir) y algunas instrucciones dan problemas debido a que el sistema operativo Linux pide permisos de **sudo** al usuario, que requiere de varias interacciones con la terminal (usuario, contraseña, etc.).

Representación de las vistas. Documento XML

Como en el modelo anterior, especificaremos el apartado visual de las diferentes vistas en el archivo XML. Cabe destacar que la vista de formulario en este caso será muy extensa debido a la cantidad de campos y elementos tales como botones que se deben introducir y configurar.

Vista de formulario, configuración y distribución de los elementos:

```
<!-- FORM LISTA MIBS -->
<record model="ir.ui.view" id="lista_mib_form_view">
  <field name="name">listamib.form</field>
  <field name="model">lista_mib.listamib</field>
  <field name="arch" type="xml">
    <form string="MIBS" create="false">
      <sheet>
        <group string="Datos MIB" col="8">
        <field colspan="3" name = "dispo_id" string="IP dispositivo: "/>
        <separator colspan="5"/>
        <field colspan="3" name = "name"/>
        <separator colspan="5"/>
        <field colspan="3" name = "user" string="Usuario: "/>
        <separator colspan="5"/>
        <separator colspan="8"/>
        <field colspan="3" name = "mib_oid" string="OID: "/>
        <separator colspan="5"/>
        <field colspan="3" name = "value" string="Valor: "/>
        <button colspan="2" name="valor_actual" string="Actualizar valor" style="width:55%%" type="object" icon="fa-eye" class="oe_inline oe_stat_button"/>
        <separator colspan="8"/>
        <field colspan="3" name="imagen" widget="image"/>
        <separator colspan="5"/>
        <separator colspan="8"/>
        <field colspan="3" name = "refresh_time" string="Tiempo de comprobación (minutos): "/>
        <separator colspan="5"/>
        <button colspan="3" name="copiar_log" string="Activar compr." type="object" icon="fa-toggle-on" class="oe_inline oe_stat_button" attrs="{'invisible': [('temp_activo', '=', True)]}"/>
        <field colspan ="2" name="temp_activo" string="Activo" readonly="True"/>
        <button colspan="3" name="eliminar_cron" string="Eliminar" style="width:50%%" type="object" icon="fa-toggle-off" class="oe_inline oe_stat_button" attrs="{'invisible': [('temp_activo', '!=', True)]}"/>
        <separator colspan="8"/>
        <button colspan="3" name="crea_grafico" string="Ver gráfico" style="width:50%%" type="object" icon="fa-bar-chart" class="oe_inline oe_stat_button"/>
        </group>
        <field name="condiciones_id" widget="one2many_list">
          <tree editable="top">
            <field name="compara" string="Condiciones"/>
            <field name="valor" string ="Valor a comparar"/>
            <field name="o_log" string="Y/O"/>
          </tree>
        </field>
        <group string="Seleccionar alerta para condiciones anteriores: " col="8">
        <field colspan="2" name="email" string="E-mail" eval="0"/>
        <field colspan="2" name="set_val" string="Cambio valor de MIB" eval="0"/>
        <field colspan="2" name="consola" string="Comando de consola" eval="0"/>
        </group>
        <group string="Alerta por E-mail: " col="8" attrs="{'invisible': [('email', '!=', True)]}">
        <field colspan="2" name = "host_id" string="Host SMTP: "/>
        <separator colspan="6"/>
        <separator colspan="8"/>
        <field colspan = "1" name = "alerta_mail" string="Activar alerta: "/>
        <separator colspan="8"/>
        <field colspan = "8" name = "body" string="Cuerpo del mensaje: "/>
        <separator colspan="8"/>
        </group>
        <group string="Cambiar valor de un MIB: " col="8" attrs="{'invisible': [('set_val', '!=', True)]}">
        <field colspan="2" name="valor_set" string="Nuevo valor:"/>
        <field colspan="2" name="mib_id" string="Seleccionar MIB:"/>
        <separator colspan="4"/>
        <field colspan = "1" name = "alerta_set" string="Activar alerta: "/>
        <separator colspan="8"/>
        <button colspan="2" name="valor_actual" string="Valor actual" style="width:55%%" type="object" icon="fa-eye" class="oe_inline oe_stat_button"/>
        <separator colspan="8"/>
        </group>
        <group string="Comando de consola: " col="8" attrs="{'invisible': [('consola', '!=', True)]}">
        <field colspan="3" name="comando_terminal" string="Introducir comando: "/>
        <field colspan = "2" name = "alerta_consola" string="Activar: "/>
        <separator colspan="3"/>
        </group>
      </sheet>
    </form>
  </field>
</record>
```

Imagen 38: Código XML de formulario de *OID*

Cabe destacar la configuración del botón de activar el **cron**, ya que se ha configurado para que se cree un **cron** cuando se pulsa y para detener la comprobación por tiempo

se pulsa otro botón que elimina el registro correspondiente al **cron** que se ha creado previamente. Se puede ver el estado del temporizador mediante el boolean *temp_activo*.

También mencionar los campos correspondientes a las funciones de alerta, que solo serán visibles cuando se seleccione cualquiera de ellos. Se mostrarán debajo de la tabla de condiciones, que se representa mediante la cariable *condiciones_id* y muestra los campos de *valor, comparador* y *operador lógico.*

Por último, se ha creado un botón al principio del formulario que permite actualizar el campo del valor del OID para que se muestre el valor actualizado. (*valor_actual*)

Ejemplos de visualización del modelo *OID:*

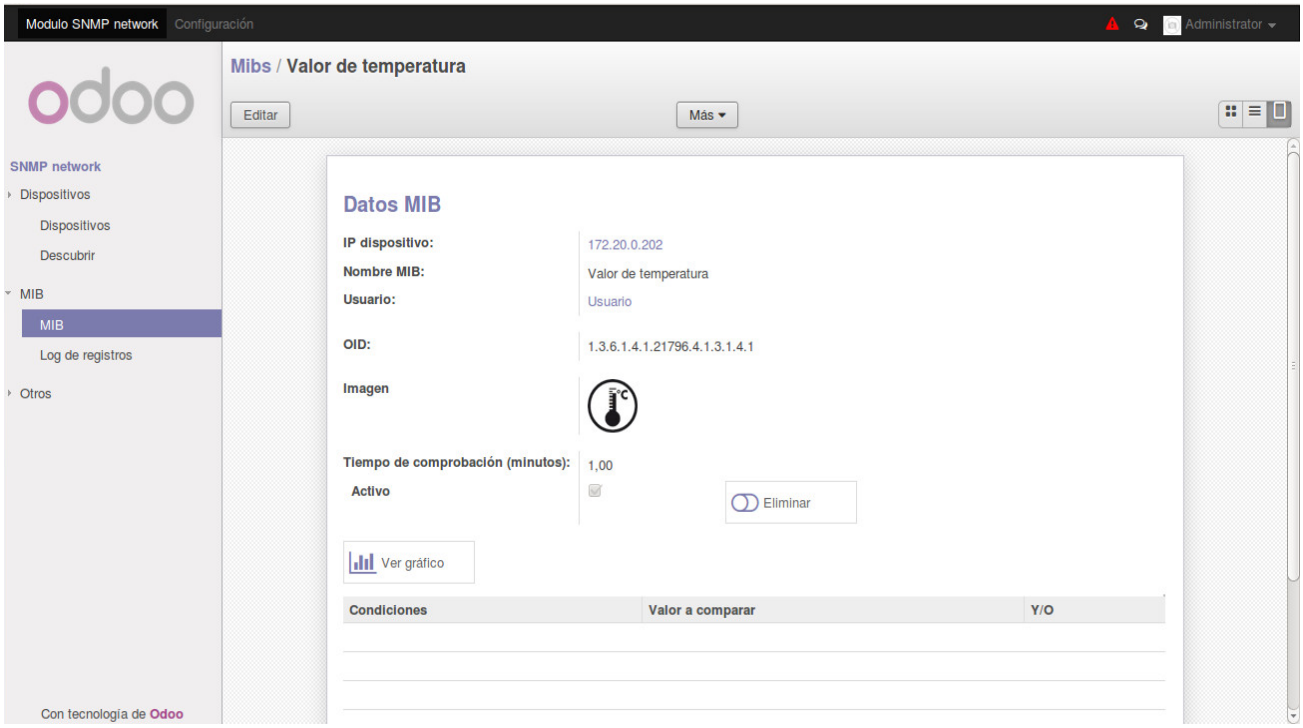


Imagen 39: Formulario *OID*

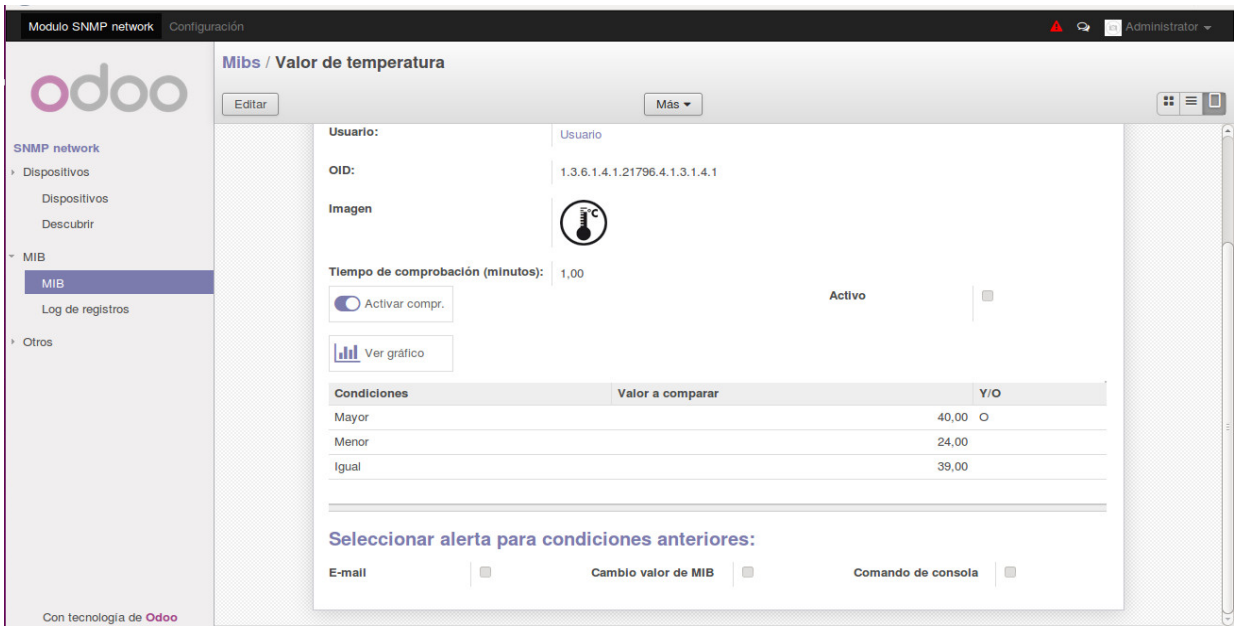


Imagen 40: Tabla de condiciones

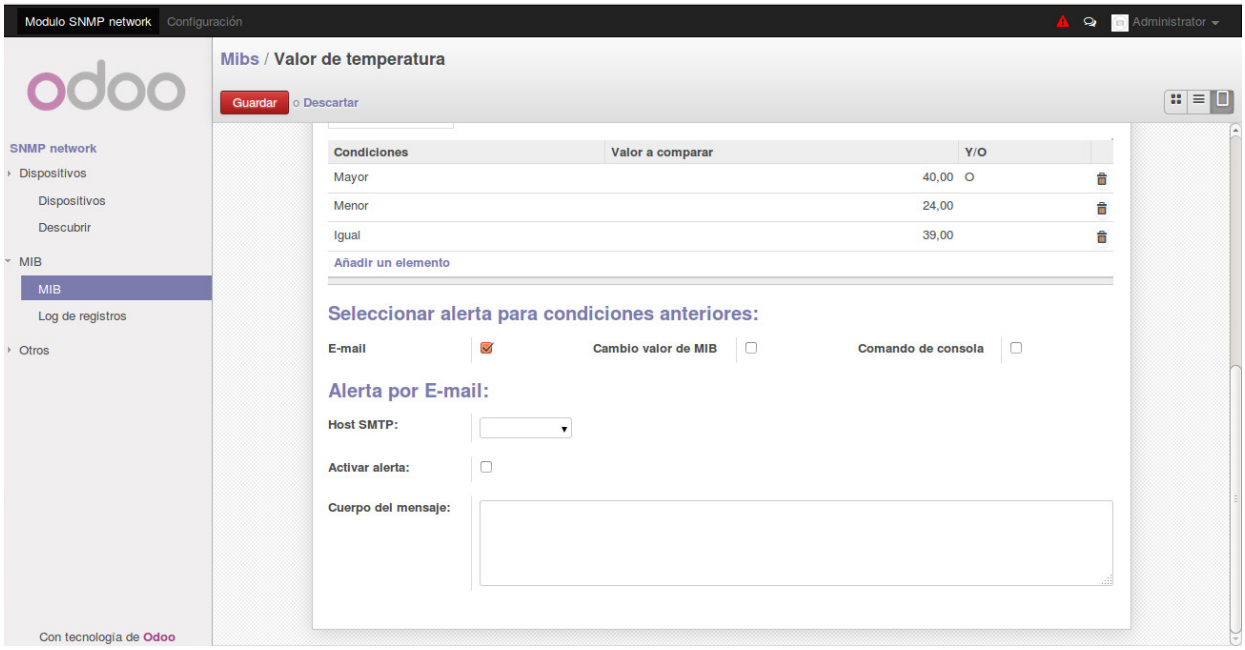


Imagen 41: Alerta E-mail activada

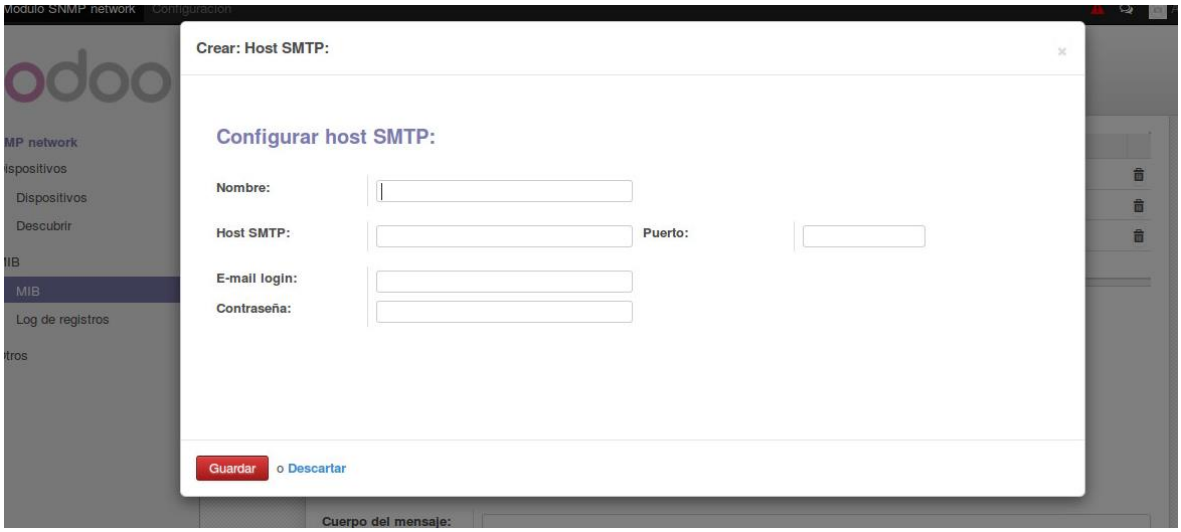


Imagen 42: Formulario Host SMTP

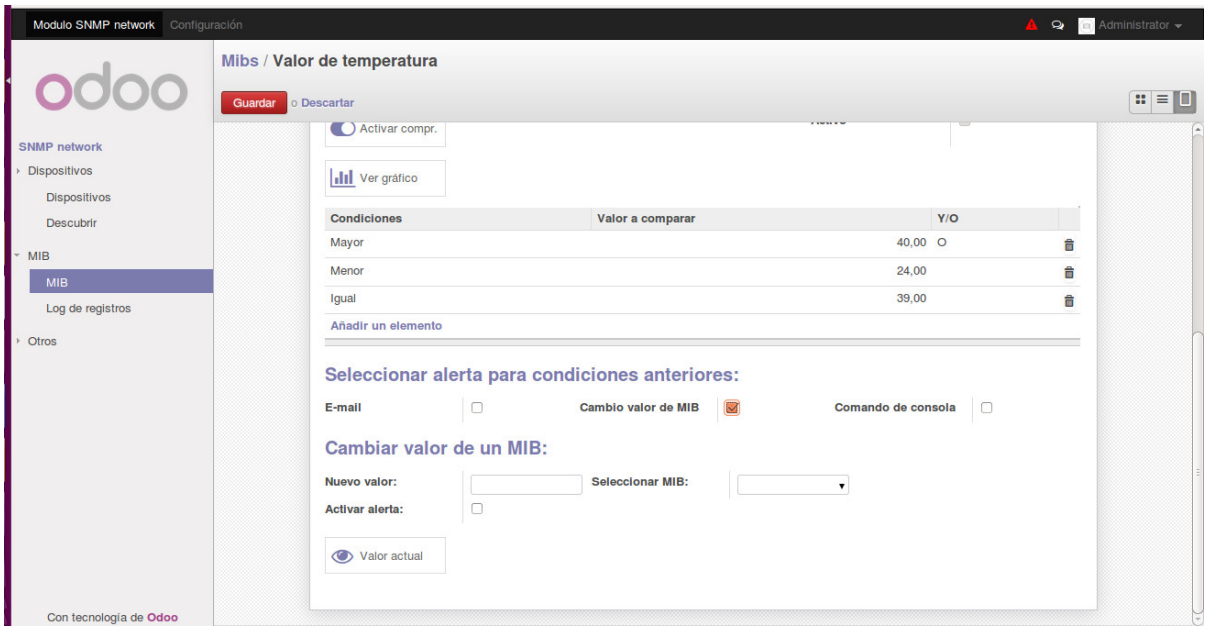


Imagen 43: Cambio de valor de un MIB. Alerta activada

4.9. Otras funciones

Para completar el módulo, se han creado otras funciones y formularios menores para complementar otros puntos.

Clase condiciones

```
class condiciones(models.Model):

    _name = 'condiciones.condicion'

    compara = fields.Selection((('mayor', 'Mayor'),
                                ('menor', 'Menor'),
                                ('mayorigual', 'Mayor o igual'),
                                ('menorigual', 'Menor o igual'),
                                ('igual', 'Igual'),
                                ('distinto', 'Distinto')))
    valor = fields.Float(string="Valor")

    o_log = fields.Selection((('y', 'Y'),
                              ('o', 'O')))

    condicion_id = fields.Many2one('lista_mib.listamib', ondelete='cascade',
    string="Condicion")

condiciones()
```

Imagen 44: Clase condiciones

Clase necesaria para poder visualizar la tabla de condiciones del formulario *OID*. Incluye un Many2one para relacionar cada registro con uno de los mencionados objetos *OID.*

Clase host_smtp

```
class host_smtp(models.Model):

    _name = 'hosts.host'

    name = fields.Char()

    host_smtp = fields.Char()
    puerto_smtp = fields.Char()

    usuario_smtp = fields.Char()
    contra_smtp = fields.Char()
```

Imagen 45: Clase host_smtp

Clase para configurar el formulario del Host SMTP de la alerta por E-mail.

Clase clear_log

```
class clear_log(osv.osv_memory):

  _name = 'clear_log.clear'

  def clearlog(self, cr, uid, ids, context=None):

    obj_log = self.pool.get('log_mib.logmib')
    ids_log = obj_log.search(cr, uid, [])

    obj_log.unlink(cr, uid, ids_log, context=None)

clear_log()
```

Imagen 46: Clase clear_log

Clase ubicada en el archivo **wizards.py** que permite al usuario eliminar todos los registros del *registro de valores*. Para ejecutar la función se abre una ventana emergente pop-up que advierte de que los datos no pueden ser recuperados.

5. Bibliografía

- R. Kenneth. *Python Guide Documentation*. Release 0.0.1.
- Andrés Marzal, Isabel García. *Introducción a la programación con Python*.
- Main Informática. *Guía de instalación de Odoo en Ubuntu.* Documento interno de la empresa
- Documentación Oficial de Odoo proporcionada por la Web Oficial.
- Aportaciones de usuarios en Webs de ayuda al programador:
    - StackOverflow
    - Foro de ayuda de Odoo [https://www.odoo.com/es_ES/forum/help-1]