Universidad Carlos III de Madrid

“DESIGN AND IMPLEMENTATION OF A PROTOTYPE FOR MODEL-DRIVEN NETWORK SERVICE MANAGEMENT”

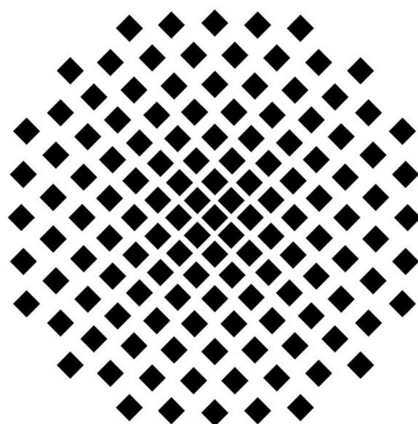
PROYECTO FIN DE CARRERA

Autor: Silvia de los Ríos Pérez

Tutor: José Ignacio Moreno Novella

INGENIERIA TÉCNICA DE TELECOMUNICACIONES: TELEMÁTICA

Proyecto fin de carrera realizado en periodo Erasmus en la Universidad de Stuttgart (Alemania)



Universität Stuttgart

Con tutores en Alemania:
Jochen Kögel
Yongzheng Liang

Según la normativa vigente en la Universidad Carlos III con respecto a este tipo de proyectos se incluye a continuación un resumen de 10 páginas en español.

Resumen

Hoy en día, la dependencia de los servicios de red en las empresas es bastante grande. El uso de redes para comunicar y distribuir información en entornos empresariales es muy común y por ello, es muy importante mantener estos servicios y redes funcionando 24 horas al día. Con este objetivo nace el concepto de administración y gestión de red.

Existen diferentes tecnologías que facilitan el desarrollo de aplicaciones empresariales, tal y como Administración dirigida por Modelos. Esta tecnología busca abstraer un sistema separando la arquitectura del sistema de la arquitectura de la plataforma, para hacer más fácil el entendimiento de un sistema ya que éste se divide en capas. Los modelos trabajan en alto nivel, lo que significa que no es necesario entender protocolos de bajo nivel.

Si estas tecnologías se mezclan con un protocolo de gestión de red, como Simple Network Management Protocol (SNMP), se puede conseguir una aplicación de gestión mucho más poderosa, fácil de desarrollar, manejar y modificar. Estas características son deseables para desarrollar aplicaciones fácilmente escalables.

Pero este proyecto pretende encontrar un nuevo enfoque para la Administración dirigida por Modelos: gestión de red a través de un modelo en tiempo de ejecución. Esto es posible gracias a model@runtime (tecnología que estudia los modelos en tiempo de ejecución). Este concepto busca comunicar un modelo con el mundo real sin desarrollo adicional de código, directamente desde el modelo.

Este proyecto está centrado en el estudio de la aplicabilidad de los conceptos de Administración dirigida por Modelos y modelos en tiempo de ejecución. Se trata de aprovechar las ventajas del uso de modelos en el desarrollo de aplicaciones de negocio, pero aplicado a la gestión y administración de redes.

Para este propósito, ha sido desarrollado un modelo UML que servirá de monitor de red de un conjunto de conmutadores de repuesto. Este modelo usará una librería SNMP existente desarrollada en C++. Esta aplicación tiene el propósito de comunicar el modelo con los conmutadores utilizando mensajes SNMP que contienen comandos SNMP.

En este caso concreto, el escenario real es la Universidad de Stuttgart, Alemania, y más concretamente un departamento específico llamado Centro de Cálculo (con siglas RUS en alemán). Este departamento opera una red con diferentes tipos de dispositivos como pueden ser servidores, ordenadores, impresoras... Éstos están interconectados por medio de conmutadores, y utilizan routers para conectarlos con Internet y otras redes, tal y como muestra la Figura 1.

Se desea para dicha red que no haya pérdida de conexión entre dispositivos, o entre otros departamentos o Internet. Pero las redes ideales no existen. A veces los conmutadores o routers se caen, y si no hay otra ruta alternativa, existe una pérdida de conexión con un dispositivo o grupo de ellos.

La motivación de esta tesis está centrada en resolver este tipo de pérdida de conexiones tan pronto como sea posible. Esto puede hacerse, por ejemplo, por medio de nuevas rutas alternativas usando routers de repuesto.

El RUS tiene 10 conmutadores de repuesto en un banco de pruebas. Un conmutador de red es un dispositivo que conecta diferentes segmentos de red. Si cualquier dispositivo se cae, puede ser usado un conmutador de red para reemplazar el

dispositivo caído, y, gracias a esto, enviar el tráfico a través del nuevo conmutador con el fin de alcanzar el mismo destino.

Ésta es la idea del RUS: utilizar los conmutadores de repuesto para resolver una pérdida de conexión. Pero no se conoce si los conmutadores están listos para usarse, es decir, si están funcionando, configurados, si todos los puertos están habilitados, etc. Es necesario monitorizar estos dispositivos y, dependiendo de la información recibida, tomar esa decisión por medio de casos de prueba y diagramas de flujo de trabajo.

En este punto, se presenta un importante enfoque del proyecto. Con el fin de especificar estos casos de prueba y flujos de trabajo, se va a utilizar Administración dirigida por Modelos, ya que ofrece una serie de ventajas en el momento del desarrollo de aplicaciones.

Es preferible desarrollar una aplicación flexible que pueda ser fácilmente modificable, esto es, con mínimo impacto en el código, portable entre diferentes plataformas, fácil de diseñar, etc. Para conseguir estos propósitos se utilizará Ingeniería dirigida por Modelos.

Este tipo de ingeniería ha sido creada como una metodología de desarrollo de software que se centra en crear modelos o abstracciones más cerca de algunos conceptos de dominio particular en lugar de conceptos de computación. El modelado introduce abstracción para reducir la complejidad. Esta idea busca potenciar la compatibilidad entre sistemas, simplificar el proceso de diseño y promover la comunicación entre particulares y equipos trabajando en el sistema. Con estas técnicas de modelado multicapa, cada capa puede proveer una vista del sistema simple y fácil de comprender.

Para obtener los datos de monitorización se ha utilizado el protocolo de gestión de red SNMP, que comunica al manager con los dispositivos monitorizados a través del intercambio de mensajes SNMP que contienen comandos SNMP. Este protocolo será explicado con más detalles en el capítulo 2.1.

El uso de modelos permite al desarrollador diseñar sistemas de una forma gráfica y más comprensible. La Ingeniería dirigida por Modelos utiliza el Lenguaje de Modelado Unificado, con siglas UML en inglés, para definir un modelo. Este lenguaje usa principalmente diagramas para describir las diferentes capas de un sistema. Todos los diagramas y características de UML se muestran en el capítulo 2.2.3 de la memoria.

Cuando se utiliza la Ingeniería dirigida por Modelos no es necesario entender los protocolos de bajo nivel, como SNMP, ya que los modelos están en un nivel superior. Estas características permiten desarrollar aplicaciones de forma más rápida y sencilla.

Pero la Gestión dirigida por Modelos no permite comunicar el modelo con el mundo real. Sólo define una serie de normas y enfoques de desarrollo para hacer más fácil, portátil y comprensible el desarrollo de un sistema.

El propósito de conectar un modelo directamente con el entorno de tiempo de ejecución está definido por la tecnología model@runtime. Ésta extiende los enfoques de la Ingeniería dirigida por Modelos al entorno de ejecución. Con esto se busca añadir comportamiento real en un modelo ya desarrollado y, debido a esto, conseguir la comunicación con el mundo real.

Model@runtime busca extender la aplicabilidad de los modelos producidos en la

Ingeniería dirigida por Modelos al entorno de tiempo de ejecución. El principal objetivo de model@runtime es desarrollar un modelo que se pueda comunicar con el mundo real, sin desarrollo adicional de código: un modelo directamente conectado al entorno de tiempo de ejecución. Un modelo de ejecución pues verse como un desarrollo vivo de un modelo que permite la evolución dinámica y la realización de diseños de software.

Con estas tecnologías se presenta un nuevo enfoque del proyecto. Se busca desarrollar una aplicación que pueda monitorizar los conmutadores de repuesto a través de SNMP. Pero se quiere que la aplicación sea un modelo. Esto introduce un nuevo concepto de gestión de red: gestión de red a través de un modelo de ejecución.

La figura 2 muestra un esquema de lo que se quiere desarrollar, el lugar donde se encuentra el modelo, el mundo real y el nexo de unión, que en este caso es la tecnología model@runtime.

En esta tesis el modelo desarrollado servirá como un monitor de red. Se conectará a una estación de gestión que usará SNMP como protocolo de gestión de red, utilizando una librería SNMP existente. Esta librería está desarrollada en C++ y proporciona servicios SNMP a cualquier desarrollador de aplicaciones de gestión de red. Se puede utilizar para diferentes funciones. Ofrece la ventaja de programar orientado a objetos en gestión de red, y, debido a esto, hace más fácil el desarrollo de aplicaciones más poderosas, portables y robustas.

Para desarrollar la aplicación requerida en este proyecto se han utilizado los siguientes lenguajes de modelado: UML y SDL. UML define un sistema utilizando diferentes diagramas y SDL describe su comportamiento. Ambos serán explicados con más detalle en el capítulo 2.2.3.

La figura 3 muestra una vista general del proyecto. El modelo se ha desarrollado en el ordenador administrador de red. Este administrador habla SNMP con los conmutadores. El modelo se comunica directamente con los conmutadores, usando el enfoque de model@runtime.

Primero, se quiere probar la comunicación con un agente SNMP controlado por Windows XP, instalado en un ordenador portátil. Éste será el escenario de prueba. Después, el modelo se quiere probar en el escenario real, con los conmutadores de repuesto. La aplicación desarrollada permitirá añadir nueva funcionalidad al modelo por medio de nuevos casos de prueba y flujos de trabajo.

La herramienta de desarrollo empleada en el diseño del modelo es Telelogic TAU, de IBM.

A continuación se pasará a explicar los 2 escenarios en los que se ha desarrollado y probado el modelo: el escenario de prueba y el real.

A lo largo del desarrollo del proyecto, la funcionalidad de la aplicación ha sido probada en un escenario de prueba. Este escenario consiste en un ordenador administrador que contiene el modelo desarrollado, y un dispositivo monitorizado, un agente, el cual está corriendo en un portátil con Windows XP como sistema operativo. Microsoft Windows ofrece un servicio SNMP que implementa un agente SNMP. Si este servicio está encendido, permite enviar respuestas SNMP con la información solicitada desde una aplicación de gestión SNMP. Existe otro servicio SNMP que permite al agente enviar mensajes no solicitados llamados “traps”. Pero este tipo de mensajes no son importantes en esta tesis.

Entonces, el agente SNMP controlado por Windows y el administrador utilizan la funcionalidad de gestión de red del modelo desarrollado. Ambos, administrador y agente, pertenecen a la misma subred 129.69.30.0/24.

La figura 4 muestra un esquema de este escenario de pruebas, así como las direcciones IP de administrador y agente.

Al principio, se realizó una aplicación simple que enviaba diferentes comando SNMP desde el administrador al agente y recibía correctamente las respuestas correspondientes. Poco a poco se ha ido añadiendo más funcionalidad hasta llegar al modelo final.

El modelo final fue probado en el escenario real, con los 10 conmutadores de repuesto. En este escenario, se probó la correcta funcionalidad del modelo desarrollado, y el correcto cumplimiento de los objetivos propuestos al inicio, así como la demostración de los conceptos aplicados de las tecnologías usadas.

Los conmutadores están preconfigurados para saber hablar SNMP y están preparados para ser monitorizados. Hay un agente SNMP en cada conmutador.

EL administrador se comunicará con el agente del correspondiente conmutador a través del intercambio de mensajes SNMP, dependiendo del conmutador solicitado para ser monitorizado.

La figura 5 muestra el plano lógico del escenario real, así como las direcciones IP del administrador y de los 10 conmutadores. En este caso todas pertenecen a la red 129.69.99.0/24.

El plano lógico es diferente del físico. Éste último se muestra en la figura 6.

A continuación se va a proceder a explicar la implementación del modelo. Éste se divide en una primera versión del modelo de gestión y la versión final con toda la funcionalidad, que está basada en la primera.

Como objetivo para este proyecto, se ha dicho que se busca el poder utilizar los conceptos de Gestión dirigida por Modelos y model@runtime para desarrollar una aplicación que monitorice un conjunto de conmutadores de repuesto.

Con este propósito, se intenta definir un modelo UML que sirva de monitor de red. Dicho monitor de red se utilizará para obtener información de los dispositivos de seguimiento, en este caso, los conmutadores. Después de esto, dependiendo de los datos recibidos, la aplicación decidirá el estado del dispositivo, es decir, si está listo para utilizarse o si hay algún problema.

Otro objetivo es conectar este modelo UML con una estación de gestión SNMP utilizando una librería SNMP existente. Esta librería es un conjunto de clases desarrolladas en C++ que definen el comportamiento de los comandos SNMP. Hay una clase para cada comando. Cada clase define los métodos necesarios para utilizar la funcionalidad del comando SNMP correspondiente.

El modelo hablará SNMP a través de la librería SNMP mencionada.

La figura 11 muestra la idea del proyecto, de lo que hay y de lo que es necesario

desarrollar. Así, se tiene un banco de pruebas con 10 conmutadores de repuesto, una librería SNMP con una serie de clases desarrolladas en C++, y se quiere desarrollar un modelo UML que sirva de monitor de red y que se comunique con los conmutadores utilizando la librería SNMP existente.

Es necesario que la aplicación permita a los conmutadores monitorizados utilizar un flujo de trabajo. En este proyecto, se ha desarrollado una implementación básica de una aplicación de gestión que comunica el administrador con los conmutadores. Esta aplicación está desarrollada utilizando un modelo UML que define e implementa el comportamiento del administrador. Todavía no hay ningún flujo de trabajo de prueba que diseñe una serie de tareas para controlar los dispositivos, esto sería un trabajo futuro. La aplicación desarrollada para este trabajo de tesis permite añadir flujos de trabajo con diferentes propósitos y más modelos para aumentar la funcionalidad.

El primer modelo desarrollado sirve de base para implementar el modelo final. Por eso, es importante mostrar cómo ha sido desarrollado, aunque se muestre mayor énfasis en el desarrollo del modelo final.

Este primer modelo se centra en permitir la comunicación en el escenario de prueba. Se busca comunicar el administrador con un agente instalado en un ordenador portátil.

La estructura del primer modelo está organizada en clases que están incluidas en paquetes. Hay 3 paquetes que contienen todas las clases y diagramas del modelo. La estructura de los paquetes es la misma en ambas, la primera y la implementación final.

Estos paquetes son los siguientes:

snmpModel: contiene todas las clases principales y los diagrams del modelo.
snmpFiles: contiene las clases C++ externas importadas de la librería SNMP al modelo.
Signals: contiene todas las señales e interfaces para comunicar el administrador y el agente. Pertenece al paquete snmpModel.

Dentro del paquete snmpModel hay 3 clases principales que representan los componentes del modelo. Éstas son: SNMP_System, SNMP_Manager y Agent_Adapter. Existe también otra clase, Manager, que representa al usuario y simula el entorno. Es una clase informal, esto significa que no se genera código para esta clase en la simulación del modelo.

La clase SNMP_System representa el conjunto del sistema de seguimiento dentro del dispositivo administrador. Esta clase contiene a las clases SNMP_Manager y Agent_Adapter. Ambas contienen el comportamiento del modelo.
La clase SNMP_Manager contiene la lógica del administrador. Recibe una señal de entrada del usuario y envía la correspondiente señal al Agent_Adapter (adaptador de agente SNMP).

Agent_Adapter es el vínculo entre el administrador y el agente SNMP instalado en el recurso monitorizado. El adaptador de agente traduce UML a SNMP, es decir, recibe una señal del administrador (desde dentro del modelo) y utiliza la librería SNMP para enviar el correspondiente comando SNMP al dispositivo de seguimiento (en el mundo real).

En la figura 13 se puede ver toda la estructura de paquetes mencionada y cómo

acceden unos paquetes y clases a otros.

Si una clase o paquete puede acceder a otro paquete, esto significa que puede utilizar los recursos contenidos en él, tales como clases C++ (de la librería SNMP), señales o interfaces.

En cuanto al funcionamiento de la aplicación, el primer desarrollo de la aplicación tiene una funcionalidad simple. Sólo trata de probar la conectividad entre el modelo y el mundo real. Envía y recibe señales SNMP que contienen comandos SNMP. Hay 4 señales que comunican el usuario con el administrador, y el administrador con el agente. Son las siguientes: user_snmpGet_req(), snmpGet_req(), snmpGet_reply() y user_snmpGet_reply(). Las señales terminadas en “req” corresponden con las peticiones y las señales terminadas en “reply” son las respuestas.

El usuario (la clase Manager) y el administrador (la clase SNMP_Manager) se comunican mediante el uso de las señales user_snmpGet_req() y user_snmpGet_reply(). Las señales usadas en la comunicación entre SNMP_Manager y Agent_Adapter son snmpGet_req() y snmpGet_reply(). Entre el Agent_Adapter y el agente SNMP en el dispositivo monitorizado se envían las señales definidas en la librería SNMP.

En la figura 14 se muestra el esquema de comunicación de la primera implementación. Se puede ver cómo el usuario envía una señal al administrador. Éste lo recibe y envía la petición con el correspondiente comando SNMP al adaptador del agente. Este último utiliza la librería SNMP para enviar la petición SNMP al agente a través de la red. Cuando el agente recibe la señal de entrada, la procesa y envía la respuesta al administrador. El adaptador de agente la recibe, porque es el enlace entre el dispositivo controlado y la clase SNMP_Manager, y reenvía la respuesta. Cuando el administrador recibe la respuesta, envía al usuario la señal user_snmpGet_reply().

Este modelo ha sido probado con los siguiente comandos SNMP: snmpBulk, snmpDiscover, snmpGet, snmpNext y snmpWalk. El resultado fue satisfactorio para todos ellos.

Los resultados de estas pruebas para la primera implementación del modelo se muestran en el apartado 4.2.3 de la memoria. En ellos se muestran los paquetes capturados en la interfaz de red entre el ordenador administrador y el dispositivo monitorizado, en este caso el agente instalado en el portátil.

Las capturas de red se han realizado con las siguientes herramientas de red: Wireshark y CommView.

A partir de ahora se va a hablar del desarrollo e implementación del modelo de gestión desarrollado para el escenario real. Este modelo está basado en el modelo anterior. Adapta la funcionalidad de éste al escenario real añadiendo nuevas clases y diagramas.

Este modelo comunica al administrador con los agentes SNMP instalados en cada uno de los conmutadores. Cuando el administrador recibe una señal del usuario, envía la señal al correspondiente adaptador de agente, dependiendo de la dirección IP solicitada. Hay un adaptador de agente por cada conmutador.

La estructura de este modelo es la misma que en la primera implementación. Pero en el modelo final hay más clases que añaden funcionalidad. Hay 3 paquetes, como en el anterior modelo: snmpModel, snmpFiles y Signals. Su contenido es muy similar al del

modelo primero.

La figura 18 muestra la estructura en árbol del modelo final, con todos sus paquetes, clases y diagramas.

Como muestra la figura, el paquete snmpFiles contienes un diagrama de clases con los tipos importados de C++ y las clases importadas al modelo en C++, en este caso snmpGet. Éste es el único comando usado en este modelo.

Dentro del paquete snmpModel están todas las clases y diagramas principales para la aplicación. La idea principal de la estructura de las clases es la misma que en la primera implementación: una clase Manager, una clase SNMP_System que representa en sistema de seguimiento, una clase SNMP_Manager y 10 clases Agent_Adapter, una por cada agente de conmutador. Hay además 2 clases más llamadas Input y Switch_Selector.

La primera de ellas sirve para almacenar la dirección IP y el OID solicitados. El OID es el identificador de objeto utilizado en SNMP para identificar de manera única a un objeto gestionado en una base de información de gestión (MIB) jerarquizada. Esta clase Input contiene 2 atributos que contienen los datos mencionados.

La segunda clase, Switch_Selector, contiene una operación llamada selectSwitch() que recibe un número que representa a un conmutador y devuelve la dirección IP correspondiente.

Dentro del paquete snmpModel está el paquete Signals. Éste contiene 3 interfaces diferentes que contienen las señales para comunicar al usuario con el manager y al manager con el adaptador de agente.

En la parte alta del árbol hay un artefacto. Éste es un elemento del modelo UML, con propiedades dedicadas al proceso de construcción. Se usa para traducir un modelo UML en un programa ejecutable.

Hay una clase de actividad llamada Functionality. Ésta contiene diagramas de actividad y permite añadir más flujos de trabajo para incluir diferentes funcionalidades de control al modelo.

Las relaciones entre clases y paquetes es la misma que en la primera implementación y se muestra en la figura 19. Como puede verse, casi todas las clases del paquete snmpModel pueden acceder al paquete de señales, Signals, excepto las clases Input y Switch_Selector porque éstas no son clases con comportamiento, son clases funcionales.

También en la figura podemos ver la relación entre los paquetes snmpModel y snmpFiles. Es la misma que en modelo anterior: el paquete snmpModel puede acceder a snmpFiles para así poder usar las clases C++ importadas al modelo, y, gracias a esto, poder comunicarse con el mundo real.

Por último, en relación a la estructura, es necesario decir que todos los adaptadores de agente (clases Agent_Adapters) son parte de la clase SNMP_Manager. Ésta es la mejor manera de manejar las llamadas a los adaptadores de agente. Dentro del modelo, la clase SNMP_Manager llama al Agent_Adapter correspondiente directamente desde sus atributos.

EL modelo final implementa una funcionalidad completa. Se busca obtener

información de monitorización sobre los conmutadores de repuesto del RUS. Para este propósito, el usuario escoge un conmutador para ser monitorizado y un identificador de objeto específico, la aplicación envía la petición y espera la respuesta. Cuando la obtiene, muestra la salida. En este modelo, el usuario controla qué conmutador es monitorizado.

Existen diferentes señales en el modelo para usar entre el usuario y la clase SNMP_Manager y entre SNMP_Manager y Agent_Adapter:

- Entre el usuario y SNMP_Manager hay sólo una señal llamada prepare_snmpGet(switch_number, OID), la cual contiene 2 parámetros: el número de conmutador que se quiere controlar y el identificador de objeto solicitado.
- Para comunicar SNMP_Manager con Agent_Adapter hay 10 diferentes señales, cada una de ellas asociada a un Agent_Adapter concreto. Sus nombres son: snmpGet_switch1(), snmpGet_switch2(), snmpGet_switch3(), y así hasta snmpGet_switch10(). Sólo tienen 1 parámetro, una instancia de la clase Input que contiene la dirección IP y el OID solicitados.

Las señales entre SNMP_Manager y el correspondiente adaptador de agente son enviadas como operaciones implementadas en cada clase Agent_Adapter. Ésta es la manera más útil y fácil de implementar esta funcionalidad y comunicarse con el correspondiente Agent_Adapter.

La figura 20 muestra un esquema de cómo funciona este modelo. Describe el diagrama del sistema y el intercambio de mensajes entre usuario, manager y conmutadores.

La clase SNMP_Manager contiene casi toda la lógica del sistema. Esta clase recibe señales entrantes del usuario, y las redirige al correspondiente Agent_Adapter para comunicarse con el conmutador correspondiente. El comportamiento de la clase antes mencionada está representado en el diagrama de la máquina de estados de la figura 21.

Hay 9 clases más, una para cada conmutador. Depende del conmutador escogido, se envía una petición y se espera por más señales de entrada.

A continuación, se va a explicar el funcionamiento, paso a paso, del esquema de trabajo de esta implementación:

El usuario decide que conmutador se va a monitorizar y envía al manager la señal prepare_snmpGet() con parámetros el número de conmutador y el OID.

La clase SNMP_Manager recibe esta señal y llama a la operación selectSwitch() en la clase Switch_Selector. Esta operación contiene una tabla de correlación entre el número de conmutador y su dirección IP. Por eso, dependiendo del número recibido devolverá la dirección IP correspondiente. Con esta información, SNMP_Manager guarda la IP y el OID en una instancia de la clase Input.

Dependiendo del número de conmutador, la clase SNMP_Manager llama al correspondiente Agent_Adapter y manda la petición con la clase Input que contiene almacenada la información.

El Agent_Adapter recibe esta información y se comunica con el correspondiente conmutador utilizando la librería de SNMP. Después de esto, espera la respuesta. Cuando la recibe, muestra el resultado. La figura 22 muestra un ejemplo del funcionamiento de la aplicación.

Los resultados obtenidos se muestran a partir de la figura 23. Las pruebas fueron realizadas en el escenario real y los resultados mostrados son capturas de pantalla obtenidas de la comunicación entre el manager y el conmutador 2.

Después del desarrollo de este proyecto, se puede decir que se han logrado todos los objetivos propuestos al principio y se han probado los conceptos del Servicio de Gestión dirigido por Modelos.

Se ha conseguido desarrollar un modelo en UML que sirva como monitor de red.

Se ha logrado total conectividad entre el modelo desarrollado y el mundo real.

Se ha demostrado la mejora de la utilización y el manejo del sistema cuando se usan modelos.

Se ha demostrado que un modelo desarrollado con UML se puede comunicar con dispositivos de red.

Se ha logrado integrar la librería SNMP existente en TAU y usarla para conectar el modelo UML con los agentes SNMP instalados en los conmutadores de repuesto.

Se ha demostrado simplicidad en la notación de las clases de prueba y la facilidad para añadir nuevos flujos de trabajo al modelo.

En este punto, se proponen las siguientes adiciones y mejoras a este modelo de gestión y administración:

Crear un interfaz gráfico de usuario para hacer más fácil y comprensible en manejo de sistema de monitorización.

Mejorar el manejo de errores del modelo. En este momento, hay un control de errores simple: la aplicación envía la petición desde el adaptador de agente al conmutador un máximo de 2 veces, si el agente SNMP del conmutador no responde, el adaptador de agente devuelve el control a la clase SNMP_Manager.

Desarrollar aplicaciones de negocios. Una ventaja del uso de la Gestión dirigida por Modelos es la facilidad de añadir nuevos flujos de trabajo al modelo implementado. Por eso, se propone desarrollar grandes flujos de trabajo que puedan manejar diferentes aspectos de la gestión de red.

Abstract

At present, business are very dependent on networking services. So, network management is increasingly important.

There are different technologies that seek to facilitate development of management business applications, such as Model-driven Management. This technology aims to abstract a system by way of separating system architecture from platform architecture. This serves to make easier to understand a system because it is divided in layers. Models work in a high level, it means that is not necessary to understand low level protocols.

If these technologies are mixed with a network management protocol, such as Simple Network Management Protocol (SNMP), it can be achieved a management application more powerful, easier to develop, easier to manage and easier to modify. These features are desirable in order to develop applications easily scalable.

But this project aims a new approach of Model-driven Management: network management via a runtime model. It is possible thanks to model@runtime. This concept seeks to communicate a model with the real world without additional code development, directly from the model.

This project is focused in the study and applicability of Model-driven Management and model@runtime concepts. It seeks to take advantage of use of models in the development of business applications, but applied to network management.

For this purpose, it will be develop an UML model which serves as a network monitor of a set of spare switches. This model will be use an existing SNMP library developed in C++ in order to communicate the model with the switches using SNMP messages that contain SNMP commands.

Index

Abstract......i
Index......ii
1. Introduction......1
1.1. General scenario and problem formulation......1
1.1.1. Test scenario......4
1.1.2. Real scenario......4
1.2. Structure of the report......6
2. Related work......8
2.1. Simple Network Management Protocol......8
2.1.1. What is SNMP?......8
2.1.2. Different versions: SNMPv1, SNMPv2 and SNMPv3......11
2.1.3. Importance of Network Management......13
2.2. Model-driven Engineering......13
2.2.1. Generic aspects of MDE......14
2.2.2. Model@runtime: extension of MDE to runtime environment......14
2.2.3. UML/SDL Modeling Languages......15
3. Runtime environment......17
3.1. Development tool (Telelogic UML Tau)......17
3.2. Network analyzer tools......18
3.2.1. Wireshark......19
3.2.2. CommView......19
3.3. Object-Oriented Network Management Development......19
3.3.1. SNMP++ (SNMP and C++)......19
4. Implementation of model......21
4.1. Goals......21
4.2. First implementation: Simple Management Model......22
4.2.1. Structure......22
4.2.2. Working scheme......23
4.2.3. Testing......24
4.3. Final implementation: Final Management Model......26
4.3.1. Structure......27
4.3.2. Working scheme......29
4.3.3. Testing......31
5. Conclusions and Outlook......34
Appendix A: Index of figures......35
Appendix B: Glossary......36
Appendix C: Bibliography......37

1. Introduction

This first chapter outlines the reasons that motivated to do this project and goals to be achieved. It defines the technology framework of the developed application. It also describes the test scenario and the real scenario, where it was checked its functionality. Then, it describes the structure of the document.

1.1. General scenario and problem formulation

Use of networks to communicate and distribute information between the different departments of a business is very common today. So, it is very important to keep working these networks 24 hours a day. To achieve this purpose, it is useful the network management in order to monitor and check the devices and components of the network.

In this case, the business is the University of Stuttgart, Germany. It has different departments, such as Institut für Kommunikationsnetze und Rechnersysteme (IKR), Institut für Intelligente Systeme (IIS), Institut für IT-Services (IITS), etc.

This document is focused in a network operated by a specific department of the University, called Rechenzentrum Universität Stuttgart (RUS). This network has different types of devices, such as servers, computers, printers... They are interconnected by way of switches, and use routers to connect them to Internet and other networks.

The following picture depicts a general overview of how this network is:

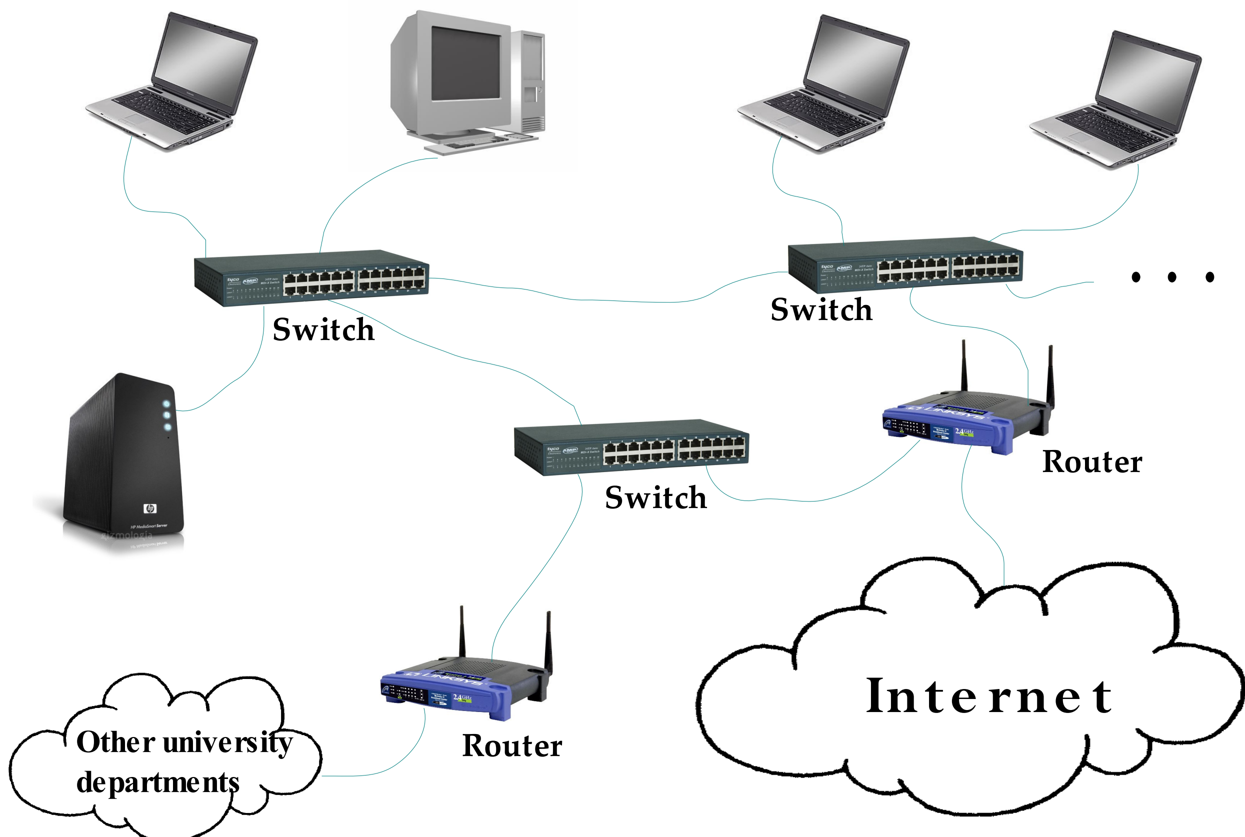


Figure 1: General scheme of the network operated by RUS

RUS operates this network. It is desirable not to lose connection between devices, or between other departments or Internet. But there are no ideal networks. Sometimes, switches or routers fall down, and, if there is not any alternative route, there is a connection loss with a device or group of them.

The motivation of this thesis is focused in solve such a connection loss as soon as possible. It could be realized, for example, by way of creating new alternative routes using spare devices.

RUS has 10 spare switches in a test rack. A network switch is a computer networking device that connects different network segments. If any device falls down, it could be used a spare switch to replace the device fallen down, and, because of this, send the traffic through the new switch in order to reach the same destination.

This is the idea of RUS: use the spare switches to solve a connection loss. But, it is not known if the switches are ready to use; this means if they are working, if they are configured, if all the ports are enabled, and so on. It is necessary to monitor these devices and, depends on the information received, make this decision by way of test cases and workflows

At this point, it presents an important approach of the project. In order to specify these test cases and workflows, it is going to be used Model-driven Management, because it offers a set of advantages at the time of development.

It is preferable to develop an application that can be easy to change, this means with minimal impact on code, portable between different platforms, easy to design, etc. To get these purposes and more that will explain in chapter 2.2, it is used Model-driven Engineering.

To get the monitoring data, it will be used the SNMP protocol (Simple Network Management Protocol). It communicates the manager and the monitored devices through the exchange of SNMP messages that contains SNMP commands. It will be explained with more detail in chapter 2.1.

Using models allows the developer to design systems in a graphical and more understandable way. Model-driven uses Unified Modeling Language (UML) to define a model. This language mainly uses diagrams to describe the different layers of a system. All of the diagrams and features of UML will show in chapter 2.2.3.

When it uses Model-driven Engineering, it is not necessary to understand low level protocols, such as SNMP, because models are in a higher level. These features allow to develop applications easier and faster.

But Model-driven Management do not allow to communicate the model with the real world. It only defines a set of rules and development approaches, to make easier, more portable and more understandable a system development.

The purpose of connect a model directly to the runtime environment is defined by model@runtime. It extends the approaches of Model-driven Engineering to runtime environment. It aims to add real behavior in a developed model and, because of this, achieve communication with the real world. Model@runtime will be explained in chapter 2.2.2.

With these technologies, it presents a new approach of the thesis. It seeks to develop an application that can monitor the spare switches via SNMP. But the application is required to be a model. This introduces a new concept of network management: network management via a runtime model.

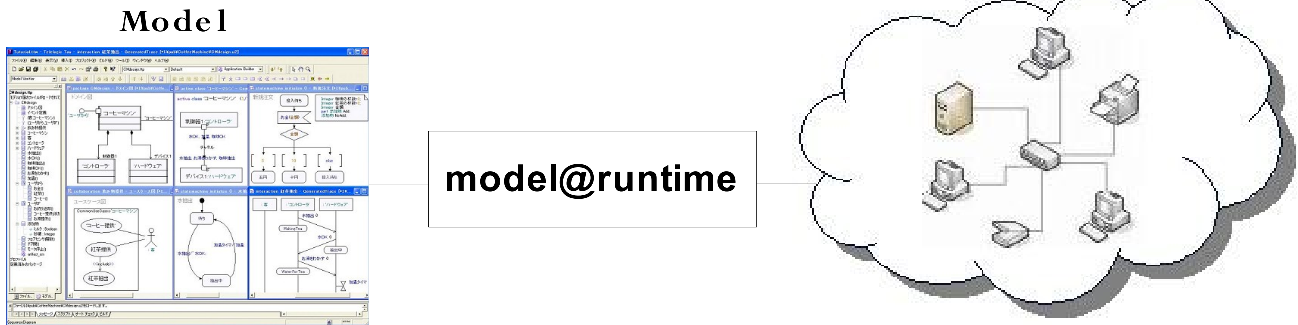


Figure 2: Model@runtime schema

The picture above shows how to communicate a model with the real world. Model@runtime is the link between the model to be developed and the real environment. It adds behavior without additional code development, directly from the model.

In this thesis, the developed model will serve as a network monitor. It will be connected to an SNMP management station using an existing SNMP library. This library is developed in C++. In chapter 3.3.1 it will be explained. To develop this application it will be used UML and SDL modeling languages. UML defines a system using different diagrams and SDL describes its behavior. Both of them will be describe with more detail in chapter 2.2.3.

The picture below gives a general overview of the project. The model will be deployed in the network manager. This manager will talk SNMP to the switches. The model will communicate directly with the switches, using the approach of model@runtime.

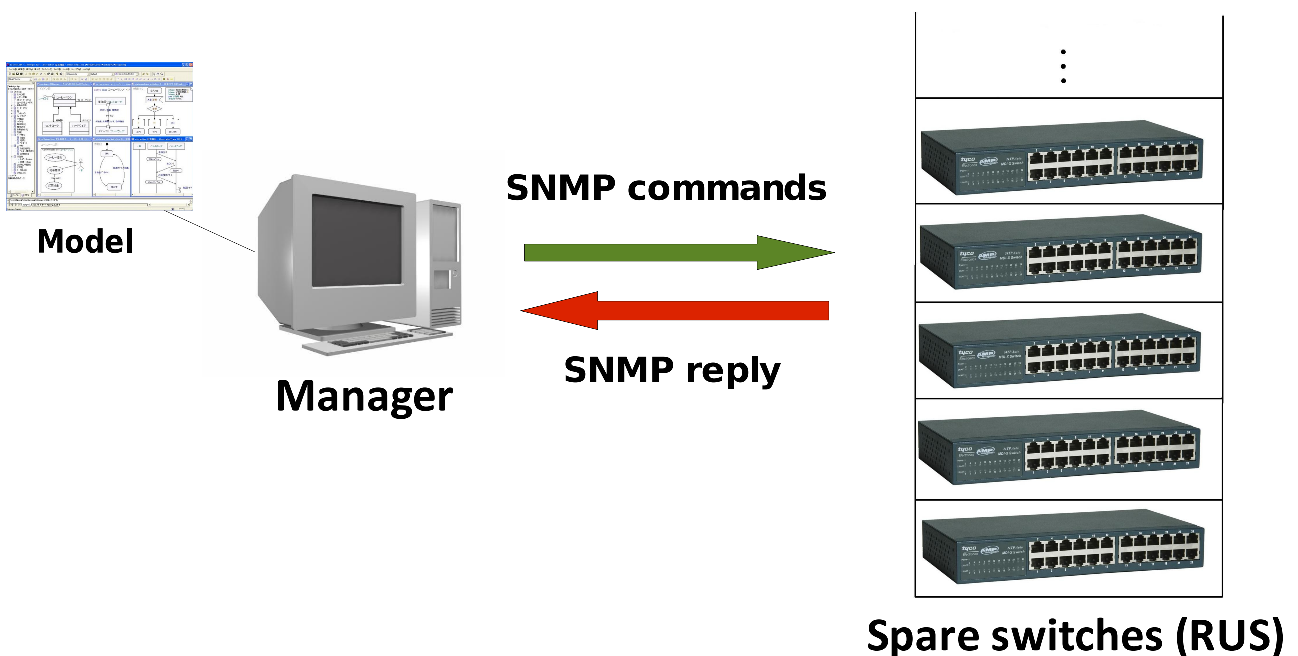


Figure 3: General schema of the project's development

First, it will communicate with an SNMP agent controlled by Windows XP, installed in a laptop. It will be the test scenario. Then, the model will be checked in the real scenario, with the spare switches. Both scenarios will be explain in next chapters. This application will allow to add functionality by way of adding new test cases and workflows to the model.

The development tool to design the model will be Telelogic TAU. This IBM tool will be detailed in chapter 3.1.

1.1.1. Test scenario

Along the development of the project, the functionality of the application has been checked in a test scenario. This scenario consists in a manager that contains the developed model, and a monitored device, an agent, which is running in a laptop with Windows XP operating system. Microsoft Windows offers an SNMP service that implements an SNMP agent. If this service is turned on, it allows to send SNMP responses with the information required from an SNMP management application. There is another SNMP service that lets the agent to send unsolicited messages called traps. But they are not important in this thesis.

So, the SNMP agent is controlled by Windows and the manager uses the network management functionality of the developed model. Both of them, manager and agent, belong to the same sub net 129.69.30.0/24.

The figure below describes an schema of this scenario, including IP addresses.

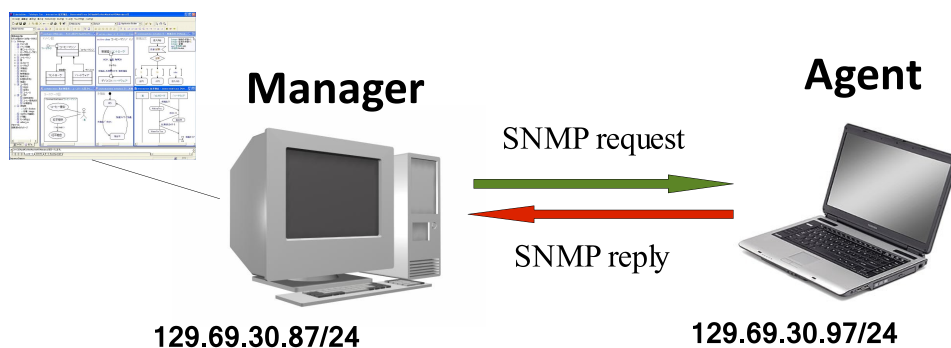


Figure 4: Schema of test scenario

At first, it performed a simple application that sent different SNMP commands from the manager to the agent and received the SNMP responses. Then, it added more functionality until develop the final model. The development of this final application will be detailed in following chapters, besides testing and result of test scenario and real scenario.

1.1.2. Real scenario

The final model was tested in the real scenario, with the 10 spare switches. In this scenario, it was proved the right functionality of the developed model, the fulfillment of goals agreed at the beginning and the proof of used technology concepts.

Switches are preconfigured to talk SNMP and ready to be monitored. There is an SNMP agent in each switch.

Manager will communicate with the agent of the corresponding switch through the exchange of SNMP messages; it depends on the required switch to be monitored.

The following picture shows the logical plan of this scenario:

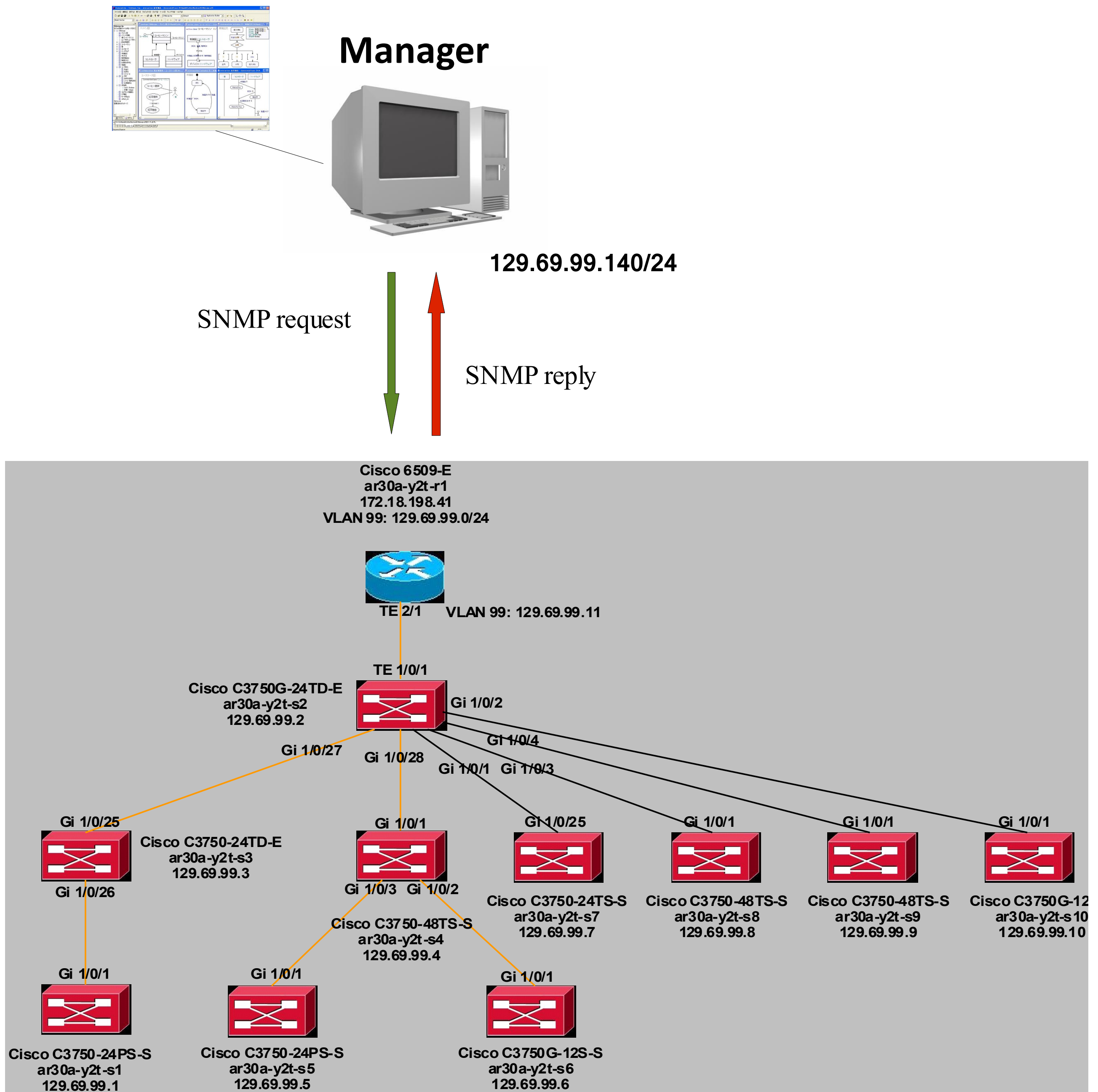


Figure 5: Logical schema of real scenario

Red devices are spare switches in the test rack mentioned before, and blue device is a router that communicates them with other networks and devices. In this case, both the manager and the agents belong to network 129.69.99.0/24.

This set of switches establishes the testbed of the project.

Physical plan and logical plan are different. Figure 6 depicts the physical structure of the switches, it means how they are interconnected.

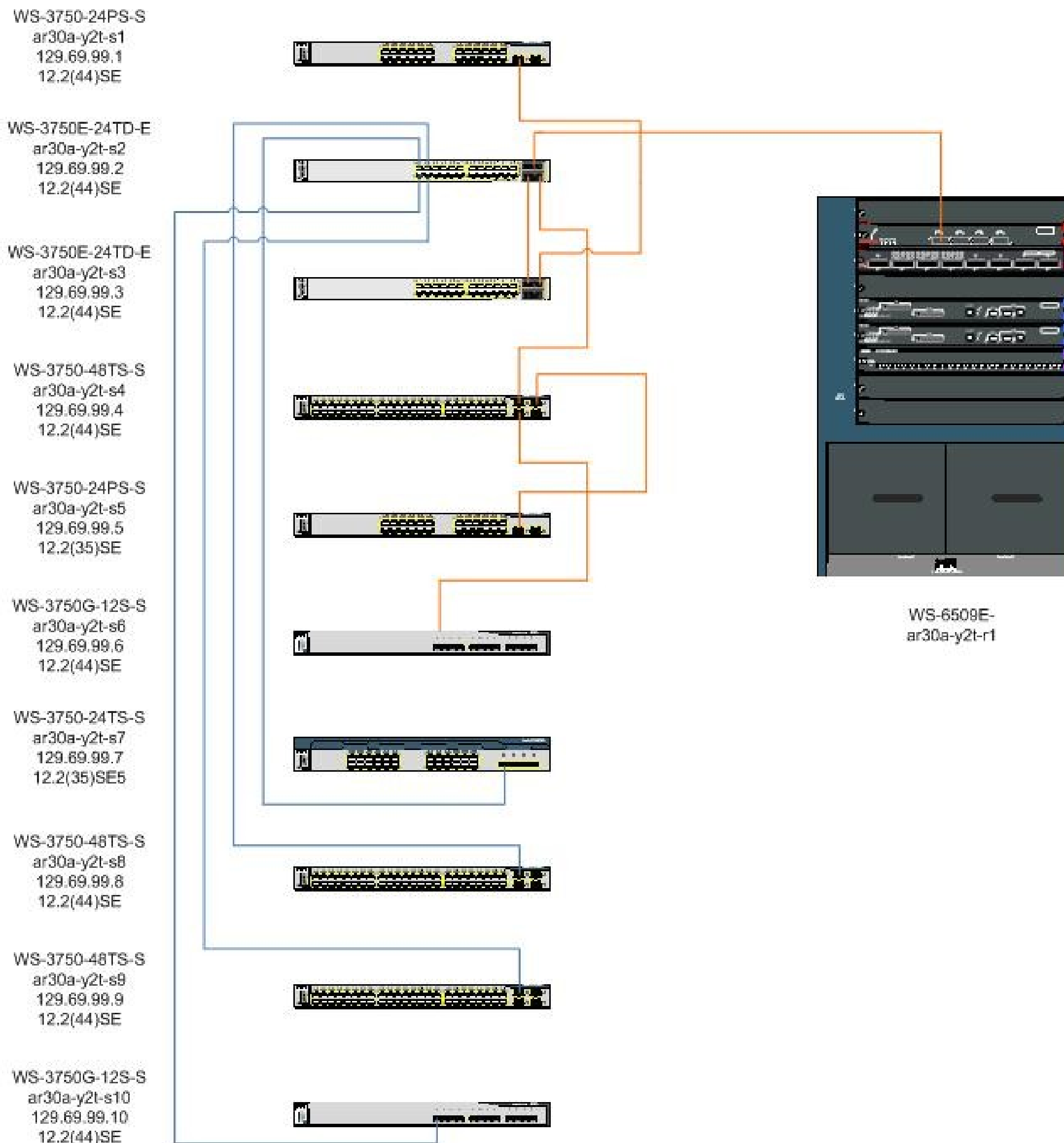


Figure 6: Physical plan of real scenario

1.2. Structure of the report

This project is structured into 5 chapters and 3 appendices which are summarized below:

- Chapter 1: Introduction
  This chapter presents the motivation that led to the realization of this thesis and the goals outlined in its development. Finally, it presents a brief section that describes the content of this report.

- Chapter 2: Related work
  Chapter 2 shows a brief survey of technologies, protocols, development languages, etc..., used in the implementation of this application.

- Chapter 3: Runtime environment
  In the third chapter, it describes the tools used to develop the system.

- Chapter 4: Implementation of model
  This chapter details the design and development of the implemented application.

- Chapter 5: Conclusions and Outlook
  Chapter 5 discusses the conclusions obtained after the project development. It also includes future research work and development in order to improve the implemented system.

- Appendix A: Index of figures
  Appendix A contains an index of the figures of this document.

- Appendix B: Glossary
  Appendix B contains a glossary with the acronyms used in this report.

- Appendix C: Bibliography
  This appendix shows the consulted references to make this project.

2. Related work

2.1. Simple Network Management Protocol

2.1.1. What is SNMP?

SNMP is the acronym of Simple Network Management Protocol. This protocol provides a standardized network management framework for enabling the control and monitoring of an internetwork. It is used mostly in network management systems to monitor network-attached devices for conditions that warrant administrative attention. It uses the manager/agent model, and its protocol operates at the application level of the TCP/IP model.

SNMP is a component of the Internet Protocol Suite as defined by the Internet Engineering Task Force (IETF). It consists in a set of standards for network management which are published as Request for Comments (RFC). These RFCs are available for unlimited distribution.

An SNMP managed network consists of three key components:

- Managed device (slave device)
  It is a network node which contains an SNMP agent. These devices get and store management information, which can be exchanged with the Network Management System. Managed devices can be routers, access servers, switches, bridges, hubs, IP telephones, computer hosts or printers.

- Agent
  It is the network management software in a managed device. This processing entity receives requests from Network Management Systems in its community, processes them if they are valid, and sends the appropriate response. Agents can also be configured to send trap messages to report asynchronous events.
  Each agent has its own Management Information Base (MIB), which contains its collection of variables of interest.

- Network Management System (NMS)
  It is the processing entity that monitors and controls the agents that it is responsible for on the internetwork. The NMS and its agents make up a community. The NMS can read and write, depends on the privacy, certain MIB objects in each agent to manage that network device.
  One or more NMSs may exists on any managed network.

Messages in SNMP are sent over UDP. Typically, this protocol uses the following UDP ports: 161 for the agent and 162 for the manager. The manager may send requests from any available source port to port 161 in the agent (destination port). The agent response will be sent back to the source port. The agent may generate notifications (TRAPs and INFORMs) from any available source port and send them to the port 162 in the manager.

With the SNMP protocol, monitoring a network device's state is usually by polling.

However, the agent can also send a limited number of traps to guide the Network Management Systems focus and timing of that polling.

The SNMP communicates its management information through the exchange of SNMP protocol messages. Each message is completely and independently represented within a single datagram as presented to the UDP transport service. Use of this no connection oriented transport protocol ensures that management tasks will not influence on overall system performance. In this way, it will be avoid the use of control and recovery mechanisms as in a connection oriented service, for example TCP.

The general format of SNMP messages consists of a message header and a message body. Each of these messages contains a version identifier, the SNMP community name and the PDU. Last one is the Protocol Data Unit, that depends on the executed operation. The version identifier is a constant used for version control that the Network Management System and agents must know. If the NMS or agent receives an SNMP message containing an invalid or unsupported version number, the message is discarded. The SNMP community name is a string that identifies a particular group of NMSs and agents. Members of a community enforce authentication by using a crude password scheme. This simple use of a non-encrypted, plain-text community name by communicating NMSs and its agents is called the Trivial Authentication Scheme. For SNMP Version 1 it is the only SNMP security measure in place. The community name is represented as a string of octets. The default community name often used is *public*.

SNMP is based in the following three major components: Structure of Management Information (SMI), Management Information Base (MIB) and the protocol itself.

- **Structure of Management Information (SMI)**

The Structure of Management Information defines the rules for describing management information. It shows how to describe and encode objects in the MIB.

The SMI defines the common structures and generic types, along with the identification scheme, that are to be used in the implementation. It describes the format and the layout of the objects in the database.

Abstract Syntax Notation One (ASN.1) is the standard used in SMI. It allows defining type of data that can be used and recognized in two different entities which communicate. Some examples of simple type of data are INTEGER, BOOLEAN, REAL, BIT STRING, OCTECT STRING, NULL, OBJECT IDENTIFIER, etc. Also, complex types can be formed from these simple types. In this thesis, OBJECT IDENTIFIER (OID) will be used to get the information of monitored devices.

- **Management Information Base (MIB)**

The MIB describes the structure of the management data that can be accessed through the network management protocol. It uses a hierarchical namespace based in object identifiers (OIDs). Each OID identifies a variable that can be read or set via SNMP. An object identifier uniquely identifies a managed object in the MIB hierarchy.

Whereas the SMI provides the general framework for the definition of the managed objects, the Management Information Base declares the particular instances for each object and then binds a value to each one.

A MIB can be depicted as a tree structure. Different levels are assigned by different organizations: top-level MIB OIDs belong to different standards organizations, while lower-level object IDs are allocated by associated organizations. The OID shows the path of the object requested.

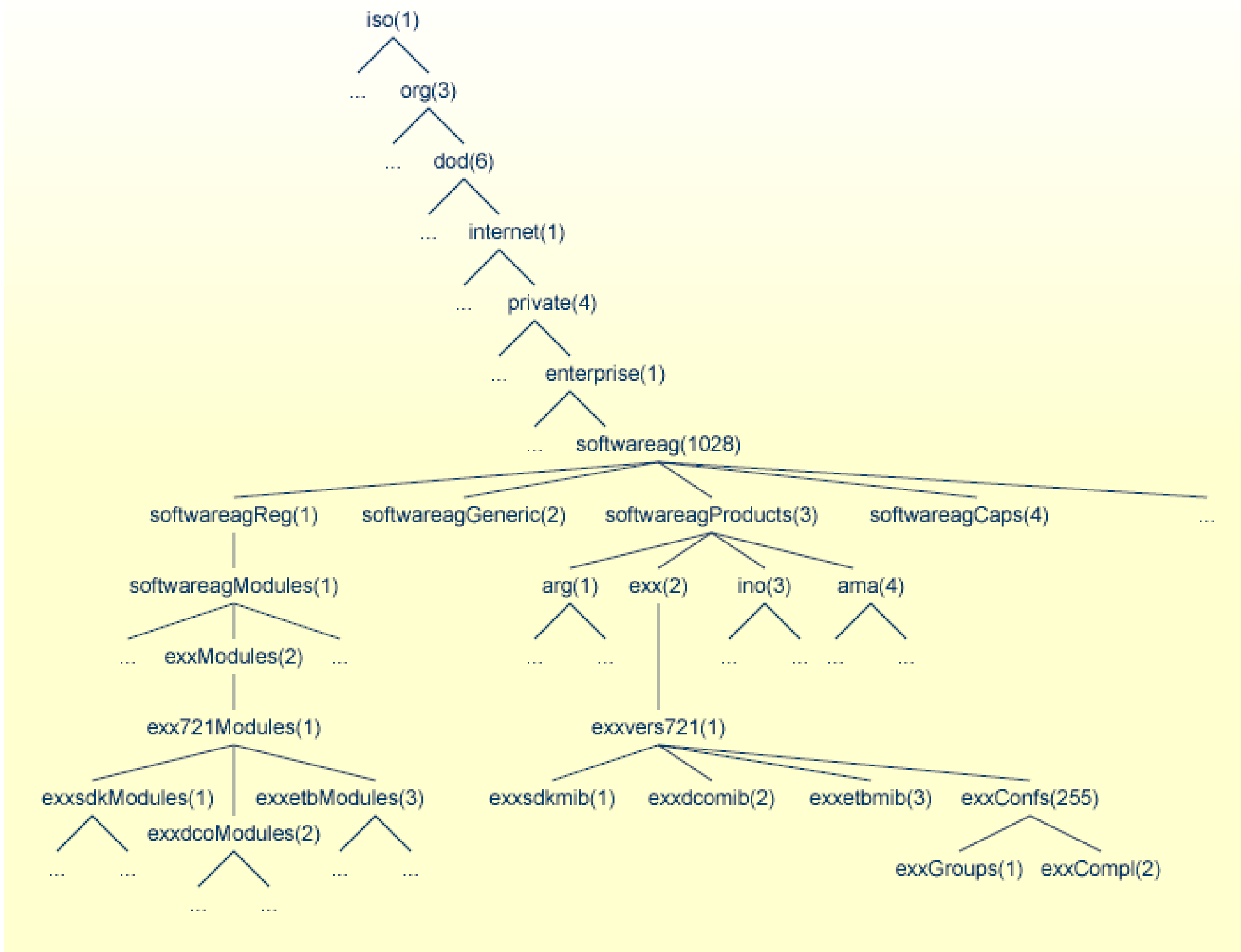


Figure 7: MIB structure

As the picture shows, an example of OID could be:

exxsdkModules: OID = (1.3.6.1.4.1.1028.1.1.2.1.1)

- **SNMP Protocol**

SNMP is the definition of the application protocol for the network management service. It is an asynchronous request and response protocol between an Network Management System (NMS) and the agent. Initially, the NMS was capable of sending three different messages containing the Protocol Data Units (PDUs): GetRequest-PDU, GetNextRequest-PDU and SetRequest-PDU. The agent was capable of sending two different messages: a response acknowledgment with the GetResponse-PDU to a proper request from an NMS, and also a message with a Trap-PDU. Last one is an unsolicited event sent when the agent has discovered a predefined extraordinary event. This PDUs are defined for the first version of SNMP (SNMPv1). Later versions implement more PDU messages, as it will be discussed in next chapter.

2.1.2. Different versions: SNMPv1, SNMPv2 and SNMPv3

In the late 80s, the Internet Architecture Board (IAB), responsible of define Internet politics, decides to create a framework standard for network management. That is the reason SNMP was developed, in order to solve management problem in TCP/IP networks. This protocol defines the message formats for commands and responses and also delineates authentication and authorization schemes used by the administrative framework.

Currently, there are three different versions of Simple Network Management Protocol. The third version of SNMP was created from the previous versions, SNMPv1 and SNMPv2. All versions have the same basic structure, components and architecture.

SNMP Version 1 fulfills the three major goals for the protocol architecture defined in RFC 1157:

- minimize the number and complexity of management functions.
- design should be extensible to handle future needs of the network's operation and management.
- architecture should be independent of other network devices and vendor specific issues.

About security, SNMPv1 uses a trivial authentication mechanism. This technique is simple and easy to implement but is not too safe; it offers the barest of protection against malicious or erroneous SNMP messages. The key component of this trivial authentication scheme is the use of a community name. As it said before, a community name is a string that represents the set of NMSs and agents that belong to a common and known group. When an agent receives a command, it checks the community field in order to compare it to the community name string stored in its configuration. If they match, the message is considered authentic by the protocol and is passed on for further processing. If not, the message is discarded.

The SNMPv1 message has the following fields:

- Version number: it is used for SNMP compatibility. For version1, according to RFC 1157, it is represented as an integer type with value 0.

- Community string: this is the community name explained before.

- Protocol Data Unit (PDU): it contains the request or response of the SNMP protocol. In SNMPv1 there are five supported PDU types:
  - GetRequest-PDU
  - GetNextRequest-PDU
  - SetRequest-PDU
  - GetResponse-PDU
  - Trap-PDU

First three of them are sent from the NMS to the agent, while the others are sent from the agent to the NMS.

The figure below shows an SNMP version 1 message:

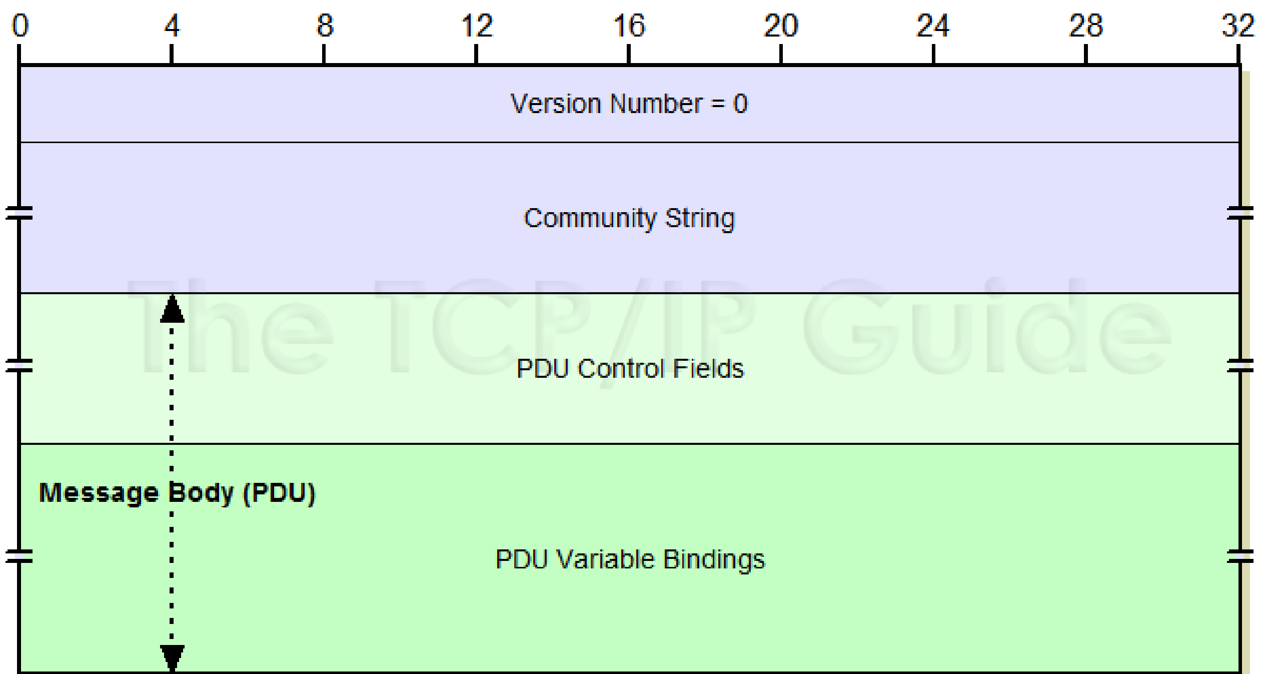


Figure 8: SNMP version 1 (SNMPv1) General Message Format

SNMP version 2 protocol builds upon the version 1 protocol. In fact, both of them are compatible. This version was created to enhance SNMPv1 in many areas, such as MIB objects definition and protocol operations. This SNMPv2 tried to improve the security of the previous version. That is the reason of the several different SNMPv2 variants (RFCs 1901, 1905, 1907...) . Finally, it was decided to use the SNMPv1 framework keeping the new protocol enhancements introduced by SNMP Version 2.

The SNMPv2 uses the same authentication and authorization scheme used in Version 1. Its message format is also the same as SNMPv1. The only difference is in version field, where the value is 1 for SNMPv2.

For Version 2 eight PDU types now exists:

- GetRequest-PDU
- GetNextRequest-PDU
- SetRequest-PDU
- Response-PDU
- GetBulkRequest-PDU
- InformRequest-PDU
- SNMPv2-Trap-PDU
- Report-PDU

These PDU types add functionality to the SNMP protocol.

In the late 1990s, Version 3 was created mainly to address the deficiencies related to security and administration of the previous versions. The SNMPv3 framework adopts many components that were created in SNMPv2, including the SNMPv2 protocol operations, PDU types and PDU format. The Request for Comments that describe SNMPv3 are RFCs 3411 and 3412.

2.1.3. Importance of Network Management

Networks exist to distribute information. This information needs to be communicated in an efficient, effective and reliable manner. Network management is an essential factor in successfully operating a network. It monitors, in real time, the network activity, it controls the operations of the devices, and it can be responsible for any number of other related tasks. Network management provides management not just on networks themselves but also of services running over those networks.

Effective network management ensures quality network services and accurately tracks service provider performance.

Today's companies invest more heavily in enterprise network infrastructure than in any other type of information technology solutions. Therefore, enterprises become increasingly dependent on networking services. That is the reason of importance of network management, because keeping those services running is synonymous with keeping the business running.

"Network management software is designed to enhanced the operation, administration, and provisioning of networks and related assets within the network. These powerful, comprehensive applications allow businesses to better control and maintain their networks by centralizing and automating all activities related to tracking, monitoring, servicing, and fine-tuning its components. " [1]

Simple Network Management Protocol was introduced in previous chapters, because it has become the most popular network management solution in use today.

2.2. Model-driven Engineering

Information systems today are heterogeneous and decentralized: there are some different platforms, operating systems, network protocols, etc. So it is increasingly important platform independence, portability and interoperability between applications.

In this sense, Model-driven Engineering (MDE) has been created as a software development methodology which focuses on creating models, or abstractions, more close to some particular domain concepts rather than computing concepts. Modeling introduces abstraction to reduce complexity. This idea seeks: compatibility between systems, simplifying the process of design and promoting communication between particulars and teams working on the system. With multiview modeling techniques, each view can provide a simple, easy to understand view of a system.

A modeling paradigm for MDE is considered effective if its models make sense from the point of view of the user and can serve as a basis for implementing systems. It seeks total convergence between business management and software engineering.

The best known MDE initiative is from the Object Management Group (OMG). This consortium has developed a set of standards called Model-driven Architecture (MDA).

2.2.1. Generic aspects of MDE

Model-driven Engineering refers to a range of development approaches that are based on the use of software modeling as a primary form of expression. Sometimes models are constructed to a certain level of detail, and the code is written by hand in a separate step. MDE aims to enhance portability by way of separating system architecture (abstract) from platform architecture (concrete).

With the introduction of Unified Modeling Language (UML), MDE has become very popular today with a wide body of practitioners and supporting tools. The continued evolution of MDE has added an increased focus on architecture and automation.

Model-driven Engineering separates business and application logic from underlying platform technology. Platform-independent models of an application or integrated system's business functionality and behavior, built using UML or other modeling standards can be realized. These platform-independent models document the business functionality and behavior of an application separate from the technology-specific code that implements it, insulating the core of the application from technology. MDE seeks that design and architecture keep separated.

With MDE, the business and technical aspects of an application or integrated system are no longer tied to each other. Business logic can respond to business need, and technology can take advantages of new developments, as the business required.

2.2.2. Model@runtime: extension of MDE to runtime environment

As said before, today's applications are complex, inevitably distributed and operate in heterogeneous and rapidly changing environments. These mission-critical software systems are often expected to safely adapt to changes in their execution environment. But, it is often inconvenient to take them offline to adapt their functionality. So, they are required to be adaptable, flexible, reconfigurable and, increasingly, self-managing, in order to adapt their behavior at runtime with little or no human intervention. Such features make systems more prone to failure when executing and thus the development and study of appropriate mechanisms for run-time validation and monitoring is needed. In the model-driven software development area, research effort has focused primarily on using models at design , implementation and deployment stages of development. It is needed to research on self-adaptive and self-managing software.

Model@runtime seeks to extend the applicability of models produced in model-driven engineering approaches to the runtime environment. As historical studies in reflection (computer science) say: *"Models should represent the system and should be linked in such a way that they constantly mirror the system and its current state and behavior; if the system changes, the representations of the system – the models – should also change, and vice versa."* [2] In model@runtime, it is also sought appropriate self-representations for adaptive systems.

The main goal in model@runtime is to develop a model that can communicate with the real world, without code development: models directly connected to the runtime environment. A runtime model can be seen as a live development model that enables

dynamic evolution and the realization of software designs.

Another definition of this models can be: *“a model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective.”* [3]

2.2.3. UML/SDL Modeling Languages

UML is the acronym of Unified Modeling Language and SDL means Specification and Description Language. Both of them are modeling languages that serve to design and define a system.

UML is a mainly graphical notation used to specify, visualize, modify, construct and document the different diagrams of an object-oriented software intensive system under development. UML combines data modeling (entity relationship diagrams), business modeling (work flows), object modeling and component modeling. It aims to be a standard modeling language which can model concurrent and distributed systems.

UML diagrams represent two different views of a system model:

- Structural view: Structural diagrams define the static architecture of a model. They are used to model the components that make up a model: classes, objects, interfaces, attributes, and also the relationships and dependencies between elements.
- Behavioral view: Behavior diagrams capture the interactions and instantaneous states within a model as it executes over time. They show how the system will act in a real-world environment, by describing the dynamic behavior of the system using collaborations among objects and changes to the internal states of objects.

Current UML specification, UML 2, defines thirteen basic diagram types categorized hierarchically as shown in the following class diagram:

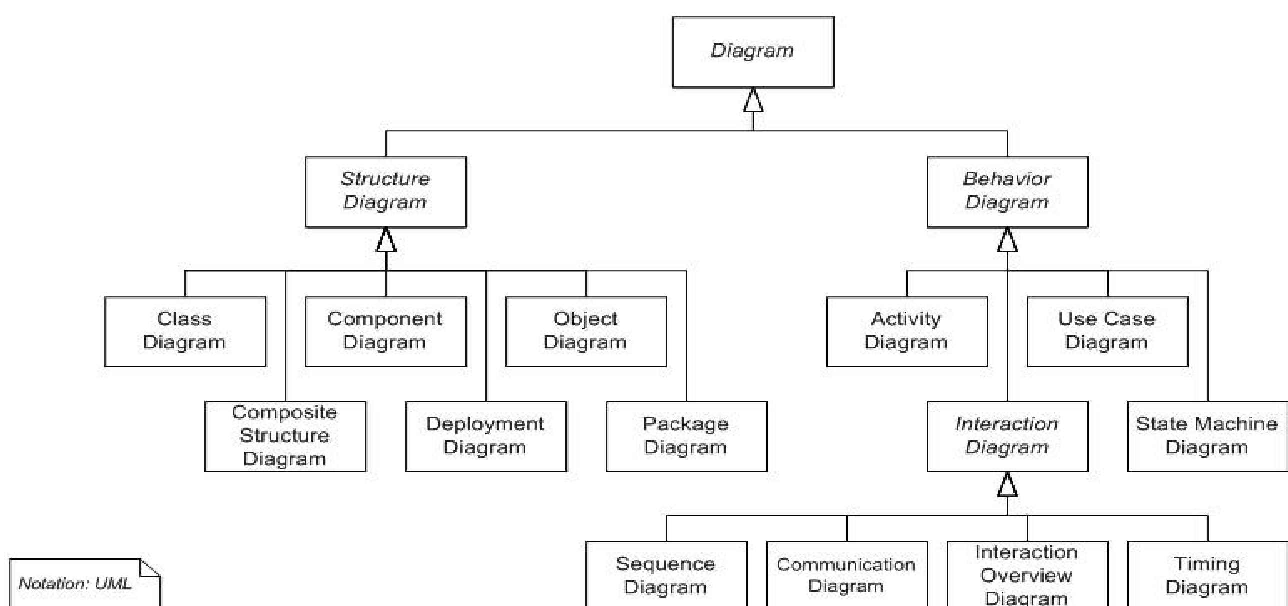


Figure 9: UML class diagram

Class diagram defines the basic building blocks of a model by showing the system's classes, their attributes and the relationships among the classes.

Composite structure diagram describes the internal structure of a class and its collaborations. It provides a means of layering a structure of an element.

Component diagram is used to model higher level or more complex structures. It shows how a software system is split up into different components and the dependencies among these components.

Deployment diagram shows the physical disposition of significant artifacts within a real-world setting.

Object diagram shows how instances of structural elements are related and used at run-time.

Package diagram is used to divide the model into logical groupings, or 'packages', and describes the interactions between them at a high level.

Activity diagram shows the overall flow of control. It represents the business and operational step-by-step workflows of components in a system.

Use case diagram is used to model the interactions between user and system.

State machine diagram is essential to understand the run state of a model when it executes.

Sequence diagram represents an interaction among objects in a sequential way by time. It shows the sequence of messages passed between objects using a vertical time-line.

Communication diagram shows the network and sequence of messages or communications between objects at run-time. It describes both the static structure and dynamic behavior of a system.

Interaction overview diagram fuses activity and sequence diagrams to allow interaction fragments to be easily combined with decision points and flows.

Timing diagram fuses sequence and state diagrams to provide a view of an object's state over time and messages which modify that state.

The Specification and Description Language is a specification language that describes the behavior of distributed systems. Currently, it is focused on process control and real-time applications in general. SDL covers five main aspects: structure, communication, behavior, data and inheritance.

The language is formally complete, so it can be used for code generation for either simulation or final targets.

SDL has a hierarchical level structure. Usually a system is specified as a set of interconnected state machines that contribute to the action carried out by the system. Each state machine is contained in a process agent. Signals are sent between the environment and an agent or between agents. When a signal is received, it is placed in a queue (the input port). When the state machine is waiting in a state, if the first signal in the input port is enabled for that state it starts a transition leading to another state. Transitions can output signals to other agents or to the environment. It is also allowed to call a remote procedure type to invoke a procedure in another agent (or even another system) and wait for a response.

3. Runtime environment

3.1. Development tool (Telelogic UML Tau)

In this chapter, it will be explained the software tool used in the development process of this project.

Telelogic Tau is a UML 2.1-based modeling tool that supports automated code generation and model verification. It provides Model-Driven development of complex systems and real-time software for enterprise applications, including Service Oriented Architectures. Today, Tau is made by IBM. It was part of the IBM April 2008 acquisition of Telelogic.

This tool permits create models by using diagrams and state machines to design and describe the behavior, the different views of the model and the relations between them. Tau includes a check functionality tool, with the purpose of finding errors in the model as soon as possible. It can also be tested the behavior of the design by way of the model verifier. It can evaluate the responses to received signals.

These are the Tau user interface main areas:

1. The Workspace window.
2. The Desktop.
3. The Output window.

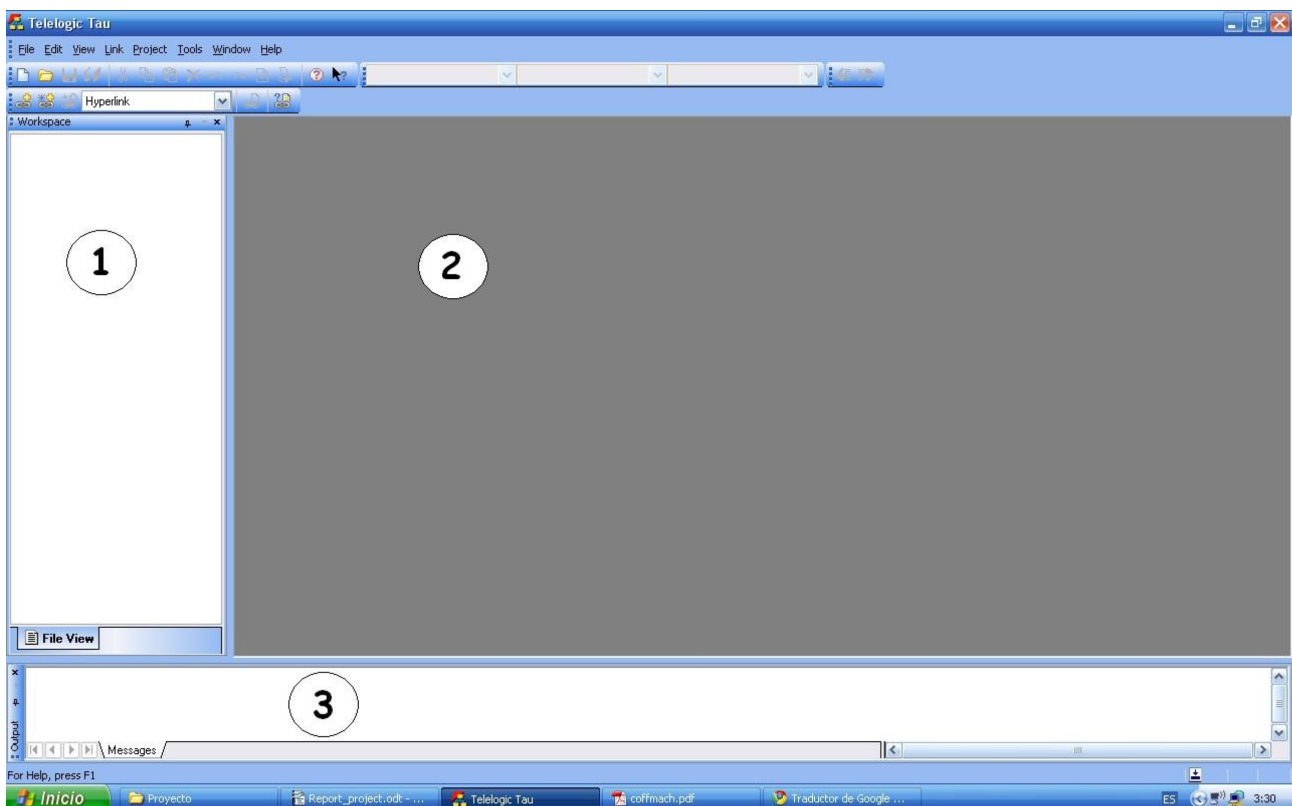


Figure 10: Telelogic Tau user interface

1. The Workspace window: it presents the entities contained in a model. There are different views available for displaying different kinds of information: ***File View*** (it shows all elements that are represented as files), ***Model View*** (it contains all UML elements) and ***Instances*** (it is only active when simulating the system).

2. The Desktop: it is the area where diagrams and documents appears when opened.

3. The Output window: it is used for logging events and displaying errors and warnings.

Telelogic Tau uses projects as a way of grouping the contents within a workspace. A project contains all the diagrams and documents corresponding to a system development and they can be moved between projects. A project can be shared between users but a workspace cannot.

To design a system, Tau permits the use of different UML diagrams, and signals to communicate the different classes. It defines three different kinds of classes:

Active class: it contains behavior.
Passive class: it contains only definitions.
Informal class: it is an entity that is not a part of the model designed itself. For this kind of class no code will be generated during the simulation of the model.

The relations between classes defined in Tau are the following:

Association: it represents a relationship between objects.
Composition: it specifies that the lifetime of the dependent class is directly related to the lifetime of the main class. The dependent class is part of another class (main class). In a composition relationship, data usually flows in only one direction.
Aggregation: it depicts a class as a part of, or a subordinate to, another class. A part class can belong to more than one aggregate class and it can exist independently of the aggregate. Aggregation relationships do not have to be unidirectional.
Aggregation is closely related to composition.

Tau uses a build artifact to translate a UML model into an executable program. A build artifact represents the model or a subset of it. It is an element in the UML model, with properties dedicated to the build process.

Tau's iterative requirements-based approach, comprehensive error-checking, and automated simulation increase developer productivity from initial requirements through final documentation and deployment.

3.2. Network analyzer tools

During the development of this thesis it has been necessary prove the connectivity and analyze network packets between the manager and the agent. For this purpose, different tools were used. This chapter will show a brief outline of the used network analyzer tools: Wireshark and CommView.

3.2.1. Wireshark

Wireshark is the world's foremost network protocol analyzer. It is a free open-source software. It is a cross-platform, it runs on various operating systems including Linux, Mac OS, Solaris and Microsoft Windows.

Wireshark is used for network troubleshooting, analysis, software and communications protocol development and education. Its original name was Ethereal, but in May 2006 the project was renamed Wireshark due to trademark issues.

This tool allows the user to see all traffic being passed over the network by putting the network interface into promiscuous mode.

In this project, Wireshark was used over Windows XP. In this case, it captures all traffic which passes through network interface but not through local interface (localhost). On Linux, it is allow to capture this traffic but on Windows this feature is disabled. So, in the development of this project, Wireshark was used to capture traffic over the network interface and the next tool to analyze traffic over local interface.

3.2.2. CommView

CommView is a powerful network monitor and analyzer. It is similar to Wireshark and offers the user alike functions. But it is not free. For this project, it was used an evaluation version that is available in Internet.

This tool was used, as said before, to capture traffic over the local interface.

3.3. Object-Oriented Network Management Development

Due to increasing growth of networks, the need to develop and deploy network management applications has become very important today. There are various SNMP application programming interfaces (APIs) that allow the creation of network management applications. But the majority of these APIs require the programmer to be familiar with the details of SNMP and SNMP resource management. Most of these APIs are platform specific, resulting in SNMP code that is specific to a particular operating system or network operating system platform, and thus, not portable.

However, there are software development tools based on object-oriented design methodology. They allow developers to create powerful network management applications quickly without necessarily having to be experts on network protocols.

Hewlett-Packard developed a freely available open specification called SNMP++. It uses a set of C++ classes for network management that provide many benefits including ease of use, safety, portability and extensibility.

3.3.1. SNMP++ (SNMP and C++)

SNMP++ is a set of C++ classes that provide Simple Network Management Protocol services to a network management application developer. It is a SNMP library

developed in C++ which can be used for different SNMP functions. SNMP++ brings the object-oriented advantage to network management programming, and in doing so, makes it much easier to develop powerful, portable, and robust network management applications.

SNMP++ provides many benefits, such as:

Ease of use: Using SNMP++, application programmers do not need to know low level SNMP mechanisms. It provides an interface in which the user does not have to be an SNMP or C++ expert to use its features. It encapsulates and hides the internal mechanisms of SNMP.
One of the major goals of SNMP++ was to develop an API that would be scalable to SNMP version 2 with minimal impact on code.

Programming safety: SNMP++ manages all the resources, such as memory and sockets, automatically in a safe way. This includes SNMP structures, sessions and transport layer management.
It provides error checking and automatic timeout and retry. SNMP++ addresses a variety of communication errors, such as lost datagrams, duplicated datagrams and reordered datagrams, and provides the user with transparent reliability.

Portability: Another major goal of SNMP++ was to provide a portable API across a variety of operating systems, network operating systems and network management platforms. A programmer who use SNMP++ does not need to make changes to move the program to another platform. This set of C++ classes can also run across a variety of protocols. Currently, it operates over the Internet Protocol (IP) or Internet Packet Exchange (IPX) protocols.

Extensibility: SNMP++ can be extended by way of supporting new operating systems, network operating systems, network management platforms, protocols, SNMP version 2 and other new features. Through C++ class derivation, SNMP++ users can inherit what they like and overload what they wish to redefine.

The application programmer can subclass the base SNMP++ classes to obtain specialized behavior and attributes. The base classes of SNMP++ are meant to be generic and do not contain any vendor-specific data structures or behavior, in order to facilitate portability. New attributes can be easily added through C++ subclassing and virtual member function redefinition.

In the development of this thesis, it was used SNMP++ to manage the corresponding resources.

4. Implementation of model

In this chapter, it will be explained the developed model. This is divided in a first version of a simple management model and the final version which is based on the first. Explications for each of these applications are divided in three parties: structure, working scheme and testing.

This chapter also talks about the goals agreed for the concrete work.

4.1. Goals

As said in previous chapters, this project aims the use of Model-driven Management and model@runtime concepts to develop a management application to monitor a set of spare switches.

With this purpose, it seeks to define an UML model which serves as a network monitor. This network monitor will be used to get different information of the monitored devices, in this case, the switches. After that, it depends on the received data, the application will decide the state of the device; it means, if it is ready to use or if there is any problem.

Another goal is to connect this UML model to an SNMP management station using an existing SNMP library. This library is a set of classes developed in C++. It defines the behavior of the different SNMP commands. There is one class for each command. Each class defines the needed methods to use the functionality of the corresponding SNMP command.

This model will talk SNMP by way of the mentioned SNMP library.

The picture below shows the idea of the discussed goals.

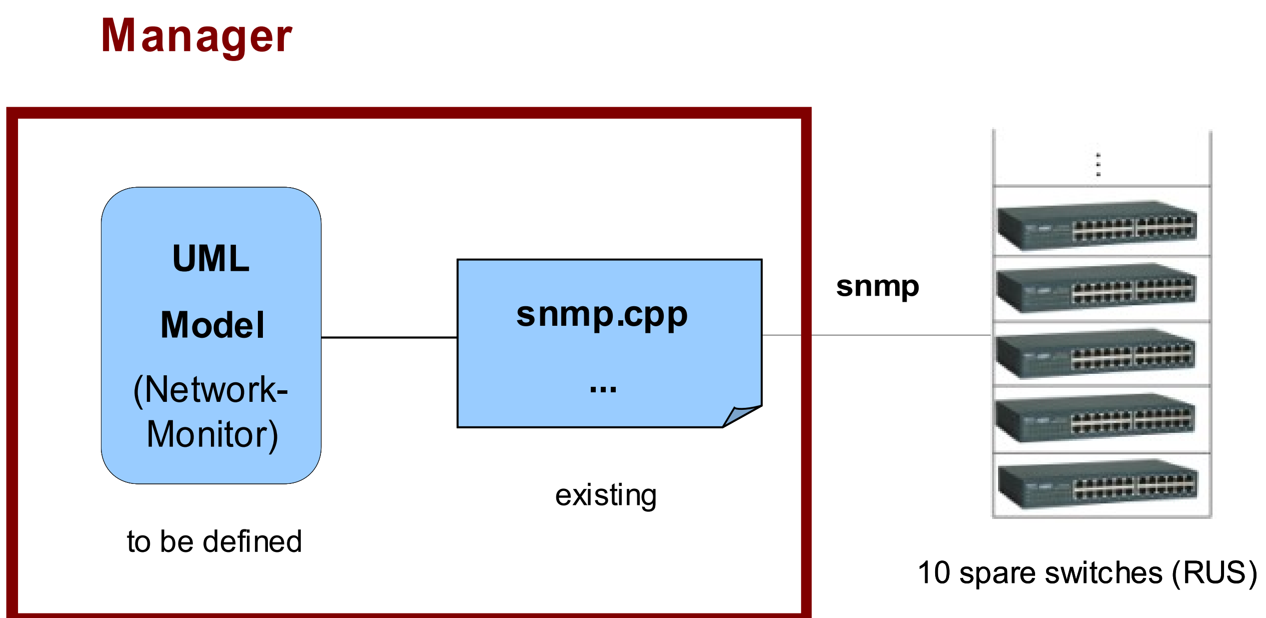


Figure 11: Schema of concrete work

It is required that the application lets the monitoring switches using a test workflow. In this project, it was developed a basic implementation of a monitoring application that communicates the manager with the switches. This application is developed using an UML model that defines and implements the behavior of the manager. There is not any test workflow yet to design a set of tasks to monitor the devices; this would be a future work, but this application allows to add workflows with different purposes and more models to increase the functionality. The following figure is an example of a test workflow:

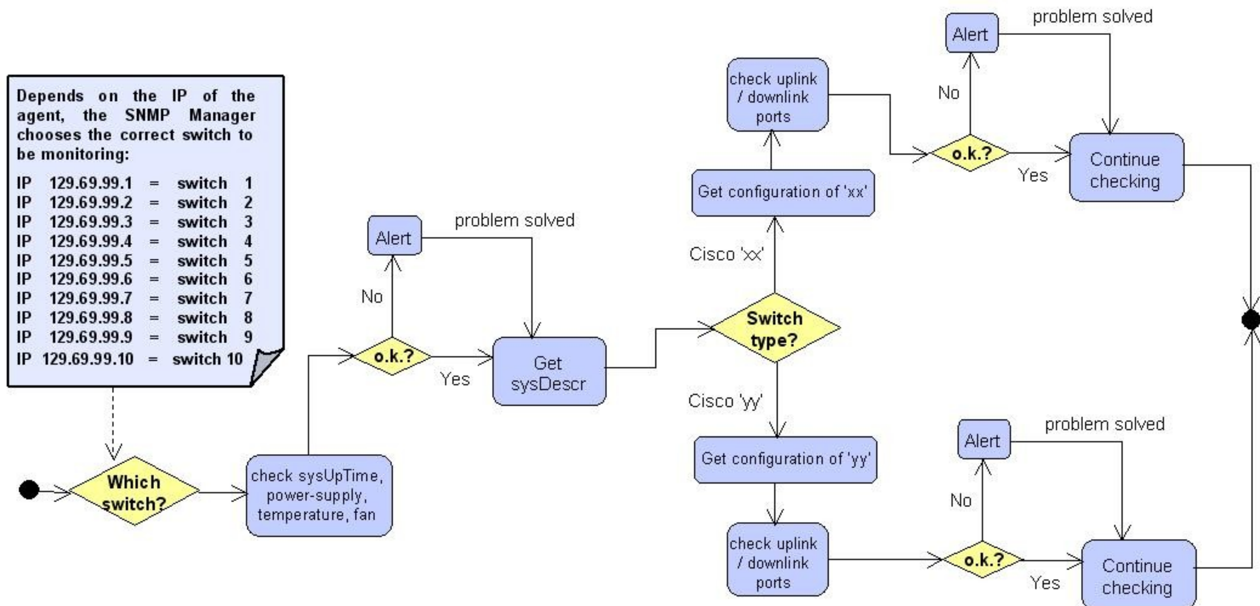


Figure 12: Example workflow

4.2. First implementation: Simple Management Model

In this chapter, it will be explained the first developed application which serves as a base to implement the final model. So, it is important to show how it is implemented but this document will focus with more emphasis in the final developed model.

This model is focused in allow the communication in the test scenario. It aims to communicate the manager with an agent installed in a laptop.

In next chapters, it will be shown structure, working scheme and results achieved for this model.

4.2.1. Structure

Classes of the model are organized into packages. There are three packages that contain all the classes and diagrams of the model. This structure of packages is the same in both, first implementation and final implementation.

These packages are the following:

snmpModel: It contains all the main classes and diagrams of the model.
snmpFiles: It contains the external C++ classes from the SNMP library imported to the model.

Signals: It contains all the signals and interfaces to communicate manager and agent. It belongs to snmpModel package.

Inside the snmpModel package there are three main classes that represent the components of the model. They are SNMP_System, SNMP_Manager and Agent_adapter. There is also another class, Manager, that represents the user and simulates the environment. This is an informal class, it means that there is no code generation for this class during the simulation of the model.

SNMP_System represents the monitoring system inside the manager device. It contains the SNMP_Manager and the Agent_Adapter. Both of them contain the behavior of the model.

SNMP_Manager contains the logic of the manager. It receives an input signal from the user and sends the corresponding signal to the Agent_Adapter.

Agent_Adapter is the link between the manager and the SNMP agent installed in the monitored resource. It “translates” UML to SNMP, it means that the adapter receives a signal from the manager (inside the model) and uses the SNMP library to send the corresponding SNMP command to the monitored device (real world).

This picture shows the relations among classes and packages:

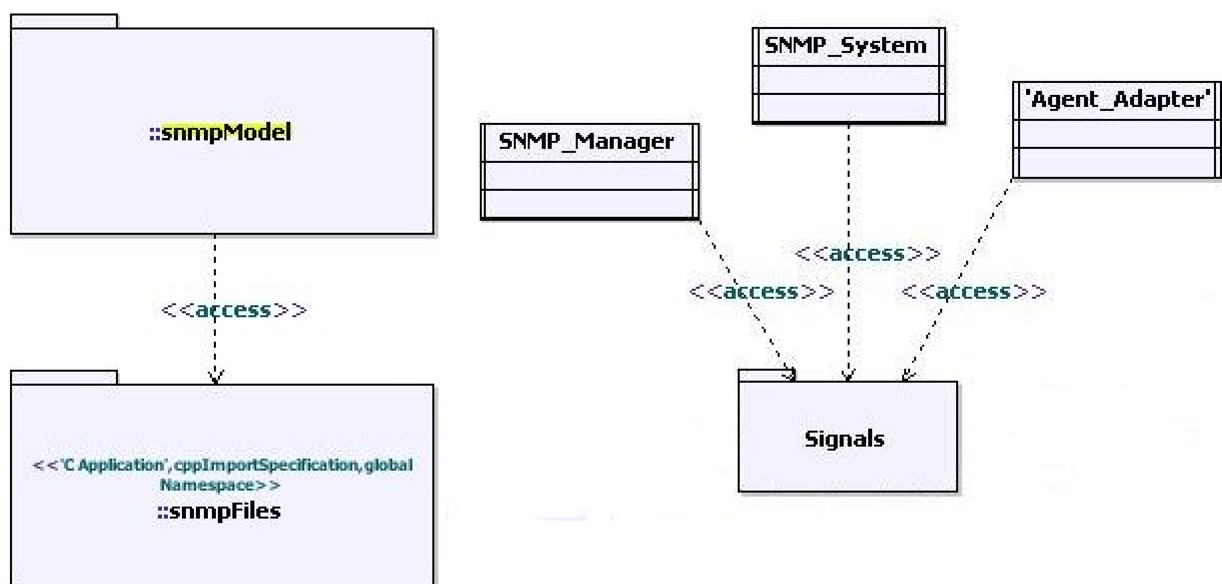


Figure 13: Package structure of first implementation

If a class or package can access to another package, it means that it can use the resources contained in it, such as C++ classes (SNMP library) or signals and interfaces.

4.2.2. Working scheme

This implementation has a simple functionality. It just try to prove connectivity between the model and the real world. It sends and receives SNMP signals containing SNMP commands. There are four signals to communicate the user with the manager, and the manager with the agent. These are the following: user_snmpGet_req(), snmpGet_req(), snmpGet_reply() and user_snmpGet_reply().

User (Manager class) and manager (SNMP_Manager class) are communicated by way of user_snmpGet_req() and user_snmpGet_reply() signals. Signals used in the communication between SNMP_Manager and Agent_Adapter are snmpGet_req() and snmpGet_reply(). Among the Agent_Adapter and the SNMP agent in the monitored device, they are sent SNMP signals defined by SNMP library.

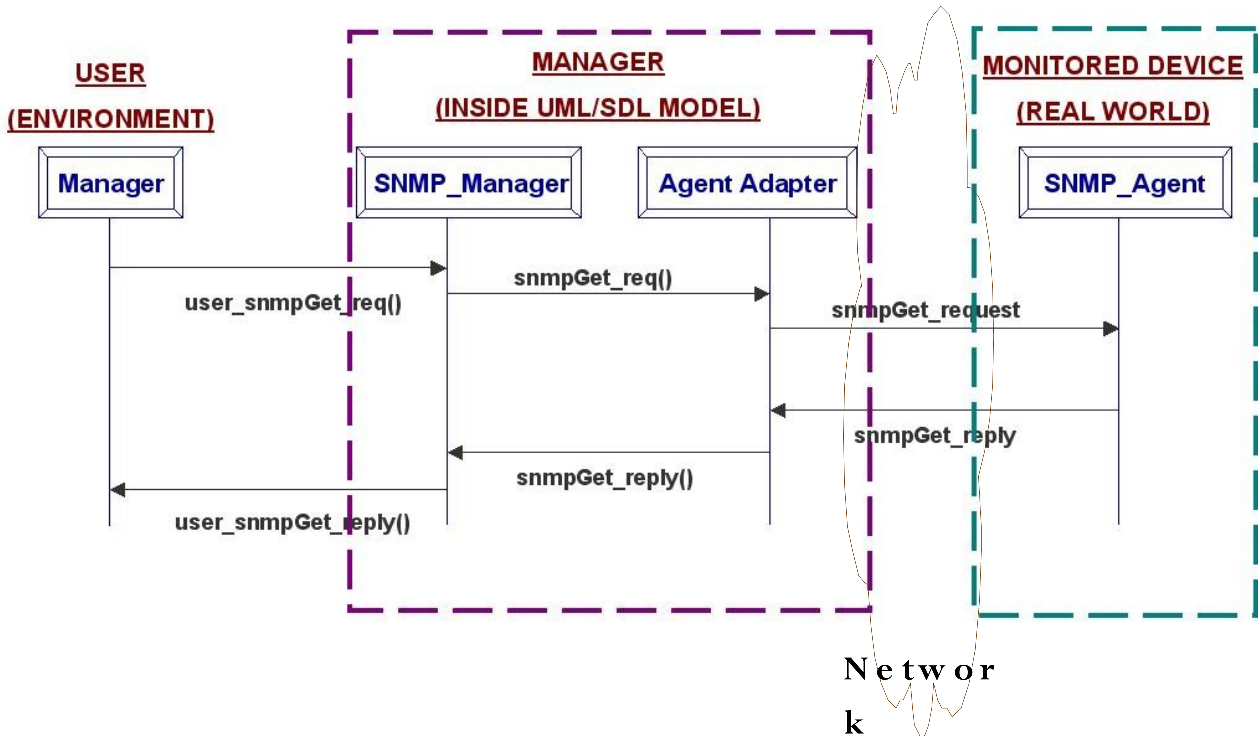


Figure 14: Communication scheme of first implementation

As seen in the image above, user sends a signal to the manager. The manager receives it, and sends a request with the corresponding SNMP command to the Agent_Adapter. This last one uses the SNMP library to send the SNMP request to the agent through the network. When the agent receives the input signal, it processes it and sends the response to the manager. Agent_Adapter receives it, because it is the link between the monitored device and the SNMP_Manager, and sends the reply to it. When the manager gets the response, it sends user_snmpGet_reply() signal to the user.

This model was proved with the following SNMP commands: snmpBulk, snmpDiscover, snmpGet, snmpNext and snmpWalk. Their result was successful.

4.2.3. Testing

In order to check that the model works, they were used the network analyzer tools mentioned in chapter 3.2, such as Wireshark and CommView.

Results shown later are a testing of the use of snmpGet command. They show packets captured in network interface between the manager and the monitored device of the test scenario.

Their IP addresses are:

Manager = 129.69.30.87/24
Agent = 129.69.30.97/24

The following picture is a screenshot from Wireshark. It shows an SNMP GetRequest and its corresponding response from the agent.

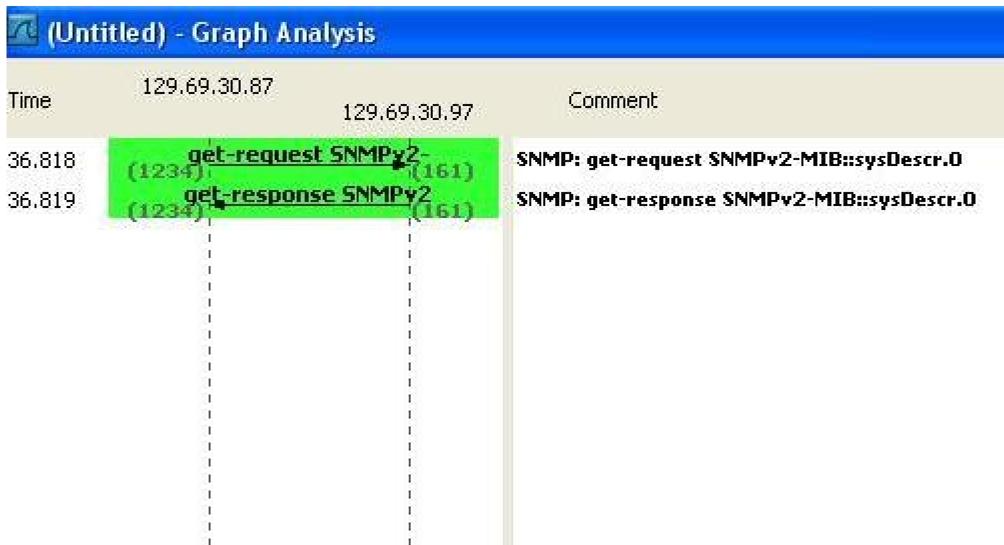


Figure 15: Testing of first implementation (trace diagram: request and reply)

The next two figures show the details of each packet, get-request and get-response.

(Untitled) - Wireshark

File Edit View Go Capture Analyze Statistics Telephony Tools Help

Filter: snmp Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Info
241	36.817851	129.69.30.87	129.69.30.97	SNMP	get-request SNMPv2-MIB::sysDescr.0
242	36.819093	129.69.30.97	129.69.30.87	SNMP	get-response SNMPv2-MIB::sysDescr.0

```
⊞ Frame 241 (85 bytes on wire, 85 bytes captured)
⊞ Ethernet II, Src: 3com_be:78:cc (00:60:97:be:78:cc), Dst: DellPcba_e3:d3:d2 (00:0d:56:e3:d3:d2)
⊞ Internet Protocol, Src: 129.69.30.87 (129.69.30.87), Dst: 129.69.30.97 (129.69.30.97)
⊞ User Datagram Protocol, Src Port: search-agent (1234), Dst Port: snmp (161)
⊟ Simple Network Management Protocol
    version: version-1 (0)
    community: public
  ⊟ data: get-request (0)
    ⊟ get-request
        request-id: 1042
        error-status: noError (0)
        error-index: 0
      ⊟ variable-bindings: 1 item
        ⊟ SNMPv2-MIB::sysDescr.0 (1.3.6.1.2.1.1.1.0): unSpecified
          ⊞ Object Name: 1.3.6.1.2.1.1.1.0 (SNMPv2-MIB::sysDescr.0)
            unSpecified

0000  00 0d 56 e3 d3 d2 00 60  97 be 78 cc 08 00 45 00   ..V....` ..x...E.
0010  00 45 29 97 00 00 80 11  d1 ce 81 45 1e 57 81 45   .E)..... ...E.W.E
0020  1e 61 04 d2 00 a1 00 31  e0 2b 30 27 02 01 00 04   .a.....1 .+0'....
0030  06 70 75 62 6c 69 63 a0  1a 02 02 04 12 02 01 00   .public. ........
0040  02 01 00 30 0e 30 0c 06  08 2b 06 01 02 01 01 01   ...0.0.. .+......
```

File: "C:\DOCUME~1\Silvia 2\LOCALS~1\Temp\... Packets: 260 Displayed: 2 Marked: 0 Dropped: 0

start | Command Prompt | Telelogic Tau 4.2 | SNMP_Monitoring.tt... | Servicio de Correo El... | (Untitled) - Wireshark | SNMP_Monitoring

Figure 16: Testing of first implementation (get-request packet)

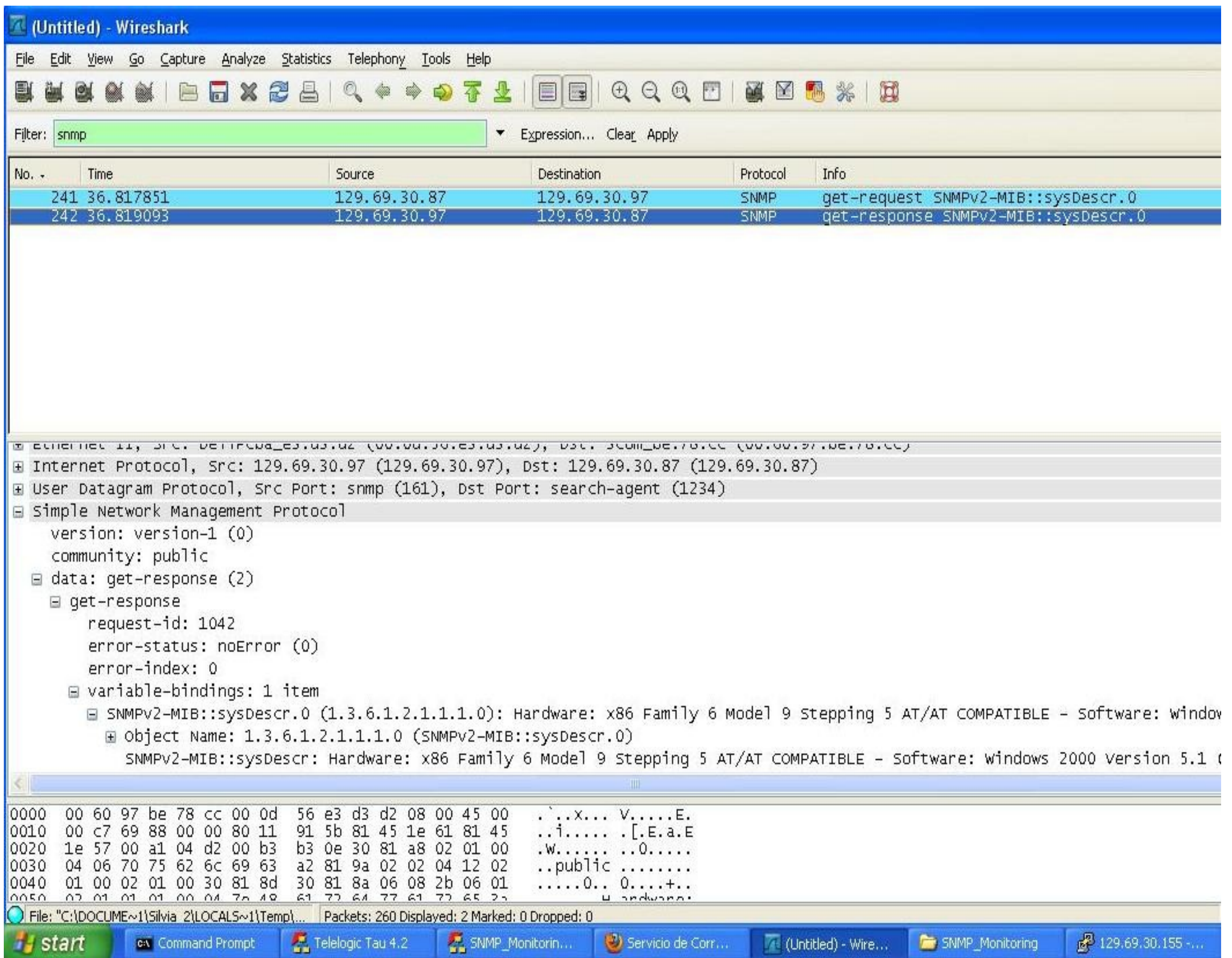


Figure 17: Testing of first implementation (get-response packet)

In both of these screenshots, it can see the SNMP fields explained in previous chapters, such as version, community name and data. The main difference between request and response is in data. In the request, in variable-bindings field, the manager asks for an object identifier (sysDescr.0 – 1.3.6.1.2.1.1.1.0) and the field for the response is empty (it shows the word: “unSpecified”). However, this field in response is filled with the requested information.

4.3. Final implementation: Final Management Model

This chapter presents the development and implementation of the developed management model for real scenario. This model is based in previous model. It adapts functionality of the previous model to the real scenario by adding new classes and diagrams.

This model communicates the manager with the SNMP agents installed in each switch. When the manager receives a signal from the user, it sends the signal to the corresponding agent_adapter, it depends on the requested IP address. There is one agent_adapter for each switch.

In the following chapters it will be explained structure, working scheme and testing of this final model.

4.3.1. Structure

Structure of this model is the same than is the first implementation of the model. But in final model there are more classes that add functionality. There are three packages: snmpModel, snmpFiles and Signals. Their content is very similar than in the previous model.

Picture on the right shows the tree structure of final model.

As seen in the figure, package snmpFiles contains a class diagram with C++ imported types and the C++ class imported to the model, in this case snmpGet. This is the only command used in this model.

Inside snmpModel package there are all main classes and diagrams for the application. The main idea of the structure of the classes are the same than in first implementation: a class Manager, a class SNMP_System that represents monitoring system, a class SNMP_Manager and Agent_Adapters. There are one Agent_Adapter class for each switch. And there are also two more classes called Input and Switch_Selector.

First one serves to store IP address and OID requested. This class Input has two attributes which contain the mentioned data.

Switch_Selector contains an operation called selectSwitch() that receives a number which represent a switch and returns the corresponding IP address.

Functionality of this new classes will be explained in Working scheme chapter.

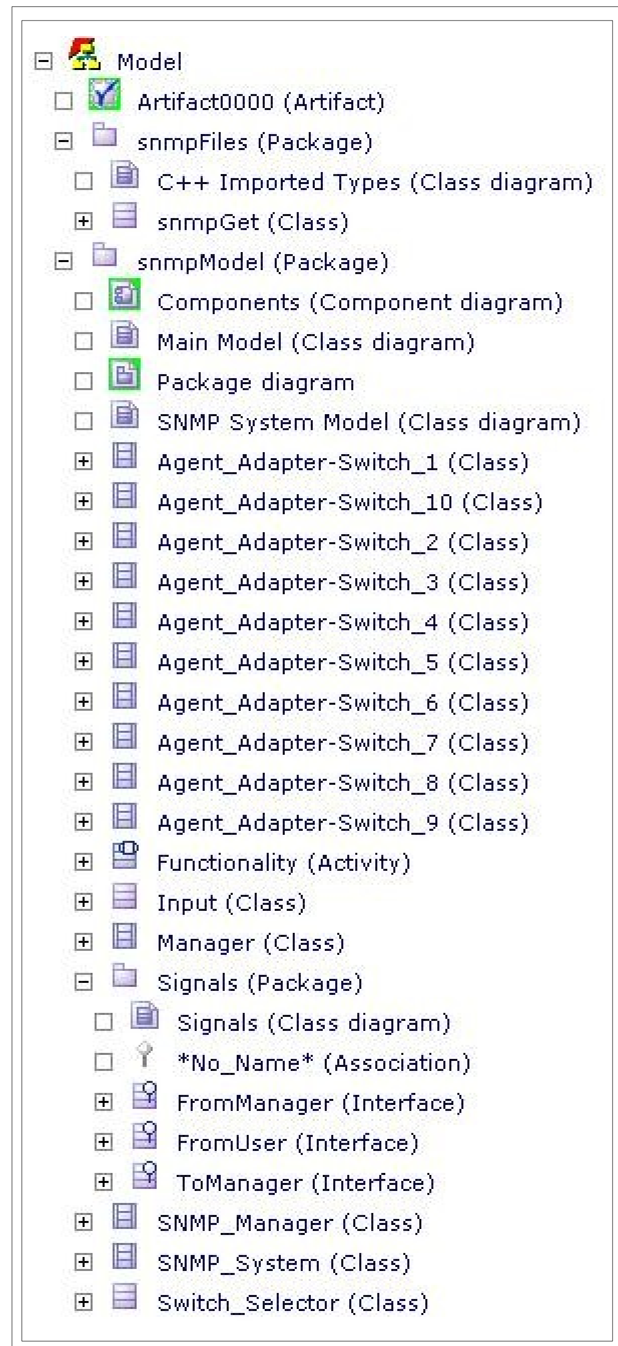


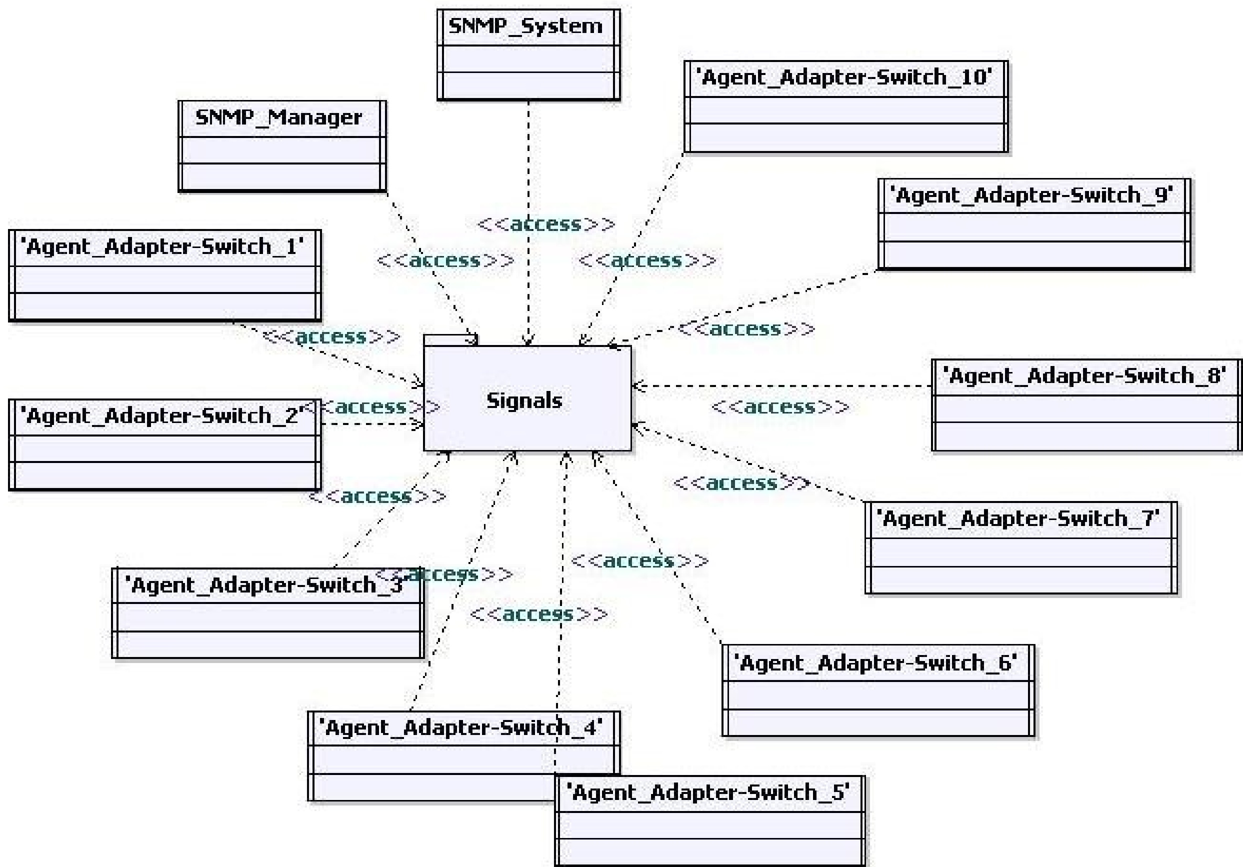
Figure 18: Tree structure of final model

Inside snmpModel it is Signals package. It contains three different interfaces that contain signals to communicate user with manager and manager with agent_adapter.

On the top of the tree there is an artifact. It is an element in the UML model, with properties dedicated to the build process. It is used to translate an UML model into an executable program.

There is an activity class called Functionality. It contains activity diagrams and allows to add more workflows in order to add different monitoring functionality to the model.

Relationships between classes and packages are the same than in first implementation. They are depicted in the following figure:



As figure shows, almost all classes from snmpModel can access to Signals package, except Input and Switch_selector because these are not classes with behavior, they are functional classes.

On the right, it shows the relationship among snmpModel and snmpFiles. It is the same than in the previous model: snmpModel package can access to snmpFiles in order to use C++ classes imported to the model and, thanks to this, can communicate with real world.

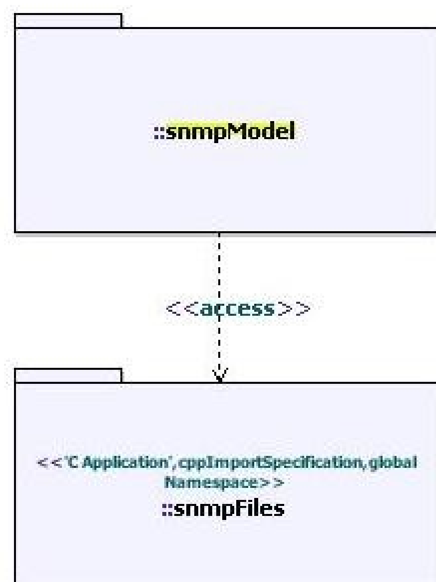


Figure 19: Package structure of final model

Finally, it is necessary to say that all Agent_Adapters are part of SNMP_Manager. This is the best way to manage calls to Agent_Adapters. Inside the model, SNMP_Manager calls to the corresponding Agent_Adapter directly from their attributes.

4.3.2. Working scheme

Final model implements a complete functionality. It seeks to get monitoring information about the spare switches of RUS. For this purpose, user chooses a switch to be monitored and a specific OID, the application sends the request and waits for the reply. When it gets the response, it shows the output. In this model, user manages which switch is monitored.

There are different signals to use between user and SNMP_Manager and between SNMP_Manager and Agent_Adapter:

- Among user and SNMP_Manager there is only one signal called prepare_snmpGet(switch_number, OID) which contains two parameters: switch number to be monitored and requested object identifier.
- In order to communicate SNMP_Manager and Agent_Adapter there are 10 different signals, each one associated with one Agent_Adapter. Their names are snmpGet_switch1(), snmpGet_switch2(), snmpGet_switch3(), and so on, until snmpGet_switch10(). They have only one parameter, an instance of Input class which contains the IP address and the OID requested.

Signals between SNMP_Manager and the corresponding Agent_Adapter are sent as operations implemented in each class Agent_Adapter, because this is the most useful and easier way to implement this functionality and talk with the corresponding Agent_Adapter.

Next figure shows an overview of how it works. It describes the trace diagram of the system, it means the exchange of messages between user, manager and switches.

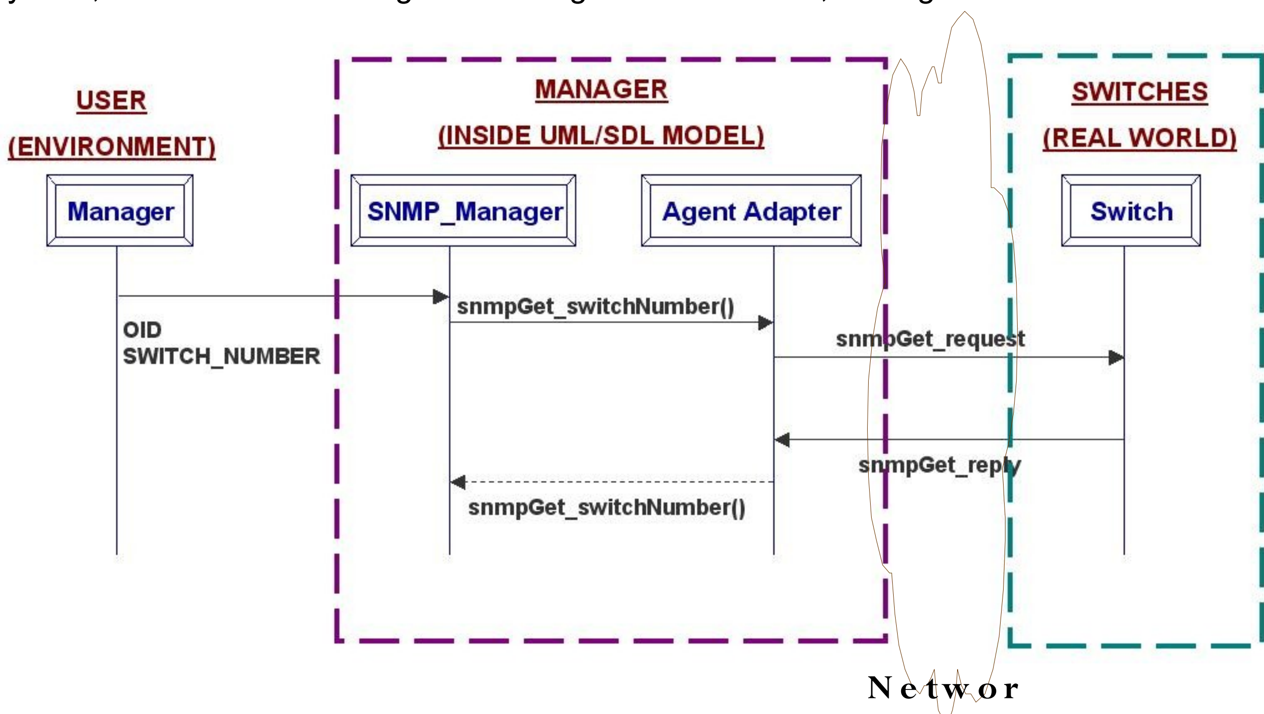


Figure 20: Communication scheme of final implementation

Class SNMP_Manager contains almost all the logic of the system. This class receives input signals from user, and redirects requests to the corresponding Agent_Adapter in order to communicate with the corresponding switch. The behavior of this class is represented in the state machine diagram shown below.

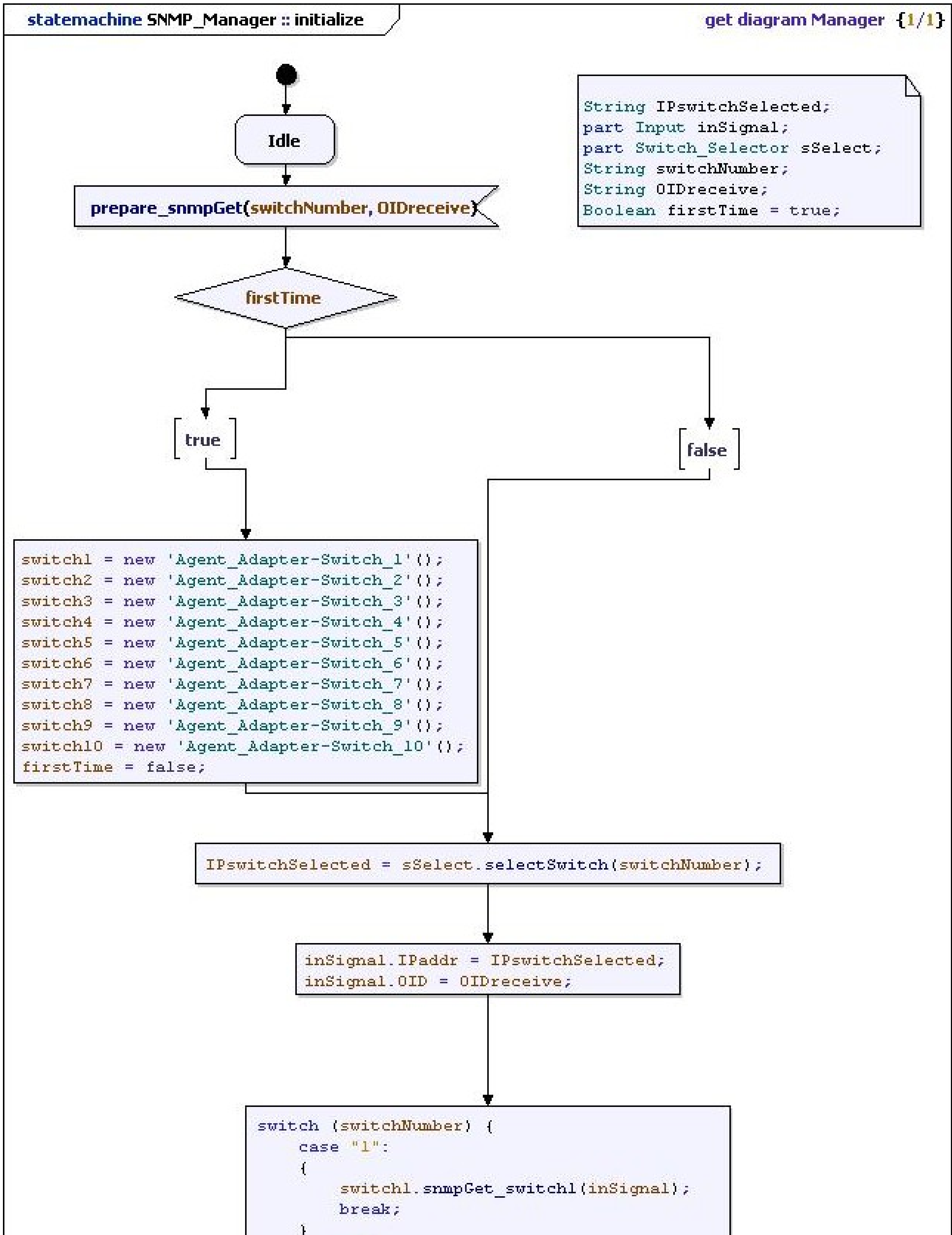


Figure 21: State machine diagram of SNMP_Manager

There are 9 more cases, one for each switch. It depends on the chosen switch, it sends the request and waits for more input signals.

In the next paragraph, it will be explain, step by step, the working scheme of this implementation:

User decides which switch to monitor and sends to the manager a signal prepare_snmpGet() with parameters switch number and object identifier.

SNMP_Manager class receives this signal and calls to selectSwitch() operation in class Switch_Selector. This operation contains a correlation table between switch number and IP address. So, it depends on the received switch number, it returns the corresponding IP address. With this information, SNMP_Manager stores IP address and OID in an instance of Input class.

It depends on the switch number, SNMP_Manager calls the corresponding Agent_Adapter and sends the request with the class Input which contains the stored information.

Agent_Adapter receives this information and communicates with the corresponding switch using the SNMP library. After that, it waits for the response. When it gets the reply signal with the response, it shows the result.

Following figure shows an example of the working scheme. In next chapter, it will show the real results of this example.

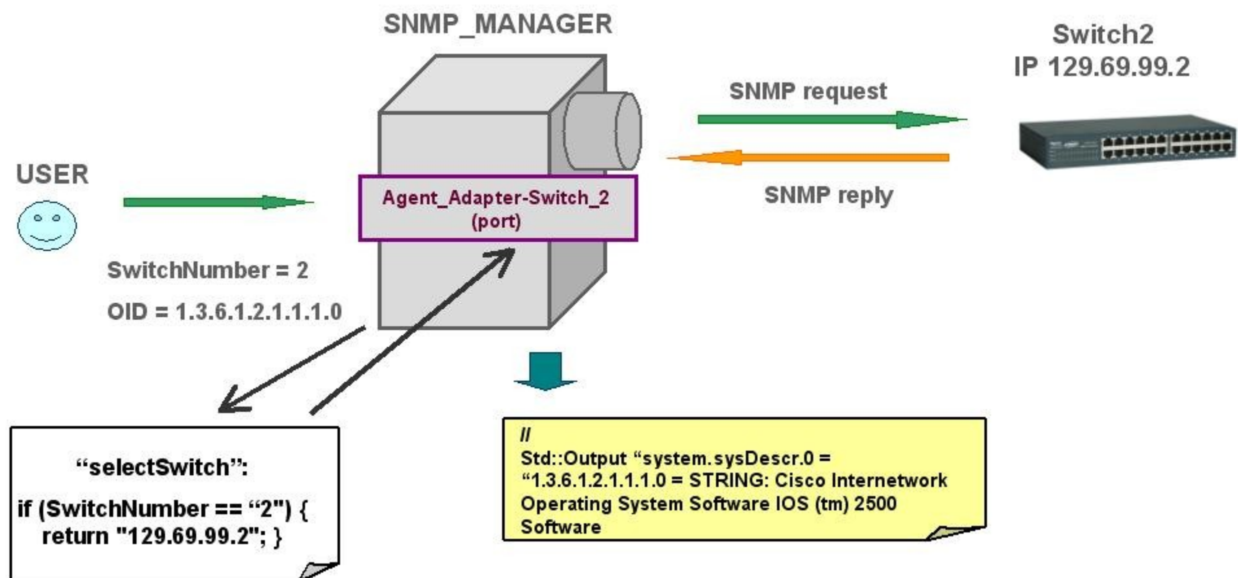


Figure 22: Working scheme of final implementation

4.3.3. Testing

In this chapter, it will show achieved results in the communication inside the model and in the network.

Testing was realized in real scenario, and shown results are a set of screenshots obtained in the communication between manager and switch 2, as in previous example.

This figure shows the exchange of messages from the environment to the manager, and inside the model, from the SNMP_Manager class to the corresponding Agent_Adapter, Agent_Adapter-Switch_2 in this case.

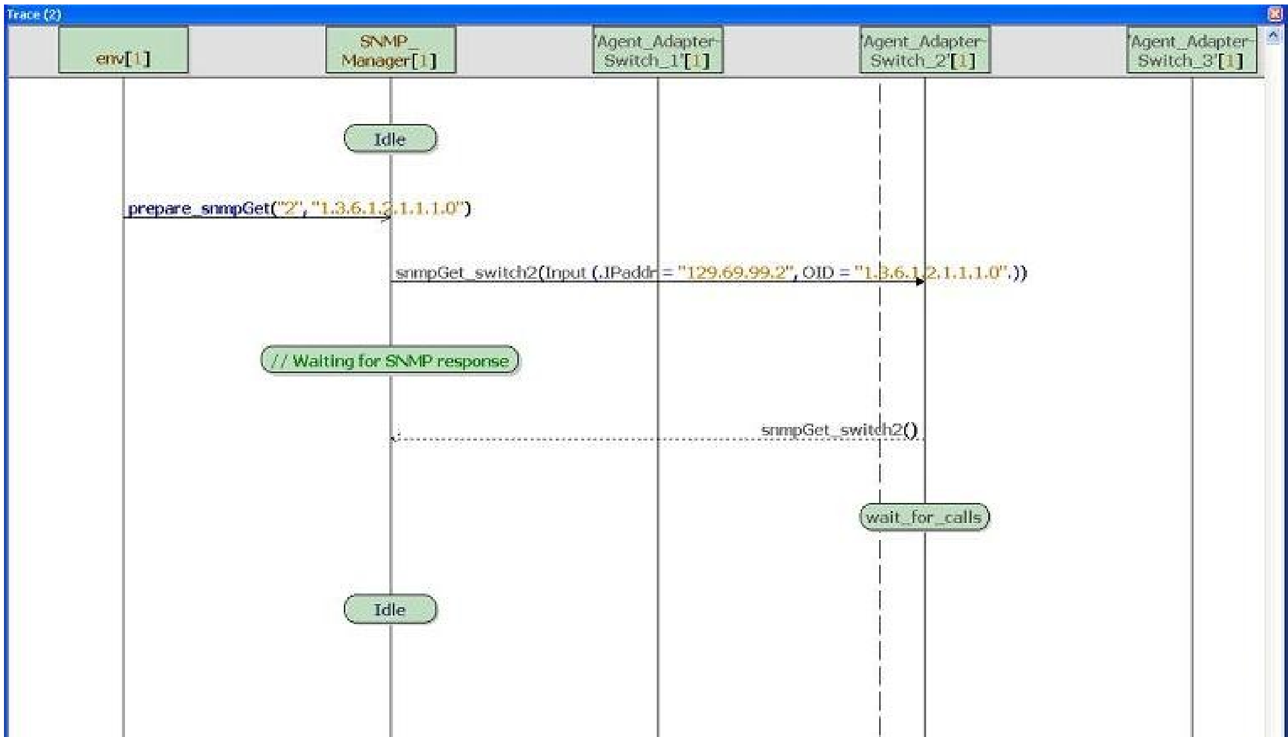


Figure 23: Trace diagram environment and inside model of final implementation

As it shows, signal snmpGet_switch2() from SNMP_Manager to Agent_Adapter-Switch_2 contains an instance of Input class which contains the IP address and the OID.

Next picture shows the exchange of SNMP messages between the model and the SNMP agent installed in the switch. In this case, manager's IP address is 129.69.99.140/24 and 129.69.99.2/24 is the corresponding IP address to switch number 2. Picture is a captured screenshot from Wireshark.

Figure 24: Testing of final implementation (trace diagram: request and reply)

It can highlight the SNMP port in the switch. It is port number 161, as said in SNMP definition, this port is in the agent which listens waiting for SNMP requests.

The next two figures show an example of packets capture in Wireshark between the manager and the agent.

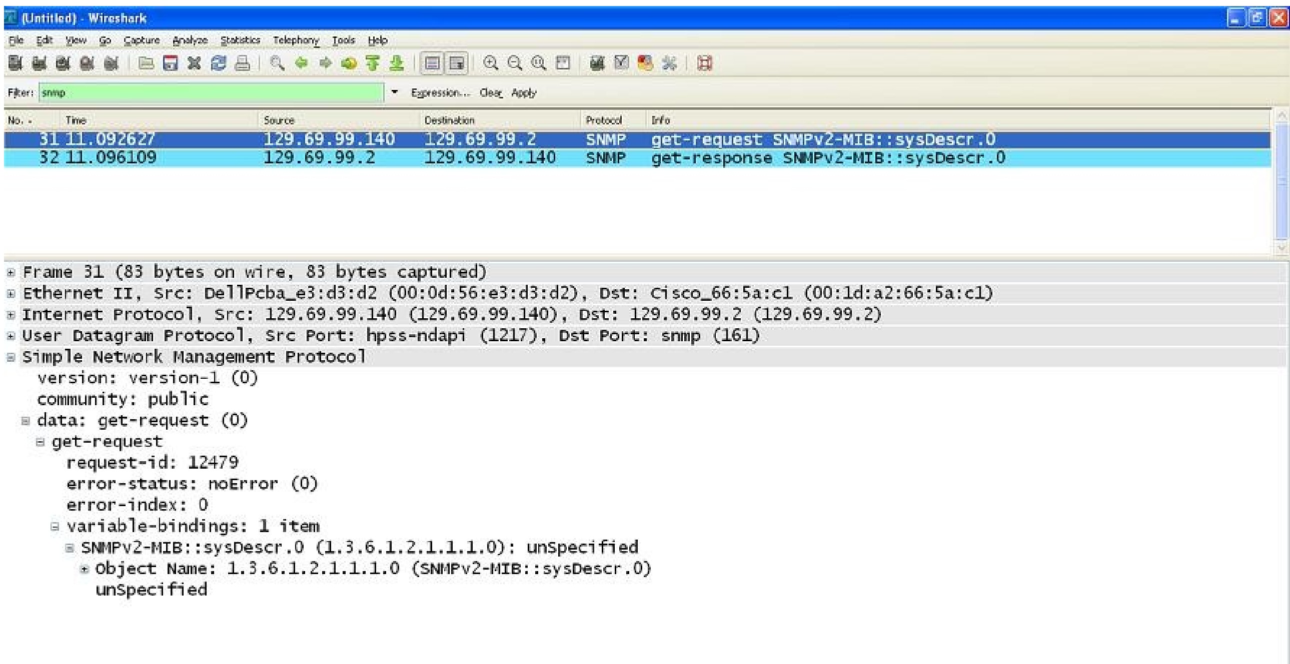


Figure 25: Testing of final implementation (get-request packet)

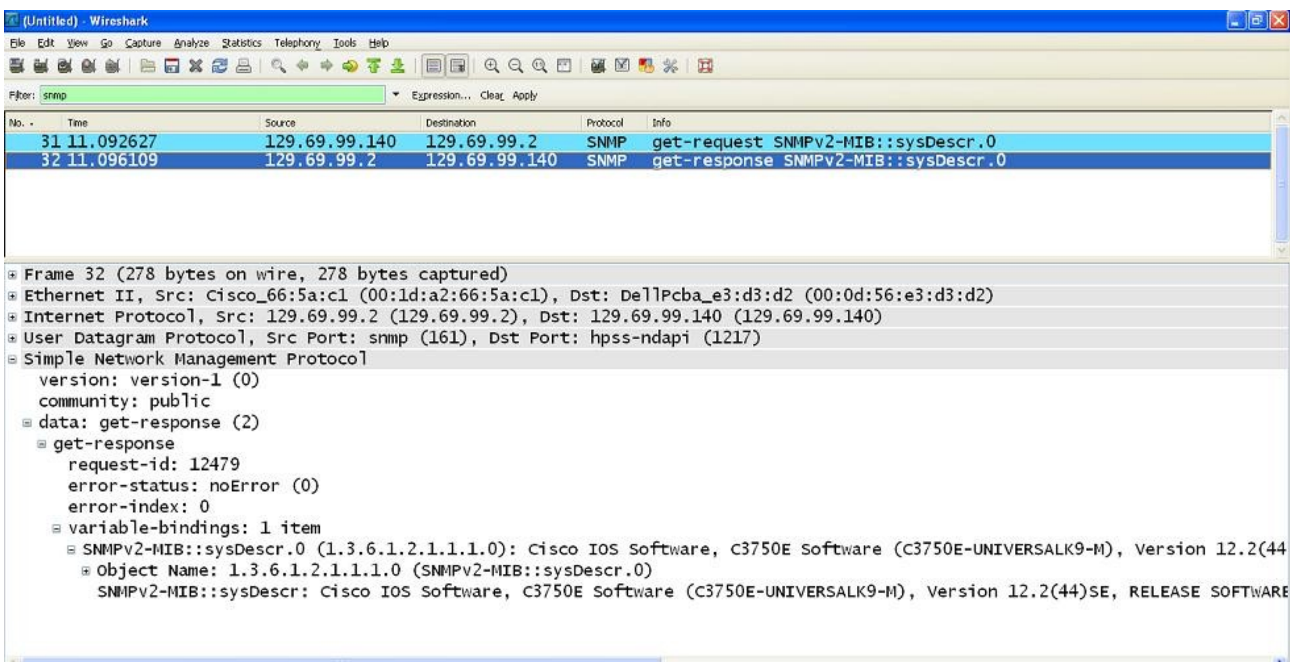


Figure 26: Testing of final implementation (get-response packet)

As in first implementation, fields of SNMP version and community name are the same in both packets, request and response. It is also similar that data field of request contains the requested OID and the response contains the corresponding data associated to the requested object identifier.

5. Conclusions and Outlook

After development of this project, may be said to have achieved all goals agreed at the beginning and to have proved Model-driven Service Management concepts.

- It has achieved to develop an UML model that serves as a network monitor.
- It has achieved total connectivity between developed model and real world.
- It has demonstrated the enhancement of usability and handling of system when to use models.
- It has demonstrated that an UML model can communicate with network devices.
- It has achieved to integrate the existing SNMP library in TAU and use it in order to connect the UML model to the SNMP agents installed in the spare switches.
- It has demonstrated simplicity notation of test cases and facility to add new workflows to the model.

At this point, additions and improvements to this management model are proposed.

- Create a graphical user interface in order to make easier and more understandable handling of monitoring system.
- Improve error handling of the model. At this moment, there is a simple error handling: the application sends the request from the agent adapter to the switch a maximum of two times, if the SNMP agent of the switch does not respond, the agent adapter returns back control to the SNMP_Manager class.
- Develop business applications. An advantage of using Model-driven Management is the facility to add new workflows to the implemented model. So, it is proposed to develop large workflows that could handle different aspects of network management.

Appendix A: Index of figures

Figure 1: General scheme of the network operated by RUS, page 1
Figure 2: Model@runtime schema, page 3
Figure 3: General schema of the project's development, page 3
Figure 4: Schema of test scenario, page 4
Figure 5: Logical schema of real scenario, page 5
Figure 6: Physical plan of real scenario, page 6
Figure 7: MIB structure, page 10
Figure 8: SNMP version 1 (SNMPv1) General Message Format, page 12
Figure 9: UML class diagram, page 15
Figure 10: Telelogic Tau user interface, page 17
Figure 11: Schema of concrete work, page 21
Figure 12: Example workflow, page 22
Figure 13: Package structure of first implementation, page 23
Figure 14: Communication scheme of first implementation, page 24
Figure 15: Testing of first implementation (trace diagram: request and reply), page 25
Figure 16: Testing of first implementation (get-request packet), page 25
Figure 17: Testing of first implementation (get-response packet), page 26
Figure 18: Tree structure of final model, page 27
Figure 19: Package structure of final model, page 28
Figure 20: Communication scheme of final implementation, page 29
Figure 21: State machine diagram of SNMP_Manager, page 30
Figure 22: Working scheme of final implementation, page 31
Figure 23: Trace diagram environment and inside model of final implementation, page 32
Figure 24: Testing of final implementation (trace diagram: request and reply), page 32
Figure 25: Testing of final implementation (get-request packet), page 33
Figure 26: Testing of final implementation (get-response packet), page 33

Appendix B: Glossary

API – Application Programming Interface

ASN.1 – Abstract Syntax Notation One

IAB – Internet Architecture Board

IETF – Internet Engineering Task Force

IIS – Institut für Intelligente Systeme

IITS – Institut für IT-Services

IKR – Institut für Kommunikationsnetze und Rechnersysteme

IP – Internet Protocol

IPX – Internet Packet Exchange

MDA – Model-driven Architecture

MDE – Model-driven Engineering

MIB – Management Information Base

NMS – Network Management System

OID – Object Identifier

OMG – Object Management Group

PDU – Protocol Data Unit

RFC – Request for Comments

RUS – Rechenzentrum Universität Stuttgart

SDL – Specification and Description Language

SMI – Structure of Management Information

SNMP – Simple Network Management Protocol

SNMPv1 – Simple Network Management Protocol Version 1

SNMPv2 – Simple Network Management Protocol Version 2

SNMPv3 – Simple Network Management Protocol Version 3

SOA – Service Oriented Architecture

TCP – Transmission Control Protocol

UDP – User Datagram Protocol

UML – Unified Modeling Language

Appendix C: Bibliography

UML:

- http://www.elguille.info/colabora/puntoNET/canchala_UML.htm
- http://en.wikipedia.org/wiki/Unified_Modeling_Language
- UML Distilled (Applying the Standard Object Modeling Language) [Martin Fowler with Kendall Scott, ED. Booch Jacobson Rumbaugh (Addison-Wesley)]
- http://www.sparxsystems.com/resources/uml2_tutorial/index.html

SOA:

- http://en.wikipedia.org/wiki/Service-oriented_architecture
- http://es.wikipedia.org/wiki/Arquitectura_orientada_a_servicios

SDL:

- http://en.wikipedia.org/wiki/Specification_and_Description_Language

Model@runtime:

- http://www.comp.lancs.ac.uk/~bencomo/MRT/#dates

MDA:

- http://www.sparxsystems.com/platforms/mda_tool.html?source=google&campaign=3&group=1&creative=2&wcw=google&gclid=CO7N1aGEpJ8CFYQU4wodlQ-YYw
- http://es.wikipedia.org/wiki/Model_Driven_Architecture
- http://en.wikipedia.org/wiki/Model-driven_engineering
- http://www.uclm.es/dep/tsi/pdf/UCLM-TSI-002.pdf
- http://portal.modeldriven.org/
- http://www.omg.org/mda/

SNMP:

- http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol
- Total SNMP (Exploring the Simple Network Management Protocol) Second Edition [ED. Prentice Hall]
- http://es.wikipedia.org/wiki/Simple_Network_Management_Protocol
- http://neutron.ing.ucv.ve/revista-e/No6/Briceño%20Maria/SNMPv3.html
- http://www.webnms.com/snmputilities/help/quick_tour/snmp_and_mib/snmpmib_versionssnmp.html
- http://www.tcpipguide.com/free/index.htm
- http://www.samanage.com/blog/2009/09/the-importance-of-network-management-software/
- http://www.ciscopress.com/bookstore/product.asp?isbn=1587201372
- http://www.researchandmarkets.com/reportinfo.asp?report_id=1095741

SNMP++:

- http://oss.org.cn/man/develop/snmp_pp/index.htm
- http://www.agentpp.com/snmp_pp3_x/snmp_pp3_x.html
- http://rosegarden.external.hp.com/snmp++
- http://www.hpl.hp.com/hpjournal/97apr/apr97a11.pdf

Wireshark:

- http://www.wireshark.org/
- http://en.wikipedia.org/wiki/Wireshark

CommView:

- http://www.tamos.com/products/commview/

Telelogic TAU:

- http://en.wikipedia.org/wiki/Telelogic_TAU
- http://publib.boulder.ibm.com/infocenter/rsdvhelp/v6r0m1/index.jsp?topic=/com.ibm.xtools.modeler.doc/topics/ccompasn.html

References in this document:

[1] http://www.samanage.com/blog/2009/09/the-importance-of-network-management-software/
[2] Document IEEE 2009 Models@run.time
[3] Document IEEE 2009 Models@run.time

URLs from pictures downloaded of Internet:

- http://www.taxi.com.hk/sshop/images/computer.jpg
- http://gizmologia.com/files/2009/04/hp-mediasmart-server-lx190.jpg
- http://www.mobilewhack.com/images/toshiba_satellite_a105_s4284_laptop.jpg
- http://nirewiki.com/wikisdir/wakon/files/switch.jpg
- http://documentation.softwareag.com/crossvision/eli/smhAgents/graphics/exx_oid1.png
- http://www.tcpipguide.com/free/diagrams/snmpv1format.png
- http://www.valenciablog.com/wp-content/uploads/2008/09/apple_imac_g5.jpg
- http://upload.wikimedia.org/wikipedia/commons/d/d6/Uml_diagram2.png
- http://www.scisrl.com.ar/images/router_linksys_wrt54g_wireless.jpg
- http://www.ahiva.info/Colorear/Clima/Nubes/Nube-01.gif
- http://www.necel.com/micro/ja/partner/images/TAU_Generation2_thumb.png
- http://www.gan-polarcon.de/USt_logo3_07_klein.jpg