Informe de Proyecto

Gestión de Inventario de Red
Marzo de 2004

Miguel Bartesaghi
Bernardo Fagalde
Federico Zubía

1. Introducción y objetivos del Proyecto

A lo largo de los años las redes se han vuelto mucho más **grandes**, populares y tienen mayores requerimientos de **calidad**. Los mundos de las redes empresariales, telecomunicaciones e Internet están más **interconectados** día a día, provocando que haya más **heterogeneidad** de tecnologías, fabricantes y protocolos. Además de aumentar los requerimientos de calidad, en ciertas aplicaciones se necesita asegurar ciertos niveles de servicio y se hace necesario entonces definición de acuerdos de nivel de servicio (service-level agreements, SLA).

Todo esto provoca que las redes sean cada vez más complejas de administrar y sea más importante su gestión.

Este proyecto entonces se enmarca en la **gestión de redes**. Nuestro proyecto se desarrolla en simultaneo con otro proyecto y entre ambos se pretende lograr un desarrollo de la **función de configuración** a través de un módulo de **aprovisionamiento**, hasta el nivel del elemento de red. Estos proyectos pretenden ser los primeros de una serie que tiendan a completar un sistema de gestión que permite estudiar los temas relacionados a esta área.

El componente principal a desarrollar y que da nombre a nuestro proyecto es el **Inventario de Red**. Este consiste en un repositorio de objetos gestionados al nivel de capa de red que brinda múltiples servicios.

Como mencionamos existen múltiples tecnologías y múltiples fabricantes, por lo cual, se tiene como segundo objetivo la implementación de un componente que permite comunicar con los dispositivos reales, de manera **independiente a la tecnología** subyacente.

También es importante resaltar la necesidad de una implementación basada en el paradigma de **desarrollo de componentes**, con el fin de facilitar la mejora y agregado de nuevos componentes en futuros trabajos.

Este informe consta de tres grandes partes. Primero el estudio del estado del arte que permitió introducirse en el tema de gestión de redes e investigar entre los desarrollos existentes. La segunda parte trata sobre la implementación, que decisiones hubo que tomar y los motivos de las mismas. Por último veremos las conclusiones y futuros trabajos.

1.1. Tabla de Contenido

1.2. Tabla de Ilustraciones

PRIMERA PARTE: Estado del Arte

2. Gestión de Redes

Cuando se habla de gestión de redes se piensa en la inicialización, monitorización y modificación de las funciones de una red. Primero se deberá inicializar (configurar) y luego monitorizar la red por ejemplo para detectar desperfectos o cambios en el tráfico.

2.1. Evolución

Las redes empresariales se componen de muchos dispositivos, algunos de los cuales son críticos para el funcionamiento de la misma, mientras que otros son fundamentales para los servicios que la empresa presta.

En el pasado, no había ninguna norma y cada compañía tenía que crear su propio proceso de gestión de red. En empresas con muchos y variados dispositivos, había que configurar y mantener cada uno. Esto requería una gran cantidad de mano de obra, además de costar mucho dinero. [1]

Las estrategias de gestión de red fueron creadas como una manera más razonable de gestionar una red grande. El objetivo de la gestión de red entonces, es mejorar las capacidades de la red y hacer más fluido su mantenimiento. Pero la falta de una plataforma de gestión consistente significó que los administradores de la red tenían que aprender muchas interfaces de equipos diferentes y gastar mucho tiempo con cada dispositivo. Para complicarlo aún más, cada proveedor creó su propia forma para agregar opciones de gestión a sus equipos.

A lo largo de los años varias organizaciones han dedicado esfuerzos para el desarrollo de servicios, protocolo y arquitecturas para la gestión de redes. Existieron tres grupos de interés en este sentido: Redes de Computadoras (OSI), Redes de Telecomunicaciones (ITU-T) e Internet (IETF).

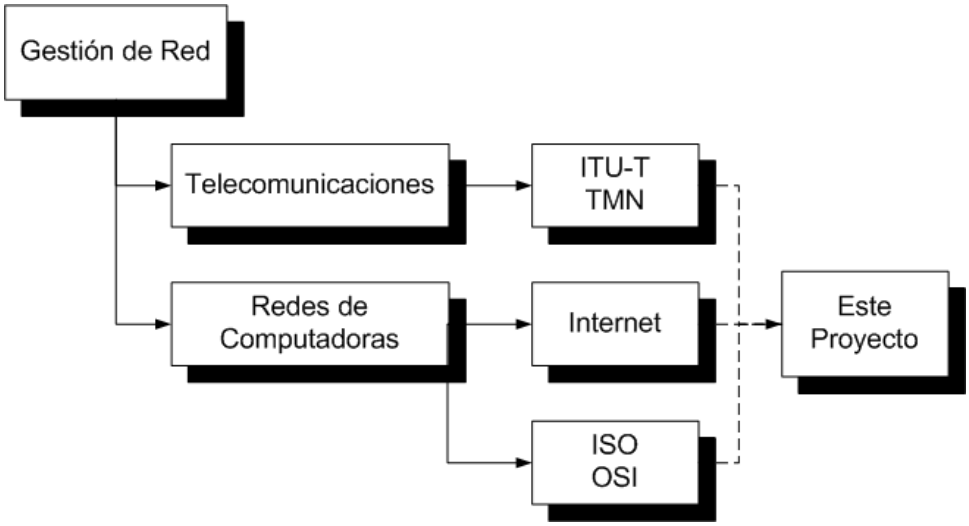


Figura 2-1 - Grupos de Interés de la gestión de red

La Organización Internacional para la Estandarización (ISO) fue la primera en comenzar en 1980 con su modelo OSI, la cual creó un marco de trabajo “OSI System Management Framework”, una visión “OSI System Management Overview” y un protocolo de gestión “Common Management Information Protocol” (CMIP).

Luego en 1985 el “Comité Consultative Internationale de Telegraphique et Telephonique” (CCITT) hoy en día llamado Unión de la Telecomunicación Internacional (ITU) empezó a desarrollar estándares conocidos como las recomendaciones “Telecommunication Management Network” (TMN).

Con el desarrollo de la Internet la “Internet Engineering Task Force” (IETF) desarrolló su protocolo de gestión de redes “Simple Network Management Protocol” (SNMP). Muchos fabricantes empezaron a producir sistemas que soportan SNMP. Y a pesar de varias deficiencias, SNMP se convirtió en un estándar de hecho. En 1993 apareció una versión mejorada, la SNMPv2 y hoy ya existe la versión 3.

Otras organizaciones han dedicado esfuerzos a la gestión de red, tales como el “TeleManagement Forum” (TM Forum o TMF). El TMF es un consorcio internacional de proveedores de servicio de comunicaciones y empresas afines. Su misión es ayudar a los proveedores de servicio y operadores de red a automatizar sus procesos comerciales, de manera efectiva en costo y tiempo.

Todas las actividades del TM Forum se concentran en facilitar la búsqueda de soluciones comunes a las necesidades operacionales, relacionadas con la industria de la información y telecomunicaciones.

El TMF funciona desde 1988 y originalmente usaba el nombre “Foro de Gestión de Redes” (NMF). Hoy posee más de 340 compañías miembro y comprende proveedores de servicio, informática, equipos de red, solución de software y clientes de servicios de comunicaciones. Entre éstas se incluyen Alcatel, AT&T, Fujitsu, Hitatchi, Lucent Technologies, Marconi Communications, Nortel Networks, Siemens AG, Telcordia Technolgies, Tellabs.

Su trabajo esta fuertemente influenciado por la ITU-T G.805 y las recomendaciones M.3100, ambos de la ITU.

2.2. Utilidades

Algunas de las razones por las que necesitamos gestionar las redes son:

1. Reducción de costos: A veces en una organización se tienen diferentes necesidades de funcionamiento de la red. Si se tuviera una única red multipropósito y se gestionara para ajustarle los parámetros necesarios, de acuerdo los requerimientos de los distintos usuarios, entonces sería más barato que sobredimensionarla o tener más de una.
2. Falta de experiencia: Cuando se diseña una red no es posible prever todas las situaciones potenciales. Por ejemplo, pueden ocurrir eventos imprevistos o situaciones no contempladas a priori por los diseñadores de la red, que desaten una congestión.
3. Manejo de Fallos: Siempre es posible que ocurran desperfectos en los elementos de la red. El hecho de monitorizar y comparar el funcionamiento de la red permite detectar desperfectos en la misma.
4. Flexibilidad: A lo largo de la operación de una red nuevos requerimientos pueden surgir, por lo que es muy importante poder adaptarse a los cambios sin mayor impacto.

2.3. OSS

Las tareas de administrar un conjunto de servicios y la infraestructura en la que ellos se basan es uno de los desafíos mayores de un Operador de Red. Involucra el reclutamiento de un buen grupo de personas, organizando el trabajo en los procesos eficaces e infraestructura de los sistemas de apoyo. [2]

Los problemas, soluciones y principios que aplican a esta área los llamaremos “Operations Support Systems” (OSS). Éstos a menudo también son llamados “Support Systems, Operations & Maintenance”, OSS/BSS o “Telecom Management”.

2.4. Modelos de Gestión de Redes

Como hemos visto, la evolución originó que hoy existan tres grandes modelos de gestión. El modelo ISO, el modelo Internet y el modelo TMN. [3]

El modelo de gestión de redes ISO esta muy acoplado a las estructuras del modelo OSI, es muy ambicioso y con muchas funcionalidades, pero por otro lado es muy complejo y pesado. La falta de productos comerciales hizo necesario buscar una forma alternativa de gestionar las computadoras y se usó el modelo Internet. Éste es poco hábil al momento de manejar redes heterogenias como OSI, aunque de todas maneras su funcionalidad se considera adecuada para muchas de las redes reales.

Finalmente en el área especifica de las redes de computadoras la ITU-T aprobó el modelo TMN beneficiándose de la mayoría de los conceptos del estándar del modelo ISO.

2.4.1. El Modelo ISO

La gestión de redes de ISO está basada en tres elementos básicos: la estructura de la información gestionada, el protocolo y las funciones.

La estructura de la información gestionada está basada en un modelo orientado a objetos donde cada dispositivo está asociado a una clase particular. Los objetos están organizados de acuerdo a un árbol de dependencias jerárquicas. El conjunto de objetos gestionados construyen la "Management Information Base" (MIB) y resulta en un modelo con gran capacidad descriptiva y apto para lidiar con situación particular que puede ocurrir en redes reales.

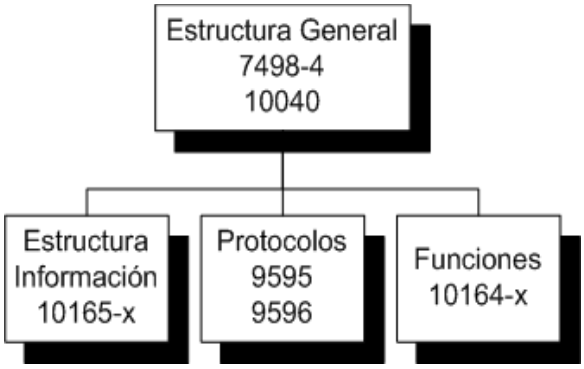


Figura 2-2 - Modelo ISO

Entre los protocolos que define la ISO está "Common Management Information Protocol" (CMIP) que puede soportar las complejas estructuras de la MIB y permite la gestión en ambientes distribuidos.

El elemento básico de la gestión de red del modelo ISO son las funciones. Las funciones de gestión se dividen en cinco áreas que son fallas, configuración, contabilidad, prestaciones y seguridad. Estas áreas funcionales se detallaran más adelante.

2.4.2. El Modelo Internet

El modelo de Internet esta basado en un pequeño conjunto simple de protocolos y estructuras de información. La estructura de información sigue el modelo orientado a objetos pero introduciendo muchas simplificaciones con respecto al modelo ISO. Esto hace que el modelado de dispositivos complejos sea más difícil, pero por otro lado hace que las aplicaciones sean considerablemente más fáciles.

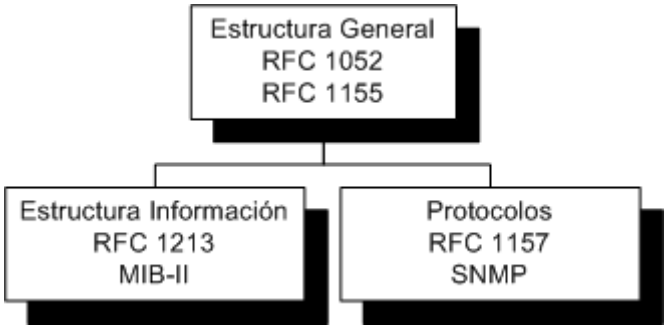


Figura 2-3 - Modelo Internet

El protocolo de gestión de Internet es SNMP, el cual se ubica sobre el stack TCP/IP. Su diseño esta pensado para ser simple y es capaz de manejar propiedades de muchos dispositivos pero tiene limitaciones para manejar objetos complejos o estructurados.

Las limitaciones del modelo son al mismo tiempo sus ventajas e inconvenientes. Tiene limitaciones para gestionar redes, pero por otro lado permite un fácil desarrollo de agentes y gestores. Esto ha provocado una rápida expansión y una baja en los costos. Actualmente existen muchos dispositivos que soportan SNMP.

2.4.3. El Modelo TMN

El modelo de gestión de la ITU-T llamado formalmente TMN, esta basado en el modelo OSI y también se estructura en tres elementos: estructura de información de gestión, protocolos y funciones. Usando el núcleo de ISO, el modelo TMN se expandió en dos direcciones:

1. Expandir el concepto y funciones para amoldar las características de las redes de telecomunicación.
2. Establecer un conjunto de estándares específicos para cada tecnología en particular. Por ejemplo, hay estándares para manejar las tecnologías de SDH, ISDN, ATM, etc.

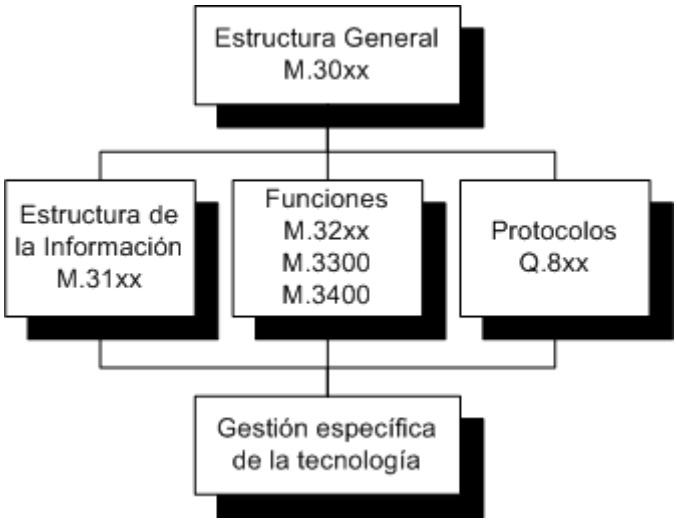


Figura 2-4 - Modelo TMN

Las funciones en el modelo TMN también están organizadas en cinco funciones, pero cada una de estas funciones está estructurada en cuatro capas: gestión de elementos, gestión de red, gestión de servicio y gestión del negocio. Más adelante se describirán cada una de estas capas cuando se analice la pirámide TMN.

2.5. Áreas Funcionales

Tanto en el modelo de gestión de ISO como el de TMN se hace una división en cinco áreas funcionales, estas son conocidas como FCAPS: Fallas (**F**ault), Configuración (**C**onfiguration), Contabilidad (**A**ccounting), Desempeño (**P**erformance), y Seguridad (**S**ecurity).

A continuación describiremos brevemente cada una de ellas. [4]

2.5.1. Gestión de Fallos

La gestión de fallos incluye una manera estandarizada para supervisar los dispositivos de la red y generar una alarma cuando algo sale mal.

El software de gestión de red puede configurarse para escuchar un evento específico de la red conocido como "trap" o excepción. Cuando un "trap" ocurre, activa una alarma para alertar a los administradores para que el problema pueda corregirse.

En la mediada de lo posible se pretende corregir automáticamente el problema y mantener la red corriendo en forma eficiente.

La gestión de fallos primero determina los síntomas y aísla el problema. Luego el problema es corregido y la solución es probada en todos los subsistemas importantes. Finalmente, se graban la detección y resolución del problema.

2.5.2. Gestión de Configuración

El software de gestión se comunica con los dispositivos gestionados (ME) y lee su base de información para descubrir detalles sobre su configuración. El software de gestión puede salvar la información para el futuro análisis o cambiarla de manera que el dispositivo funcione de acuerdo a su nueva configuración.

2.5.3. Gestión Contable

El enfoque principal de la gestión contable es monitorizar y almacenar el uso de los componentes de red. Estos datos se utilizan para análisis de tráfico y carga en una cuenta para el cobro del servicio.

2.5.4. Gestión de Desempeño

La gestión de desempeño es el proceso de recolección y análisis de datos de los componentes de la red, para supervisar el desempeño actual y planear las necesidades actuales y el crecimiento futuro.

Primero se deben decidir que variables son de interés para los administradores. Luego se deben establecer los valores normales. Finalmente se determinan los umbrales para cada variable importante. De esta manera si alguno es excedido, se indique que hay un problema en la red.

Las entidades gestionadas están continuamente monitorizando las variables de desempeño y cuando una de éstas es excedida se genera una alerta que es enviada al sistema de gestión (NMS).

La gestión de desempeño también permite métodos proactivos, por ejemplo la simulación de redes puede ser utilizada para proyectar cómo el crecimiento de la red afectará las métricas de desempeño. Esta simulación puede alertar a los administradores de problemas inminentes para que puedan tomarse medidas correctivas.

2.5.5. Gestión de Seguridad

El objetivo de la gestión de seguridad es controlar el acceso a los recursos de la red según las políticas, para que la red no pueda ser saboteada, y qué información sensible no pueda ser accedida o modificada sin la debida autorización.

Los subsistemas de seguridad realizan varias tareas: identificar los recursos sensibles de la red y determinar mapeos entre estos recursos y la configuración de cada usuario. También monitorizan los puntos de acceso a los recursos sensibles y registran accesos inapropiados a éstos.

2.6. Pirámide TMN

Hoy día, el modelo TMN se ha aceptado como una manera lógica de estructurar las actividades que los operadores de telecomunicaciones necesitan realizar en sus redes.

El modelo de TMN se organiza de manera jerárquica en cuatro niveles, que son: negocio, servicio, red y elemento según se muestra en la siguiente figura.

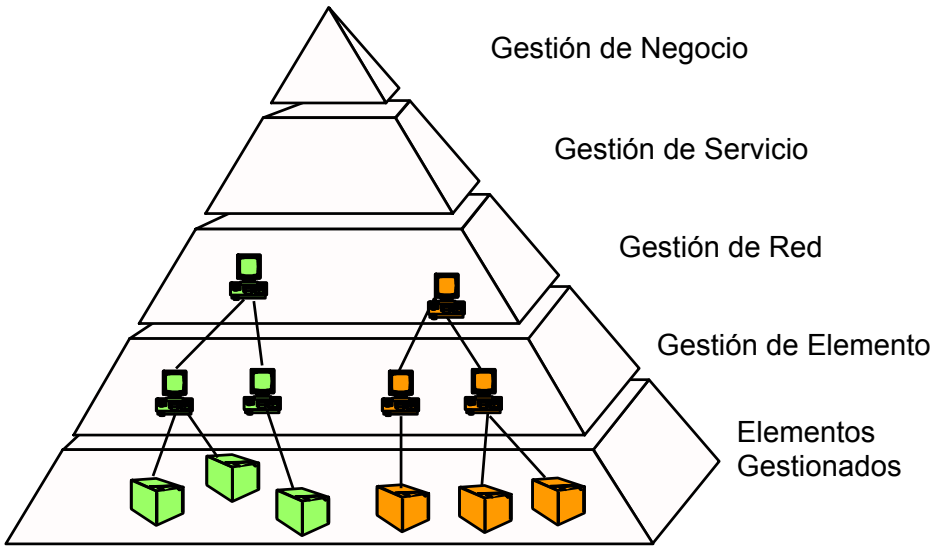


Figura 2-5 - Pirámide TMN

Se describirá brevemente las capas de la pirámide pues en sucesivos capítulos profundizaremos en las capas principales: [2]

Elementos de Red

En la base de la pirámide se encuentran los elementos gestionados, que son todos los equipos conectados en una red de computadoras que se quiere gestionar.

Gestión de Elementos

Esta capa actúa directamente sobre los elementos gestionados y posee la funcionalidad de gestión que es requerida para operar una sola pieza de equipo independientemente.

Gestión de Red

Cuando múltiples elementos de red interconectados forman una red, una conexión punto a punto, o una llamada telefónica, usan un conjunto de recursos de la red. La capa de gestión de red contiene la funcionalidad requerida para controlar la red, integrando las funciones de capa de elemento.

Gestión de Servicios

Una red proporciona servicios. Una suscripción a una línea arrendada, una cuenta de correo y una suscripción del teléfono son ejemplos de estos servicios. La gestión de Servicio se refiere al control de éstos.

Gestión de Negocio

Los servicios se proporcionan a los suscriptores (clientes). La gestión de cliente y los problemas relacionados tales como el cobro a los mismos, se les llama Gestión de Negocio.

2.7. Pirámide TMN y FCAPS

Nótese que el modelo FCAPS y la pirámide TMN son complementarios, como se puede observar en la figura. Las áreas funcionales son aplicables a todas las capas.

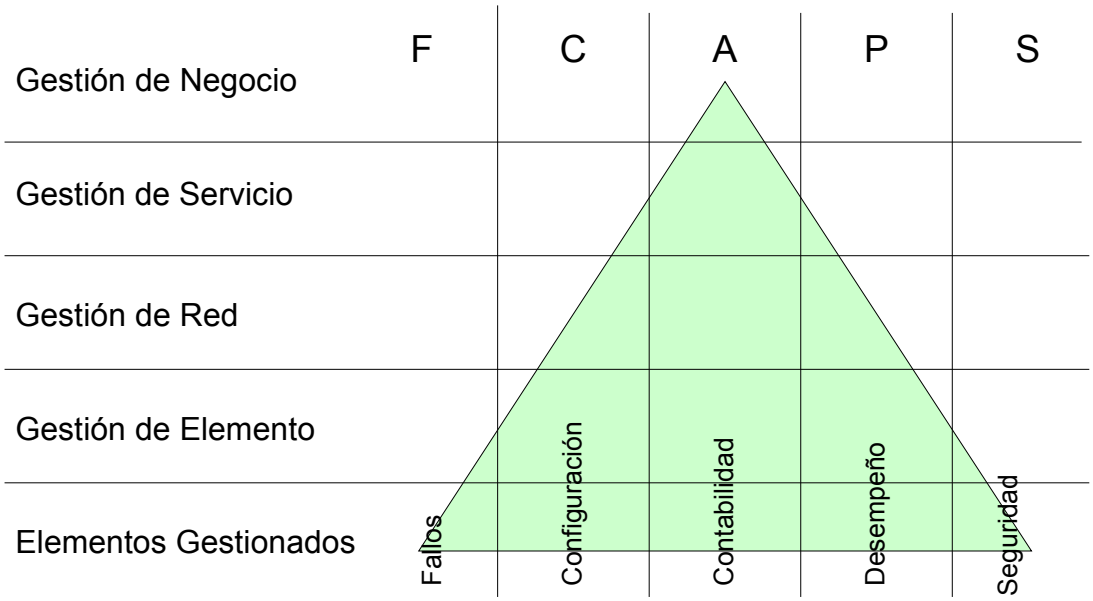


Figura 2-6 - Pirámide TMN y FCAPS

Ejemplo: una condición de error en un puerto físico (capa de elemento de red) es una interrupción de servicio (capa de gestión de servicio) qué es un informe de problema de cliente y posiblemente un descuento en la próxima factura (capa de gestión de negocio). [2]

2.8. Mapa de Operaciones de Telecomunicaciones (TOM)

TMF, vía su visión de los procesos, el “Telecom Operations Map” (TOM), contiene una descripción detallada de los procesos más importantes involucrados en la ejecución de operaciones de un operador de red. [5] [6]

El TOM provee una dirección de alto nivel como modelo común para los procesos de las operaciones de comunicaciones. Mantiene un lenguaje común y una estructura general para el apoyo e implementación de la integración de las operaciones punto a punto y su automatización.

El TOM usa las capas del modelo TMN de la ITU-T para organizar los procesos de negocio, pero divide la capa de Gestión de Servicio en dos partes: Procesos de Atención al Cliente y Procesos de Desarrollo de Servicio y Operaciones. En un sentido simple, esta división refleja la diferencia entre los procesos activados por necesidades individuales de clientes, de aquéllas aplicadas a un grupo de clientes subscriptos a un servicio o familia de servicios. Además refleja la contabilidad para el manejo del contacto directo con el cliente en Procesos de Atención al Cliente y la necesidad crítica en la integración, automatización y soporte de los procesos de atención al cliente. [4]

En la siguiente figura se presenta gráficamente el TOM con sus procesos.

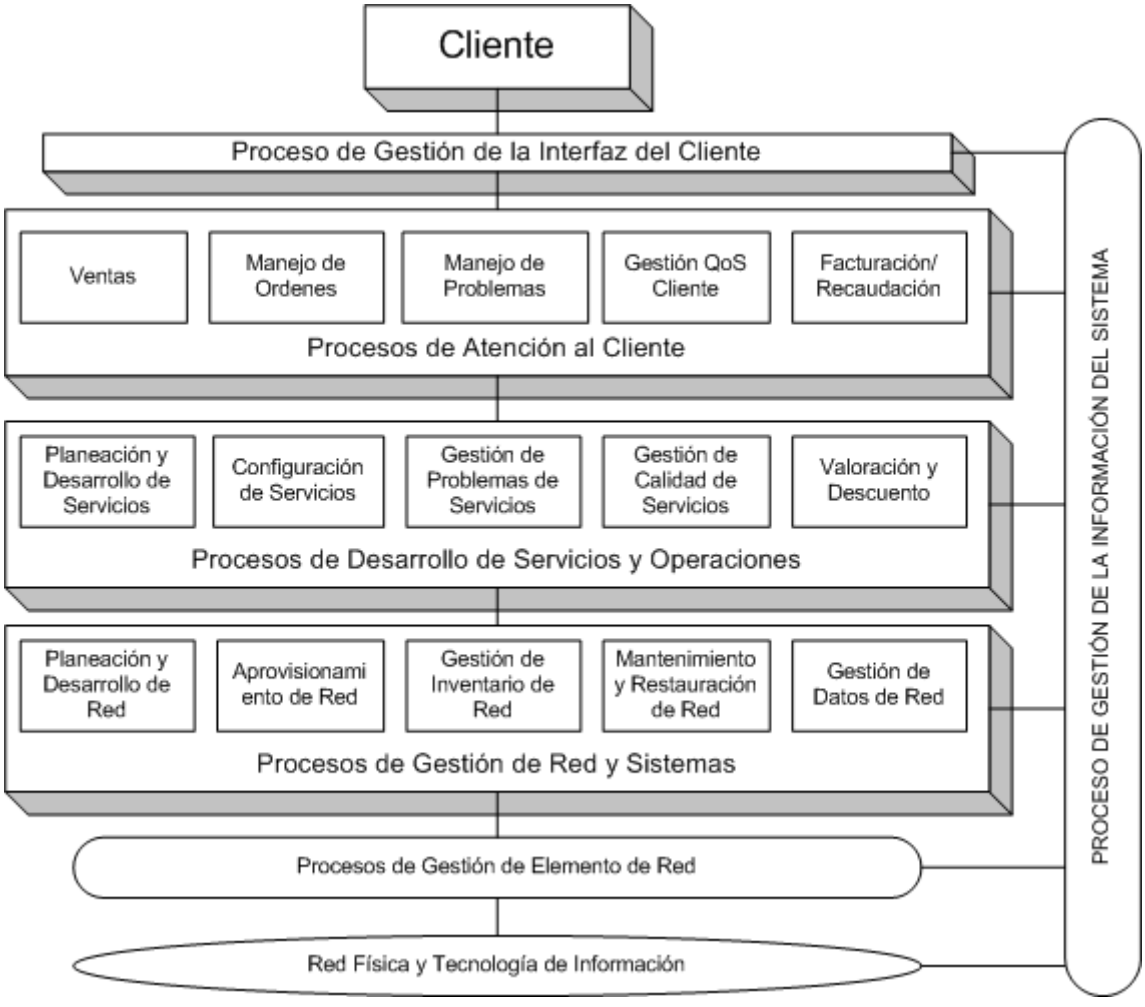


Figura 2-7 - Telecom Operations Map, Estructura General del Negocio

Los objetivos TOM son continuar el progreso hecho, estableciendo:

- Un modelo del proceso comercial común propio de la industria.
- Definiciones comunes para describir procesos de un proveedor de servicio.
- El acuerdo en la información básica exigido al realizar cada proceso, subproceso, y actividad del proceso.
- Una estructura general para identificar qué procesos e interfaces es necesario integrar y automatizar, dependiendo en especial de los acuerdos de la industria.
- Suficiente Información de alto nivel para servir como punto de arranque para los requerimientos comerciales y modelos de información de desarrollo, y la satisfacción de esos requerimientos a través de los acuerdos de la industria.

Flujo de Procesos Descendente

Así como la pirámide TMN se podía dividir verticalmente en las funciones FCAPS, también el TOM puede dividirse en tres procesos comunes descendentes para cualquier negocio orientado al servicio:

- "Fulfillment": Realización de servicio (aprovisionamiento oportuno y correcto de los requerimientos del cliente)
- "Assurance": Aseguramiento de servicio (manteniendo oportunamente el servicio de respuesta y resolución de problemas de red o del cliente, rastreando, informando, gestionando y tomando acciones para mejorar el desempeño en todos los aspectos de un servicio)
- "Billing": Cobro del Servicio (facturación oportuna y exacta, apoyo a la investigación de facturas)

En la siguiente figura se puede ver gráficamente esta división vertical.

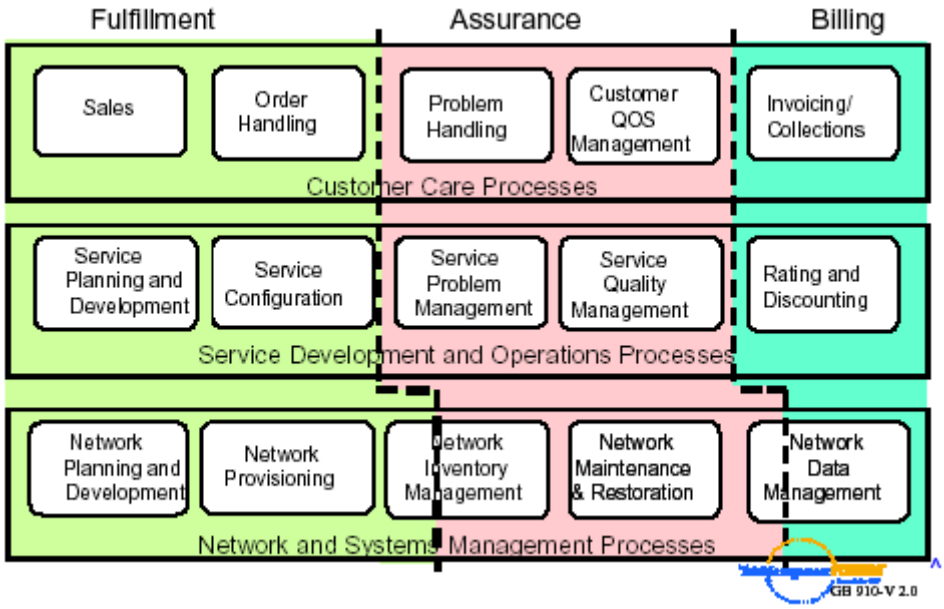


Figura 2-8 - F.A.B., División vertical de procesos

3. Elementos Gestionados

A la hora de estudiar la Gestión de Red es importante comprender los elementos que se quieren gestionar, que como se mencionó anteriormente, se encuentran en la base de la pirámide TMN o sobre el cuál se apoyan los procesos del TOM.

Primero se expondrán las diferentes arquitecturas de red y luego se describirán algunos protocolos de gestión.

3.1. Tecnologías de Medio Óptico

Se mencionarán algunas tecnologías aplicadas a la de transmisión sobre medios ópticos y su gestión.

3.1.1. Multiplexación

Para la transmisión de varios flujos de datos a través de un único medio óptico existen múltiples tecnologías, entre ellas tenemos la multiplexación en el tiempo y la multiplexación en frecuencia (o largo de onda). [7]

La multiplexación en tiempo se denomina “Time División Multiplexing” (TDM) y consiste es asignar marcos temporales para la transmisión de cada flujo de datos.

La multiplexación en frecuencia se denomina “Wavelength División Multiplexing” (WDM) y consiste en enviar canales de información a distintas frecuencias luminosas a través de la misma fibra.

Generalmente estas tecnologías se usan en conjunto para utilizar al máximo la capacidad de transmisión de la fibra óptica.

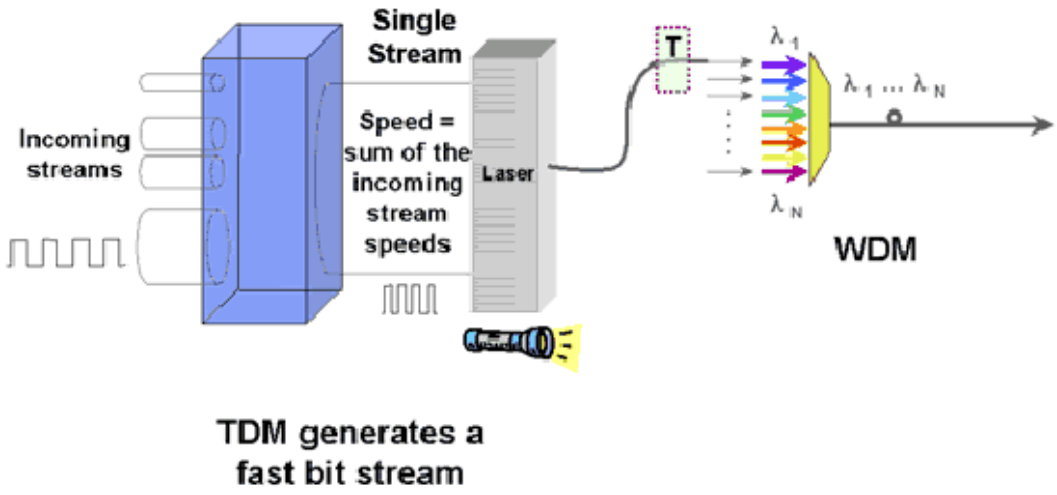


Figura 3-1 - Integración TDM WDM

En la figura se observan la multiplexación en tiempo (TDM) a la izquierda , en la cual un flujo de bits de entrada es multiplexado para obtener un solo flujo de bits de salida. Posteriormente distintas fuentes de LASER provenientes de distintos flujos de datos son multiplexadas en largo de onda para ser portadas en conjunto por una misma fibra óptica.

3.1.2. SDH/SONET

"Synchronous Digital Hierarchy" (SDH) y "Synchronous Optic Network" (SONET) son estándares para la transmisión bajo TDM. SDH fue definido por la "European Telecommunications Standards Institute" (ETSI) y SONET por ANSI. [7]

SDH contiene la mayoría de las características de SONET y es un estándar internacional, pero es especialmente contemplado como un estándar Europeo, dado que solo soporta el ETSI.

La unidad básica de transmisión de SONET es a 51.84 Mbps, pero pudiendo transportar 140 Mbps, SDH puede transportar 155.52 Mbps, con una apropiada elección de opciones, un subconjunto de SDH es compatible con un subconjunto de SONET.

Aunque el tráfico entre SDH y SONET es posible, las alarmas y gestión de desempeño en el "Interworking" generalmente no son compatibles. Éste solo posible en algunos casos con determinadas características entre vendedores de SDH y un poco mas entre vendedores de SONET.

Gestión

SDH incluye una capa de gestión donde dicha información es transportada por un canal dedicado (DCC). Hay un "profile" estándar para la estructura de los mensajes del gestor de red, independientemente de cual sea el vendedor u operador.

Sin embargo no ha habido acuerdos en la definición de un conjunto de mensajes para transportar, así como no hay canales de "Interworking" para gestión entre distintos equipos SDH.

Por otro lado cada nodo comúnmente tiene una interfaz de gestión de red que es típicamente vía LAN, que ha sido mas aceptada. El estándar ITU-TS define la interfaz **Q3em** entre un equipo SDH y su manager. Los vendedores de SDH están migrando su software para ser compatible con esta interfaz.

3.1.3. DWDM

DWDM ("Dense Wavelength División Multiplexing") se refiere al sistema WDM. La diferencia fundamental entre WDM y DWDM es que esta última permite la transmisión más densa de señales de distinto largo de onda que WDM, por lo tanto posee una mayor capacidad de transmisión. [8] [9]

Asimismo permite la amplificación de todos los canales de señales luminosas de una sola vez, sin tener que demultiplexar y procesar cada uno de estos canales por separado para luego ser retransmitidas, esto se debe a las tecnologías de amplificadores ópticos existentes, y redunda en una baja en los costos de instalación de la infraestructura física.

Así entonces el objetivo de todas las redes ópticas DWDM es la eliminación de las conversiones entre electricidad y luz. Toda esta red óptica es llamada como la "transparent network".

Otra de las ventajas que ofrece DWDM es la interconexión prácticamente directa con SONET y ATM, eliminando capas de interconexión entre las mismas, haciendo un uso más simple de estos.

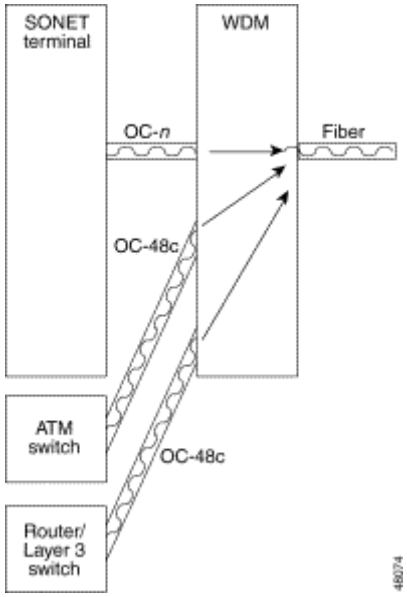


Figura 3-2 - Conectividad directa SONET a DWDM

Gestión

Una de las ventajas principales que ofrece SONET es la posibilidad de tener un "data communication channel" (DCC), usado para funciones de operación, alarmas, información administrativa, señales de control y mensajes de mantenimiento. Cuando SONET es transportado sobre DWMD, los canales DCC continúan funcionando dentro de los elementos de red SONET. Así entonces el sistema DWDM puede tener su propio canal de gestión en la capa óptica. Para el manejo "inband" una pequeña porción de ancho de banda de 8kHz es reservado en los canales básicos de DWDM, y para la gestión "out-of-band" canales adicionales, como por ejemplo el 33o y 32o son utilizados como "optical supervisory channel" (OSC). Dentro de los sistemas de gestión dispone de la Interfaz que dialoga el lenguaje transaccional estándar 1 ("*Standard transaction language 1 interface")*.

3.2. Protocolos de red

3.2.1. IP

"Internet Protocol" (IP) es un protocolo de capa de red, que como su nombre lo indica es el usado en Internet y fue diseñado desde el principio con la interconexión de redes en mente. Su trabajo es proporcionar un medio (de mejor esfuerzo) para el transporte de datagramas de origen al destino, sin importar si las máquinas están en la misma red o hay redes entre ellas. [10]

Las redes IP han tenido un crecimiento en las ultimas décadas fuera de toda predicción, ésta había sido concebida para un tráfico mucho menor al que soporta actualmente. [7]

La característica de IP de no ser orientado a conexión representa un problema en cuanto al aseguramiento de la calidad de servicio (QoS), algunos protocolos

orientados a brindar *servicio de conexión* sobre IP han comenzado a surgir, como por ejemplo MPLS que se verán más adelante.

Actualmente existe tecnologías heterogéneas que pueden soportar IP, como por ejemplo ATM, SDH / SONET o DWDM. En las siguientes secciones se mencionaran éstas y también los medios de gestión de las mismas.

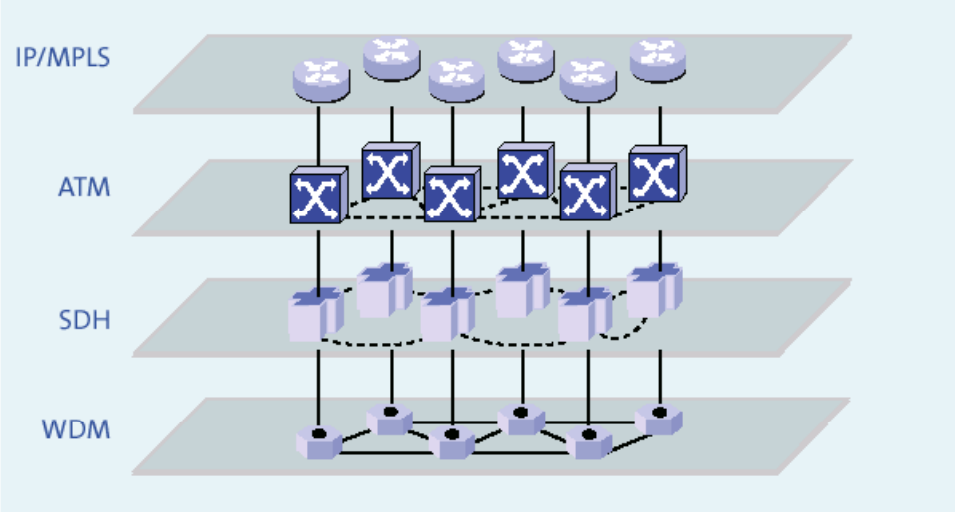


Figura 3-3 - Capas de transporte

3.2.2. MPLS

Multiprotocol Label Switching (MPLS) es un estándar de arquitectura multinivel creado por la Internet Engineering Task Force (IETF) para soportar cualquier tipo de tráfico. En particular nos interesa el tráfico IP pues le agrega funcionalidad a éste para resolver problemas de administración, VPNs, ingeniería de tráfico y calidad de servicio (QoS) para Voz/Video sobre IP, que IP por si solo no puede ofrecer. [11] [12]

Permanece independiente de los protocolos de capa 2 y 3 por lo que puede funcionar sobre enlaces existentes ATM, Frame Relay u otra tecnología.

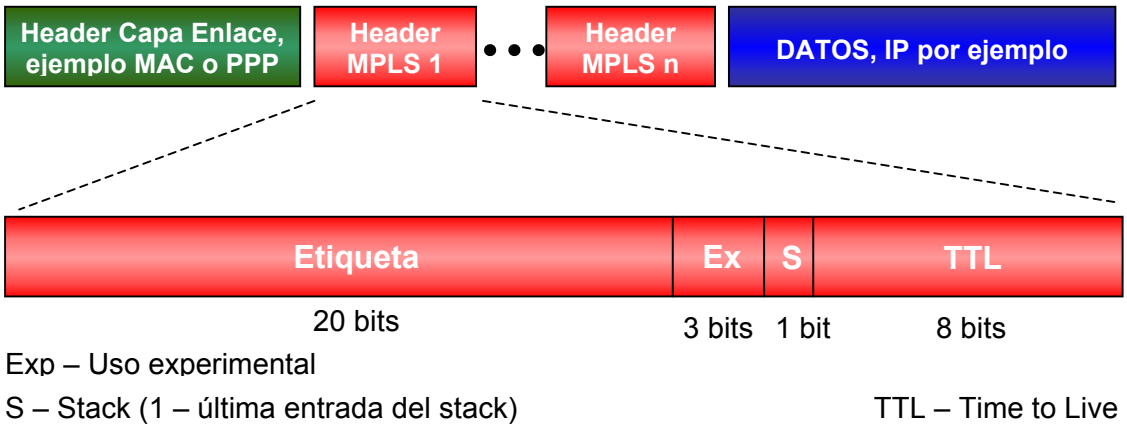


Figura 3-4 - Cabecera MPLS

Una red MPLS se compone de dispositivos (nodos MPLS) interconectados entre sí. Éstos son llamados **LSR** (Label Switching Router) y los que limitan la red son llamados

LER (Label Edge Router). Los nodos extremos (LER) añaden o quitan etiquetas, mientas que los nodos interiores (LSR) sustituyes unas etiquetas por otras. Para realizar esta tarea los nodos mantienen tablas con correspondencias entre: interfaz de entrada, etiqueta de entrada, interfaz de salida y etiqueta de salida.

Las **Etiquetas** (Labels) son identificadores con significado local empleado para identificar un FEC.

Las **FEC** (Forwarding Equivalence Class) son agrupaciones de paquetes que comparten los mismos atributos (por ejemplo: dirección destino, VPN) y/o requieren el mismo servicio (multicast, QoS). El FEC se asigna en el momento en que el paquete entra a la red. Todos los paquetes que forman parte del FEC, siguen un mismo LSP.

Un **LSP** (Label Switched Path) es un camino a través de uno o más LSRs en un nivel de jerarquía que sigue un paquete de un FEC en particular.

Antes de cualquier envío de paquetes, es necesario que los LSRs cuenten con un acuerdo acerca de la relación existente entre las etiqueta y los LSPs. Esto se consigue utilizando el Protocolo de Distribución de Etiquetas (**LDP**, Label Distribution Protocol).

Sobre esta base, un LSR de acceso recibe un paquete IP, analiza tanto la cabecera IP como el puerto de entrada y determina el destino. Este LSR de acceso (LER) añade una etiqueta al paquete IP que identifica el trayecto orientado hacia el destino (LSP) y envía el paquete hasta el siguiente LSR del backbone. El siguiente LSR conmuta el paquete basándose únicamente en la etiqueta. No inspecciona en absoluto la cabecera IP. Cuando el paquete alcanza el LSR de salida, éste elimina la etiqueta y lo envía al siguiente salto según el enrutamiento específico de la red a la que se entrega dicho paquete.

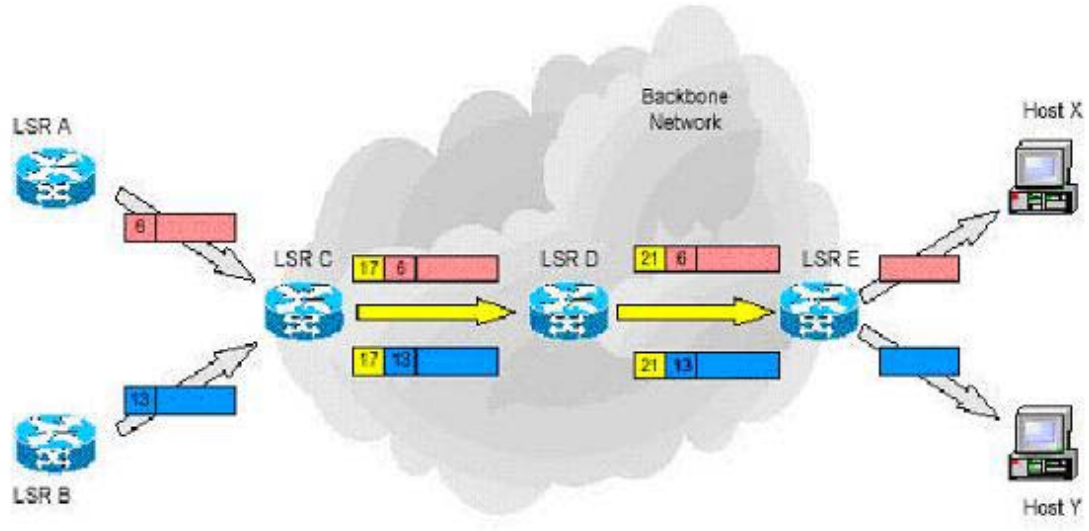


Figura 3-5 - Stack de etiquetas a través del backbone MPLS

En la figura anterior podemos ver como los LSPs (rojo y azul) están en paralelo y encaminan tráfico juntos hacia un túnel LSP de nivel más alto (amarillo). Esto permite diferentes niveles de granularidad, también permite reducir el tamaño de las tabas y la complejidad de la gestión.

3.2.3. ATM

ATM (Asynchronous Transfer Mode) es una tecnología diseñada para la transmisión de datos a alta velocidad. Fue creada por la CCITT para la industria telefónica, con la meta original de reemplazar la red telefónica pública conmutada y obtener una red digital de servicios integrada (ISDN). Luego, en 1991, muchos proveedores de computadores se unieron con las compañías telefónicas para crear el ATM Forum y guiar el futuro de ATM. [10]

La idea en ATM consiste en transmitir toda la información en paquetes pequeños de tamaño fijo (53 bytes) llamados **celdas**. Esto hace posible construir conmutadores genéricos que reciben celdas por una línea de entrada, pasan por la estructura de conmutación y se entregan en una línea de salida.

Las celdas no se alternan de manera rígida entre las fuentes, por esto decimos que ATM es **asincrónico**. Pero se requiere que lleguen en le mismo orden que fueron envidadas y con tasas de pérdida muy bajas (menor a 1 celda cada 10^{12}). Los conmutadores tienen colas para entregar celdas, si por ejemplo llegan dos celdas simultáneamente para la misma salida.

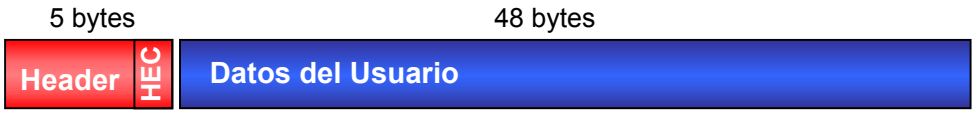


Figura 3-6 - Celda ATM

La cabecera de cada celda es de 5 bytes y el último de los 5 bytes es un control de error de la cabecera llamado **HEC** (Header Errror Control). Sólo se hace control de la cabecera, dado que fue diseñado para fibra óptica donde la tasa de error es muy baja. El HEC también se utiliza para detectar el comienzo de un celda buscando un HEC correcto. Existen dos tipos de cabeceras, pero en ambos casos, en los 4 bytes útiles de ésta, se incluye un identificador de trayectoria virtual (VPI), canal virtual (VCI) y tipo de carga útil.

ATM no se proyecta muy bien en el modelo OSI. Cumple funciones de capa 2 y 3, aunque muchas veces se considera como capa de enlace pues se pone IP encima de ésta. Existe una capa de adaptación ATM (**AAL**, ATM adaptation layer) que permite enviar paquetes más largos y se encarga de desensamblarlos y reensamblarlos.

Las redes ATM son orientadas a la conexión. El elemento básico es el circuito (o canal) virtual (VCI). Éstos son permanentes o conmutados. Los primeros son solicitados manualmente por el cliente, mientras que los segundos tienen que establecerse cada vez que se usan, como al hacer una llamada telefónica. El establecimiento de la conexión no es parte de la capa ATM y es manejado en un plano de control mediante el protocolo Q.2931 de la ITU.

Como vimos los conmutadores ATM son los encargados de encaminar el tráfico y lo hacen con el campo VPI, mediante tablas que relacionan para cada VPI de entrada, una línea de salida y un VPI de salida. El campo VCI se utiliza para el salto final entre un conmutador y un host.

3.3. Protocolos de Gestión

Se analizaran distintos protocolos y lenguajes de gestión de redes. Entre ellos se verá SNMP, la estrella principal, y otros competidores que nunca alcanzaron el éxito que se suponía, como por ejemplo CMIP.

La popularidad que alcanzó SNMP marco un camino, seguido por sus versiones mejoradas SNMPv2 y SNMPv3.

Otros decidieron no luchar contra el enemigo y unírsele, tal es el caso de RMON.

3.3.1. SNMP

Analizaremos SNMP como protocolo de gestión, éste también es un modelo de gestión y lo analizaremos en conjunto.

El protocolo

Como vimos SNMP es un protocolo de gestión simple, que fue diseñado a mediados de del los 80´s como “band-aid”, para solucionar problemas ocasionados por el crecimiento exponencial que tenían las redes. [13] [14]

SNMP permite a los sistemas de administración la localización y corrección de problemas en una red TCP/IP.

El SNMP es una versión extendida de “Simple Gateway Monitoring Protocol” (SGMP).

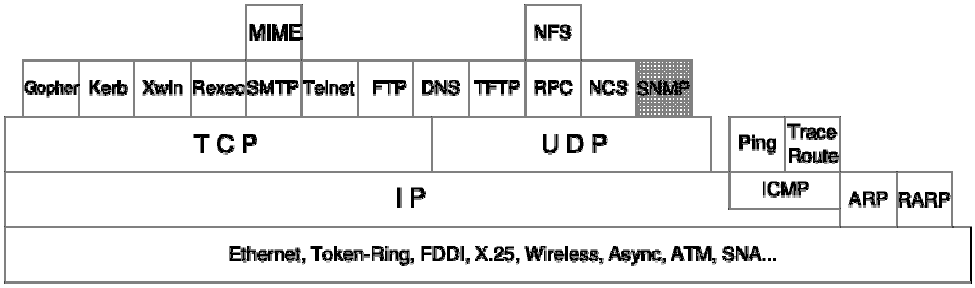


Figura 3-7 - Ubicación de SNMP en el stack TCP/IP

El modelo

Los elementos del modelo de gestión se basan en la existencia de **gestores SNMP** y **agentes SNMP,** el protocolo SNMP se basa en el paradigma **“fetch-store”,** en el cual el gestor mantiene una serie de valores conceptuales que pueden incluir estadísticas, así como variables complejas que corresponden a estructuras de datos TCP/IP (como por ejemplo Cache ARP o tablas de enrutamiento IP) de sus agentes.

Los mensajes SNMP especifican si el gestor debe recuperar valores de las variables de sus agentes o debe almacenar valores en ellas; por lo que toda solicitud de operación que se le demande al gestor deberá traducirse a este conjunto de operaciones básicas. Dado que el protocolo no incluye otras operaciones, todo el control debe ser ejercido mediante el paradigma fetch-store.

3.3.1.1. PDU

Un mensaje de SNMP consiste en un identificador de la versión, un nombre de la comunidad SNMP y un "Protocol Data Unit" (PDU). Toda implementación de SNMPv1 debe soportar las cinco PDUs siguientes:

- **GetRequest**: Recuperar los valores de un objeto del MIB
- **GetNextRequest**: Recorrer parte del MIB
- **SetRequest**: Alterar los valores de un objeto del MIB
- **GetResponse**: Respuesta de GetRequest, GetNextRequest y SetRequest
- **Trap**: Capacidad de los elementos de red para generar eventos como la inicialización, reinicio o fallo en el enlace con el “Management Agent” (MA). Hay siete tipos de traps definidos en el RFC 1157: coldStart, warmStart, linkDown, linkUp, authenticationFailure, egpNeighborLoss y enterpriseSpecific.

3.3.1.2. SNMP NMS-MA

El RFC 1157 define "Network Management Station" (NMS) como una estación que ejecuta aplicaciones de gestión de red (NMA) que monitorizan y controlan elementos de red (NE) como hosts, pasarelas y servidores de terminales. Estos elementos usan un agente para realizar estas funciones. El SNMP para la comunicación de información entre las NMS y los MA.

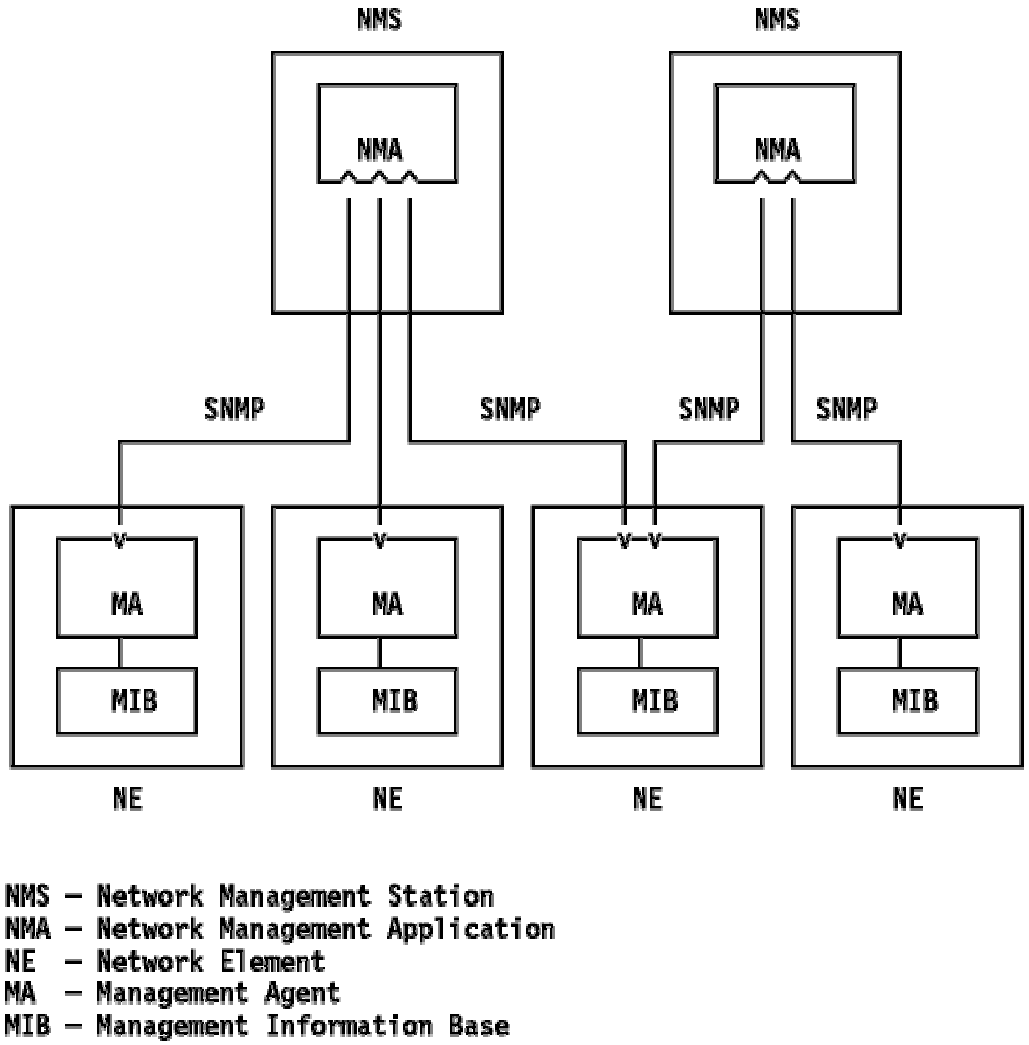


Figura 3-8 - Funcionamiento SNMP

Las entidades que residen en las NMS y los elementos de red que se comunican con otros a través de SNMP se denominan entidades de aplicación de SNMP. Los procesos que las implementan son las entidades de protocolo. Un agente SNMP con un conjunto arbitrario de entidades es una comunidad SNMP, en la que cada entidad se nombra con una lista de bytes que debe ser única para esa comunidad.

3.3.1.3. SNMPv2

La infraestructura de la versión 2 de SNMP se publicó en abril de 1993 y consiste en 12 RFCs, incluyendo el primero, el 1441, que es una introducción. En agosto de 1993 los 12 RFCs se convirtieron en un estándar con status electivo.

Se mejoró el sistema de seguridad respecto a la versión 1 con la incorporación al *framework* de un nuevo *sistema administrativo*. La infraestructura de este sistema administrativo incorpora el "*user-based security model for SNMPv2"* y el "*community-based SNMPv2"*.

La idea de una *Community based security* consta simplemente del establecimiento de un nombre de comunidad que se adjunta encriptado en cada mensaje que es enviado.

Obviamente este sistema no ofrece protección alguna contra intrusiones, por lo que veremos mas adelante las mejoras realizadas en SNMPv3 al respecto.

Las siguientes son otras diferencias fundamentales entre SNMPv1 y SNMPv2

SNMPv2 Entity

Una *entidad* SNMPv2 es un proceso real que realiza operaciones de gestión de red mediante la generación y/o respuesta a/de mensajes SNMPv2. Todas las posibles operaciones de una entidad se pueden restringir a un subconjunto de las operaciones que puede efectuar el entorno de gestión "SNMPv2 Party" o entidades gestoras (EG). Una *entidad* SNMPv2 podría pertenecer a múltiples *entidades* gestoras, y mantiene las siguientes bases de datos locales:

- Una base de datos para todos los EG que conoce la entidad, que podrían ser:
  - Operación local.
  - Operación local realizada por interacciones con EG o dispositivos remotos.
  - Operación realizada por otras entidades SNMPv2.
- Otra base de datos que representa todos los recursos de los objetos gestionados que conoce la entidad.
- Una base de datos que representa una política de control de acceso que define los privilegios de acuerdo con los EG conocidos.

Una entidad SNMPv2 puede actuar como agente o como manager SNMPv2.

SNMPv2 party

Un *SNMPv2 party* es un entorno virtual de ejecución, en el cual las operaciones son restringidas, sea por seguridad u otros propósitos.

Esta define un subconjunto de posibles operaciones que pueden realizan las entidades en ellas.

3.3.2. SNMPv3

SNMPv3 es mucho más complejo que SNMPv1 y SNMPv2. Su complejidad radica en el mecanismo de seguridad usado en el mismo. SNMPv1 y SNMPv2 proveen solo primitivas básicas de seguridad basadas en "community names" fácilmente violables. [15] [16]

El modelo de seguridad usado en SNMPv3 el "User Security Model" (USM) definido en el RFC 2574.

El USM provee autenticación y servicios privados para SNMP y esta diseñado para asegurar los siguientes aspectos:

- **Modificación de la Información**: Una entidad puede alterar el tránsito interno de mensajes generados por una entidad autorizada, con el fin de realizar operaciones no autorizadas de administración.
- **Enmascaramiento**: Una entidad no autorizada puede realizar operaciones asumiendo la identidad de una entidad autorizada.

- **Modificación** de flujo de mensajes: dado que SNMP esta diseñado para operar sobre un protocolo no orientado a conexiones los mensajes pueden ser reordenados, retrazados o replicados para realizar operaciones no autorizadas.
- **Apertura**: Una entidad puede observar el intercambio de mensajes entre *managers* y *agents* y conocer los datos manipulados.

Los casos para los que el USM no ofrece seguridad son:

- **Negación de servicios**: Si un atacante puede bloquear intercambio de mensajes entre *managers* y *agents*.
- **Análisis de trafico**: Un atacante puede observar patrones generales de trafico entre *managers* y *agents*.

Funciones Criptograficas: Existen dos funciones criptográficas definidas en USM: autenticación y encriptación. Para soportar estas funciones los elementos SNMP requieren dos valores, una clave privada (privKey) y una clave de autentificación (authKey). [17]

Estas claves son mantenidas separadamente por los siguientes usuarios:

Usuarios locales: Entidades principales para las cuales las operaciones son autorizadas.

Usuarios remotos: Entidades remotas a las cuales son envidas las operaciones.

Estas claves son guardadas por cada uno de los usuarios y no son accesibles vía SNMP.

3.3.3. CMIP / CMOT

CMIP fue pensado para ocupar el lugar que SNMP tiene. Básicamente su diseño es similar a SNMP con un enfoque distinto del mismo. [18]

CMIP es un protocolo basado en sistemas orientados a objetos. A diferencia de SNMP, en el cual la MIB consiste únicamente en una colección de atributos, en CMIP este posee una MIB basada en una colección de objetos, los cuales poseen atributos, comportamientos, pueden ser creados y eliminados, y pueden también realizar acciones especificas que los administradores pueden requerir.

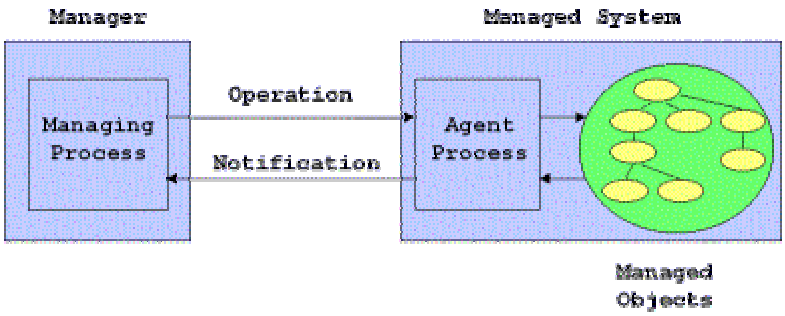


Figura 3-9 - Arquitectura Manager - Managed System en CMIP

Así entonces el comportamiento de cada objeto esta vinculado directamente con el recurso al cual representa.

En CMIP los objetos poseen las siguientes funcionalidades :

- **Atributos***:* estos representan las variables simples (datos).
- **Conductas***:* en la cual acciones pueden ser disparados por los mismos.
- **Notificaciones**: en la cual la variable genera un reporte de evento.

La gran diferencia de CIMP es que los objetos no solo relacionan información de las terminales, sino que también pueden ser usados para hacer tareas, cosa imposible con SNMP.

En comparación con SNMP, en el cual uno debe explícitamente seguir los cambios de la red, CMIP permite notificaciones generadas por los agentes que redundan en un manejo más eficiente y menos complejo para mantener actualizado el estado de la red.

Así también CMIP posee un sistema de seguridad que soporta autorización, control de acceso y logueos de seguridad. Características solo comparables con SNMPv2.

3.3.4. SMI y MIB

El "Structure and Identification of Management Information" (SMI) define las reglas para describir los objetos gestionados y cómo los protocolos sometidos a la gestión pueden acceder a ellos. La descripción de los objetos gestionados se hace utilizando un subconjunto de ASN.1 ("Abstract Syntax Notation 1, estándar ISO 8824), un lenguaje de descripción de datos. [14]

Tanto SNMP y CMOT utilizan los mismos conceptos básicos en la descripción y definición de la información de gestión denominada *SMI* descrita en el RFC 1155 y "Management Information Base" (MIB), descrita en el RFC 1156. [1]

El estándar MIB define las variables que el servidor SNMP debe mantener. EL MIB define un conjunto de *variables conceptuales* a las que el servidor SNMP debe acceder.

Estas variables pueden dividirse en dos clases: *simples y tablas.*

MIB utiliza ASN.1 para nombrar todas las variables. El ASN define un espacio de nombres jerárquico, de modo que el nombre de una variable indica su posición en la jerarquía. El objetivo de todo esto es distribuir cuidadosamente la *autoridad* para asignar nombres en relación a organizaciones.

Hay dos versiones, MIB-I y MIB-II, la primera fue definida en el RFC 1156, y está clasificado ahora como protocolo *histórico* con status *no recomendado*.

3.3.5. RMON

La generalización de SNMP trajo consigo diversas ampliaciones, una de ellas fue “Remote Monitor” (RMON), que permite monitorizar globalmente una subred, no solo a sus dispositivos. [14]

---

[1] La razón de una compatibilidad con respecto a la MIB entre CMOT y SNMP radicaba en la esperanza de una convergencia, pero actualmente CMOT languidece y SNMP se ha impuesto.

La MIB de RMON define estadísticas a nivel de capa MAC y objetos de control que permiten capturar información en tiempo real en toda una red.

RMON es un estándar de definición de SNMP MIB descripta en el RFC1757.

AL igual que SNMP este posee una Estación de gestión central y dispositivos remotos de monitoreo llamados *agentes,* estos agentes corren en diferentes segmentos de una red, usualmente uno por subred. Un ejemplo es el siguiente:

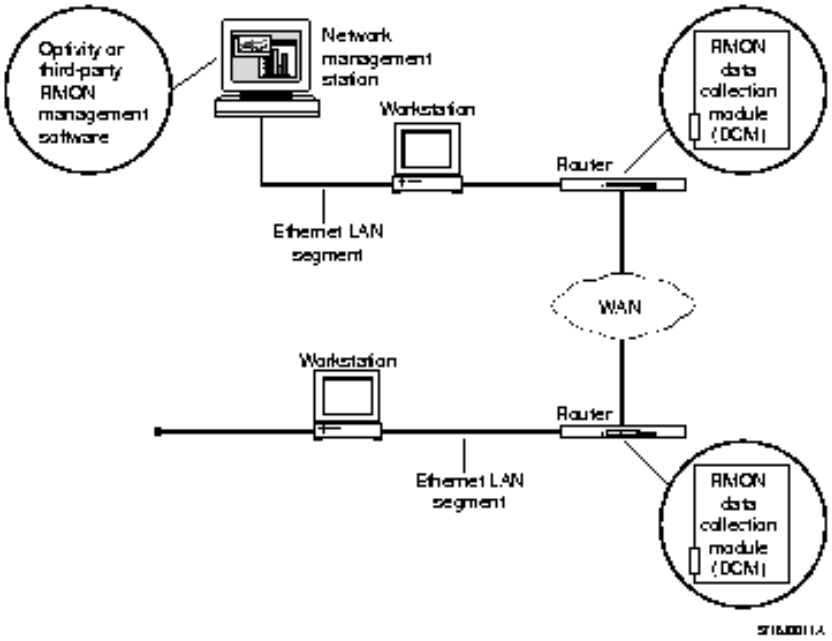


Figura 3-10 - Distribución de Agentes RMON

La MIB de RMON contiene ampliaciones a la MIB de SNMP para administrar diferentes aspectos de la red, éstos son llamados *grupos* y se identifican de la siguiente manera:

- Estadísticas
- Historia
- Host
- HostTopN
- Matrices
- Filtros
- Capturas
- Alarmas
- Eventos

3.3.6. CLI

La mayoría de los elementos de red poseen “Command Line Interfaces” (CLIs). Estas interfaces son accesibles vía telnet o sobre líneas seriales.

Los operadores escriben comandos en ASCII para realizar consultas y controlar los elementos de red. [30]

En muchos casos todo el complejo soporte, incluido el “turn up”, es realizado enteramente a través del CLI. Muchos CLIs llegan a soportar incluso el pedido de “help” para dar facilidades al usuario.

El lenguaje TL1 es un ejemplo muy difundido en Norteamérica de CLI.

3.3.6.1. TL1

En 1984 la “Regional Bell Operating Copanies” (RBOC) especifica un lenguaje estándar llamado “Man Machine Language” (MML) para el control de los elementos de red. Ese lenguaje se llamó TL1.

En los siguientes años luego de su creación los primeros OSS, como el NMA de Bellcore entraron al mercado, y ellos requerían que los NE soportaran interfaces TL1. Así que los vendedores rápidamente implementaron TL1 en sus dispositivos y quedó consolidada su difusión.

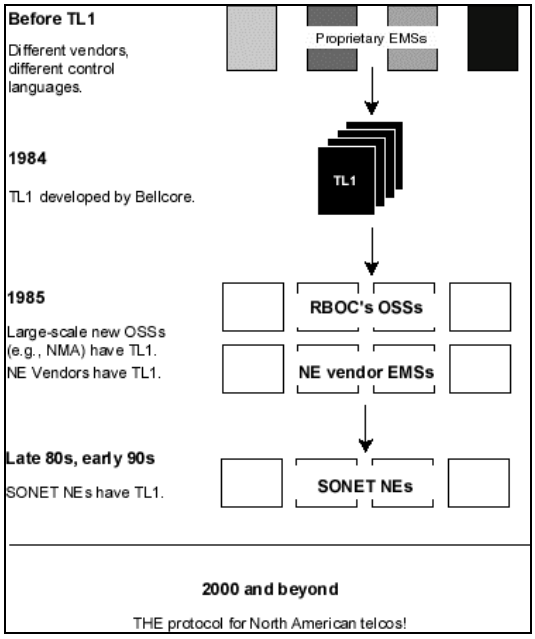


Figura 3-11 - Desarrollo TL1

TL1 posee 2 formas de comunicación, humano-maquina y maquina-maquina:

- **Humano-maquina**: los operadores y desarrolladores pueden leer fácilmente los mensajes , TL1 no requiere sofisticados analizadores de protocolo, SE OBTIENE LO QUE SE VE (“what you see is what you get”).
- **Maquina-maquina**: Su comunicación es tan efectiva como la binaria, pero sus mensajes pueden ser leídos por los humanos, esta es una gran diferencia con otros productos, el control humano es posible directamente.

4. Capa de Gestión de Elementos

Como vimos al estudiar la pirámide TMN la gestión de elementos es la capa que brida las funcionalidades para operar los elementos gestionados.

El estudio de esta capa se centrará en la interfaz del TM Forum llamada MTNM, que especifica un ejemplo de funcionalidad de un "Element Management System" (EMS).

4.1. MTNM

"Multi-Technology Network Management" (MTNM) es una interfaz entre capa de red y capa de elementos (NML-EML), creada por el *TeleManagement Forum* y alineada con el TOM. Pretende resolver una necesidad inmediata de la industria con una solución aplicable a múltiples tecnologías. [19]

Antes de MTNM se tenía que para administrar redes heterogéneas la capa de red NML tenía que tener una interfaz para cada tipo de red. Con la utilización de MTNM es posible administrar la capa EML con una única interfaz común.

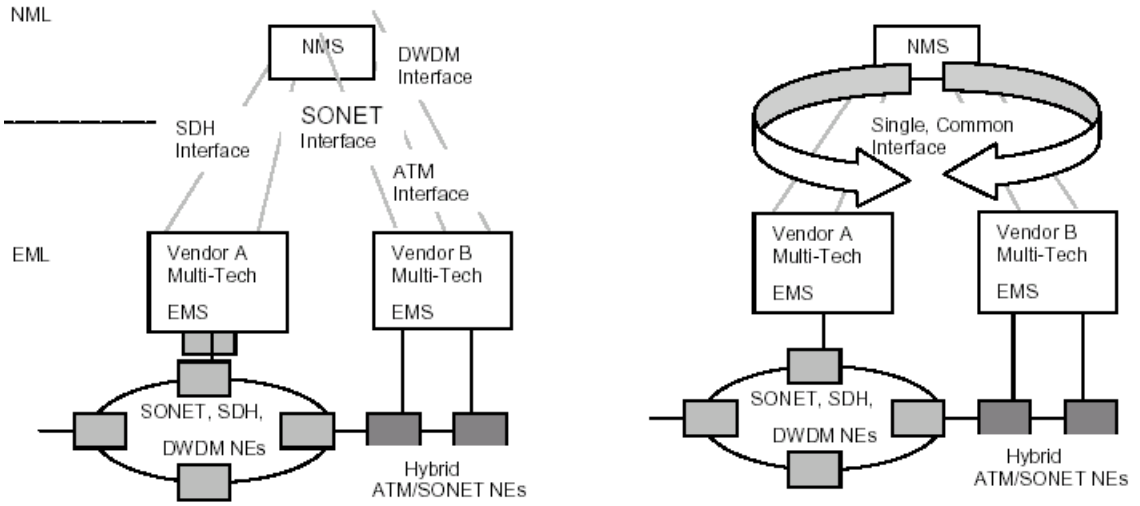


Figura 4-1 - Diferencia entre una Interfaz común e interfaces individuales

El hecho de independizar la gestión de red de la tecnología subyacente trae grandes beneficios para proveedores, consumidores y fabricantes.

Beneficios a los proveedores de servicios porque permitirá gestionar las redes de la próxima generación compuestas de elementos híbridos de manera eficiente. También porque permite la rápida inclusión de nuevas tecnologías.

Beneficios al consumidor final y clientes porque como consecuencia del beneficio de los proveedores los clientes se beneficiaran con mejores servicios y reducción de costos.

Beneficios para los fabricantes porque si no hay beneficios en los proveedores no los habrá para los fabricantes. A su vez esta interfaz permite agregar nuevas tecnologías dada su extensibilidad y flexibilidad.

Documentos

La especificación MTNM no solo contiene las interfaces sino también una descripción detallada de los módulos que las implementan y diagramas de clases. Además se provee un conjunto de soluciones con todas las IDL CORBA.

Los siguientes documentos del TM Forum tienen toda la información sobre MTNM:

Documento	Nombre
TMF 513	Multi-Technology Network Management Business Agreement
TMF 608	Multi-Technology Network Management Model *Incluye diagramas de casos de uso, de clases y de colaboración en UML*
TMF 814	Multi-Technology Network Management Solution Set

5. Capa de Gestión de Red

En esta sección estudiaremos a partir del estudio de una interfaz (CaSMIM) las funcionalidades de la capa de gestión de red. Como vimos al estudiar la pirámide TMN para prestar un servicio (conexión punto a punto) sobre una red es necesario gestionar simultáneamente múltiples elementos de red, posiblemente gestionados por distintos EMS. La función de esta capa es proveer dichos servicios integrando los diferentes elementos de red.

5.1. CaSMIM

El "Connection and Service Management Information Modeling" (CaSMIM) es una interfaz desarrollada por el TM Forum para comunicar un "Service Management System" (SMS) con un "Network Management System" (NMS), en definitiva para comunicar la capa de servicio (SML) con la capa de red (NML) de la pirámide TMN. [20]

La interfaz fue pensada esencialmente para Aprovisionamiento y Configuración de un servicio orientado a conexión, aunque también se pretende que alguna información relacionada a las fallas pase por la interfaz si es pertinente al SMS.

Aunque esta interfaz esta principalmente diseñada para comunicar la capa de servicio con la de red (SML-NML) también puede ser utilizada entre NMSs o también directamente sobre un EMS.

Al igual que vimos en la descripción de la interfaz MTNM, esta interfaz plantea beneficios para los proveedores de servicios, para los clientes, los fabricantes y también para los proveedores de software, basándose en el hecho de que se puede integrar tecnologías con un modelo común.

Posible Escenarios

A continuación se verán dos posibles escenarios que ilustran los objetivos de CaSMIM.

En un primer escenario imaginemos un servicio de aprovisionamiento ("Provisioning") de conexión punto a punto. Éste es el principal escenario en ambientes multi-fabricante y multi-tecnológicos. Cualquier combinación de tecnologías puede ser hecha, entre ellas todos las tecnologías orientadas a la conexión como ATM, SONET / SDH, WDM, y *Frame Relay* así como tecnologías de acceso como ADSL y HFC. En este servicio se pretende descubrir información topológica de la red, descubrir capacidades de la red, crear modificar y borrar conexiones. También descubrir y consultas rutas, además de las restricciones para las mismas.

Otro posible escenario es el mapeo de alarmas de la red a sus correspondientes notificaciones en clientes de más alto nivel, como la gestión de servicio y sistemas de gestión de conexión.

Mapeo de los procesos a la estructura general del TOM

Como vimos en secciones anteriores el TOM (Telecom Operations Map) provee una dirección de alto nivel para los procesos de la operación de comunicaciones. Ahora veremos que procesos del TOM son comprendidos por CaSMIM.

En la siguiente figura tenemos en *color naranjo* ■ las áreas cubiertas en esta interfaz y en *azul* ■ el área de resolución de problemas parcialmente cubierta.

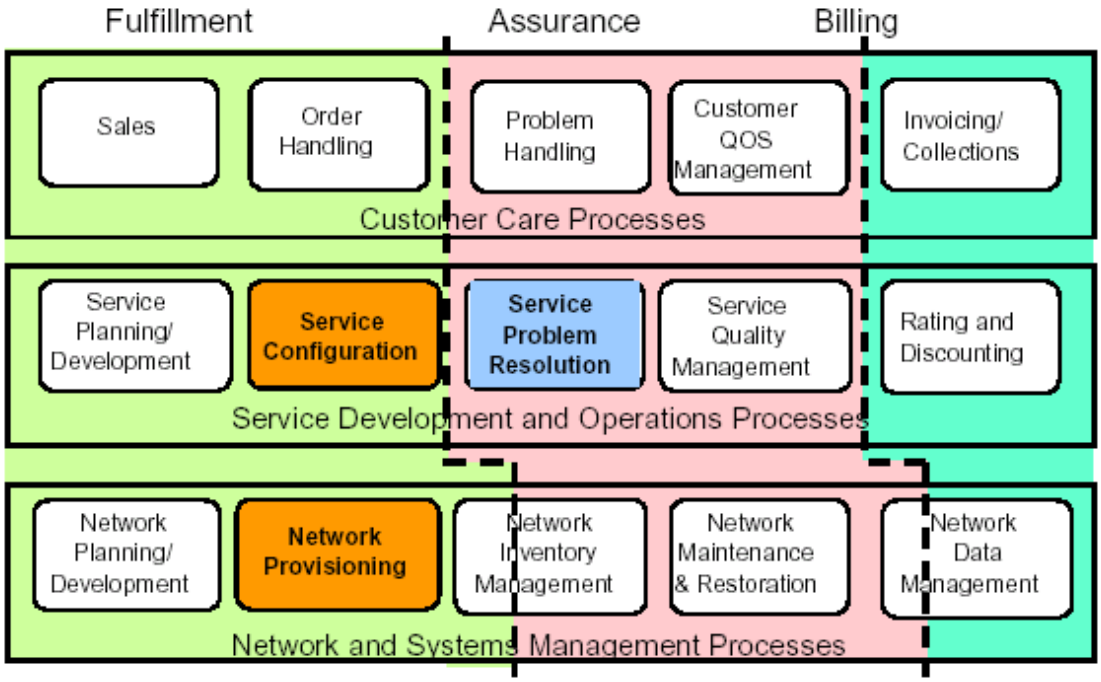


Figura 5-1 - Alcance de CaSMIM sobre el TOM

Requerimientos

Ahora veremos los requerimientos que se plantearon en este acuerdo comercial en su versión actual (versión 3).

Entre los requerimientos generales que se le pusieron a CaSMIM se tienen: que sea multi-tecnológico, que incluya varios niveles de interoperabilidad utilizando perfiles, que soporte servicios de manera flexible y eficiente, que sea escalable, que sea programable (permitir reservar capacidad a futuro), que se puedan manejar condiciones anormales.

Entre los requerimientos referidos a la topología se tienen:

- Get Connection
- Get Connection Info
- History of connections
- Get Acess Group
- Get Acess Group Info

También se plantearon grupos de requerimientos para la comunicación de fallas, la Gestión de Resincronización, Gestión de Notificaciones y Gestión de Accesos, Gestión de Conexión, otro para Cancelaciones y Enmiendas, y finalmente otro para Gestión de Pruebas.

Se analizaran los requerimientos del grupo de Gestión de Conexión pues algunos de ellos tienen impacto directo en nuestra implementación:

- **Crear Conexión de Red**: la interfaz permitir que el cliente le pida al servidor que cree una nueva conexión. Entre los parámetros se tendrán: "Terminal Points" (TP) de comienzo y fin, y parámetros de conexión. Opcionalmente se podaran incluir los siguientes parámetros: identificación del "customer" (cliente), identificador del servicio, número de versión, tiempo requerido, portador (sí aplica) y tiempo finalización. Inmediatamente recibido el pedido se responderá: si el requerimiento fue Aceptado o Rechazado, y luego sí se Reservó (junto con el identificador de conexión) o no.
- **Activar Conexión de Red**: esta interfaz permite que el cliente solicite al servidor la activación de una conexión previamente solicitada. Entre los parámetros se incluirá el identificador de conexión y opcionalmente: número de versión, el tiempo requerido, tiempo de finalización y parámetros de conexión. Inmediatamente recibido el pedido se responde si se Aceptó o se Rechazó, y más tarde si el servicio pudo ser activado o no.
- **Crear y Activar Conexión**: permite crear y activar una conexión en una única solicitud. Algunos proveedores puede querer que solo se soliciten conexiones de esta manera para que no haya capacidad reservada que no sea utilizada.
- **Desactivar Conexión de Red**: permite desactivar una conexión pasando el identificador de la misma, opcionalmente se pasa el número de versión y tiempo requerido. Al igual que el resto de las operaciones se responde si se aceptó o rechazó, y luego si se pudo desactivar o no.
- **Borrar Conexión de Red**: permite borrar una conexión. Los parámetros y respuestas son iguales que los requerimientos anteriores.
- **Desactivar y Borrar Conexión**: desactiva y borra una conexión en una único pedido.
- **Modificar Conexión**: permite modificar cualquiera de los parámetros pasados en la creación de la conexión. Todos los parámetros de la conexión son opcionales con excepción del identificador de conexión.

Documentos

Los siguientes documentos del TM Forum tienen toda la información sobre CaSMIM:

Documento	Nombre
TMF 508	Connection and Service Management Business Agreement
TMF 605	Connection and Service Management Information Model (CaSMIM) Information Agreement
TMF 807	NGN Connection and Service Management Information Model CORBA IDL Solution Set

6. Ejemplos de Herramientas

A continuación se hará una breve introducción a algunas herramientas de gestión relacionadas con los objetivos del proyecto.

6.1. WINMAN

““WDM and IP Network Management” (WINMAN) es un sistema de gestión de red desarrollado por un consorcio de países europeos.

La idea de WINMAN es proveer una solución de gestión de redes basado en TMF, utilizando la interfaces CaSMIM y MTNM para proveer servicios de conectividad IP por contratos de Niveles de Servicios. WINMAN captura los requerimientos, define y especifica una arquitectura de gestión abierta, distribuida y escalable para servicios de conectividad IP en redes híbridas de transporte (caso de ATM, SDH y WDM). Las plataformas son independientes, ya sea a la hora de capturar los requerimientos o a la hora de definir la arquitectura. [23]

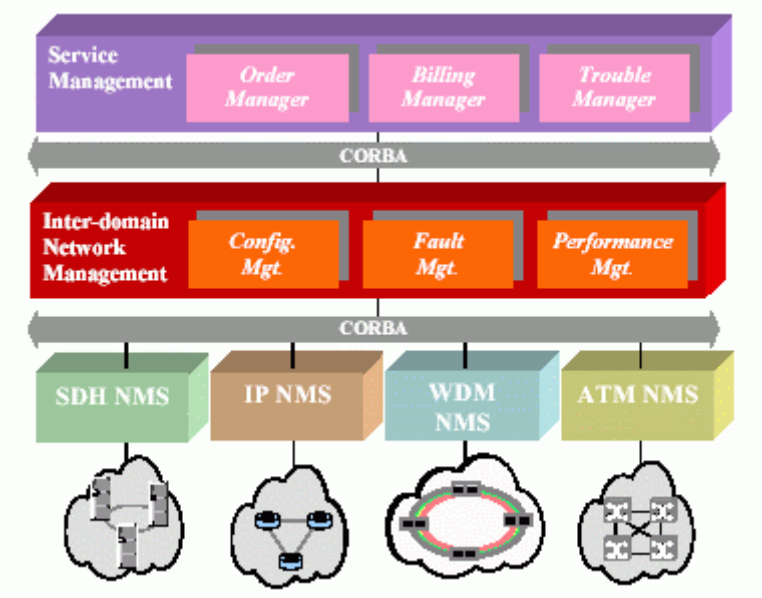


Figura 6-1 - Arquitectura de alto nivel de WINMAN

Desde el punto de vista de la implementación, el proyecto maneja en forma separada lo que es una red IP, de las WDM. La integración de los gestores a nivel de Red está separada por dominio de tecnología. Comprende un sistema de gestión de redes de dominio Inter-tecnológico llamado “Inter-domain Network Management System” INMS, como subcapa de capa de Red que soporta conectividad IP expandiendo diferentes subredes WDM, e integra la gestión de redes IP y WDM.

El INMS y los sistemas de gestión IP y WDM implementan funciones de aplicaciones de Configuración, Fallas y Performance (CFP).

6.2. AdventNet

AdventNet básicamente provee soluciones para los proveedores de gestión de servicios. La idea general a la hora de brindar este tipo de soluciones por parte de AdventNet es cubrir las siguientes necesidades: [24]

- Servir a una gran cantidad de clientes eficientemente.
- Proveer productos de gestión equipados para acompañar el crecimiento de los proveedores de servicios de gestión (MSP), para que puedan escalar y distribuirse.
- Evitarle a los clientes el uso de VPN, que pueden encarecer sus soluciones.

A nivel de desarrollo de aplicaciones, AdventNet nos ofrece un software orientado a la integración de tecnologías emergentes de hoy en día.

- AdventNet Web NMS: Es un framework para desarrollar aplicaciones EMS, NMS, Aprovisionamiento y sistemas OSS. Dicha herramienta soporta muchos tipos de elementos de red gestionados, entre ellos se encuentran SNMP, CORBA y TL1.
- AdventNet TMF EMS & NMS: Construido sobre AdventNet Web NMS, conforma una interfaz entre NML y EML. Este software facilita el rápido desarrollo de servicios de aplicaciones basados en el estándar TMF, tales como administración de Inventario, provisioning, conexiones, fallas y equipamiento para redes SONET / SDH, DWDM y ATM. Las aplicaciones que se basen en TMF EMS serán parametrizables con distintos dispositivos específicos, y pueden ser extendidas a nuevos dispositivos o tecnologías.

AdventNet diseño estos productos para brindar flexibilidad y soporte TMF a los nuevos EMS que surjan, o a los ya existentes. Los proveedores de equipamiento pueden proveer fácilmente una interfaz TMF para integrarse a esta tecnología, permitiendo el fácil agregado y quitado de dispositivos de la red.

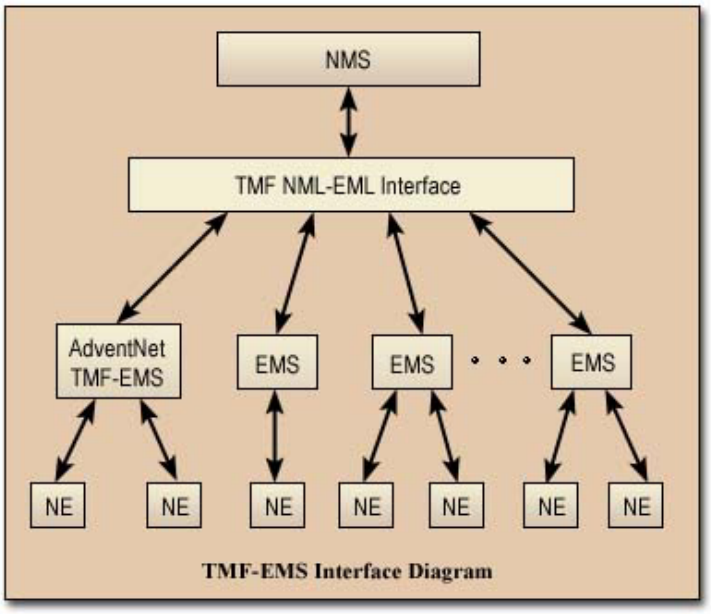


Figura 6-2 - Interfaces de AdventNet TMF-EMS

7. Desarrollo de software basado en componentes

Dado que uno de los objetivos planteados en el proyecto es la utilización del paradigma de desarrollo basado en componentes, se analizarán las plataformas más populares.

7.1. CORBA

"Common Object Request Broker Architecture" (CORBA) significa "Arquitectura Común de Corredores de Solicitudes de Objeto". Es una arquitectura para comunicar aplicaciones distribuidas, de la OMG. Actualmente se compara con DCOM (Microsoft), dependiendo de las aplicaciones a relacionar es el peso de cada arquitectura, pero debido a que el consorcio OMG esta formado por muchas corporaciones, el alcance de CORBA es mayor y abarca cualquier tipo de plataformas. [25] [26]

Fue diseñado para permitir que componentes se descubran entre sí, e interoperen en un bus de objeto. Asimismo, son similares a los objetos clásicos manejados en C++, Java, etc, soportando herencia, polimorfismo, etc. Las aplicaciones CORBA están compuestas de objetos, los cuales algunos serán objetos servidores, brindando servicios a otros objetos, posiblemente remotos, en calidad de clientes. La clase de objeto determina que servicio da el mismo, y a su vez una aplicación servidor sobre CORBA puede tener mas de una instancia de clase. Para esto se define un lenguaje estándar llamado IDL.

IDL (Lenguaje de Definición de Interfases)

Para cada clase de objeto servidor, se define una interfase en el lenguaje IDL, la cual estipula un vinculo contractual con clientes potenciales, de modo que todo cliente que invoque una operación del objeto debe utilizar esta especificación.

Es independiente del lenguaje, pero se mapea a los lenguajes mas utilizados. El mapeo también esta estandarizaos por la OMG, definiendo mapeos entre tipos, operaciones e interfases. De esta manera CORBA se vuelve sumamente portable.

ORB (Corredores de solicitud de objetos)

Permite que los objetos hagan solicitudes a otros objetos ubicados local o remotamente. Las especificación CORBA 1.1 solo determinaba el IDL y la API para la interfaz con el ORB. Luego, en CORBA 2.0 se especifica la interoperabilidad entre ORBs de distintos proveedores. Los clientes hacen referencia a los objetos servidores mediante un identificador reconocido y resuelto por el ORB; en el código cliente, se obtiene una referencia al objeto a partir del Servicio de nombres (se detalla mas adelante), o de algún archivo o base de datos donde haya sido almacenado y lo utiliza para invocar operaciones en el objeto al que corresponda, sin importar donde este ubicado (el objeto).

Servicios CORBA

Los servicios de objetos de CORBA son conjuntos de funcionalidades empaquetadas con interfases IDL. Son agregados y complementos a la funcionalidad del ORB. Estos servicios fueron considerados por la OMG como de interés para aplicaciones distribuidas, y se basan en desarrollos anteriores. Entre ellos encontramos servicios de Persistencia (Interfaz única para almacenamiento en bases relacionales, OO, etc), servicios de Nombre (permite que componentes en el ORB localicen objetos por su nombre), servicios de notificaciones, etc.

IIOP: (Protocolo de Internet para Intercomunicación de ORB)

El IIOP es prácticamente un TCP/IP con algunos agregados de mensajes definidos por CORBA que sirven como un protocolo común. Todo ORB que se declare observante de CORBA debe implementar nativamente el IIOP, o proveer algún puente que funcione como traductor de protocolos. La OMG esta analizando tener una representación canónica de objetos (CORF).

Actualmente la ultima versión de CORBA es la 3.0 (principios de 2003). En esta versión encontramos varios agregados al CORBA habitual.

CORBA 3.0 extiende el concepto de 'valuetype', permitiendo el envío de documentos XML, como objetos pasados por valor, dando un nivel de abstracción mas alto que facilita el trabajo del programador. Otro tema que aquí no se cubre es una mayor integración con EJB (se discutirá posteriormente). Otro aspecto que corresponde a esta versión de CORBA son las referencias a objetos con formato de URL, llamados corbalocs. También incorpora mecanismos adicionales para llamadas asincrónicas. Los clientes y objetos servidores pueden controlar orden por antigüedad o prioridad de las respuestas. Pueden asignar a las mismas prioridades y tiempos de vida, especificar políticas de enrutamiento, y especificar el conteo de hops al pasar por routers (no hay mucha experiencia en aplicaciones que lo utilicen).

Es complejo coordinar aplicaciones grandes, por ejemplo el orden en el cual se levantan los procesos. Asimismo, el uso de múltiples puertos TCP o UDP que son los usuales en cualquier firewall abierto, hace que muchas veces sea descartado su uso. Un problema que ha ido disminuyendo con el tiempo, es que los ORBs comerciales son algo lentos, ineficientes y poco escalables, en comparación con otras tecnologías cliente-servidor. También son poco tolerantes a fallas. La OMG sigue lanzando especificaciones para estandarizar y solucionar estos problemas, o estandariza características para ambientes particulares, por ejemplo de tiempo critico.

Otro problema es la falta de portabilidad del código de servidor. Este problema parece empeorar en lugar de mejorar, pues cada fabricante agrega nuevas facilidades a sus ORBs, sin dejar por eso de cumplir con las especificaciones OMG.

Java está muy vinculado con CORBA en varios aspectos. Java complementa algunos servicios de CORBA, en particular servicios de externalización y ciclos de vida, permitiendo transferir el comportamiento y el estado de un objeto a otro, con la serialización de objetos. También soporta "multithreading", "garbage collection" y manejo de excepciones, necesarios para CORBA, y hace distinción de interfaces de un mismo objeto. Todos aquellos proveedores de herramientas JAVA (Sun, IBM, Oracle, etc) están tratando de agregarle vínculos y dependencias para de esta manera "cazar" Java con CORBA.

Finalmente, a la hora de desarrollar aplicaciones de "Middleware" o distribuidas, debemos tener en cuenta que CORBA es una buena opción siempre y cuando no

estemos en ambientes problemáticos como los que se describieron anteriormente, y se aprovechen las siguientes características:

- La interoperabilidad entre lenguajes.
- Los mecanismos de escalabilidad en el servidor.
- Los servicios de objetos CORBA.

Por otro lado CORBA no se ha utilizado en todo sistema cliente-servidor, como anunciaban sus fabricantes y promotores en un tiempo, ni vivimos en una época guiada por los objetos distribuidos.

7.2. Java Beans y J2EE

Los Java Beans no son otra cosa que un framework de programación Java, el cual facilita la integración con otras aplicaciones de forma transparente. Esto es, poder tener componentes clientes y componentes servidores que se comuniquen sin que el usuario programador se preocupe si lo están haciendo en forma remota o local. [27] [28]

Los Java Beans nacieron con la idea de formar parte en las aplicaciones web, donde los clientes (Browser) se conectaban a un Web Server (o servidores de aplicaciones) y abrían una sesión, transaccionaban con ciertas aplicaciones y se desconectaban. Actualmente estos conceptos han crecido y ya están incluidos a la hora de integrar aplicaciones.

Hoy contamos con una generalización de beans, que son los EJB (Java Beans corporativos o empresariales). Éstos, son de 2 tipos, los EJB de sesión (los cuales su tiempo de vida es durante la sesión que el usuario maneja, y que únicamente son accedido por ese usuario) y los EJB de entidades (que son accedidos por varios clientes a la vez; los cuales a su vez pueden manejar ellos mismos su persistencia o que lo haga el objeto contenedor de los mismos).

En particular J2EE, es una nueva arquitectura diseñada por varios proveedores, tales como SUN, IBM, Oracle, HP, y varios más; la cual cumple con ciertos estándares para el manejo de EJBs de Entidad con su persistencia y están sumamente relacionados con CORBA (mencionado en el punto anterior). También se puso mucho énfasis en las operaciones transaccionales (JAAS), de esta manera encapsula mucho más la programación de aplicaciones distribuidas.

Con los EJB, podemos implementar aplicaciones integradas, en donde tenemos una parte en el cliente y otra parte en el servidor comunicándose de distintas maneras.

JAXM (API Java para mensajería XML)

La especificación JAXM, es una API Java concebida con la intención de facilitar la programación a la hora de comunicarse con mensajes XML. Implementa el protocolo SOAP lo que permite manejar mensajes que viajen sobre HTTP, SMTP, POP, e IMAP. Trata de estandarizar una metodología de comunicación XML para componentes JAVA a través de una capa de abstracción en lo que es actualmente un mar de sistemas de mensajería y RPC. Hay que aclarar que si bien los mensajes son creados a través e JAXM, existe una nueva especificación denominada SAAJ (SOAP con agregados para JAVA) la cual esta orientada a poblar los mensajes SOAP. El nacimiento de esta especificación se debe a la existencia de una tercera especificación llamada JAX-RPC (se discute mas adelante), la cual también se maneja

con mensajes SOAP y hasta hace poco se veía obligada a utilizar primitivas propias de JAXM solo con el fin de poblar los mensajes.

JAX-RPC

Es otra API para acceder a Servicios Web a través de RPC vía XML (basado en SOAP). Permite que un cliente basado en JAVA realice llamadas a métodos de cierto Servicio Web en un entorno distribuido y heterogéneo. Esta diseñado para esconder la complejidad de SOAP, de hecho cuando se utiliza una llamada a través de ésta API, la implementación convierte la invocación en un mensaje SOAP (SAAJ mediante), y de forma análoga cuando se recibe un mensaje SOAP, JAX-RPC decodifica el mensaje y lo procesa. De esta forma facilita el proceso de programación de Servicio Web. JAX-RPC forma parte de J2EE y de Java WSDP (Paquete de Desarrollo de Servicio Web para Java).

JMS (Servicio de Mensajes de JAVA)

Solución de SUN para los sistemas de mensajeria. JMS se sitúa como Middleware en medio de 2 aplicaciones que no estén en la misma maquina necesariamente. En entornos cliente-servidor, cuando la aplicación A quiere comunicarse con la B, A necesita saber donde esta B (dirección IP, etc) y que B esté escuchando en ese momento. JMS permite la comunicación diferida, esto es que A puede enviar un mensaje a B y cuando B se conecta, este recibe el mensaje. La especificación 2.0 de EJB introdujo un nuevo tipo de EJB llamado MDB (Bean orientado a mensajes). Hasta ahora para integrar una aplicación J2EE con JMS había que programar un consumidor de mensajes, y manejarlos a través de un bean de sesión. Ahora la propia aplicación J2EE recibe el mensaje y ella misma los trata a su manera.

J2EE CONECTOR

La arquitectura del J2EE Conector permite conectar varias aplicaciones con sistemas de información heterogéneos. Cada sistema de información propietario, debe de proporcionar un adaptador estándar para su sistema de información, de esta manera se brinda conectividad con cualquier aplicación J2EE que quiera acceder a dicho sistema. Estos adaptadores se pueden incorporar como “plugins” a los servidores de aplicaciones.

7.3. Microsoft .Net

Microsoft .Net es una tecnología reciente pero muy utilizada, que se comporta de manera similar a Java, pero es orientado a plataformas Microsoft. La idea de esta tecnología es facilitar al programador visual (Lenguajes Microsoft, tales como Visual Basic, Visual J++, C#, etc) las tareas de programar en un Lenguaje y luego reprogramar para otro. Inicialmente los lenguajes soportados eran únicamente los de Microsoft, pero como la especificación (Common Language Specification) es libre se agregaron muchos otros. Esta tecnología permite generar código en cualquier de estos lenguaje y luego precompilarlo en .Net para luego generar un PRE-ejecutable (MSIL) que corriendo sobre el framework que .Net proporciona se genera el ejecutable real. [29]

Hoy por hoy el código del framework es libre, lo que implica que muchos programadores están tratando de adaptar dicho código para que corra en plataformas Linux.

A nivel de comunicaciones, la idea de Microsoft es integrar sus aplicaciones mediante DCOM (diferencia de CORBA).

SEGUNDA PARTE: El Desarrollo

8. La Solución

En esta segunda parte del informe nos concentraremos en el resultado más importante de este proyecto que es el desarrollo del componente de Gestión de Inventario de Red.

8.1. Entorno de Trabajo

Aunque el resultado de nuestro trabajo tenía objetivos concretos, había un objetivo superior que era la implementación entre este proyecto y otro proyecto simultaneo, de un sistema capaz de realizar la "función de configuración" mediante un módulo de aprovisionamiento. Estos proyectos tenían que dejar como resultado un sistema modularizado y extensible como base para futuros desarrollos en el área de Gestión de Red, en nuestra Universidad.

Inicialmente se trabajo en forma paralela y con puestas en común sobre el macro diseño de la solución conjunta. A priori se desconocía la macro arquitectura por lo tanto tampoco se tenía una división clara de componentes entre proyectos.

Luego se observó la existencia de cuatro grandes componentes como solución para el objetivo de aprovisionamiento. Se establecieron dos capas. En la primera (NMS) se tiene el componente "Provisioning", único que brinda servicios a capas superiores, y el componente "Inventory" que presta servicios internos a esta capa. La segunda capa (EMS) consta de dos componentes principales, el MSLN Manager y MEM que describiremos en sucesivas secciones.

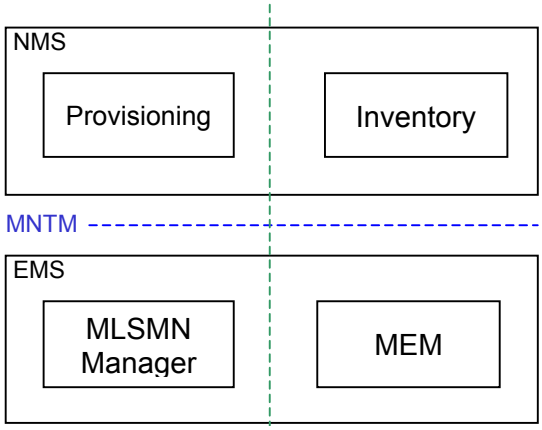


Figura 8-1- División entre proyectos

En la solución conjunta, hubo un considerable atraso debido a las dificultades en dividir el proyecto. Se dudó entre hacer una partición por capas utilizando el estándar MTNM o hacer una partición vertical según se había pensado originalmente. Se evaluaron múltiples factores y finalmente se eligió mantener la partición vertical para que ambos proyectos tuvieran participación en capa de red y capa de elemento. Quedando el Inventario en nuestro proyecto, junto al MEM.

Como consecuencia aumentó el acoplamiento entre ambos proyectos y esta decisión tuvo gran impacto en el resultado final, que analizaremos en la última parte.

8.2. Proyecto mitiNum

Como mencionamos, nuestro proyecto se desarrolló simultáneamente con otro proyecto llamado "Gestión de Red basada en componentes", este grupo se autodenominó mitiNum (por parecerse a MTNM, **mi**t**N**u**m**), nombre que utilizaremos en adelante para referirnos a este.

Dicho proyecto (de ahora en más mitiNum) tuvo como objetivo desde el comienzo proponer una arquitectura de Gestión de Red basada en componentes e implementar el componente básico de Aprovisionamiento de Conectividad (Gestión de Configuración).

Debido a la división resultante del diseño, el componente de "Provisioning" desarrollado por mitiNum se convirtió en el "cliente" del inventario a desarrollar.

8.3. Herramientas Utilizadas

En esta sección enumeraremos brevemente las herramientas utilizadas y los motivos que nos llevaron a sus elecciones.

- Como lenguaje de programación se eligió Java por ser un lenguaje moderno, orientado a objetos, de distribución libre y muy utilizado en la industria. No hubo dudas al respecto.
- Como plataforma para el desarrollo basada en componentes se eligió CORBA. Tampoco hubo dudas al respecto, primero por que es un estándar de la industria de amplia difusión, además existen implementaciones libres y fundamentalmente porque las IDLs (descripción de las interfaces) que pensábamos utilizar estaban definidas para esta arquitectura. Respecto a la implementación de ORB a utilizar se llegó a un acuerdo con el *proyecto mitiNum*. Se consideró primero la utilización de JacORB pero se descartó en primera instancia pues su versión 1.4.1 (la versión más estable hasta ese momento) no incluía servicio de notificaciones. Luego se utilizó JavaORB pero se descartó porque su funcionamiento no era del todo correcto y el proyecto de JavaORB parecía estar abandonado. Después comenzamos a utilizar la versión JacORB 2.0 (beta 3) recién salida con muy buenos resultados. Finalmente cuando estuvo disponible utilizamos el JacORB 2.1.
- El proyecto debía tener una persistencia de una gran cantidad de datos por lo que necesitábamos un manejador de Base de Datos. Aunque consideramos la utilización de una base de datos orientada a objetos, resolvimos utilizar una base de datos relacional, por su mayor familiaridad y disponibilidad. Como se pretendía tener independencia respecto a la base de datos elegida, utilizamos JDBC como mecanismo de acceso a la misma, ampliaremos más adelante. Durante el proyecto se utilizaron diferentes bases: Access, SQL Server 2000, DB2 v7.2 para Linux y Windows, e incluso mitiNum utilizó Postgre sobre Linux.
- Respecto al Sistema Operativo, todas las herramientas elegidas son independientes del S.O. por lo que se trabajó en Windows y en Linux indistintamente a lo largo del proyecto.
- JBuilder se usó como editor de código y para generar algunas interfaces gráficas, pero el proyecto final es independiente de esta herramienta y no utiliza ningún paquete propietario de Borland.

9. Arquitectura de la Solución

Ahora estudiaremos a nivel macro la solución, introduciéndonos brevemente en los componentes que la conforman, sus interfaces e interacciones. Luego en los capítulos siguientes analizaremos detalladamente los componentes desarrollados en nuestro proyecto, el MEM y el Inventory.

En el diagrama que sigue podemos ver las dos capas de la pirámide TMN que alcanza el sistema, sus componentes y la interacción con las diferentes interfaces.

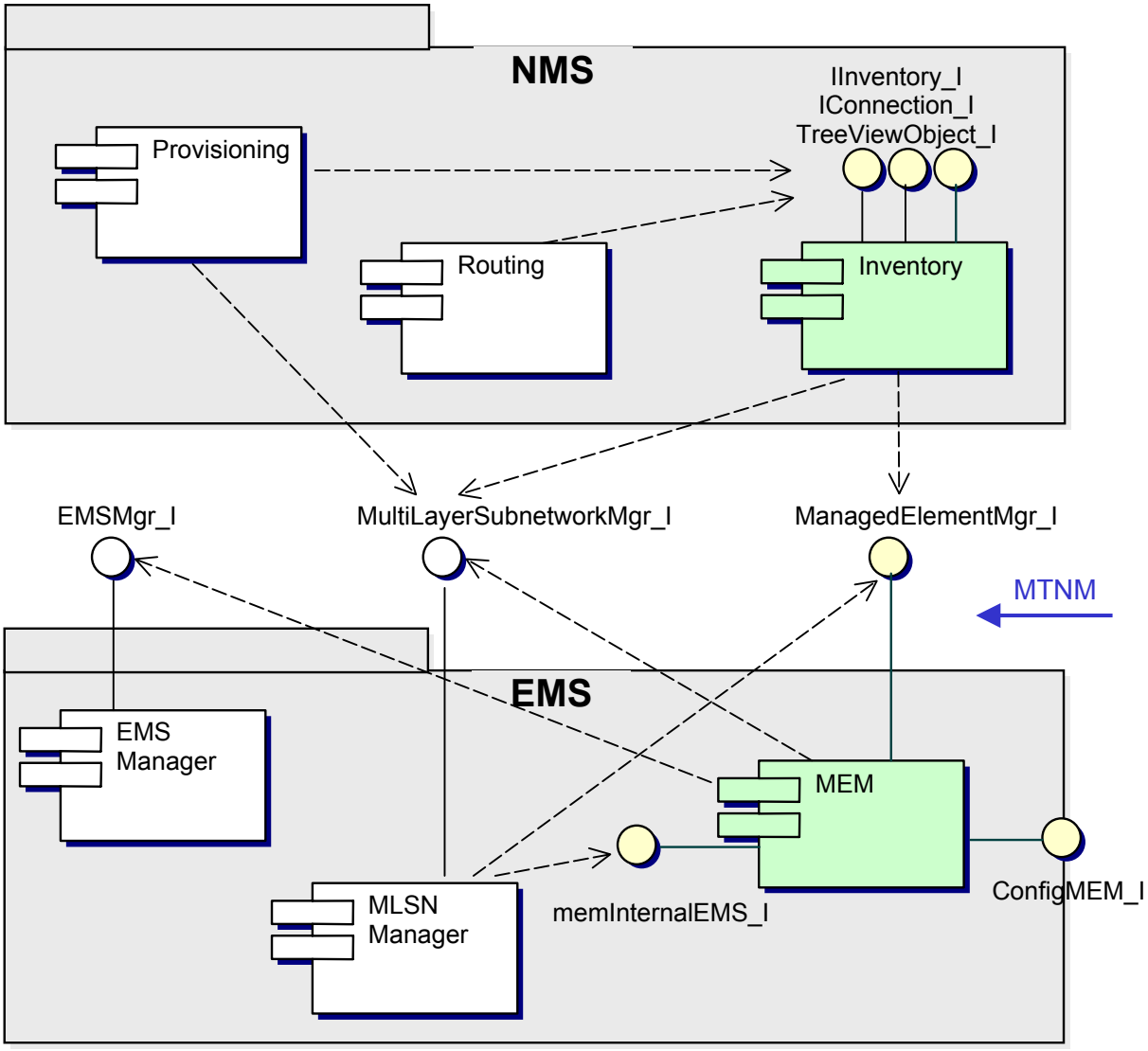


Figura 9-1 - Arquitectura de la Solución

Primero que nada podemos observar las dos grandes capas:

NMS – Network Management System (gestión de red)

EMS – Element Management System (gestión de elementos)

La primera de estas capas es la que se encarga de brindar los servicios a capas superiores (SML), mientras que la segunda es la encargada de “hablar” con los dispositivos. Para comunicar estas capas utilizamos únicamente el estándar MTNM del TM Forum. Los componentes en verde y las interfaces en amarillo son los desarrollados en nuestro proyecto.

El resto de los módulos fueron implementados por mitiNum exceptuando el componente Routing que se dejo “hueco” para futuros proyectos. El “EMS Manager” es un componente responsable del manejo de los atributos básicos del EMS, las TL y las SN. El “MLSN Manager” se encarga de gestionar las SNC. Estos conceptos serán definidos posteriormente.

Debido al hecho de que todas las interfaces utilicen CORBA permite correr los componentes y los clientes en máquinas diferentes.

10. MEM

El “Managed Element Manager” (MEM) es el componente que se desarrolló en capa de elemento (EML). Éste es el encargado de comunicarse directamente con los dispositivos (elementos) de red y hacer de intermediario con la capa de red (NML).

El diseño e implementación de este componente esta influenciado fuertemente por el estándar MTNM y casi todas las estructuras de datos esta basada en los requerimientos de éste.

10.1. Funcionamiento

Básicamente el MEM administra una colección de MEs (Manged Elements) que son los dispositivos de red y actúa como intermediario con éstos. Recibe requerimientos para un ME específico y se los pasa a éste.

Dado que los dispositivos de red son muy variados y nos interesa lograr extensibilidad e independencia de la tecnologíal, creamos unas clases llamadas drivers que son las que realmente “hablan” con el dispositivo real.

Los drivers contienen la lógica específica para cada dispositivo (ME) y son diseñados para cada tipo de ME.

Al levantarse el MEM lee de su persistencia todos los ME agregados hasta el momento, también lee que driver le corresponde a cada uno y la configuración de éste. De esta manera se instancia una clase driver para cada dispositivo.

El diseño de los drivers fue pensado para dejarle el mínimo de funcionalidades, dejando únicamente lo que no puede ser implementado sin conocer el dispositivo y simplificando el agregado de nuevos drivers. Más adelante veremos el mecanismo para crear nuevos drivers.

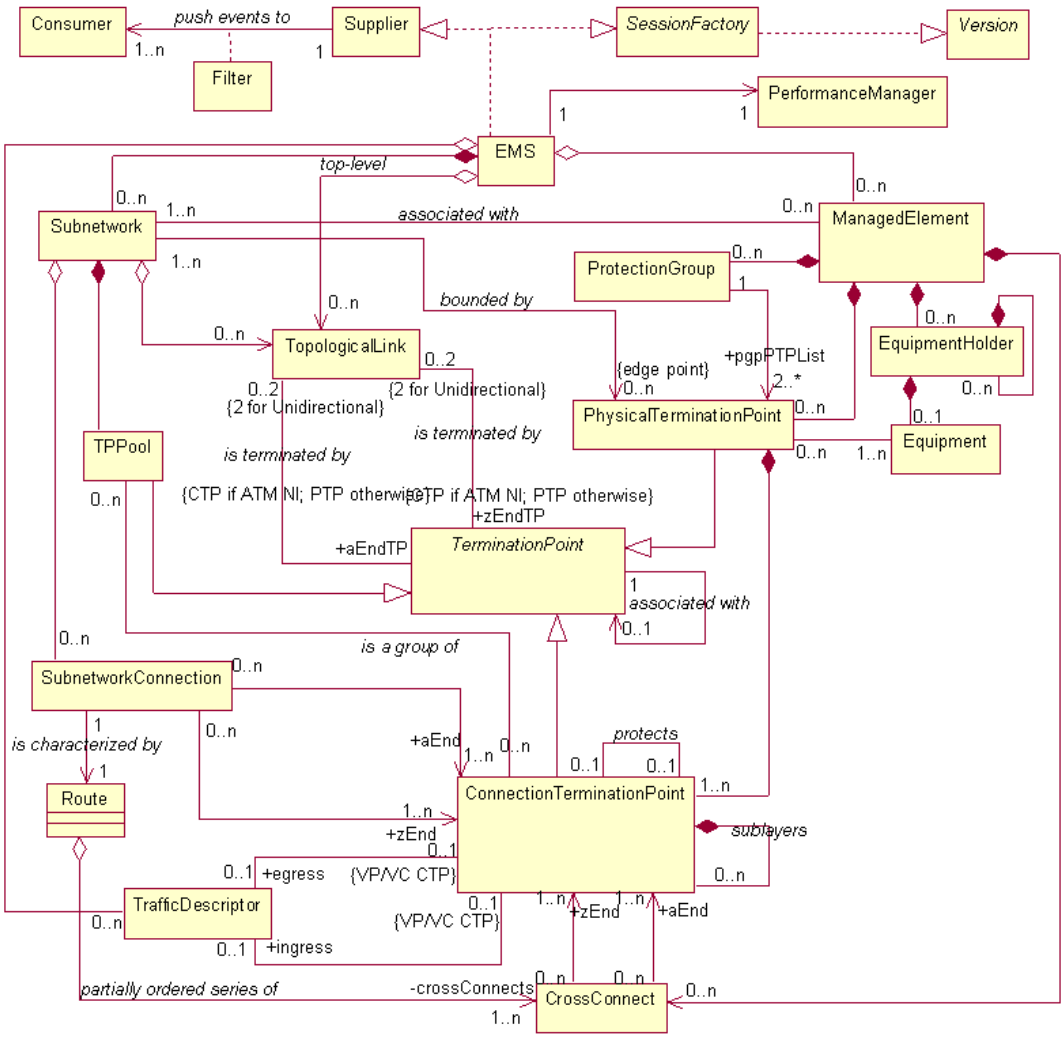
Luego de levantado el MEM, mediante una interfaz gráfica (que utiliza una interfaz CORBA) o una interfaz CORBA es posible agregar o quitar MEs y configurar los parámetros específicos de su driver.

10.2. Modelo de Datos MTNM

Dado que el principal objetivo del componente MEM es proveer servicios a la capa de red mediante la interfaz MTNM, veremos brevemente el modelo de datos MTNM.

10.2.1. Diagrama de Clases

El siguiente diagrama de clases fue extraído de la documentación MTNM. [21]



De las clases del diagrama anterior destacaremos las más importantes y las relacionadas directamente al componente MEM:

- **EMS**: Es el sistema al cual pertenecen los elementos administrados. El componente desarrollado tendrá una instancia por cada EMS. A los efectos de este proyecto en todos los casos se trabajó con un único EMS.
- **ManagedElement (ME)**: es una abstracción para representar elementos de red a través de la interfaz MTNM. Generalmente los asociaremos a un dispositivo.
- **Subnetwork (SN o MLSN)**: es una abstracción de un conjunto de dispositivos (MEs) interconectados entre sí y pertenecientes a un EMS. Un ME puede pertenecer a más de una Subnetwork, pero no en el mismo Layer Rate. El concepto de Layer Rate es importante y se refiere a una capa de una determinada velocidad y protocolo. Las Subnetwork también son llamadas Multi Layer Subnetwork (MLSN).
- **TerminationPoint (TP)**: es el punto final actual o potencial de un enlace. Es decir es un punto donde comienza o termina un enlace que existe o puede existir.

- **ConnectionTerminationPoint (CTP)**: es un tipo de TP que tiene información de conexión y estado. En particular son los extremos de SNCs y XCs que veremos más adelante. Otra particularidad es que tienen un único Layer Rate.
- **PhysicalTerminationPoin (PTP)**: son TPs que a diferencia de los CTP no tienen información de conexión y pueden manejar varios Layer Rate. No pueden ser extremos de SNCs o XCs, pero si de TLs. Cada PTP puede contener varios CTPs.
- **Topological Link (TL)**: es un enlace entre dos puntos de dos MEs diferentes, el Layer Rate es el menor de los dos puntos que conecta.
- **Subnetwork Connection (SNC)**: es una conexión que relaciona varios CTP y provee conexión transparente punto a punto a través de una Subnetwork. La creación y activación de estas SNC, es el principal servicio que brinda la capa de elementos al módulo de aprovisionamiento. Es decir que se podrán crear servicios en la capa de red creando SNCs en la capa de elementos. Aunque el MEM no gestiona las SNC están relacionadas con los objetos que éste maneja.
- **Cross-Connect (XC)**: representa una conexión física dentro de un ME, es decir que un XC se asocia a un ME. Una SNC entonces estará formada por una lista de XC saltando de ME a través de TL. El componente MEM será el que cree físicamente los XC dentro del driver de cada ME.

En la siguiente figura podemos observar una representación gráfica de las clases recién mencionadas.

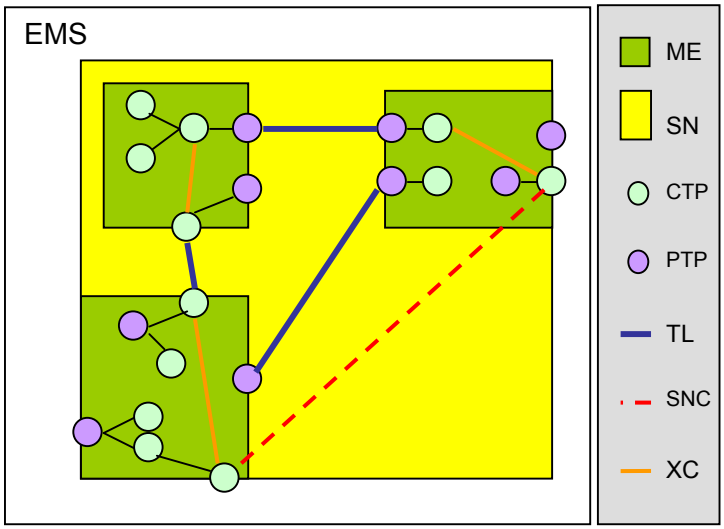


Figura 10-1- Representación gráfica del modelo de datos de MTNM

10.2.2. Nombres de Objetos MTNM

Algo que nos ayudará a entender la jerarquía de los objetos MTNM es su manera de nombrarlos. Los objetos MTNM se nombran con una lista de pares (nombre, valor). A continuación enumeramos la composición de nombres de los objetos más importantes:

- **EMS**
  1. name="EMS";value="*CompanyName*/*EMSname*"
- **Subnetwork**
  1. name="EMS";value="CompanyName/EMSname"
  2. name="MultiLayerSubnetwork";value="*SubnetworkName*"
- **SubnetworkConnection**
  1. name="EMS";value="CompanyName/EMSname"
  2. name="MultiLayerSubnetwork";value="SubnetworkName"
  3. name="SubnetworkConnection";value="*SubnetworkConnectionName*"
- **ManagedElement**
  1. name="EMS";value="CompanyName/EMSname"
  2. name="ManagedElement";value="*ManagedElementName*"
- **TopologicalLink**
  1. name="EMS";value="CompanyName/EMSname"
  2. name="TopologicalLink";value="*TopologicalLinkName*"
- **PTP**
  1. name="EMS";value="CompanyName/EMSname"
  2. name="ManagedElement";value="ManagedElementName"
  3. name="PTP";value="*PTPName*"
- **CTP**
  1. name="EMS";value="CompanyName/EMSname"
  2. name="ManagedElement";value="ManagedElementName"
  3. name="PTP";value="PTPName"
  4. name ="CTP";value="*CTPName*”

10.2.3. Ejemplos de MEs y TPs

Aunque durante el proyecto no tuvimos la oportunidad de trabajar con dispositivos reales, nos pareció importante incluir algunos ejemplos que nos permitan mapear las abstracciones de los objetos MTNM, con un dispositivo real.

En el siguiente ejemplo extraído de la documentación del estándar MTNM, podemos ver una interconexión de 3 dispositivos de red.

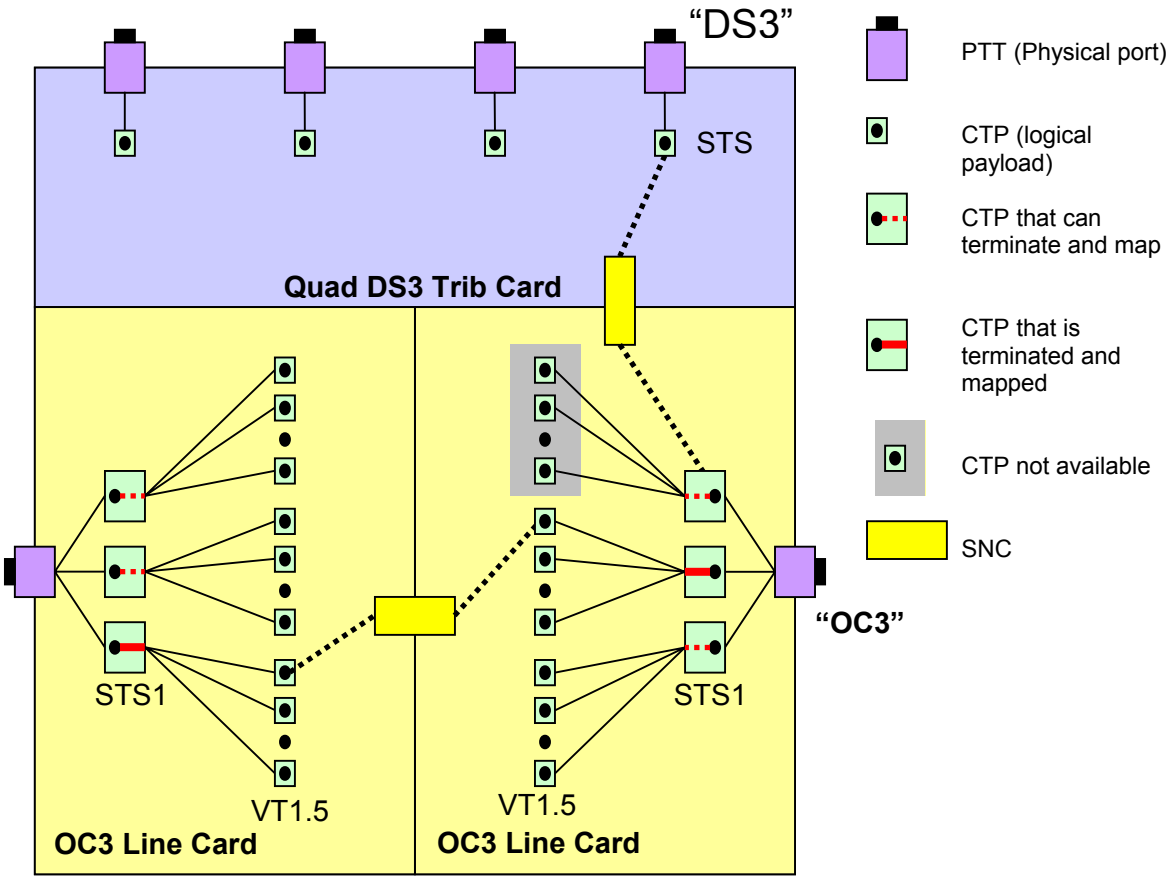
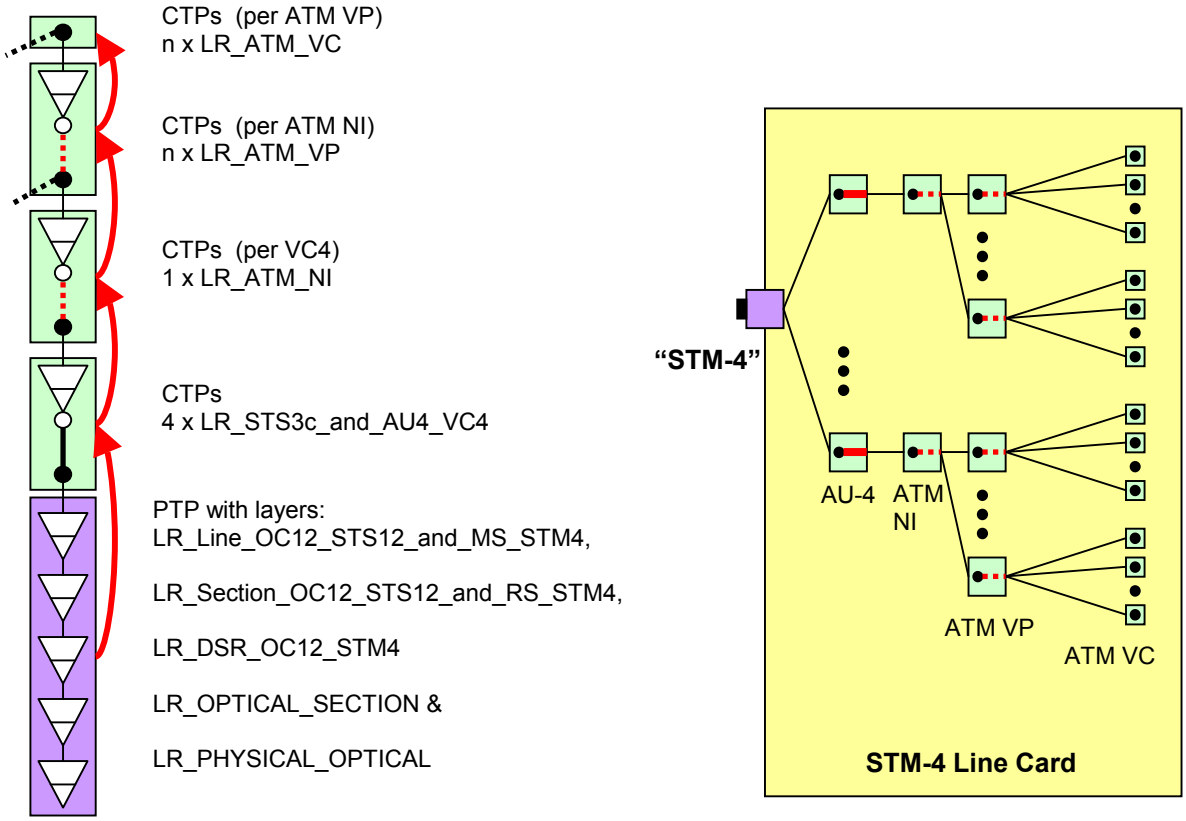


Figura 10-2 - Ejemplo de dispositivos interconectados



El siguiente dibujo también extraído de la documentación de MTNM nos permite visualizar las capas de un puerto STM-4.

Figura 10-3 - Ejemplo puerto STM-4

Es importante notar como los CTPs se encadenan en forma de árbol para un TP. También podemos observar como ya habíamos mencionado que los PTPs trabajan en varios "Layer Rates" mientras que los CTPs solamente en uno.

Para profundizar en este tema se recomienda la lectura del Apéndice C - "FUNCTIONAL MODELING CONCEPTS" de [22].

10.3. Interfaces

Como forma de comprender un poco más el componente MEM, que mejor manera que estudiar sus interfaces.

El MEM implementa tres interfaces (ver figura 9.1) que fueron desarrolladas utilizando CORBA. La primera implementa la parte del estándar MTNM de este componente. La segunda permite realizar operaciones internas a la capa de elementos, que será consumida por el componente MLSN Manager desarrollado por mitiNum. La última permite que se puedan agregar, quitar y configurar dispositivos.

A continuación describiremos cada una de estas interfaces sin entrar en el detalle de cada método, para esto se deberá recurrir a la documentación de programa.

10.3.1. ManagedElementMgr_I

Esta interfaz es la única de las tres interfaces que provee servicios a la capa de red y esta definida en el estándar MTNM versión 2 en el documento “TMF 814 - Multi-Technology Network Management Solution Set”. [22]

Esta interfaz se especifica en la idl (interface definition language) que se encuentra en el archivo *managedElementManager.idl* y define 18 métodos. No detallaremos los 18 métodos (quien esté interesado deberá referirse al estándar) pero será dada una idea de éstos.

El método *getAllActiveAlarms* fue el único no implementado por quedar fuera del alcance del proyecto. Entre los 17 métodos restantes tenemos 16 de consulta y 1 único método para setear valores.

Entre los métodos de consulta podemos consultar por cualquier o todos los ME (Managed Element), por los PTP (Physical Termination Points) de un ME o por los CTP (Connection Termination Points) de un PTP. También podemos a través del método *getContainingSubnetworkNames* obtener los nombres de las subredes (MLSNs) a las que un ME pertenece.

El único método que se tiene capaz de setear algún valor desde la capa de red es el setTPData que permite setear parámetros adicionales a PTPs o CTPs según se detalla en el estándar MTN.

10.3.2. memInternalEMS_I

Esta interfaz es la que permite a otros componentes de la capa de elemento (EMS) crear las conexiones (Cross Connects) entre diferentes puntos (Termination Points) y así poder crear las Subnetwork Connections. En particular esta interfaz es consumida por el componente MLSM (Multi Layer Subnetwork Manager).

Esta interfaz consta de dos métodos *creatCrossConnections* para crear XC (Cross Connects) y *deleteCrossConnections* para borrarlas. Los XC no se pueden crear y borran de a uno, sino que se crean por grupo para un ME y una SNC (Subnetwork Connection). Esto se resolvió de esta manera porque no hay nombres para los XC en el estándar MNTM, por lo tanto se identifica al grupo por la SNC que implementan en un ME.

Para ver el detalle de cada método referirse a la documentación incluida en el archivo idl “memInternalEMS.idl”.

10.3.3. ConfigMEM_I

La última de las tres interfaces es la que permite, agregar, quitar, modificar, activar y configurar cada ME.

Cada vez que se agrega o quita un ME se genera un evento que es capturado por el NMS para actualizar el inventario.

Esta interfaz es la que permite a la herramienta de configuración, de nombre java: gesinv.ems.mem.configApl.ConfigApl, interactuar con el MEM en tiempo de ejecución.

El hecho de haber creado una interfaz CORBA para agregar o quitar dispositivos dinámicamente, también se debió a la posibilidad de que en un futuro se pudiera anexar un módulo de descubrimiento de dispositivos, que corriera en una red específica.

10.3.3.1. Herramienta de Configuración

Existe una herramienta de configuración llamada ConfigApl que consume la interfaz ConfigMEM_I y permite especificar los dispositivos del EMS.

Para utilizar esta herramienta, primero que nada se debe elegir la opción iniciar sesión en el menú “Archivo”. Inmediatamente realizado esto se despliega una lista con los MEs (dispositivos) que posee el EMS y son administrados por el componente MEM, como se muestra en la siguiente figura.

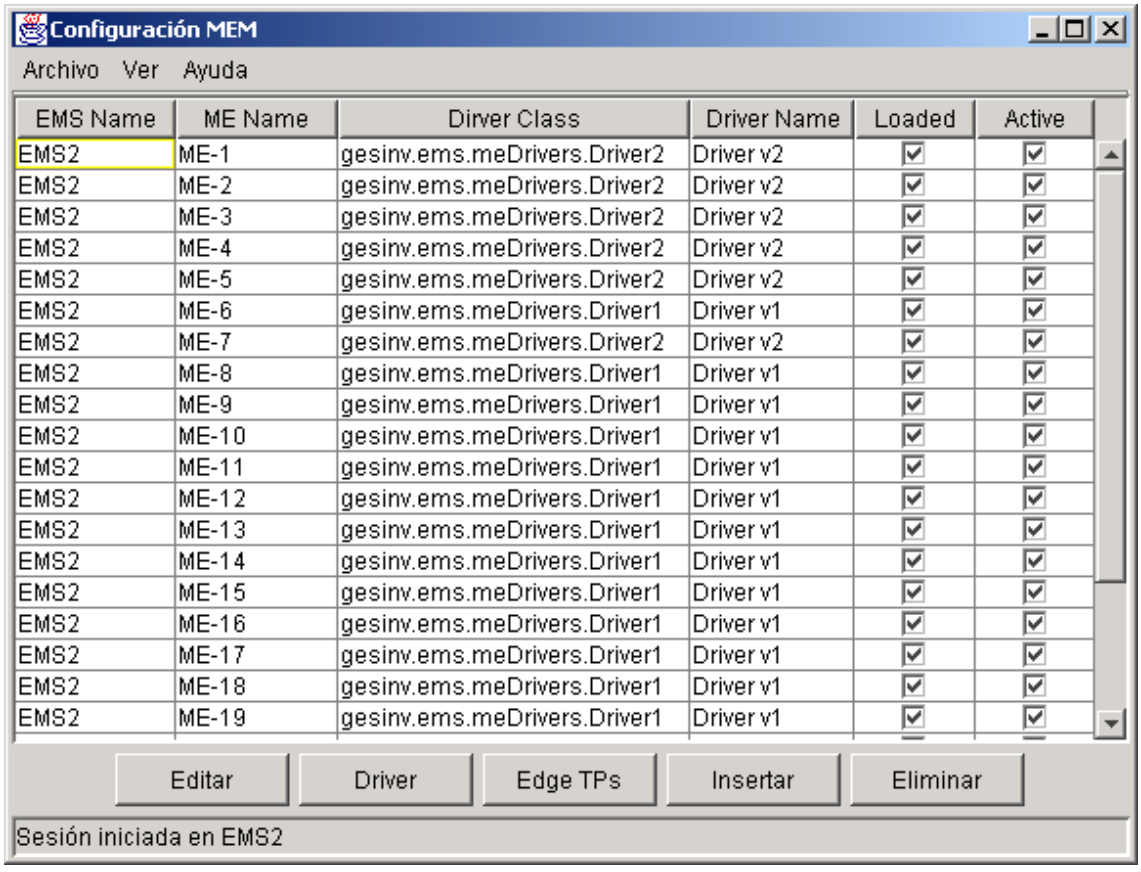


Figura 10-4 - Herramienta de configuración del MEM

Luego de seleccionado un ME de la lista podemos editarlo con el botón “Editar”.

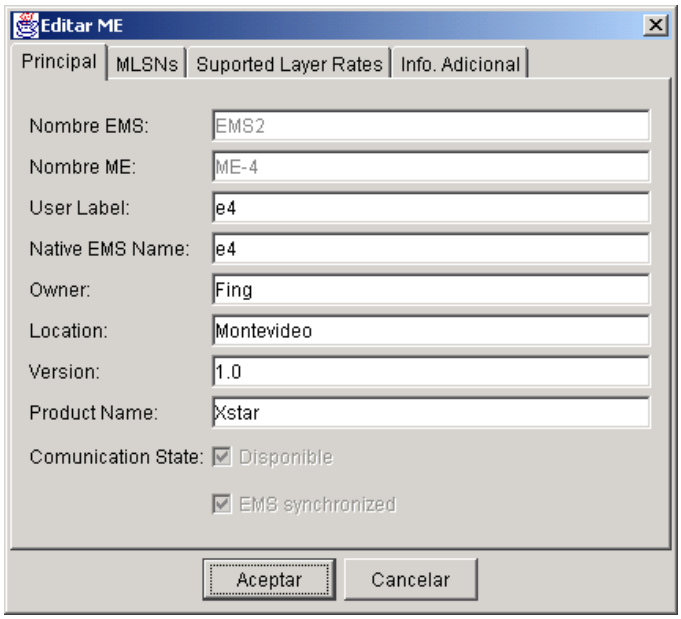


Figura 10-5 - Edición de un ME

Desde la ventana inicial, también podemos desactivar o activar un ME haciendo clic en el check box “Active”, o bien podemos cambiar el driver si el ME no está activo, escribiendo sobre el campo “Driver Class” la clase que implementa el driver.

Con el botón “Driver” podemos acceder a la configuración específica del driver cargado.

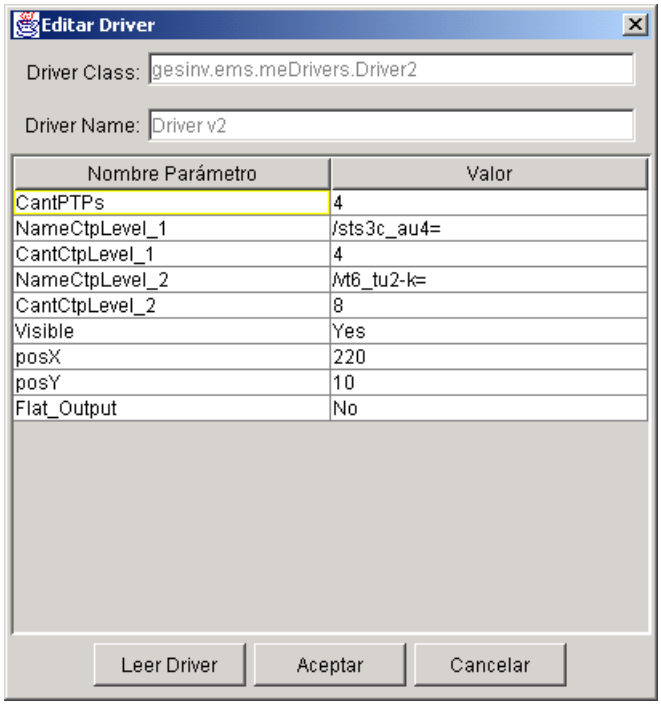


Figura 10-6 - Configuración de un Driver de un ME

Una vez que entramos a la ventana de configuración de un driver, vemos la listar de parámetros configurados por última vez. Pero si cambiamos de versión de driver, por ejemplo es posible que se necesiten más o menor parámetros para configurar. Entonces apretando el botón “Leer Driver” se le solicita al driver que indique los parámetros necesarios para ser configurado. De esta manera en la lista de la pantalla se quitan los parámetros sobrantes y se agregan los faltantes sin valor.

Que los TP sean edge (de borde) o no, es un atributo del mismo. Aunque los TPs son administrados por el driver, el dispositivo no tiene porqué conocer la topología de red. Entonces la configuración de este atributo se agregó en la herramienta de configuración.

Desde la ventana inicial con el botón “Edge TPs” saltamos a otra ventana donde podemos indicar para cada dispositivo que TPs son edge. Para esto se debe tener cargado y activado el driver.

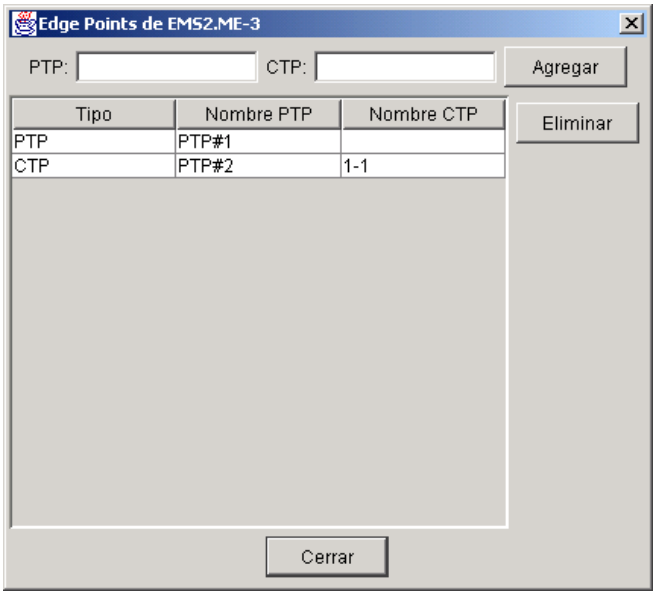


Figura 10-7 - Configuración Edge TPs

10.4. Diseño Interno

En esta sección describiremos brevemente las grandes clases del componente MEM y su visibilidad.

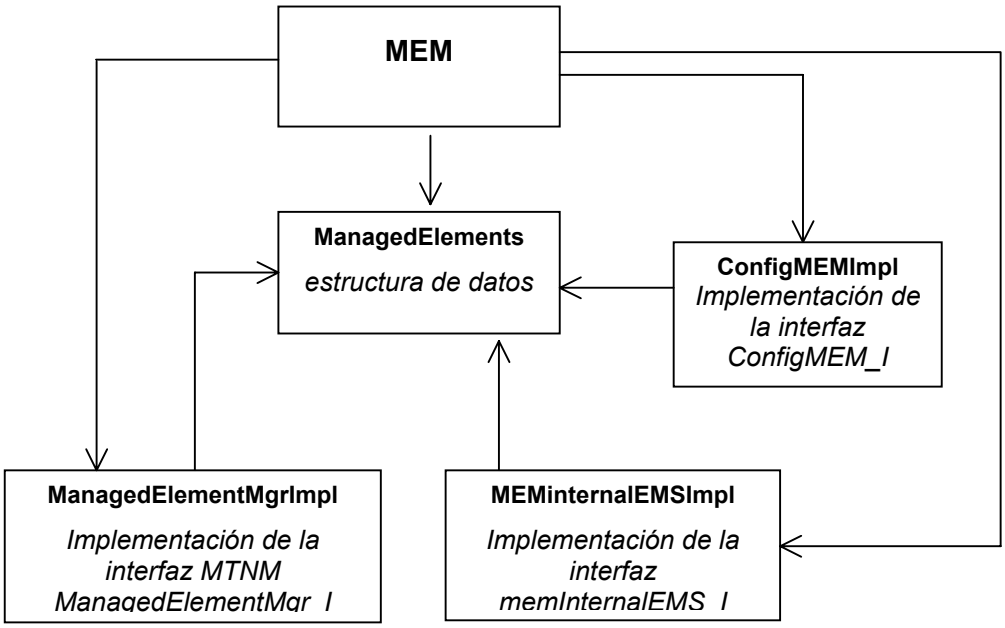


Figura 10-8- Diseño de clases del MEM

La clase MEM es la clase principal y es instanciada por el programa Loader implementado por el proyecto mitiNum. Esta clase levanta la estructura de datos y las interfaces pasándole a estas últimas la referencia a la estructura de datos. Luego registra estas interfaces en el servicio de nombres de CORBA. La función de esta clase cumple este objetivo, pero es posible dado el diseño crear otras instancias de las tres interfaces. Esto es posible pues ninguna de estas posee datos, sino simplemente lógica y acceden a una única estructura de datos.

Esto es importante para futuras mejoras en el EMS pudiéndose crear una instancia para cada cliente del EMS que lo solicite con los mecanismos previstos en MTNM, en lugar de acceder al servicio de nombres. El componente que maneja las sesiones (emsSessionFactory) desarrollado por el proyecto mitiNum obtiene las referencias a las fachadas (interfaces registradas en CORBA) del servicio de nombres. Este manejo de sesiones puede ser burlado pues es posible ir al servicio de nombres sin pedir sesión. Como solución el emsSessionFactory podría tener las referencias a las fachadas desde el inicio o mejor aún fabricar una instancia para cada sesión.

Entre las otras clases principales del diagrama tenemos las tres interfaces cuyo funcionamiento se describió anteriormente y la estructura de datos contenida en la clase ManagedElements que veremos a continuación.

10.5. Estructuras de Datos

En el MEM se modelan MEs, nombres de MLSNs, PTPs, CTPs y XCs. Todos estos objetos se agrupan en un único objeto *ManagedElements* que representa la colección de MEs.

La siguiente ilustración representa gráficamente la estructura de datos del MEM.

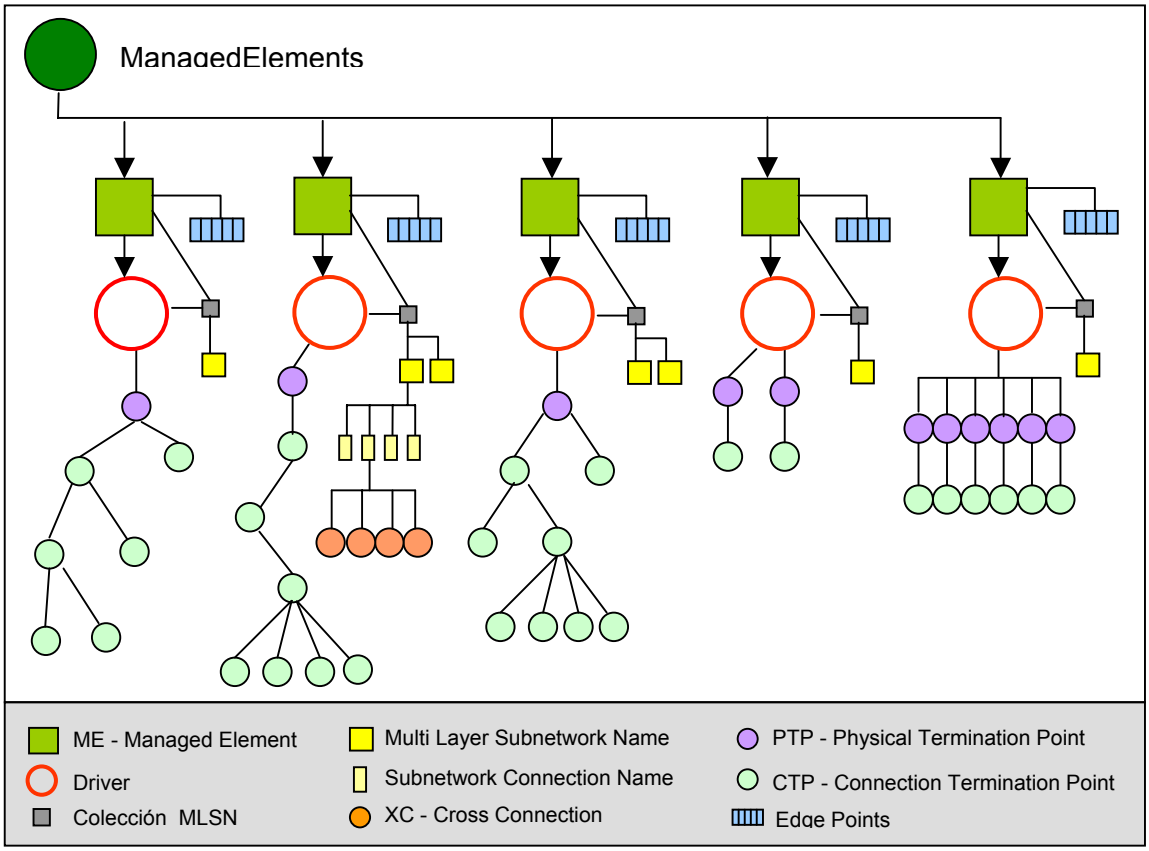


Figura 10-9 - Estructura de Datos

Cada ME tiene un driver que a pesar de ser abstracto y de no conocerse su implementación, maneja una colección de PTPs. A su vez cada PTP tiene un árbol de CTPs.

Por otro lado cada ME tiene además una colección de Subnetworks, también llamadas MLSN. Esta colección de nombres de MLSN es administrada por el ME, pero también se le pasa una referencia al driver, para que pueda crear las XCs.

Cada MLSN tiene una colección de nombres SNC, que como vimos, sirven para nombrar las XC que pasan por un ME.

10.5.1. Iteradores

En el estándar MTNM existen iteradores para todos los objetos TPs, MEs, XCs. SNCs, SNc y Nombres.

Es importante la mención de los iteradores, primero porque están regados en la mayoría de los métodos y segundo porque el diseño e implementación del MEM apunta a que el desarrollo de los drivers esté libre del manejo de estos.

En todos los métodos de MTNM que se pueden devolver varios objetos se pasa un parámetro con la cantidad máxima a devolver. Luego, si la cantidad es mayor que el máximo se devuelve un iterador que permite acceder al resto de los objetos.

A la hora de implementar los iteradores surgen dos alternativas. La primera es que al solicitar un iterador se copien en memoria los objetos que éste irá devolviendo. La segunda alternativa es que el iterador no contenga en si los datos a devolver, sino que contenga un puntero al último (o el próximo) devuelto y alguna información adicional por ejemplo para filtrar los objetos. Es importante destacar que ambas alternativas tienen semántica diferente, pues en el segundo caso la cantidad de elementos a devolver puede variar a lo largo de la iteración.

Para la mayoría de los iteradores utilizamos la primera alternativa por su simpleza y practicidad. Únicamente para los iteradores de CTPs y nombres de CTPs se utilizó la segunda opción. Esto se debió a que en algunos ejemplos de la documentación de MTNM había casos en que un PTP contenía más de 200 millones de CTPs.

10.5.2. CTPTree

Como vimos cada PTP tiene un conjunto de CTPs, organizados jerárquicamente en forma de árbol. Esta organización se representa en la clase CTPTree.

Esta clase es abstracta y el paquete desarrollado provee una implementación llamada GenericCTPTree. La implementación de GenericCTPTree guarda en memoria un árbol de CTPs.

La razón de que la clase CTPTree sea abstracta radica en que las implementaciones de drivers de dispositivos con muchos CTPs, puedan realizar implementaciones más eficientes de acuerdo al comportamiento del dispositivo. Por ejemplo, cuando mencionamos un dispositivo con más de 200 millones de CTPs potenciales, el dispositivo que lo gestione podría tener una implementación de CTP que solo materializara en memoria los CTPs en uso.

La interfaz que tiene CTPTree está pensada en combinación con el CTPIterator para implementar iteradores sobre estas estructuras, aunque no estén almacenadas en memoria.

10.6. Persistencia

Alternativas

El componente MEM necesita recordar su estado entre sucesivas ejecuciones. Para esto se plantearon varias alternativas.

Dado que uno de los objetivos del proyecto era diseñar e implementar un inventario (persistencia) de red y mirando todo el proyecto como un gran sistema de gestión de red, parecía natural la utilización de este inventario como persistencia para todo el sistema. Esta opción no fue la elegida, pues desde el comienzo nos propusimos la utilización del estándar MTNM y éste partía el proyecto en dos capas. Entonces

hubiéramos violado la independencia del diseño en capas, si una capa inferior utilizara servicios internos de una capa superior.

Con esto en vista empezamos a trabajar en una persistencia propia del MEM. Luego de bastante avanzado el proyecto, en una reunión con el otro proyecto (proyecto mitiNum) nos planteamos la posibilidad de hacer una persistencia común y un componente que la administrara para toda la capa de elemento, pues había mucha interrelación entre los objetos manejados por ambos proyectos.

Esta idea fue descartada pues estábamos muy avanzados en el proyecto y se ponía en riesgo la culminación en fecha. Además agregaba demasiada interacción e interdependencia para ambos proyectos.

Finalmente se optó que cada componente de la capa de elementos tuviera su propia persistencia y se chequeará la consistencia entre los componentes mediante los métodos previstos en las interfaces existentes.

Implementación

Uno de los objetivos de este componente era que fuera independiente de la tecnología. Esto se logro mediante la incorporación de “drivers” que pueden tener implementaciones diferentes para cada dispositivo. Por lo tanto trasladamos la persistencia interna de cada ME (Managed Element) al driver, pensando además que parte de su estado podría obtenerse del dispositivo en sí y no habría necesidad de un almacenamiento secundario.

A partir de esta decisión lo que quedaba por almacenar era básicamente la información de los MEs y su configuración. Por otro lado, la información de qué dispositivos se administran en el componente parece ser bastante estable. Por esto y por razones de practicidad, la decisión fue que el componente guardara su estado en un archivo XML.

Este archivo es leído por única vez cuando se levanta el componente y se almacena cada vez que se agrega o quita un ME, o si se cambia la configuración de un ME.

Estructura XML

El archivo en el cual el componente gurda su estado se llama mem.xml y a continuación se muestra un ejemplo parcial de su contenido.

```
- <MEM>
      - <ME>
            + <name>
              <userLabel>Cisco-1</userLabel>
              <nativeEMSName>Cisco-1</nativeEMSName>
              <owner>FIng</owner>
              <location>Montevideo</location>
              <version>3</version>
              <productName>Cisco-RJ</productName>
              <additionalInfo />
            + <MLSNNames>
            - <EdgePoints>
                  - <CTP>
                        <PTPName>PTP#3</PTPName>
                        <CTPName>CTP#1</CTPName>
```

```
                    </CTP>
                  + <CTP>
                </EdgePoints>
              - <MEdriver>
                    <driverName>Driver 3 for mitiNum test</driverName>
                    <driverClass>gesinv.ems.meDrivers.Driver3</driverClass>
                    +<driverConfigInfo>
                </MEdriver>
            </ME>
          + <ME>
          + <ME>
          + <ME>
</MEM>
```

10.7. Agregando Drivers

Ya mencionamos la función de los drivers y que permiten independizar el funcionamiento de cada dispositivo.

En nuestro proyecto desarrollamos tres drivers con el objetivo de testear la aplicación y comprobar los resultados. Como vimos cada driver deberá tener su propia persistencia si la necesita, e interfaz gráfica si fuera conveniente. La siguiente pantalla muestra la interfaz gráfica desarrollada para uno de los drivers, con el ánimo de ver el impacto de las operaciones.

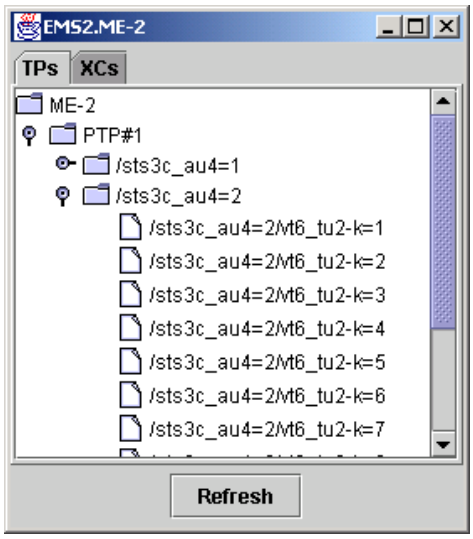


Figura 10-10 - Interfaz gráfica del driver de prueba

Los drivers son agregados dinámicamente (en tiempo de ejecución) utilizando un mecanismo provisto por Java llamado reflexión. Éste permite instanciar clases que no se conocen en el momento que se compila el programa.

Para crear un driver que permita manejar un dispositivo particular se deberá extender una clase llamada ManagedElementDriver. Esta clase está parcialmente implementada y maneja las estructuras de datos que vimos anteriormente.

A pesar de la complejidad de la interfaz MTNM, para implementar una driver necesitamos simplemente implementar unos pocos métodos heredados de la clase ManagedElementDriver:

- **getConfigParams():String[]**

  Este método devuelve una lista de String que son los parámetros que el driver necesita para ser configurado.

- **active(NameAndStringValue_T[] driverConfig)**

  Este método recibe una lista de parejas con el nombre del parámetro y su valor de configuración, y activa el driver.

- **deactivate()**

  Este método desactiva un driver lo que permite por ejemplo remplazar el driver o reconfigurarlo para un ME.

- **getSuportedRates():short[]**

  Este método devuelve la lista de los Layer Rates soportados por el dispositivo de acuerdo a la codificación establecida en el estándar MTNM.

- **setTPData(TPData_T tpInf):TerminationPoint_T**

  Este método permite setear a un Termination Point algunos parámetros y devuelve los valores resultantes.

  El driver deber ajustar las estructuras de datos internas de acuerdo a los valores seteados.

- **createCrossConnects(NameAndStringValue_T snName[], CrossConnect_T[] xcs)**

  Este método permite crear un grupo de Cross Connects que pertenecen a una Subnetwork Connection. El driver luego de crear físicamente las XCs deberá actualizar su estructura interna de datos pudiendo utilizar para esto el código escrito en los drivers de ejemplo.

- **deleteCrossConnects(NameAndStringValue_T[] snName)**

  Este método debe borrar las XCs perteneciente a la SNC pasada como parámetro. Al igual que al crearlas, deberá actualizar su estructura interna de datos.

11. Inventory

En esta sección describiremos el inventario que es el componente central de nuestra solución.

Este componente es el administrador de persistencia de objetos en la capa de red, su función principal es proveer servicios a la propia capa NMS para consultar, crear, modificar y eliminar objetos.

11.1. Consolidación del Inventario

Se comenzará a estudiar el inventario a través del proceso de decisiones que nos llevaron a obtener la solución definitiva.

11.1.1. Alternativas

El modelo a elegir para implementar el inventario representó una de las mayores decisiones del proyecto.

Inicialmente se tenía conocimiento del modelo MTNM, pero de esto no se podía desprender cómo debía ser el Inventario.

Decidimos postergar la decisión a un estado más avanzado del proyecto y resolvimos crear un inventario que cumpliera estrictamente con persistir los posibles objetos obtenidos por la **capa de red** desde la **capa de elemento** a través de la interfaz MTNM.

Almacenamiento

En principio ante la situación planteada se consideraron dos alternativas básicas, que éste fuera persistido en una base de datos relacional o en una base de datos orientada a objetos.

Tomamos la decisión de utilizar a un manejador de base de datos relacional. Esta decisión se baso en tres motivos. El primero radica en el hecho de que son las más utilizadas, conocidas y por ello confiables. El segundo fue nuestra experiencia en el manejo de este tipo de bases. Por último la abundancia de manejadores relacionales en comparación a las bases de datos orientadas a objetos, tema fundamental para permitir una futura reutilización del mismo en ambientes de desarrollo heterogéneos.

Es así pues, que fue realizada la primera presentación de código funcionando en base a un inventario que almacenaba objetos idénticos a los planteados dentro del estándar MTNM con el desempeño y claridad necesarios para ese momento.

Servicios

Posteriormente comenzó la etapa de análisis de cómo debería ser el inventario definitivo que tendría el proyecto, surgieron muchas dudas y posibilidades.

El aspecto que entró en escena en ese momento fue los servicios requeridos por el módulo Provisioning y el peso que tenía que éste perteneciera al proyecto **mitiNum**, debido al bajo nivel de acoplamiento que esto requería. La definición de estos

servicios no era del todo sencilla porque no estaba definido que requermientos necesitaba el Provisioning.

Otro aspecto que consideramos fue la futura investigación que se continuaría en Facultad de Ingeniería basándose en estos proyectos, pues otros componentes que no se desarrollaron en esta etapa probablemente requerirán de sus servicios. Fue así que nos pareció necesario hallar un estándar que nos guiara en la construcción del inventario definitivo y nos evitara caer en un diseño **"a medida**" que no seria de tanta utilidad para futuras investigaciones.

Por otro lado era razonable como una justa decisión hallar una interfaz de dialogo entre el modulo de Provisioning y el inventario que fuera independiente de los puntos de vista e intereses particulares de cada equipo e intentara proyectarse mas halla de los alcances del proyecto.

Luego de la obtención de documentos que profundizaban en el modelo de **WINMAN**, la consideramos como una opción sumamente atractiva. Asimismo considerábamos también la posibilidad de encontrar en **CaSMIM** las respuestas para ofrecer los servicios al Provisioning. Estas eran nuestras dos alternativas consideradas y debíamos decidir cual de ellas seria la que ofrecería al proyecto la mejor de las soluciones.

Luego de analizar a ambas en profundidad percibimos que **CaSMIM** se encontraba mas enfocada a la capa de servicios y considerábamos que la brecha que se creaba entre las interfaces **CaSMIM** y la interfaz MTNM era tan grande que el inventario quedaría expuesto a una indefinición gigantesca de cómo debería transformar datos de la capa de elementos a prácticamente datos de la capa de servicio.

Así también **CaSMIM** no nos aportaba ninguna información, aunque sea mínima, de cómo debería ser la arquitectura interna del inventario, porque es un estándar de interfaz.

Por el otro lado encontramos en WINMAN el planteo de un modelo sumamente completo, claro, pero no estricto, de cómo debería ser el inventario. Asimismo realizaba el planteo de la interfaz entre el inventario y el Provisioning que tenía un diálogo realmente claro y concreto, que tenía la gran ventaja de ser un punto intermedio entre el estándar MTNM y el estándar CaSMIM. Este punto intermedio nos permitía ver con más claridad cuáles deberían ser las transformaciones que se realizarían con los datos procedentes desde la capa de elemento, y nos permitía tener una idea más exacta de cómo deberíamos construir el inventario.

Por todo esto y teniendo en cuenta la falta de una especificación de requerimientos, la decisión fue contundente, elegimos **WINMAN** como base para utilizar para construir el **Inventory**.

11.1.2. Adaptación de WINMAN

Como vimos la solución esta fuertemente basada en WINMAN, en esta sección describiremos las partes que utilizamos y las adaptaciones que hicimos.

La solución WINMAN es un NMS, el cual define arquitectura de componentes e interfaces internas entre estos. Esta arquitectura tiene un componente Inventario del cual extrajimos los servicios ofrecidos por este y su modelo relacional. También utilizamos la idea de tener un "componente sur" que hablara con el EMS, aunque con alcance únicamente del inventario.

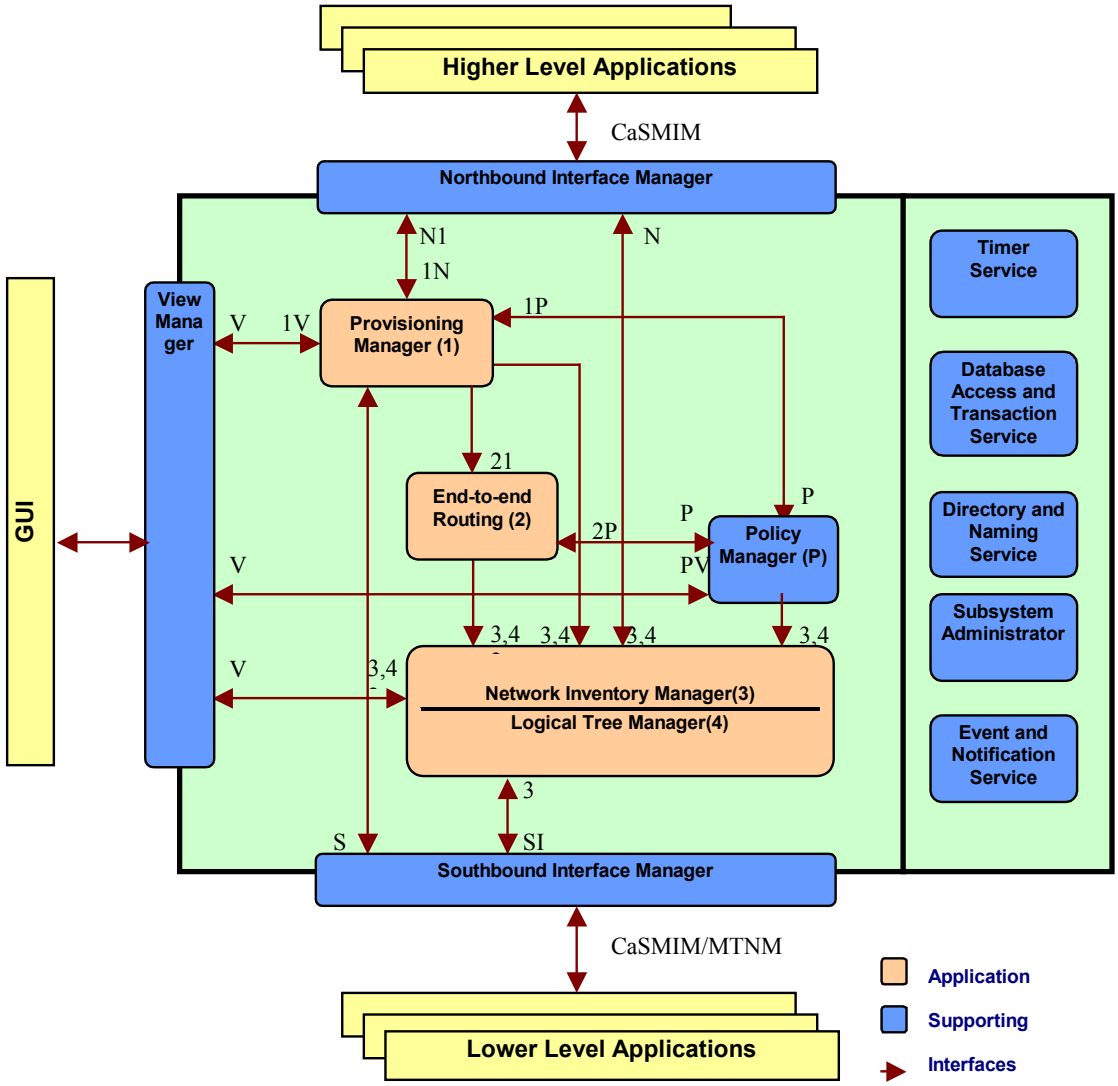


Figura 11-1 - Arquitectura de componentes de WINMAN

Fachadas

Dentro de las interfaces que define WINMAN para el inventario encontramos IInventory, IConnection y TreeViewObject, las cuales debimos pasar a IDLs CORBA para cumplir con el paradigma de desarrollo basado en componentes.

Otra ventaja de la utilización de estas IDLs es la posibilidad de tener módulos instalados en distintas terminales, interactuando a través de CORBA, lo que ofrece una versatilidad mayor.

Por otro lado estas interfaces otorgaban los objetos solicitados, cosa no recomendable en un ambiente de trabajo en el cual existan varios consumidores. Por lo cual decidimos que en lugar de ofrecer los objetos mismos, estas fachadas deberían ofrecer **fachadas especializadas** que fueran las encargadas de ser intermediarias entre los objetos administrados y los consumidores. Como aclaración, puede observarse en la interfaz **IInventory** el método **getObject.**

La ventaja de tener fachadas especializadas radica en la garantía de que todo objeto consultado por los consumidores se encontrará actualizado, dado que su manejo se encuentra centralizado internamente en el inventario, y es consultado por las mismas.

Así entonces habilitamos el uso del mismo objeto por varios consumidores sin el peligro de tener datos inconsistentes o desactualizados.

A continuación se puede ver el diagrama de clases junto con la interfaces propuestas por WINMAN (extraído de WINMAN-WP3-PTI-024i-D3-1-b1, página 417).

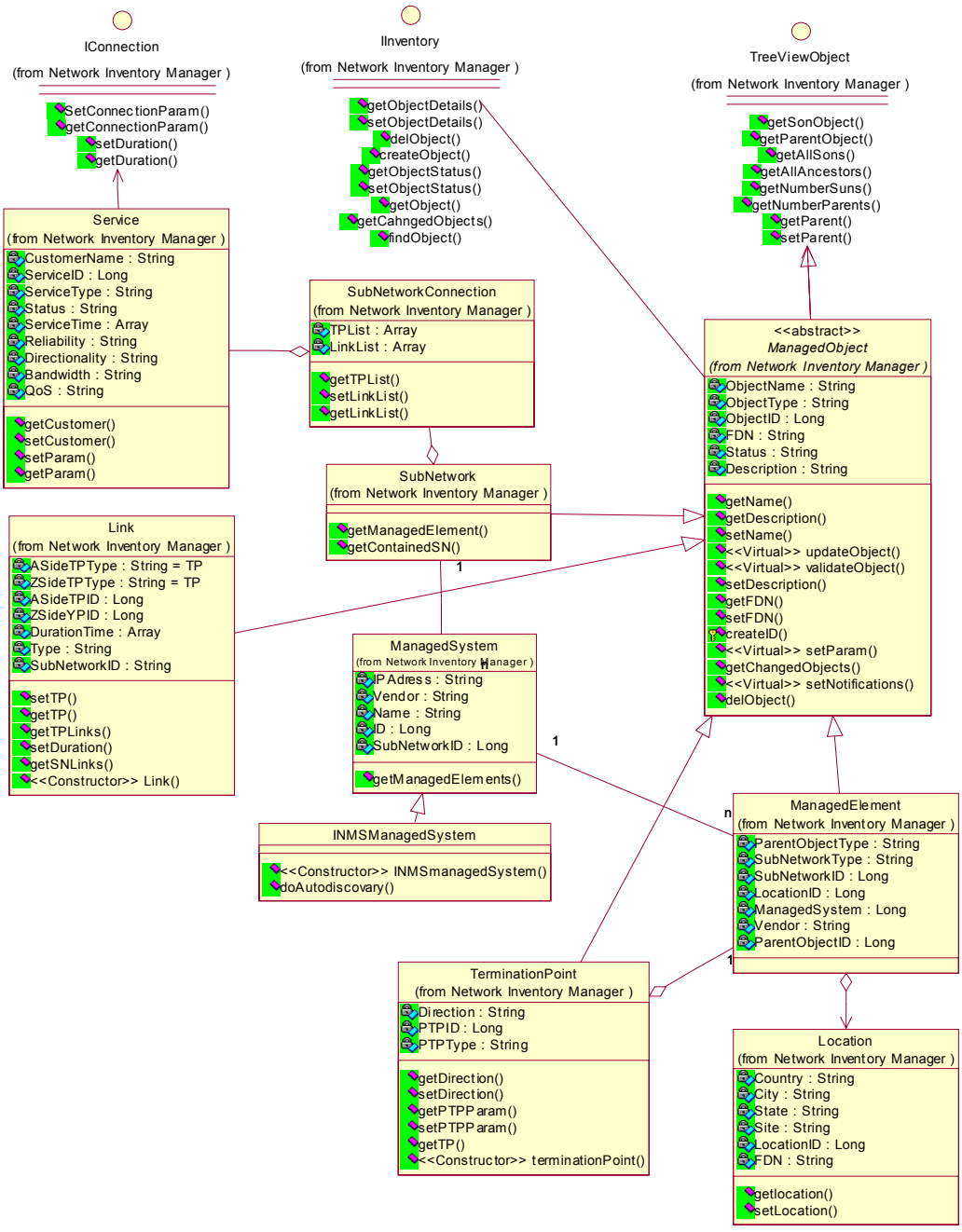


Figura 11-2 - Diseño de clases del Inventario WINMAN[2]

[2] Nótese que el diagrama extraído de la documentación de WINMAN contiene un error en la relación SubnetworkConnection-Subtnwtwork, la relación está invertida.

TreeViewObject

De las tres interfaces, la TreeViewObject fue la que requirió la modificación más radical. Dicha Interfaz se consulta para "navegar" el inventario a través de la jerarquía de sus objetos.

WINMAN plantea que cada objeto, "si corresponde", posee una referencia a sus objetos relacionados, como por ejemplo un ManagedSystem posee una referencia a la SubNetwork que lo contiene. Así las únicas vinculaciones que existen son las explicitadas en los propios objetos.

Esta interfaz fue pensada con la finalidad de permitir una navegabilidad entre objetos del inventario con el objetivo de visualización de los mismos. Ésta consiste en la administración de links que vinculan un objeto con su padre, utilizando los datos contenidos en los objetos ManagedObject (ParentObjectID).

Nosotros por nuestra parte nos encontrábamos en una situación que requería otras funcionalidades.

Primero, las relaciones entre objetos del inventario no se ajustaban a las necesidades que teníamos, y algunos casos incluso las contradecían. Segundo, en el modelo existen objetos que no heredan de la clase ManagedObject, por lo cual quedaban excluidos de toda posible navegabilidad a través de la interfaz TreeViewObject. Y tercero, la más importante, no podíamos predecir como serian los requerimientos del Provisioning en cuanto a la navegabilidad y no era conveniente estructurar un sistema de referencias entre objetos que resultara obsoleta a la hora de la puesta en marcha.

Por lo cual decidimos, primero, mantener las referencias originales entre objetos del inventario, tal cual las sugiere WINMAN, pero sin ser utilizadas para el proyecto actual. Y segundo, modificar radicalmente la forma en que es utilizada la interfaz TreeViewObject.

La nueva modalidad consiste en que las relaciones entre padres e hijos es almacenada en una estructura separada de los objetos del inventario, así pues se tendría una tabla que relacionara un objeto padre con un objeto hijo. Así también extendimos la funcionalidad para permitir que un hijo posea mas de un padre, por lo cual, estaríamos en presencia de un grafo más que de la estructura de árbol que plantea WINMAN.

Este punto fue acertado, ya que:

- En el momento de la construcción y avance del inventario aun no teníamos una definición formal de acuerdo con **mitiNum** en cuanto a la estructura de vinculaciones entre objetos, y ésta no llegó hasta etapas muy avanzadas de la integración. Esta situación hubiera ocasionado grandes dificultades en el caso de haber manejado una estructura rígida de vinculaciones entre los objetos.
- Nos permitió integrar la estructura jerárquica de vinculaciones entre objetos que provenían desde la capa de elemento con los requerimientos del Provisioning, en forma inmediata.
- Y aun más, los criterios de jerarquía entre objetos del inventario pueden ser seteados externamente por medio de esta fachada, sin conocerse de antemano esta jerarquía. En el caso de **mitiNum**, éste crea una objeto "ServiceConnection" y él mismo se encarga de crear la jerarquía con las "SNCs".

En resumen, el inventario para el marco temporal en el cual están basados los proyectos permitió una flexibilidad que garantizó la factibilidad de los mismos y una integración "natural" que no impactara de forma destructiva sobre el inventario. Consideramos que este objetivo fue cumplido salvaguardando la viabilidad del inventario.

La estructura descartada

Del diagrama de la figura 11-2 (Diseño de Clasess) se pueden obtener las relaciones de navegabilidad entre objetos WINMAN. Esta relaciones no eran suficientes para representar las necesidades que se desprendían de los requerimientos y el diseño.

En la siguiente figura se observan la diferencias entre éstas.

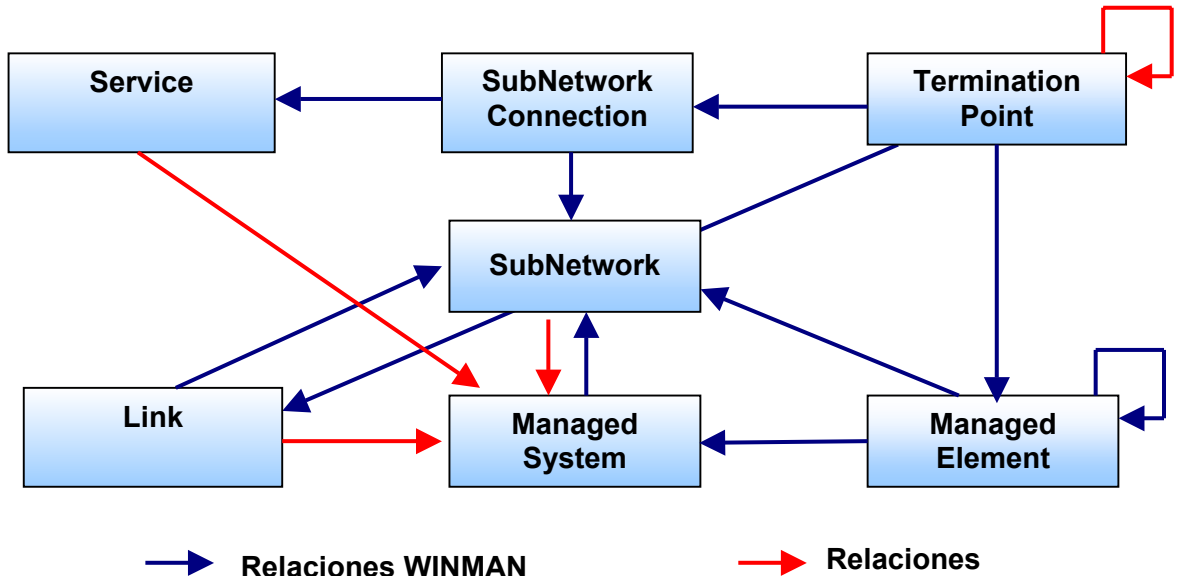


Figura 11-3 - Adaptación de relaciones jerárquicas

Servicios Modificados

Dada la adaptación que se realizó del modelo hubieron métodos y atributos que no fueron implementados y/o utilizados.

En la interfaz **TreeViewObject** se descartó el método **getAllAncestors,** ya que no tenía utilidad para el proyecto.

A continuación detallaremos para cada objeto que métodos fueron descartados:

- Para **ManagedObject**:
  - **ValidateObject** se descarto
  - **createID** deja de tener utilidad, pues se le asigna una identificación automáticamente al crear el objeto.
- **ManagedElement** queda sin atributos, por lo tanto, sin métodos.
- **SubNetwork**:
  - Método **GetConteinedSN** migra a TreeViewObject

- Método **GetManagedElement** migra a TreeViewObject

- **Link**:
  - Método **GetTPLinks** migra a TreeViewObject
  - Método **GetSNLinks** migra a TreeViewObject
- **ManagedSystem**:
  - Método **getManagedElements** migra a TreeViewObject
- **SubNetworkConnection**:
  - Método **getTPList** migra a TreeViewObject
  - Método **setLinkList** migra a TreeViewObject
  - Método **getLinkList** migra a TreeViewObject

Mapeo de Objetos

Los objetos de del inventario y los de WINMAN son prácticamente idénticos. A la clase TerminationPoint, por necesidad de Provissioning, se le agregó un atributo que indica si es “edge” (de borde) y otro que indica si es CTP o PTP. Otra variación fue el cambio de nombre de los objetos Service que se pasó a ServiceConnection, para hacerlo semejante a su correspondiente en CaSMIM.

11.1.3. Requerimientos Provisioning vs. Servicios de capa elemento

En un principio consideramos oportuno que los objetos del inventario dialogaran de dos maneras posibles, como en el estándar MTNM y como el modelo de WINMAN. Se basaba en la idea de que existía una correspondencia biyectiva entre clases de objetos de la capa de elemento y los objetos manejados por el Provisioning. Esta suposición implicaba que el inventario debería ser capaz de adaptar y transformar datos de un modelo a otro. Además, nos resultaba sumamente complejo encontrar la correspondencia de atributos entre los objetos manejados de una capa con la otra como para agregar esta funcionalidad al inventario. Finalmente consideramos que esta suposición era equivocada y que le estábamos otorgando facultades al inventario que no le correspondían.

Es por eso que tomamos dos decisiones, primero que solo necesitábamos convertir los objetos en un sentido, y segundo desplazar esa facultad a un subcomponte seprado llamado “Subcomponete South” y hacer que el inventario manejara únicamente objetos del tipo WINMAN. Este Subcomponete South es el responsable de transformar los objetos de una capa a la otra, a través del “Mapeo de atributos” entre clases de objetos y otras funciones que veremos más adelante.

11.2. Solución

En está fase se describirá el resultado final del proceso de consolidación del Inventario.

11.2.1. Arquitectura General

Comenzaremos describiendo los grandes bloques del inventario y su relación con el entorno.

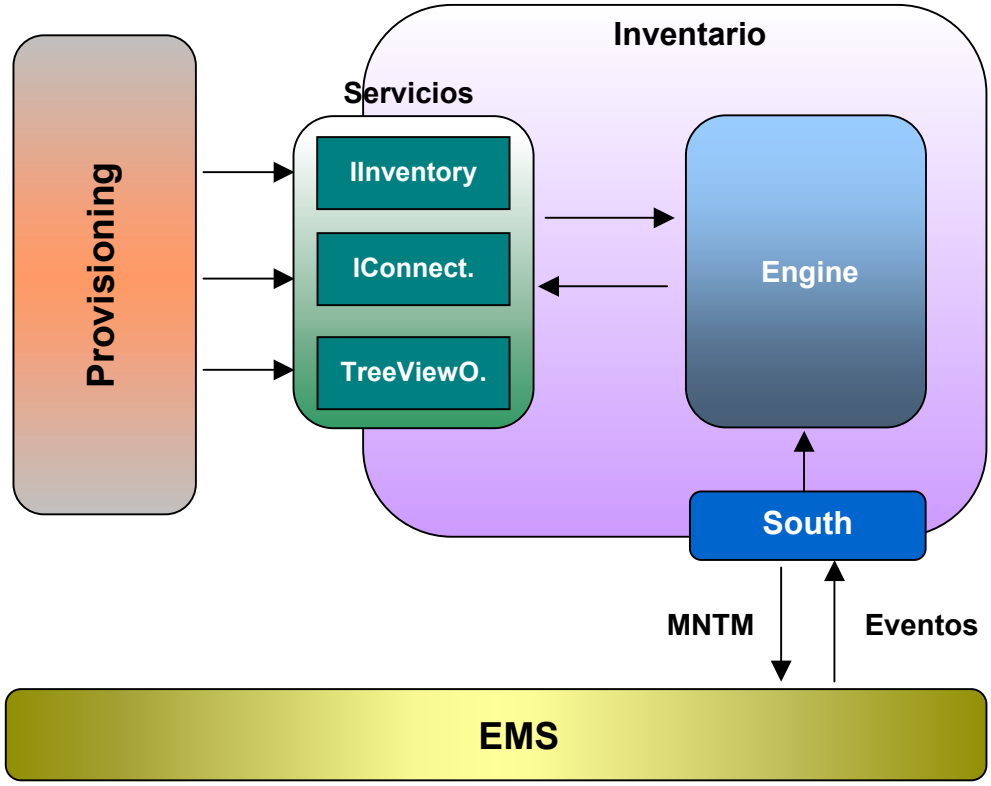


Figura 11-4 - Arquitectura general del Inventario

Entre los componentes de frontera se encuentran obviamente los encargados de prestar los servicios y además un subcomponente south (sur) que será el encargado de comunicarse con el EMS. El Engine es un grupo de subcomponentes que administra la persistencia del Inventario. Más adelante se describirá con más detalle cada uno de estos.

11.2.2. Servicios del Inventario

En esta sección describiremos los servicios brinda el Inventario, mediante la descripción de sus fachadas (interfaces).

Fachadas principales

Las fachadas principales son:

- **IInventory**: ofrece métodos que son aplicables a cualquier clase de objeto y posee el método getObject que retorna la fachada especializada para la clase especifica de objeto que se quiera administrar.
- **IConnection**: es la encargada de manejar específicamente los parámetros de conexión de los objetos **ServiceConnection** y es utilizada únicamente por el modulo Provisioning.
- **TreeViewObject**: es la explicada anteriormente y permite setear y consultar vínculos entre objetos del inventario.

Fachadas especializadas

Cada tipo de objeto manejado por el inventario tiene su propia interfaz, la que llamaremos fachada especializada. Estos métodos se basaron en los que define WINMAN para cada clase de objeto.

La ventaja de entregar una fachada especializada en vez del propio objeto garantiza que los objetos nunca dejan de estar bajo el control del inventario. De esta manera se puede atender a varios consumidores que deseen acceder al mismo objeto, ya que ellos tendrán acceso al objeto que se encuentra bajo el dominio del inventario.

Las fachadas especializadas son accesibles a través de la fachada principal IInventory con sus métodos respectivos.

Es un tema a analizar a futuro la existencia de un administrador de recursos objetos para garantizar si se desea la mutua-exclusión entre consumidores que compiten por un mismo objeto.

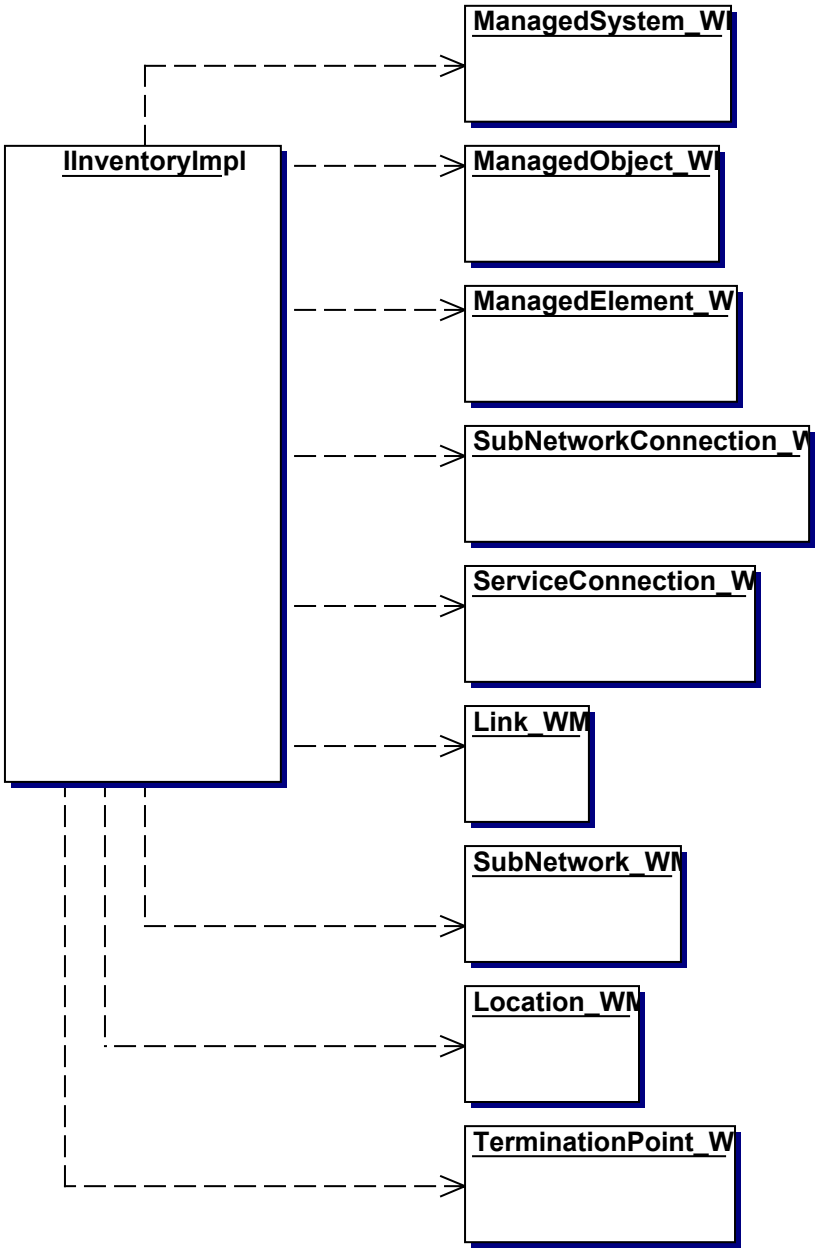
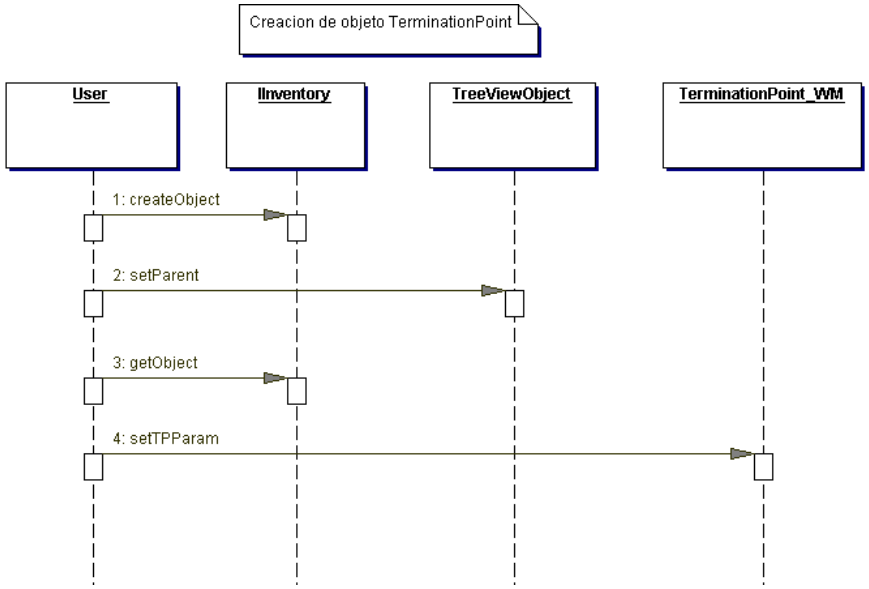


Figura 11-5 - Conjunto de fachadas especializadas

11.2.3. Uso de Fachadas

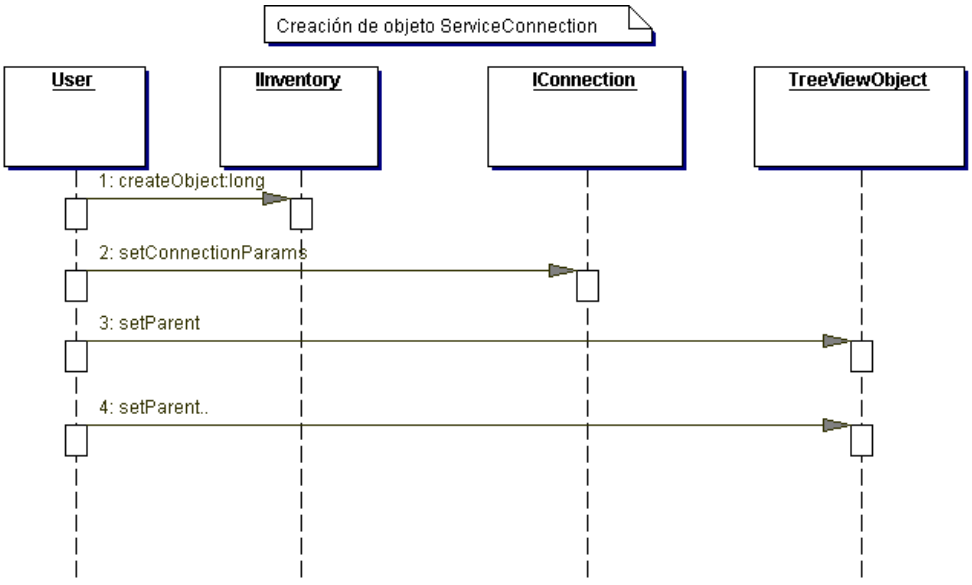
Luego de haber visto las fachadas mostraremos a través de diagramas de secuencia su funcionamiento dinámico. A continuación se verán algunas interacciones habituales del Inventario.

Creación de un TerminationPoint



1. Se crea el objeto con el tipo definido y la lista de atributos. Se obtiene el numero ID
2. Se setea el respectivo padre con el ID del objeto y el ID del padre
3. Si existe otro objeto padre se permite el enlace múltiple n veces. Idem.
4. Se setean los parámetros del objeto TerminationPoint teniendo el ID del objeto.

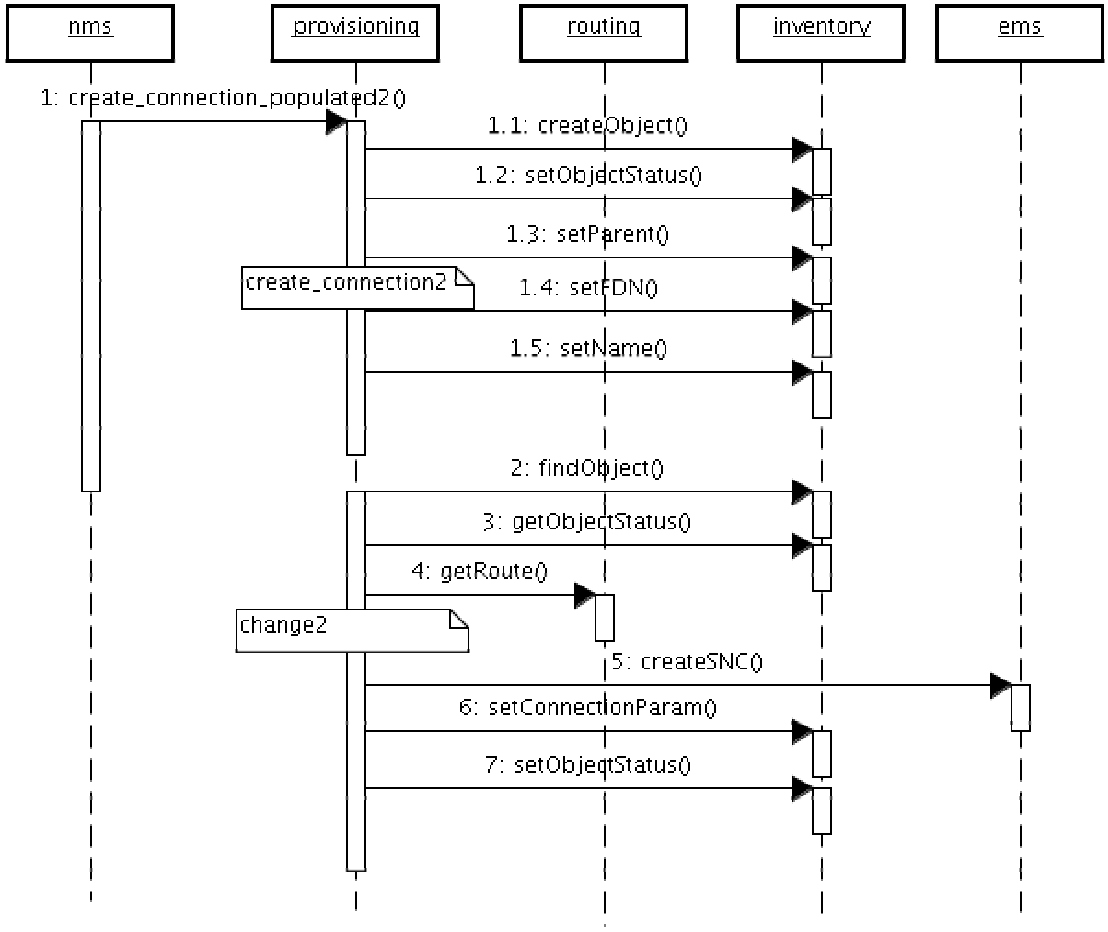
Creación de un ServiceConnection



1. Se crea el objeto ServiceConnection y se obtiene un ID.
2. Se setean los parámetros de conexión
3. Se agrega el padre respectivo
4. Se agregan todos los necesarios.

Creación de un Servicio de Conexión y su Materialización

Este diagrama de secuencia se divide en dos partes primero la creación del servicio de conexión como el visto en el caso anterior y luego su materialización, que consiste en la creación de las SNC en la capa de elemento. Este diagrama de secuencia a diferencia de los anteriores esta detallado por módulo y no por fachadas.



1. Creación ServiceConnection.
2. Busca el ServiceConnection creado por su FDN.
3. Obtiene y verifica el estado del servicio.
4. Obtiene una ruta apropiada para el servicio.
5. Crea las SNC en la capa de elemento.
6. Setea los parámetros de la conexión en el inventario.
7. Cambia el estado de la conexión.

11.2.4. Mapeo con atributos MTNM

Dado que nuestro Inventario se baso fuertemente en WINMAN hay diferencia con los objetos de capa de elemento definidos en MTNM. Por ello hubo que analizar cual sería la correspondencia correcta entre atributos del estándar MTNM y con los del Inventario, para eso primero se hizo un relevamiento de cuales son los conjunto de clases involucradas y sus mapeos semántico.

El resultado fue el siguiente:

MTNM	Inventario
EMS_T	ManagedSystem
MangedElement_T	ManagedElement
MultiLayerSubnetwork_T	SubNetwork
SubnetworkConnection_T	SubNetworkConnection
TopologicalLink_T	Link
TerminationPoint_T	TerminationPoint

Existen dos clases de objetos más del Inventario que no son mapeables con clases de MTNM, pues son conceptos de capa de red. Éstas son “Location” y “Service”. Además existe una tercera clase “ManagedObject”, que es una clase genérica de la que heredan varios objetos.

Luego, se debió vincular los atributos particulares de cada objeto MTNM con los de cada objeto del Inventario. Para ello se tuvieron en cuenta cuatro elementos:

- Descripción conceptual de cada uno.
- Requerimientos posibles que llegan al Inventario (interfaces, relaciones entre clases, etc.).
- Requerimientos mitiNum. Provisioning crea objetos en el EMS que luego son reuperados por el Inventario y consumidos por el propio Provisioning.
- Sentido Común (que información de la capa EMS, puede ser útil para el resto de las capas de arriba).

Se analizó cada tipo de elemento por separado y finalmente obtuvimos una relación coherente, aunque inicialmente estos precian no corresponderse completamente.

Necesariamente debíamos definir un “lenguaje” en común con el proyecto mitiNum para la nomenclatura de elementos (provenientes MTNM), pues de esta forma se identifican los objetos. Para ello conjuntamente resolvimos qué criterio utilizar para determinar el “Full Description Name” (FDN) de cada elemento. Dicho criterio se expresa en una función que transforma el nombre de objetos MTNM (de tipo “NamingAttributes_T”) en un “String” que lo identifica y viceversa.

MTNM: EMS_T	Inventario: ManagedSystem
Globaldefs::NamingAttributes_T name	**String name**
string userLabel	
string nativeEMSName	
string owner	**String vendor**

string emsVersion	**String version**
string type	**String type**
globaldefs::NVSList_T additionalInfo	
	Long MSID
	String IPAddress
	Double UpdateTime

MTNM: ManagedElement_T	Inventario: ManagedElement
globaldefs::NamingAttributes_T name	**String FDN**
string userLabel	
string nativeEMSName	**Long EMS**
string owner	**String Vendor**
string location	**Long location**
string version	
string productName	
CommunicationState_T communicationState	
boolean emsInSyncState	
transmissionParameters::LayerRateList_T supportedRates	
globaldefs::NVSList_T additionalInfo	
	String ObjectType
	String Status
	String Description
	Double UpdateTime

MTNM : MultiLayerSubnetwork_T	Inventario: SubNetwork
globaldefs::NamingAttributes_T name	**String FDN** **String SubNetworkName**
string userLabel	
string nativeEMSName	
string owner	
Topology_T subnetworkType	
transmissionParameters::LayerRateList_T supportedRates	
globaldefs::NVSList_T additionalInfo	
	String ObjectType
	Long ObjectID
	Double UpdateTime

MTNM: SubnetworkConnection_T	Inventario: SubNetworkConnection
globaldefs::NamingAttributes_T name	**String FDN**
string userLabel	
string nativeEMSName	
string owner	

SNCState_T sncState	String Status
globaldefs::ConnectionDirection_T direction	
transmissionParameters::LayerRate_T rate	
StaticProtectionLevel_T staticProtectionLevel	
SNCType_T sncType	
TPDataList_T aEnd	
TPDataList_T zEnd	
Reroute_T rerouteAllowed	
NetworkRouted_T networkRouted	
globaldefs::NVSList_T additionalInfo	
	String ObjectType
	String ObjectID
	Double UpdateTime
	List: (LinkType,LinkID,UpdateTime)

MTNM: TerminationPoint_T	Inventario: TerminationPoint
globaldefs::NamingAttributes_T name	**String FDN**
string userLabel	
string nativeEMSName	
string owner	
NamingAttributes_T ingressTrafficDescriptorName	
NamingAttributes_T egressTrafficDescriptorName	
TPType_T type	**String PTPType**
TPConnectionState_T connectionState	
TerminationMode_T tpMappingMode	
Directionality_T direction	**String direction**
LayeredParameterList_T transmissionParams	
TPProtectionAssociation_T tpProtectionAssociation	
boolean edgePoint	**Long isEdge**
NVSList_T additionalInfo	
	String ObjectType
	Long ObjectID
	Long PTPID
	Double UpdateTime

MTNM : TopologicalLink_T	Inventario: Link
NamingAttributes_T name	**String FDN**
string userLabel	
string nativeEMSName	

string owner	
ConnectionDirection_T direction	
LayerRate_T rate	
NamingAttributes_T aEndTP	**String ASideTPType** **Long ASideTPID**
NamingAttributes_T zEndTP	**String ZSideTPType** **Long ZSideTPID**
NVSList_T additionalInfo	
	String ObjectType
	Long ObjectID
	List : Duration (Start ,End)
	String Status
	String Type
	String SubNetworkIdentifier
	Double UpdateTime

11.2.5. Jerarquía de Objetos

Como vimos anteriormente es posible navegar el Inventario a través de la interfaz TreeViewObject. Existe un objeto llamado RootObject que es raíz de todos. Aunque el inventario no ofrece restricciones en las relaciones, cada vinculo establecido obedece al análisis de requerimientos y casos de uso de mitiNum, estableciéndose la siguiente jerarquía.

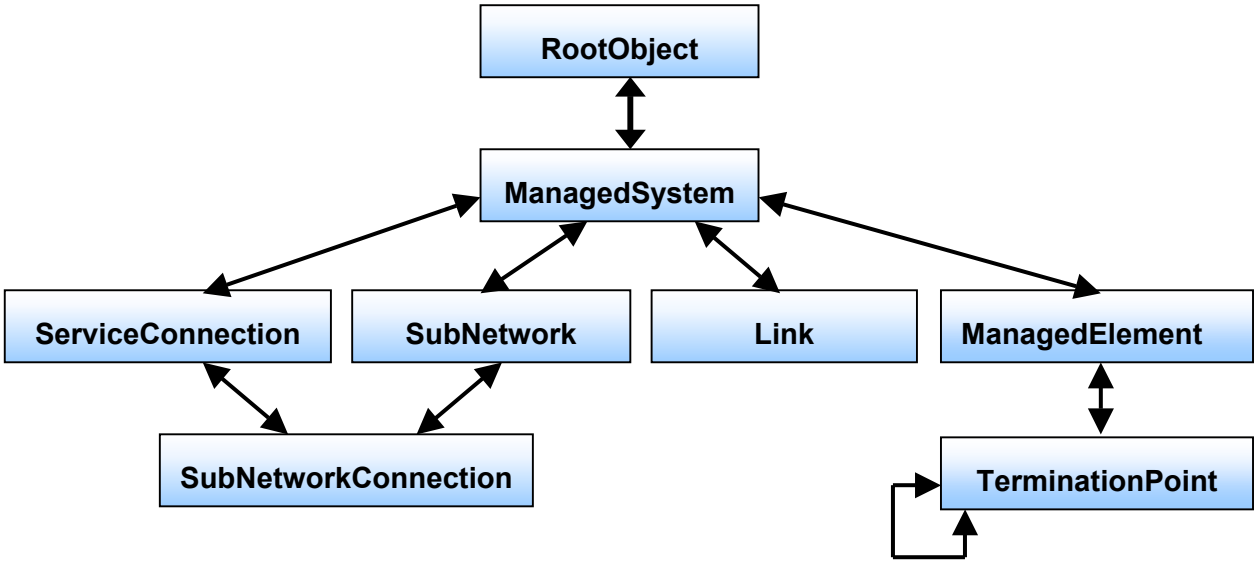


Figura 11-6 - Jerarquía de objetos del Inventario

Carga de relaciones

Las relaciones así como los objetos son almacenados en el inventario por el **subcomponente South** o por el módulo **Provisioning**.

El subcomponente South es el encargado de crear los vínculos entre:

- RootObject y el ManagedSystem
- ManagedSystem y ManagedElement
- ManagedSystem y Link
- ManagedSystem y SubNetwork
- ManagedElement y TerminalPoint
- TerminalPoint y TerminalPoint
- SubNetwork y SubNetworkConnection

Y el módulo Provisioning es el encargado de crear los vínculos entre:

- ManagedSystem y ServiceConnection
- ServiceConnection y SubNetworkConnection

Otros futuros componentes podrían agregar nuevas relaciones entre objetos sin necesidad de modificar el inventario.

11.3. Conceptos generales

Se describirán algunos conceptos importante antes de introducirnos en conceptos de diseño interno. Estos son fundamentales a la hora de comprender las ideas básicas que rigen al Inventario.

11.3.1. Diseño en 3 Capas

El inventario fue concebido basado en el paradigma de las tres capas: Servicios, Lógica y Persistencia. Esto no facilita la extensibilidad y mantenibilidad del Inventario.

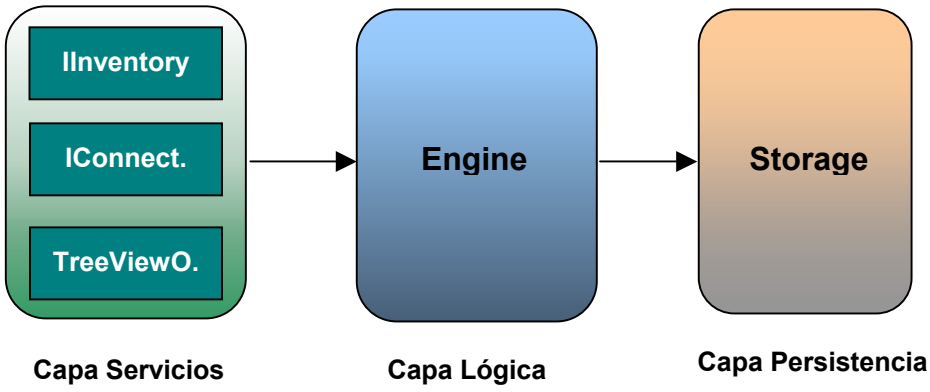


Figura 11-7 - Patrón de diseño en tres capas

La capa de servicios es la que brinda las funcionalidades del Inventario, a ser usadas por los clientes del inventario. Dicha capa delega la mayoría de las funciones a la capa lógica. La capa lógica esta compuesta por el motor del inventario y brinda funcionalidades básicas de manipulación de objetos de manera independiente de

almacenamiento. Finalmente la capa de persistencia atiende a la capa lógica y se encarga del almacenamiento real.

La capa lógica se divide en los servicios que administran los objetos y los propios objetos.

La comunicación entre la capa lógica y la de persistencia se realiza a través de la interacción entre dos tipos de objetos, objetos lógicos y objetos storage respectivamente.

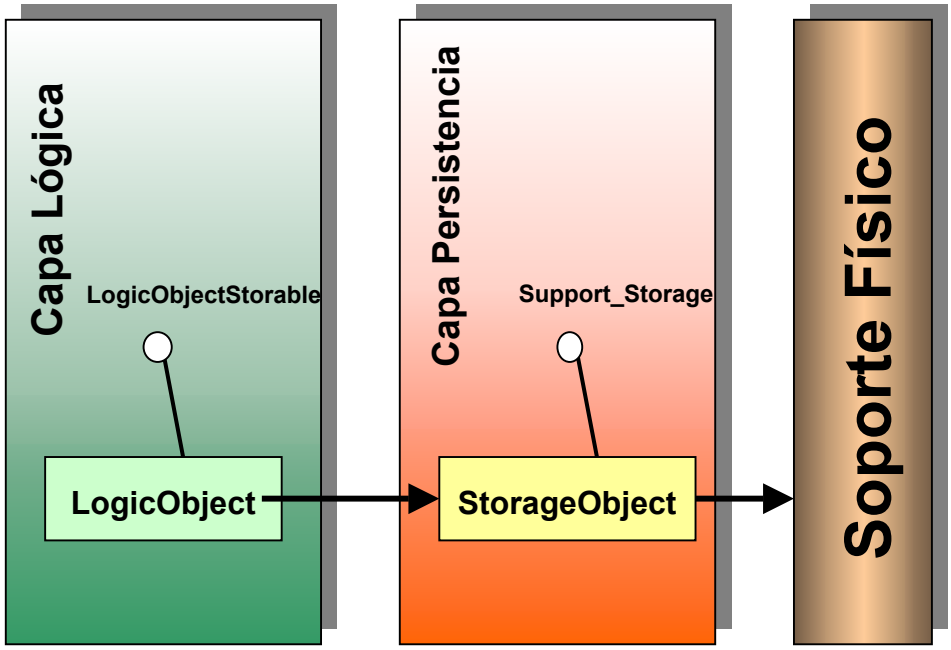


Figura 11-8 - Separación capa lógica y persistencia

11.3.2. Objetos Lógicos

Concepto

Los objetos Lógicos son las entidades básicas manejadas por el Inventario.

Este tipo de objetos son extensible y el inventario esta diseñado para permitir manejar cualquier clase de objeto siempre que éste respete determinados requerimientos.

Todo objeto Lógico que quiera ser persistido en el Inventario debe ser acompañado por un objeto del tipo **Support_Storage**, el cual se encargará de dialogar con el manejador de la BD con el fin de persistir al objeto Lógico.

En nuestro caso los objetos lógicos implementados para el proyecto se mapean directamente con los objetos de nuestro modelo.

Todo objeto lógico tendrá sus propios atributos, pero no tendrá métodos propios.

El ciclo de vida de estos objetos consiste en:

- Su creación y carga de sus atributos.
- Su almacenamiento en la DB.
- Sus sucesivas recuperaciones y/o modificaciones.

- Su eliminación del Inventario.

Identificación de los Objetos Lógicos

Todo objeto lógico puede ser identificado dentro del inventario por su tipo y su numero de identificación. El tipo de objeto es un String que lo identifica dentro de todos los tipos posibles de objetos que existen en el inventario, y el numero de identificación es un valor numérico del tipo (long) que es retornado en el momento que el objeto es almacenado en la DB y permanece incambiado hasta la eliminación del objeto. Así pues, siempre que deseemos recuperar un objeto almacenado en el inventario debemos solicitarlo a través de la identificación del tipo e ID del objeto.

Herencia

Los objetos lógicos también permiten formar estructuras de herencia, estas estructuras permiten la herencia de atributos de las súper clases. Esta funcionalidad no requiere un manejo especial y es “transparente” al usuario.

11.3.3. Persistencia de objetos

Clase LogicObjectStorable

Esta es la clase base de la cual deben extender todos los objetos lógicos.

Esta clase contiene los atributos y métodos para permitir la persistencia del objeto.

Así entonces tiene los métodos:

- **Load**: se encarga de cargar el objeto desde el inventario.
- **Save**: se encarga de salvar el objeto en el inventario.
- **Remove**: borra el objeto del inventario.
- **SetID_Object**: setea el identificador en el objeto lógico.
- **GetID_Object**: obtiene el identificador del objeto lógico.

Asimismo contiene los atributos:

- Referencia al objeto **Support_Storage** encargado de la persistencia.
- **Existe en DB**: permite saber si existe o no en la DB.
- **Tipo Objeto**: Identifica la clase de objeto lógico.
- **ID**: Identificación del numero de objeto.
- **ID seted**: permite saber si la ID fue seteada anteriormente.

Dinámica

La dinámica de funcionamiento será comentada brevemente dado que esta es transparente al usuario del inventario y no es necesario su comprensión en este momento.

Al instanciar un nuevo objeto lógico éste obtiene la referencia instancia del objeto Storage que será el encargado de persistirlo, así entonces, los métodos **load**, **save** y **remove** utilizaran las primitivas de esta clase Storage para realizar la acción pertinente.

Posteriormente se puede proceder a cargar los atributos del objeto.

Cuando se desee almacenar el objeto en el inventario (Save), éste solicitará al Storage un ID libre que pueda ser usado y posteriormente será salvado utilizando las primitivas del este objeto Sotrage.

Los métodos **load** y **remove** utilizan en forma similar las primitivas de la clase Storage.

Posteriormente si se desea salvarlo nuevamente al objeto, tendrá conocimiento que ya existe en la DB y no solicitara más ID libres al Storage. (Existe en DB)

Lo fundamental de esta dinámica es que el objeto lógico desconoce como es persistido, siendo la clase Storage la encargada de este objetivo, pudiendo ser reemplazada, si es deseado, por otro tipo de almacenamiento sin necesidad de cambiar la clase lógica.

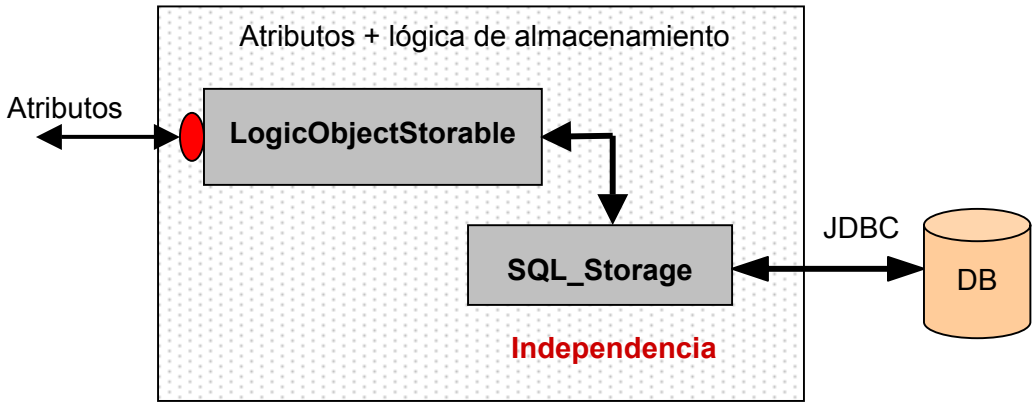


Figura 11-9 - pareja Logic-Storage

11.3.4. Objetos Storage

Concepto

Estos objetos implementan la interfaz Support_Storage.

Esta interfaz ofrece las primitivas:

- **Get**: obtiene un objeto de la base de persistencia.
- **Insert**: inserta un objeto en la base de persistencia.
- **Update**: actualiza un objeto en la base de persistencia.
- **Delete**: borra un objeto de la base de persistencia.
- **GetIDFree**: obtiene un ID de la base de persistencia.

Los Objetos Storage están agrupados en una única clase que los administra y existe una única instancia por cada tipo de objeto lógico. Al instanciar el Oobjeto Storage,

éste puede seleccionar entre un pool de posibles conexiones a distintos medios de persistencia que se pueden utilizar (ver Administración de conexiones).

Las primitivas pueden ser implementadas para usar el medio de conexión que se prefiera, incluso varios medios de conexión simultáneamente en forma híbrida. Para cada medio de conexión podríamos tener un medio de almacenamiento distinto.

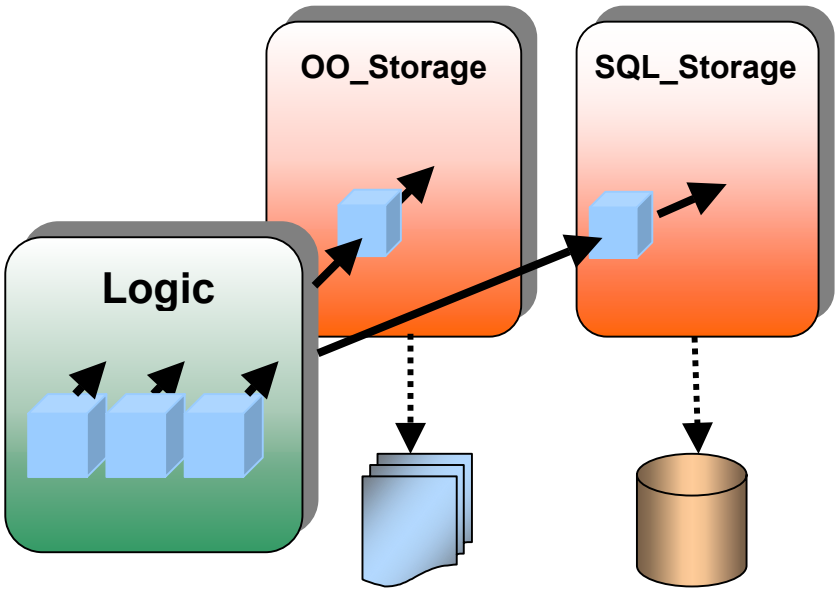


Figura 11-10 - Múltiples tipos de almacenamiento

En el caso del proyecto estos objetos utilizan una única conexión a una DB relacional, por lo cual los agrupamos en un package llamado **storSQL** y estas clases tienen la terminación "SQLStorage".

Como ejemplo tenemos:

PACKAGE	CLASE
logic	ManagedElement
storSQL	ManagedElement_SQLStorage

Dinámica

Veremos el caso completo cuando se ejecuta el método **Save** de un objeto lógico para comprender globalmente el funcionamiento Logic-Storage:

ExisteEnBD?	ID seteado?	Acción
False	True	storageMgr.insert(this); existeEnBD=true

	False	setID_OBJECT(storageMgr.getIdFree()) storageMgr.insert(this) existeEnBD=true
True	True (*)	StorageMgr.update(this)
		StorageMgr.update(this)

(*) Tiene ID seteada ya que: o fue salvada anteriormente o proviene de la base de persistencia, por lo cual fue cargada junto con su ID.

Los casos de **Load** y **Remove** son análogos y se basan en el análisis de las dos variables **existeEnDB** e **IDSeted** para el uso de las primitivas del objeto Storage.

11.3.5. Jerarquía Objetos Lógicos – Objetos Storage

Como mencionábamos anteriormente los objetos lógicos pueden estructurarse en jerarquías de herencia, esta estructura de herencia debe replicarse con las clases Storage correspondientes.

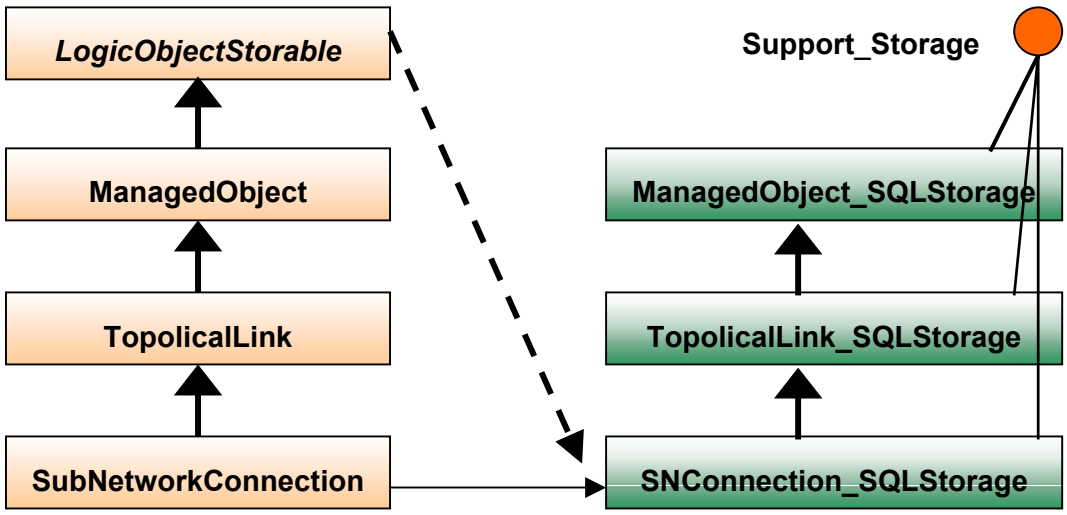
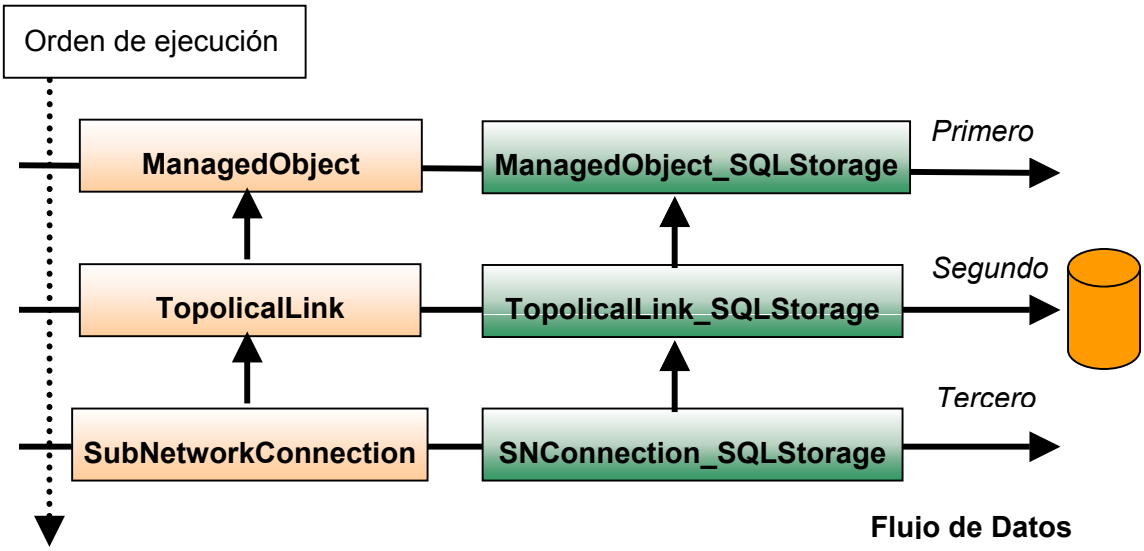


Figura 11-11 - Jerarquía de herencia Logic-Storage

Se puede observar en el ejemplo la estructura de herencia del objeto lógico SubNetworkConnection y su estructura paralela de objetos SQLStorage. Recordemos que todos los objetos Lógicos extienden de la clase base abstracta LogicObjectStorable (de ahora en mas lo llamaremos LOS) que es la encargada de implementar los métodos load, save y remove.

La clase lógica final, en este caso la clase SubNetworkConnecion, luego de obtener la referencia a su objeto SQLStorage pasa esa referencia a la clase base LOS para que esta pueda utilizar las primitivas necesarias de la clase Storage final, o sea la clase SNConnection_SQLStorage. Cuando se ejecuta el método load, save o remove del objeto lógico éste inicia un proceso en los objetos Storage, que se inicia en la clase final Storage y desencadena la ejecución de la misma primitiva en todas sus súper clases. Así entonces, la persistencia de un objeto lógico se descompone en una serie

de persistencias de cada una de sus partes, comenzando por la final hasta la clase base Storage.



En este ejemplo podemos ver el orden en que se salvan las distintas clases que componen al objeto lógico desde la clase base hasta la final.

Figura 11-12 - Orden jerárquico de ejecución de persistencia en clases heredadas

Es importante aclarar que cada uno de los objetos Storage tiene únicamente la responsabilidad de salvar los datos de su clase lógica respectiva y no tiene conocimiento de datos de las súper clases del objeto.

Entonces, es necesario que sea replicada la estructura jerárquica de los objetos lógicos en los objetos Storage para permitir esta vinculación “uno a uno”.

Este diseño permite una extensibilidad modular que garantiza la independencia de salvado de cada clase. Si se deseara extender un nuevo objeto lógico simplemente debe heredar de una clase y debemos crear su respectiva clase Storage que heredará de la misma manera que el objeto lógico pero en la jerarquía de objetos Storage.

11.3.6. Tablas en la Base de Datos

La estructura de tablas del Inventario está basada en la definición de tablas del modelo WINMAN. En dicho modelo se especifican las tablas, tipos y largo de campos de las mismas.

A continuación podemos ver como ejemplo la tabla encargada de almacenar los atributos de un ManagedSystem.

EMS table	Type	Index	Description
MSID	Number	Primary Key	

SubNetworkID	Number	Foreign Key	
MSName	Varchar(50)		
IPAddress	Varchar(250)		
Vendor	Varchar(50)		
Version	Varchar(50)		
Type	Varchar(10)		
UpdateTime	Double		The number of seconds from 1970. Updated with the system time each time an object is updated.

La estructura de las tablas tomada de WINMAN se corresponde con la idea que vimos anteriormente de que: la persistencia de un objeto es descompuesta en su estructura de herencia. Es decir, cada una de las clases que componen su estructura de herencia es salvada en su respectiva tabla.

El conjunto básico de tablas es el siguiente:

- **MO**: proviene del objeto abstracto MangedObject.
- **ME**: proviene del objeto ManagedElement.
- **Location**: proviene del objeto Location.
- **SubNetwork**: proviene del objeto Subnetwork.
- **EMS**: proviene del objeto ManagedSystem.
- **Link**: proviene del objeto Link.
- **Termination_Point**: proviene del objeto TerminatioPoint.
- **SubNetwork_Connection**: proviene del objeto SubnetworkConnection.
- **Service**: proviene del objeto ServiceConnection.

11.3.7. Jerarquía Fachadas-Objetos Lógicos

Las **fachadas especializadas** poseen una jerarquía de herencia idéntica a los objetos lógicos.

Cada fachada que extiende de otra hereda sus métodos, de la misma forma que sucede con los objetos lógicos.

Así entonces es posible consultar en una fachada los métodos específicos de la misma, así como los métodos de sus “súper-fachadas”. Y cada una de las fachadas se relacionará directamente con el objeto lógico imagen de la misma.

La ventaja de esta estructura radica en la **transparencia** ante el cliente de la existencia de múltiples clases. Haciendo innecesaria una lógica por parte del mismo para obtener atributos de distintas súper-clases de un mismo objeto.

IDL Inventory e InventroyDef

Para la implementación de las fachadas tomadas del modelo WINMAN se requirió la creación de dos definiciones IDL de CORBA.

Las funcionalidades se encuentran divididas en dos, en la IDL "**Inventory**" se encuentran las definiciones de las fachadas **IInventory**, **IConnection** y **TreeViewObject**, mientras que en la IDL "**InventroryDef**" se encuentran las definiciones de las **fachadas especializadas** vistas anteriormente.

Fueron separadas dado que el concepto de **fachada especializada** no se encuentra explicito dentro del modelo WINMAN y consideramos que era oportuno marcar dicha diferencia distribuyendo las mismas en definiciones independientes.

Así entonces tenemos los módulos **Inventory_Module** que define las interfaces:

- `IInventory_I`
- `IConnection_I`
- `TreeViewObject_I`
- `IPersistecicy_I`

Y el módulo **InventoryDefs_Module** que define las interfaces siguientes con su relación de herencia que corresponde con la herencia de los objetos lógicos (la relación de herencia es hijo: padre):

- `ManagedObject_WM_I`
- `ManagedElement_WM_I:ManagedObject_WM_I`
- `Location_WM_I`
- `SubNetwork_WM_I:ManagedObject_WM_I`
- `Link_WM_I:ManagedObject_WM_I`
- `ManagedSystem_WM_I`
- `TerminationPoint_WM_I:ManagedObject_WM_I`
- `SubNetworkConnection_WM_I:Link_WM_I`
- `ServiceConnection_WM_I:ManagedObject_WM_I`

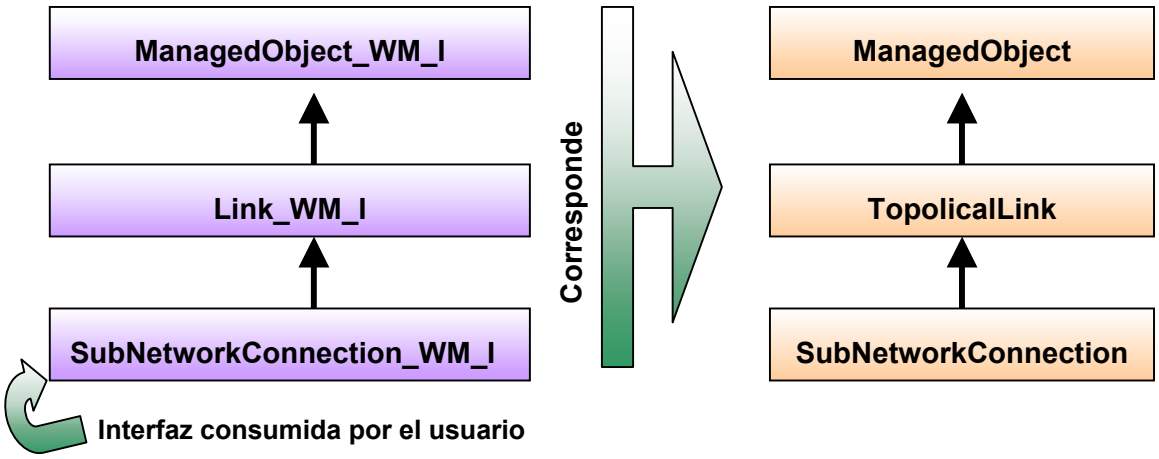


Figura 11-13 - Transparencia de uso jerárquico de fachadas

11.4. Diseño Interno

Luego de haber dado un enfoque de alto nivel del diseño y haber introducido los conceptos básicos del inventario, se explicara el funcionamiento interno del mismo.

11.4.1. Flujo de Objetos

Como ya vimos se puede dividir al inventario en 3 capas, la capa de servicios, la capa lógica y la capa de persistencia que cumple la base de datos. Existen tres componentes que se encarga de brindar los servicios internos del inventario. Estos son el Dispatcher (administrador de objetos), OmniVision (realiza funciones de búsqueda global) y TreeViewManager (administra jerarquía de objetos).

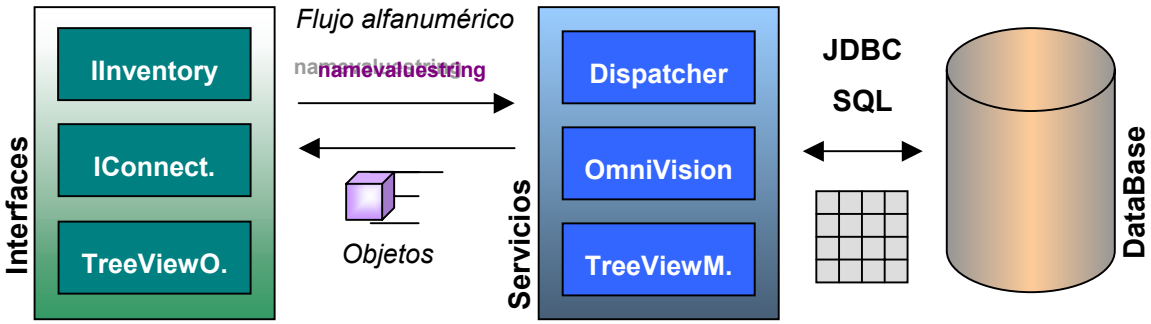


Figura 11-14 - Flujo de objetos

La capa de servicios atiende a los clientes manteniendo un dialogo basado enteramente en un flujo alfanumérico de información, esta información es parcialmente procesada y enviada a la capa lógica. En la capa lógica son atendidos los requerimientos de la capa de interfaces y dependiendo del caso ésta retorna objetos lógicos en el caso de respuestas del Dispatcher o indentificadores de objetos en el caso de los servicios de OmniVision o TreeViewObject. Los objetos retornados por el Dispatcher son tomados por las interfaces y son utilizados para atender el pedido realizado por el cliente.

Como vimos nuestra capa de persistencia esta implementada sobre una base relacional. Este diálogo con el manejador de DB se realiza por medio de lenguaje SQL sobre JDBC para obtener o salvar el contenido de los objetos lógicos.

Fachadas Especializadas

La fachada IInventory no otorga a sus clientes los objetos lógicos. En su lugar ofrece al cliente fachadas especializadas que interpretan los atributos de los objetos lógicos específicamente manejados por ellas.

Estas fachadas especializadas dialogan directamente con el Dispatcher y obtienen el objeto lógico que requiere el cliente para luego realizar todo el manejo necesario de los atributos del objeto obtenido.

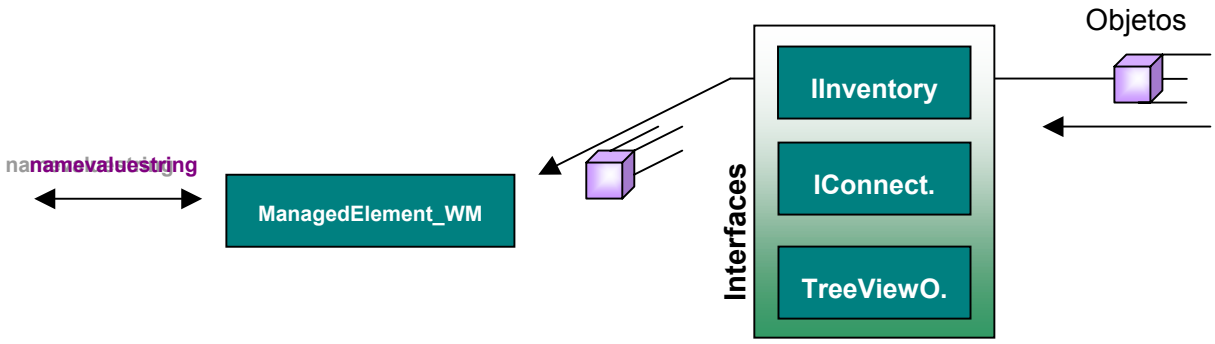


Figura 11-15 - Flujo de fachadas especializadas

Dichas fachadas poseen un flujo de datos alfanumérico con el cliente y un manejo de objetos del lado del inventario.

11.4.2. Server_Inventory

11.4.2.1. Principios

La clase Server_Inventory es la encargada de levantar el Inventario y sus funciones principales son:

- Inicializar los componentes internos.
- Levanta el subcomponente South que sincroniza el Inventario con la capa de elemento y queda esperando eventos de la misma.
- Publicar las fachadas CORBA que brindan los servicios del Inventario.

Secuencia de inicialización

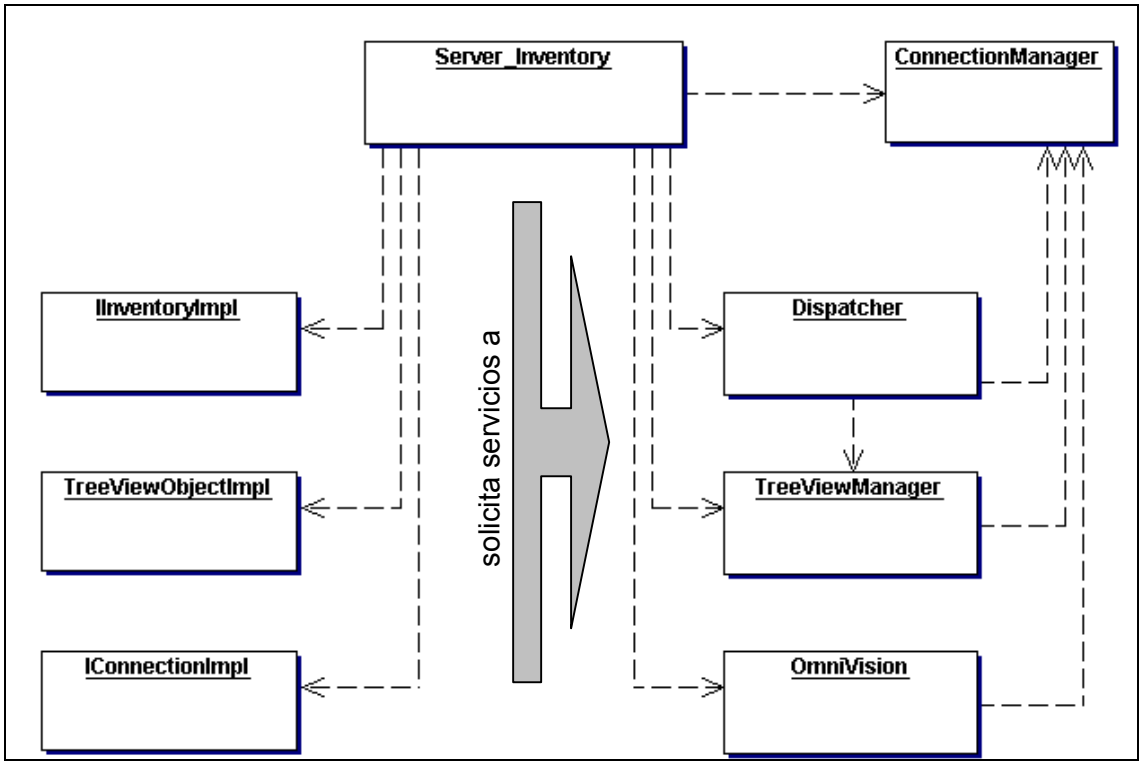
El primer paso que realiza el **Server_Inventory** es instanciar un objeto de la clase **ConnectionManager** e inicializar todas las conexiones que correspondan. Luego lo hace con las clases **TreeViewManager**, **Dipatcher** y **OmniVision** otorgándoles visibilidad hacia el administrador de conexiones.

En este momento el inventario se encuentra en condiciones de poder brindar las primitivas básicas a la capa de servicios compuesta por las instancias de las fachadas. Posteriormente procede a instanciar las clases **IInventoryImpl, TreeViewObjectImpl, IconnectionImpl** y les otorga visibilidad sobre las clases de primitivas básicas de servicio.

Una vez que están prontos los servicios internos al inventario se instancia el subcomponente South, que se encarga de sincronizar el inventario con la capa de elementos y conectarse al canal de eventos del EMS.

Finalmente publica las clases de la capa de servicio en el **Servicio de Nombres CORBA.**

A continuación se puede observar el diagrama de clases que muestra las clases principales instanciadas por el inventario.



11.4.3. Servicios Internos del Inventario

Dispatcher

Esta clase es la encargada de administrar y otorgar los objetos lógicos que son almacenados en el inventario a las instancias de las fachadas.

Posee las primitivas básicas para manejar objetos lógicos que son:

- CreateObject
- DelObject
- GetObject
- GetObjectDetails
- SetObjectDetails

El Dispatcher es usado por las fachadas **IInventory** e **IConnection** para obtener los objetos requeridos para realizar el servicio.

TreeViewManager

Esta clase posee las primitivas que permiten la navegabilidad en el inventario.

Sus primitivas básicas son:

- AddRelation
- ClearRelations
- ExistRelation
- GetAllChilds
- GetAllParents
- RemoveRelation
- InitializeTreeView

Esta clase es usada por la fachada **TreeViewObject** y por el **Dispatcher** en un caso particular que se explicara luego.

OmniVision

Esta clase brinda servicios de búsqueda que incumben a todos los objetos del inventario.

Obedece a la necesidad de poder hallar objetos dentro del inventario que son requeridos por determinados atributos.

El nombre **OmniVision** se refiere a la posibilidad de poder ver y hallar un objeto en cualquier lugar del inventario.

Este brinda específicamente los métodos :

- GetChangedObjects
- GetUnchangedObjects
- FindObject

Esta clase es usada por la fachada **IInventory** cuando son solicitados los servicios:

- GetChangedObjects
- FindObject

Y el método **getUnchangedObjects** es utilizado por la interfaz **South** al sincronizar el inventario con la capa de elemento.

Resumen de Relación entre los servicios y las Fachadas

A continuación veremos cada una de las fachadas que servicios internos consume.

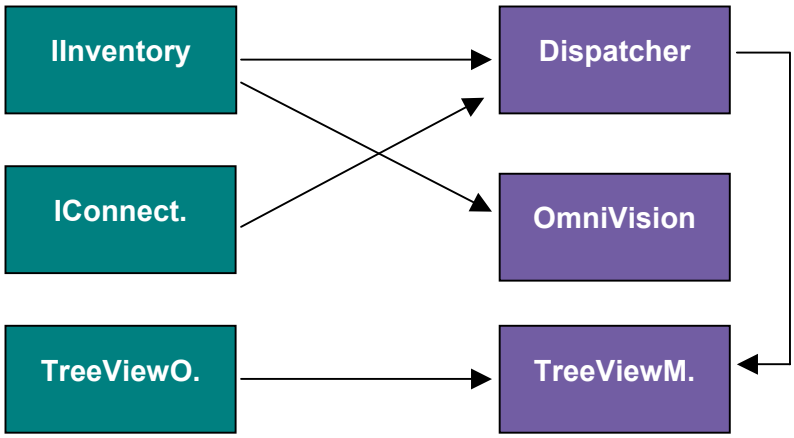


Figura 11-16 - Relación fachadas servicios internos

11.4.4. Dispatcher

El Dispatcher es la clase central de todo el inventario. Es el responsable de instanciar los objetos lógicos que son requeridos por las fachadas, así como de salvar en la base de persistencia los objetos modificados, por medio de los métodos **CreateObject, DelObject y GetObject.**

Es responsable de instanciar la clase **GroupObjectsStorage**. Dicha clase contiene el conjunto completo de objetos Storage. Este clase es otorgada por el Dispatcher a los objetos lógicos en el momento de su construcción, con el fin de que estos puedan obtener una referencia al objeto Storage que corresponda con su clase.

Así también utiliza los servicios de la clase **ManagerBufferObjects**, esta clase implmenta un buffer de objetos lógicos con el fin de mejorar la performance del inventario.

Al ser un elemento central de todo el inventario también es responsable de realizar la **mutua exclusión** en caso de colisión entre solicitudes de consumidores externos. Esta mutua exclusión se realiza para los métodos de **creación** y **borrado** de objetos ya que son métodos potenciales causantes de inconsistencias en el repositorio de objetos.

11.4.5. Buffer

11.4.5.1. Principios

El sistema de buffer de objetos lógicos permiten mejorar la performance del inventario cuando los requerimientos de **"recuperación de objetos recientemente utilizados"** son altos.

Aplicación al contexto

El uso de un buffer se debe a que existen “ráfagas de uso” sobre el mismo objeto. Estas “ráfagas de uso” son debidas a la estructura de las fachadas, en las cuales solo se pueden obtener o setear los atributos del objeto de 1 en 1 en los casos habituales.

11.4.5.2. La arquitectura

La arquitectura está basada en la utilización de un ManagerBufferObjects que posee una colección de objetos buffer, cada uno de ellos posee una “cola” de objetos container que encapsulan los objetos lógicos y contienen etiquetas para su rápida identificación.

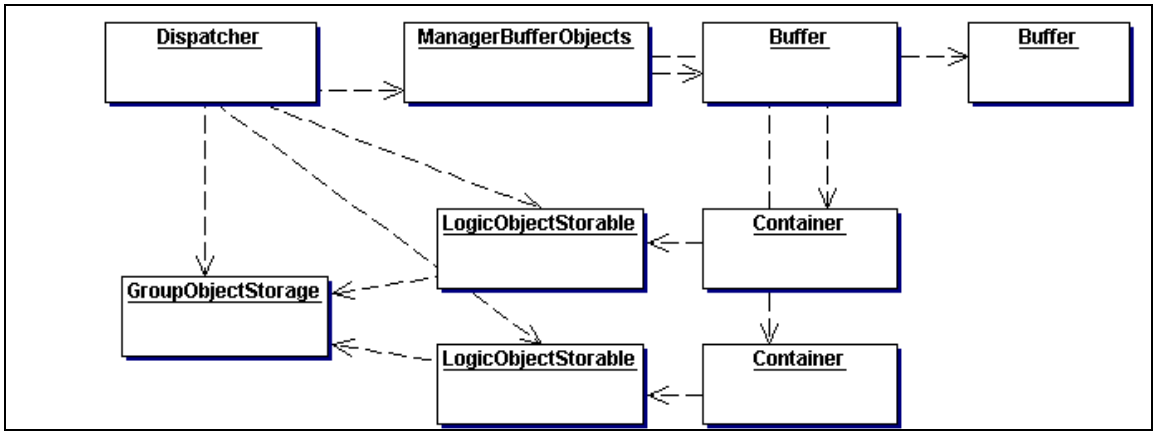


Figura 11-17 - Arquitectura de ManagerBufferObjects

Dispatcher

El Dispatcher crea una instancia del la clase ManagedBufferObject que permite administrar un pool de objetos buffer.

La utilidad de tener este pool radica en la utilización aplicada a distintos flujos de consumo que son volcados en cada uno de ellos. Por ejemplo, en el caso de tener distintos consumidores que utilizan frecuentemente conjuntos de clases de objetos, estos pueden ser colocados en buffer distintos para asegurar una memoria apropiada para cada caso de uso.

El Dispacher solo tiene visibilidad sobre los objetos LogicObjectStorable por lo que el manejo interno del **buffer** es transparente para el mismo.

ManagedObjectBuffer

Esta clase permite parametrizar la cantidad de instancias buffer a crear y permite direccionar las primitivas a la instancia apropiada.

Buffer

El buffer maneja objetos “container” que son independientes de los tipos de objetos almacenados. Los objetos **container** son manejados internamente y las clases anteriores hacen un uso transparente de los mismos.

La arquitectura del mismo consiste en una COLA de objetos container de largo fijo parametrizable. La primitiva "**push**" coloca un objeto al comienzo de la COLA y desplaza el resto de objetos. La primitiva "**getAndActualize**" retorna el objeto requerido y lo coloca al comienzo de la COLA. Esta dinámica mantiene los objetos en el buffer de acuerdo a su orden desde los mas recientemente invocados hasta el ultimo en la COLA.

Container

Esta clase encapsula los objetos del tipo "LogicObjectStorable" y mantiene los atributos claves por los cuales puede ser identificado el objeto al alcance del buffer. Ésta es solo visible dentro del objeto **buffer.**

Interacción Dispatcher - ManagedBufferObjects

A continuación se especifica el manejo que realiza el Dispatcher sobre el ManagedBufferObjects ante cada invocación que se le es realizada.

Primitiva Dispatcher	Lógica Buffer			
	Existe Objeto		No existe objeto	
	Buffer	Objeto	Buffer	Objeto
createObject	-	-	push	save
getObject	getAndActualize	-	push	load
delObject	deleteIfExist	Remove	-	remove
setObjectDetails	getAndActualize	Save	push	GetObject-save
getObjectDetails	getAndActualize	-	push	getObject

11.4.5.3. Posibles parametrizaciones

El manejo de un pool de buffers permite distintas configuraciones para optimizar la performance del inventario. Ésta quedará determinada por la cantidad consumidores y las clases de objetos a ser consumidos por cada uno.

En el caso de su aplicación a las fachadas del Inventario no es posible identificar el consumidor que realiza un pedido, por lo cual no es posible administrar los consumos independientemente.

Queda este tema planteado en el caso de querer optimizar en este aspecto el desempeño del mismo.

Respecto a la aplicación en particular del diseño realizado, debemos considerar que existe una **fachada principal** que ofrece **fachadas especializadas** para cada tipo de objeto existente en el inventario. Cada una de estas fachadas especializadas tiene las primitivas para solicitar los atributos de un objeto determinado. Como caso de uso frecuente es razonable que un consumidor realice una "ráfaga" de consultas a una fachada en particular para obtener un conjunto de atributos de un mismo objeto.

Si suponemos este caso, debemos mantener el último objeto consultado en cada una de estas fachadas para optimizar el desempeño. Esto nos lleva a tener la misma cantidad de buffers que de fachadas especializadas. Si a esto le sumamos que podemos tener N consumidores, deberíamos tener tantos buffer como tipo de objetos y cada uno debería ser de largo mínimo N por una constante.

Para el caso de este proyecto, teniendo una cantidad fija de consumidor podemos aproximar el desempeño utilizando un único buffer de largo igual a la cantidad de usuarios por una constante razonable, que represente el valor máximo esperado de objetos concurrentemente utilizados por un usuario.

11.4.6. Administración de conexiones

El diseño del inventario permite el manejo de múltiples conexiones a distintos medios de persistencia (por ejemplo BD relacional), en esta sección se detallará el funcionamiento del mismo.

Principios

El inventario esta diseñado para soportar varios tipos de conexiones e incluso concurrentes.

Vemos el diagrama general que relaciona las clases involucradas en el manejo de las conexiones en la siguiente figura.

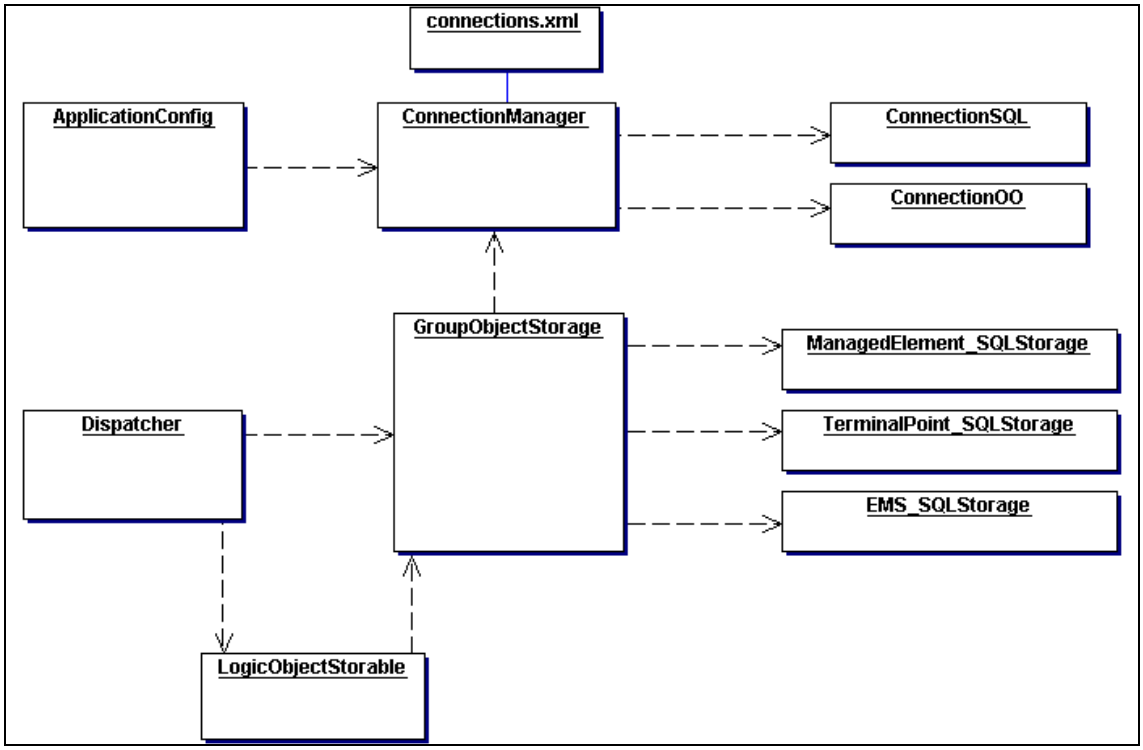


Figura 11-18 - Administración de Conexiones

A continuación se detallara el funcionamiento del mismo para comprender la potencialidad que permite obtener.

Clases Connection

Estas clases son las encargadas de establecer la conexión con el medio de almacenamiento. No existen limitaciones en el tipo de conexión, permitiendo por ejemplo conexiones a DB relacionales, orientadas a objetos, archivos planos, etc. Tampoco deben implementar una interfase definida, ya que los objetos Storage serán los encargados de su utilización.

Estas clases no poseen persistencia por si mismas, siendo instanciadas por la clase ConnectionManager.

En el proyecto se implementó la clase ConnectionSQL que almacena en una DB relacional y posee un pool de conexiones que permite mejorar el acceso concurrente a la base,

Clase ConnectionManager

Esta clase es encargada de instanciar los objetos Connection. Es responsable por la configuración de las conexiones y su persistencia. Para esto posee el archivo "connections.xml" en el cual son guardados todos los parámetros mencionados.

Esta clase debe tener conocimiento de cómo obtener y setear los atributos en cada objeto conexión administrado, para poder persistir su configuración.

Esta clase es instanciada en dos etapas distintas, cuando es configurado el inventario y en régimen de trabajo.

En la etapa de configuración la clase ApplicationConfig es la encargada de administrarla y en la etapa de régimen de trabajo es la clase GroupObjectsStorage.

ApplicationConfig

Esta clase ofrece una interfaz grafica para el seteo de los parámetros de configuración de las conexiones. Interactuando con el ConnectionManager puede cargar y salvar la configuración deseada en el archivo Connection.xml

GroupObjectStorage

Esta clase es la encargada de instanciar un objeto de cada tipo de la familia de Storage, utilizando al ConnectionManager.

A cada objeto Storage le es pasado el ConnectionManager para que selecciones la o las conexiones necesarias para su persistencia.

Obsérvese que es posible que cada instancia de objeto Storage pueda administrar un medio de almacenamiento distinto, incluso puede un mismo objeto administrar varios medios de almacenamiento simultáneamente.

En esta clase se explicita la relación de correspondencia entre los objetos lógicos y su Storage correspondiente, el objeto lógico utilizará la instancia de Storage que se encuentre cargada en la etiqueta definida con su propio nombre en la clase GroupObjectStorage.

Dispatcher

Recordemos que el Dispatcher es el encargado de instanciar los objetos lógicos.

Los objetos lógicos no tienen conocimiento de cómo se realiza su persistencia por lo cual deben delegar esa responsabilidad a agentes externos.

Asimismo el Dispatcher también mantiene independencia de esta relación entre objetos lógicos y Storage.

Esto se logra otorgando al objeto lógico la tabla de objetos GroupObjectStorage para que el mismo obtenga su clase Storage.

Definición de medio de persistencia

Podemos entonces a modo de ejemplo definir una tabla en la cual cada objeto Storage puede utilizar una o varias conexiones para su persistencia.

		Connection Manager		
		SQL	OOBD	FILE
Group Objects Storable	EMS	X		
	ME	X		X
	LINK		X	
	TerminalPoint	X		
	..	X		
	..	…		

Así entonces el objeto ManagedElement puede ser persistido en un DB relacional así como parte del mismo es salvado en un archivo plano.

Y los objetos Link son únicamente persistidos en una DB orientada a objetos.

11.4.7. TreeViewManager

Como se menciono anteriormente esta clase brinda servicios a la fachada TreeViewObject y posee las primitivas **AddRelation, ClearRelations, ExistRelation, GetAllChilds, GetAllParents, RemoveRelation, InitializeTreeView.**

Las relaciones entre objetos lógicos son salvadas actualmente en la base de datos relacional dentro de la tabla “GENERIC_TREE”.

Es importante mencionar que existe un caso de uso en el cual esta le brinda servicios al Dispatcher. Éste se origina cuando es utilizado el método **delObject** en el Dispatcher por lo cual éste invoca el método **ClearRalations** para eliminar todos los posibles vínculos que tuviera el objeto a eliminar dentro de la tabla de relaciones como padre o como hijo. Así pues un objeto puede estar incluido en varios vínculos, como hijo y como padre sin importar la cantidad de apariciones del mismo.

Cada tupla de la tabla donde se almacena físicamente el vínculo tiene la siguiente estructura, donde todos los atributos son clave.

TIPO PADRE	ID_PADRE	TIPO HIJO	ID_HIJO

11.4.8. OmniVision

Esta clase se originó debido a la necesidad de poder recuperar objetos sin conocer su tipo por medio de determinados atributos clave, dentro de todo el conjunto de objetos del inventario.

Esta recuperación puede realizase por el **timeStamp de update** o por el atributo **Full Description Name** (FDN) que es único para todos los objetos del inventario.

La clase OminiVision tiene los métodos:

- **FindObject**: recupera un objeto del inventario por medio del **tipo** y **FDN** pasados como parámetro.
- **GetChangedObject**: recupera todos los objetos que han sido creados o actualizados desde un tiempo determinado dentro del inventario.
- **GetUnchangedObject**: retorna el complemento de objetos del método anterior.

11.4.9. Subcomponente South

El subcomponente “South”, es el encargado de consumir y procesar lo eventos que llegan al inventario, así también de realizar la sincronización cada vez que se inicia. Está incluido en el servidor de Inventario, por lo que no funciona en forma independiente.

Se tomó como base para la implementar este componente la idea de WINMAN de utilizar una Interfaz sur. WINMAN maneja la Interfaz sur como un componente independiente que forma parte de toda la capa NMS (tanto para el Inventario, como para Routing, y Provisioning), mientras que nuestro diseño el subcomponente South es parte del inventario y solo se comunica con éste. Vamos a especificar este subcomponente en 2 partes, la sincronización y el manejo de Eventos.

11.4.9.1. Sincronización

El sincronizador obtiene las interfaces correspondientes y consulta tres gestores en forma sincrónica; ellos son:

- **EMS Manager**: Es el primero que se invoca y a partir de éste se consultan los sistemas administradores de elementos (EMS - “Element Manager System”),las Subredes (MLSN - “Multi Layer SubNetwork”) y los links (TL - “Topological Link”) pertenecientes a estos.
- **ME Manager**: En éste se consultan los elementos gestionados (ME - “Managed Element”), y toda la herencia a partir de este. Esto implica que se procesan los puntos terminales (PTP – “Physical Termination Point”, y CTP – “Connection Termination Point”).
- **MLSN Manager:** Por último se consultan todos los elementos que proporciona este manager, en el cual se encuentran los elementos de conexión, tales como las conexiones de red (SNC – “SubNetwork Connection”) y las Subredes en caso de ser generadas por algún evento.

Este proceso actualiza el inventario elemento por elemento, creando los nuevos, actualizando los existentes y finalmente borrando los que no existan más. Esto provoca que el tiempo de sincronización crezca linealmente con la cantidad de objetos existentes en el EMS. Esto no es mejorable dado las primitivas provistas por MTNM que no nos permiten consultar objetos modificados o borrados.

11.4.9.2. Captura y Procesamiento de eventos.

Una vez que el Inventario está sincronizado con la persistencia del EMS, está pronto para empezar a recibir eventos del canal. Estos eventos serán generado por cada componente del EMS, y consistirán en varios tipos de eventos, tales como CREACIÓN, ELIMINACIÓN, MODIFICACIÓN, Cambios de Estado y otros que no serán tomados en cuenta para ésta versión del inventario. Luego para cada tipo de elemento el procedimiento es similar; consistiendo en que cada vez que llega un evento se llama al procesador del manager correspondiente en la mayoría de los casos; luego si el estado del elemento lo amerita se consultan a todos los elementos hijos del mismo.

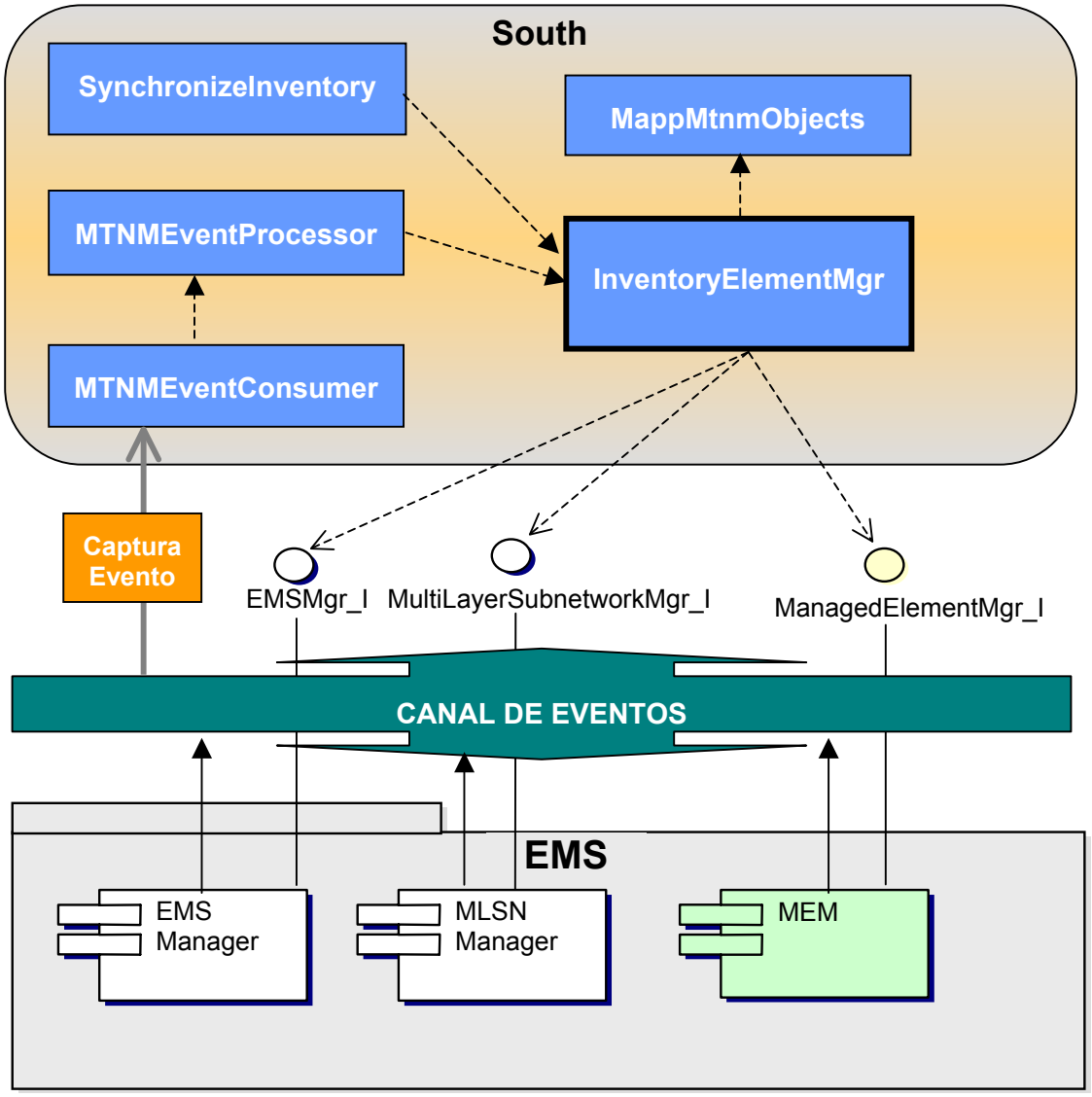


Figura 11-19 - Subcomponente South

Canal de Eventos

El canal de eventos es creado por el EMS y ofrece un mecanismo para mantener, a quien se suscriba, actualizado con los cambios en la capa de elemento.

InventoryElementMgr

Este objeto se invoca tanto en la sincronización inicial como al procesar un evento. Éste es el ejecutor de toda acción en el inventario proveniente del EMS, es decir que si cambia algo en el EMS, es el que hace los cambios en el Inventario. Es el único que accede a los servicios internos del inventario utilizando el Dispatcher, el OmniVision, y el TreeViewManager.

MappMtnmObjects

Es el encargado de construir los objetos internos del Inventario a partir de un objeto MTNM.

SynchronizeInventory

Esta clase se ejecuta por única vez al iniciarse el subcomponente South y se encarga de hacer la sincronización mencionada anteriormente.

MTNMEventConsumer

Este objeto implementa una interfaz InterfaceNotifConsumidor que permite al canal de eventos la invocación del método pushStructuredEvent para informar de un nuevo evento. Éste recibe el evento, y en caso de pertenecer al dominio MNTM, pasa el control a MTNMEventProcessor.

MTNMEventProcessor

Esta clase se encarga de procesar el evento distinguiendo que tipo de operación es (creación, eliminación, modificación y cambio de estado) y que tipo de elemento, para finalmente pedirle a InventoryElementMgr que actualice el Inventario.

11.4.10. Configuración

El inventario dispone de una aplicación de configuración de la conexión a la base de datos, esta configuración es almacenada en un archivo XML que es levantado por el inventario al inicializar los servicios de conexión.

Esta interfaz además de configurar la conexión y la cantidad manejada por el pool, permite crear las tablas de la base y configurar los tipos de datos para cada manejador.

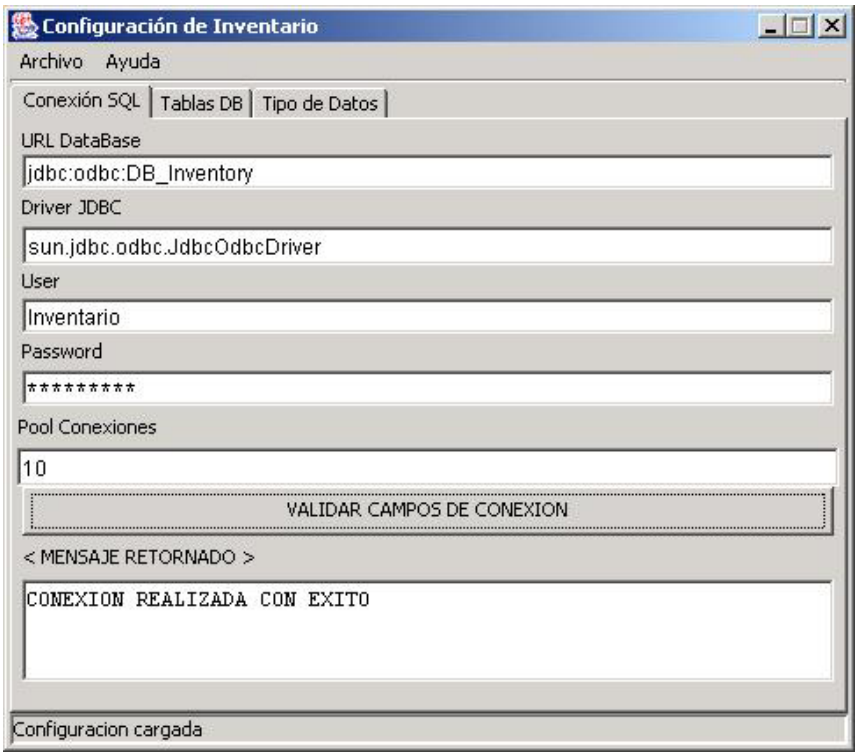


Figura 11-20 - Interfaz de configuración de conexión

11.5. Extensibilidad

Extensión de objetos lógicos

El inventario está especialmente diseñado para poder agregar nuevos objetos lógicos.

Los requerimientos que deben cumplir estos nuevos objetos son relativamente simples, debiendo extender estos objetos de la clase base **LogicObjectStorable** u otro objeto lógico existente e implementar 2 métodos para carga y retorno serializado de sus atributos.

El punto especialmente importante radica en su objeto Storage asociado que explicaremos a continuación.

Extensión de objetos Storage

Todo objeto lógico que quiera ser persistido deberá poseer un objeto Storage asociado que tenga conocimiento de los atributos que contiene éste y pueda dialogar con alguna de las formas de conexión que ofrece la clase ConnectionManager.

En caso de necesitar un forma no existente de conexión puede extenderse el conjunto de conexiones a medios de persistencia.

Extensión de conexiones

La clase **ConnectionManager** es la encargada de inicializar y otorgar el conjunto de conexiones disponibles a los objetos Storage.

Esta clase puede ser extendida e incorporar nuevas conexiones que podrán ser utilizadas por los objetos Storage para persistir a los objetos lógicos.

Extensión de servicios ofrecidos

Las fachadas así como las clases de la capa de servicio pueden ser extendidas para permitir mayores funcionalidades.

El diseño modular permite la incorporación de nuevas clases sin provocar mayor impacto en la arquitectura del inventario y permitiendo un alto grado de navegabilidad de las mismas hacia cualquier otro componente existente por medio de una "**estructura de referencias**" disponibles en todo el inventario.

TERCERA PARTE: Conclusiones

12. Conclusiones

Primero que nada debemos decir que se cumplieron los objetivos del proyecto en tiempo y forma:

- Se investigó y desarrollo en el área de gestión de redes.
- Se diseño un sistema de gestión capaz de configurar una red.
- Se implementó un Inventario de Red.
- Se desarrollo un componente capaz de configurar los dispositivos de red independientemente de la tecnología subyacente.
- Se logró coordinar con un proyecto simultaneo en el diseño y desarrollo de un sistema con un objetivo común, salvando los riesgos que esto implicaba.
- Se sentaron las bases para futuros proyectos en el área de gestión de red y en el desarrollo de un sistema de gestión de red, que permita estudiar en el área.

Otro aspecto a resaltar del proyecto es haber utilizado un estándar de la industria como es MTNM con su tamaño y complejidad, siendo ampliamente rigurosos en su implementación. Éste promete ser el futuro "estándar de hecho" en gestión de redes.

Es importante notar el gran aporte del paradigma de desarrollo basado en componentes, el cual resultó muy útil con las interfaces definidas en MTNM y las definidas para interactuar entre proyectos. Esto nos dio mejores resultados de los previstos en la interacción con el proyecto mitiNum.

Logramos crear un producto sin sobrecarga de tecnologías auxiliares las cuales hubieran aumentado la carga de la aplicación, como pueden ser servidores de aplicación (ejemplo J2EE) o herramientas propietarias (ejemplo DCOM).

El uso Java y CORBA nos permitió obtener una performance aceptable para esta aplicación, aunque existen otras plataformas más rápidas a nivel de programación distribuida y lenguajes no interpretados. Además se uso nos asegura una extensibilidad amplia y multiplataforma.

El uso de JDBC fue de gran utilizada (como esperábamos) ya que no tuvimos problemas en los múltiples manejadores utilizados.

12.1. Dificultades del Proyecto

- Falta de Requerimientos concretos inicialmente, dado que estos se desarrollaron durante le proyecto. Así también no estaba perfectamente

definido el alcance y limites del proyecto. Si bien teníamos objetivos concretos, faltaban los requerimientos.

Inicialmente decidimos almacenar directamente todos los objetos accesibles de capa de elemento, dado que ignorábamos los servicios que debíamos prestar. Esto se debía a que el proyecto mitiNum (nuestro usuario) no tenía aún sus necesidades definidas. Luego se aplicaron ideas del proyecto WINMAN para definir la interfaz. Finalmente ofrecimos dichas interfaces a mitiNum y fueron aceptadas.

- Gran Interacción con otro proyecto en desarrollo. Si bien, gracias al espíritu conciliador de ambos grupos se logró sobrellevar las diferencias que surgieron, el riesgo fue grande.
- Falta de una plataforma concreta que permitiera bajar a tierra los conceptos abstractos (que tiene un fondo concreto) manejados por los estándares utilizados. Por ejemplo, implantación en una red concreta.
- Fatal de experiencia y conocimiento en redes de gran porte donde se apliquen estos conceptos para facilitar nuestra motivación. Así como tampoco tuvimos acceso a un sistema de gestión real o sistemas con características similares a las nuestras (con excepción de WINMAN).

12.2. Lecciones aprendidas y Aportes

- Adquirimos conocimiento sobre la gestión de redes y su alcance.
- Aprendimos a trabajar en CORBA y la potencia del paradigma de desarrollo basado en componentes.
- Tuvimos una gran experiencia en la implementación de un estándar del tamaño de MTNM.
- Comprendimos el diseño WINMAN y aplicamos hasta cierto punto sus ideas.
- También tuvimos una gran experiencia al desarrollar un proyecto conjunto entre dos grupos de proyectos de grado con sus ventajas y dificultades.
- Creemos haber hecho un aporte al estudio de la Gestión de Redes y haber dejado el camino abierto para que en futuros proyectos se pueda seguir con el estudio de la gestión de redes.

12.3. Mejoras y Trabajos Futuros

- Se podría mejorar la integración con el resto de los componentes del EMS a través de un diseño en común.
    - No se respetó la seguridad básica que ofrecía MTNM, publicándose los componentes en el servicio de nombres y no garantizándose entonces que se obtenían las sesiones a través del emsSessionFactory, que fue desarrollado por mitiNum.
    - También se optó entre diferente alternativas (entre proyectos) a la hora de resolver problemas iguales dentro del EMS. Como ejemplo

de esto podemos tener la forma en que se implementaron los iteradores o el diseñó interno de cada componente.

- La independencia de componentes provocó que no se pudieran hacer controles de consistencia estrictos entre componentes y que se tuvieran que replicar estructuras de datos. Esto hubiera permitido además una persistencia única.

- Como un aspecto a ser mejorado del estándar MTNM, respecto a la sincronización del inventario, resultaría mucho más eficiente la existencia en capa de elemento de un mecanismo para obtener los elementos actualizados y eliminados en la misma desde el último momento de comunicación con dicha capa, este mecanismo podría ser basado en el manejo de un time-stamp en la capa de elemento. Esto evitaría el problema el tiempo de sincronización inicial crece linealmente con el número de objetos del inventario.
- En el componente Inventario se podrían hacer mejoras para incrementar el desempeño:
  - Permitir escritura retrasada del buffer a la DB, para evitar updates seguidos del mismo objeto cuando el usuario modifica varios atributos del mismo, de a uno.
  - Implementar servicios de bloqueo de objetos y servicios del inventario para control de competencia entre clientes concurrentes.
  - Agregar un servicio capaz de actualizar y/o crear un objeto sin necesidad de consultarlo, para reducir los accesos a la DB en el caso promedio.
- Tener un caso de estudio real. Sería muy bueno lograr una plataforma aunque sea pequeña donde se pudiera experimentar con el sistema desarrollado.
- Hacer módulo de Discovery. Quedó la puerta abierta para que en la interfaz ConfigMEM_I se pueda conectar un módulo que corra en una red particular y pueda configurar los drivers adecuados para gestionar dicha red.
- Hacer drivers que pudieran trabajar con algún tipo de simulador.
- Seguir el desarrollo de un sistema de gestión de red completo e independiente donde se puedan probar nuevas técnicas y algoritmos de ruteo.
- Ampliar el sistema para incluir funciones de gestión de fallos.

13. Glosario

ANSI	American National Standards Institute
ASN.1	Abstract Syntax Notation One
ATM	Asynchronous transfer mode
BGP	Border Gateway Protocol (RFC 1267)
BLA	Business-Level Agreements
CaSMIM	Connection and Service Management Information Modeling team
CCITT	Comité Consultative Internationale de Telegraphique et Telephonique
CMIP	Common Management Information Protocol (ISO)
CMIS	common management-information service
CORBA	Common Object request Broker Architecture
CORF	Common Object Representation Format
CoS	Class of Service
DCOM	Distributed Components
DWDM	Dense Wavelength Division Multiplexing
EJB	Enterprise Java Beans.
EML	Element Management Layer
EMS	Element Management System
ETSI	European Telecommunication Standards Institute
FCAPS	Fault, Configuration, Accounting, Performance, Security
FDDI	Fiber Distributed Data Interface
FEC	Forward Equivalence Class
IETF	Internet Engineering Task Force
IDL	Interface Definition Languaje
ISO	International Organization for Standardization
ITU	International Telecommunication Union
J2EE	Java 2 Enterprise Edition
JAAS	Java A A Security
JAXM	Java Api for Messaging
JMS	Java Messaging Service
LDP	Label Distribution Protocol
LER	Label Edge Router
MIB	Management Information Base
MPLS	Multiprotocol Label Switching
MTNM	Multi-Technology Network Management

NEL	Network-Element Layer
NML	Network Management Layer
NMS	Network Management System
OEM	Original Equipment Manufacturer
ORB	Object Request Broker.
OSI	Open System Interconnection
OSPF	Open Shortest Path First (RFC 1583)
OSS	Operations Support Systems
OXC	Optical Cross-Connect
PDH	Plesynchronous Digital Hierarchy
QA	Q adapter
RIP	Routing Information Protocol (RFC 1058)
RPC	Remote Procedure Call
SAAJ	SOAP Attached API for Java
SDH	Synchronous Digital Hierarchy
SIF	Synchronous Optical Network (SONET) Interoperability Forum
SLA	Service Level Agreement
SML	Service-Management Layer
SMS	Service Management System
SNMP	Simple Network Management Protocol
SOAP	Simple Object A Protocol
SONET	Synchronous Optical Network
TDM	Time Division Multiplexing
TOM	Telecom Operations Map
VPI	Virtual Packet Identifier
WDM	Wavelength Division Multiplexing
XML	Extended Markup Language

14. Referencias

[1] Network Management Architectures, Aiko Pras, Ph. D-thesis 1995

[2] OSS Solutions for Network Operators – white paper, 2002, Lars Andersson April 2002, Teleca

[3] TMN Versus SNMP-Based Network Management Systems, J.Luque, F.Gonzalo, J.I.Escudero, A.Carrasco, Octubre 1999

[4] Internetworking Technologies Handbook, Cisco

[5] Telecom Operations Map GB91 Version 2.1, TeleManagement Forum, March 2000

[6] Network Management Detailed Operations Map, GB908

[7] International Engineering Consortium, Proforum Tutorials, 2001

[8] Cisco "Introduction to DWDM for Metropolitan Networks", www.cisco.com/univercd/cc/td/doc/ product/mels/dwdm/dwdm_fns.htm

[9] An Overview of DWDM Networks, Shaowen Song, Wilfrid Laurier Universitiy,Waterloo, IEEE Canadian Review - Spring / Printemps 2001

[10] Redes de Computadoras, Tercera Edición, Andrew S. Tanenbaum, 1997

[11] El MPLS en las Redes Ópticas, Neil Jerram, Octubre 2001

[12] MPLS: Convergencia entre el nivel de transmisión y el nivel de enrutamiento, Ana González, Diciembre 2002

[13] Stallings, W. SNMP, SNMPv2, and CMIP. Don Mills: Addison-Wesley, 1993

[14] Stallings, W. SNMP, SNMPv2, SNMPv3, and RMON 1 and 2, Third Edition. Reading, MA: Addison-Wesley, 1998.

[15] Red Book IBM Chapter 15. Network management

[16] Hands-On SNMPv3 Tutorial & Demo Manual –NuDesign Team, Inc.

[17] IEEE COMMUNICATIONS SURVEYS "SNMPv3: A Security Enhancement for SNMP" William Stallings

[18] http://www.cellsoft.de/telecom/cmip.htm

[19] TMF 513 Multi-Technology Network Management Business Agreement

[20] TMF 508 Connection and Service Management Business Agreement, Version 3.0, April 2001

[21] TMF 608 Multi-Technology Network Management

[22] TMF 814 Multi-Technology Network Management Solution Set

[23] http://www.winman.org/

[24] http://www.adventnet.com/

[25] http://www.omg.org/technology/documents/forma1/concurrency_service

[26] http://citeseer.nj.nec.com/vinoski97corba.html

[27] http://www.java.sun.com/

[28] Manual de ayuda WSAD (Websphere Studio Application Server)

[29] http://www.microsoft.com/

[30] http://www.tl1.com/library/TL1/Overview/

15. Apéndice A: Estudio de Performance

15.1. Nivel de Carga

Para la evaluación de la performance del proyecto se debieron realizar determinadas consideraciones.

- En primer lugar no tenemos conociendo de precedentes en el tema para tomar como punto de referencia en cuanto a su desempeño esperado o deseado.
- No tenemos una perspectiva de la cantidad de componentes que puede tener un red que sea gestionada por un sistema de este tipo.

Consecuentemente consultamos a personal técnico vinculado a ANTEL, que es la mayor empresa de servicios de conectividad del país. De esta consulta obtuvimos un valor aproximado de 500 ME, el cual fue utilizado para las pruebas realizadas.

15.2. Condiciones de Medición

No se necesitaba alta precisión en las mediciones dado que el objetivo de este estudio era tener valores de tiempo de las operaciones complejas, conociendo su implementación (caja blanca) y estudiar su crecimiento con el aumento de tamaño. Por este motivo las mediciones de tiempo fueron realizadas simplemente cronometrando manualmente entre salidas.

Las mediciones de memoria fueron realizadas con herramientas del sistema operativo bajo idénticas condiciones y variando la carga.

15.3. Aspectos medidos

Se midió para cada componente (MEM e Inventory) los valores de performance pertinentes a su implementación. Ambos componentes poseen paradigmas distintos de funcionamiento. El MEM lee toda la estructura de datos al inicio, la maneja en memoria y la salva cuando hay cambios. Por otro lado el Inventory trabaja con una DB en vez de utilizar la memoria.

MEM

Para hacer las menciones en el componente MEM se utilizaron 500 ME con 16 Termination Points en cada uno (en dos niveles).

Consumo de Memoria

A medida que aumenta la cantidad de elementos administrados por el MEM aumenta la cantidad de memoria consumida, dado que por cada elemento se debe instanciar una clase Driver en memoria. Aunque la clase Driver es abstracta y cada implementación puede tener un consumo de memoria diferente, dicha implementación es un referente para tomar en cuenta.

Aumento del consumo de memoria al pasar de 0 a 500 ME: 14MB

Valor obtenido sobre Windows 2000.

Tiempo de Carga

El tiempo de carga del MEM aumenta al con la cantidad de MEs, dado que se demora en leer e instanciar cada uno de ellos. En este punto es importante aclarar que el tiempo de carga podría aumentar sustancialmente en un ambiente real pues cada ME podría demorar un tiempo considerable en conectarse al dispositivo real o realizar algún chequeo en éste. Lo que queda demostrado es la baja incidencia del software de soporte.

Aumento de tiempo de carga al pasar de 0 a 500 ME: 4 segundos

Valor obtenido con un procesado Intel Celeron 600 MHz sobre Windows 2000, no se repitió en otros equipos por su baja incidencia.

Tiempo de Salvado

Es importante analizar el tiempo de salvado pues en la implementación se graba un archivo XML con todos los datos, este tiempo aumenta linealmente en función de la cantidad de ME.

Aumento de tiempo de carga al pasar de 0 a 500 ME: 1 segundo

Tamaño del archivo XML salvado: 915 KB

Valores obtenidos con un procesado Intel Celeron 600 MHz y disco de 7500 RPM sobre Windows 2000, no se repitió en otros equipos por su baja incidencia.

Inventory

El hecho de que el inventario esté implementado sobre una base de datos relacional permite tener mayor confiabilidad en los datos y que los tiempos de respuesta (almacenamiento y acceso) se mantengan prácticamente constantes independientemente del volumen de datos. En contrapartida se requiere un mayor tiempo de acceso a los datos porque están en disco.

Tiempos por Operación

Todas las operaciones simples (que involucren el retorno o la actualización de un solo objeto) que se realizan sobre el inventario requieren un número constante de accesos a la base, la cuales se realizan a través de índices que garantizan un tiempo de ejecución prácticamente constante.

Sincronización

La sincronización es el proceso en el cual el inventario verifica todos sus elementos en capa de red. Este proceso se ejecuta al arrancar el inventario y su tiempo aumenta linealmente a mediad que aumenta la cantidad de objetos a sincronizar. Este se debe a que el numero de operaciones aumente linealmente con la cantidad de objetos y como vimos las operación son constantes.

Para ilustrar estas afirmaciones veremos los resultados empíricos.

Cantidad ME	Minutos	Segundos
500	4:23	263
400	3:31	211
300	2:35	155
200	1:46	106
100	0:54	54
0	0:01	1

Procesador PentiumIV 2.4 GHz, 512 MB RAM , HDD 7.200 r.p.m, Windows 2000 - SQLServer 2000

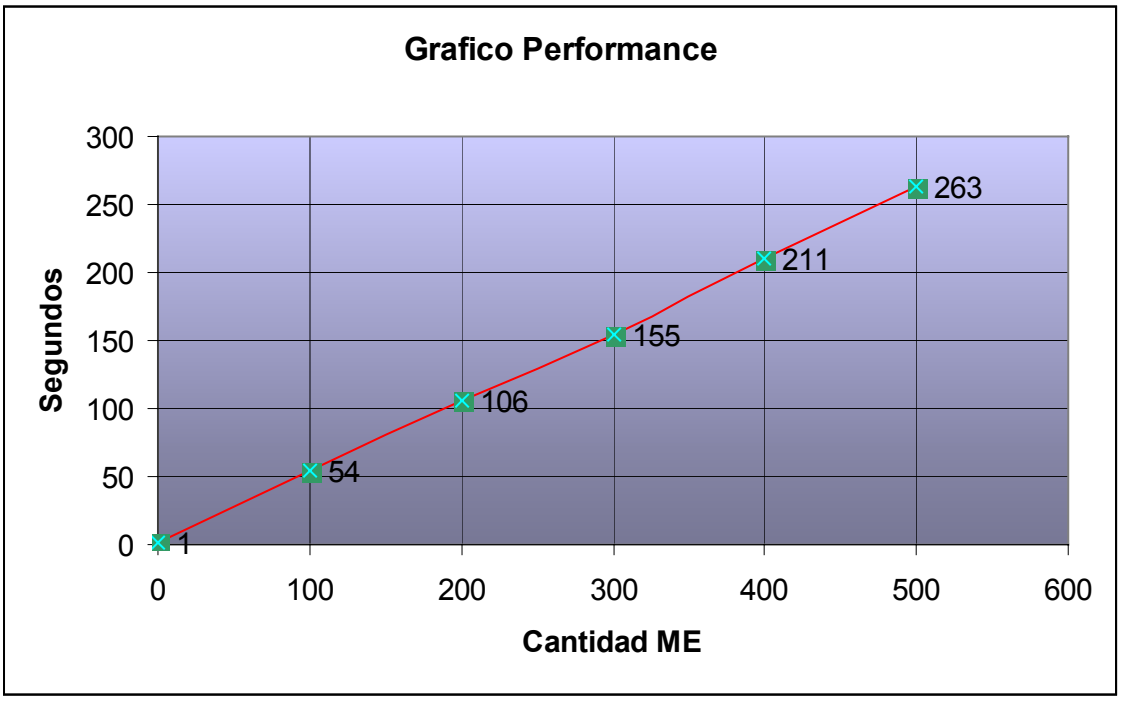


Figura 15-1 - Tiempo de sincronización en función de cantidad de ME

El punto mas débil en el tema performance es el crecimiento lineal en el proceso de sincronización. Realizando la mejora propuesta al estándar MTNM como mencionamos en “Mejoras y Trabajos Fututos” el tiempo de sincronización dependería de la cantidad de elementos modificados o agregados desde la ultima sincronización, siendo cuantitativamente menor al actual. De todas maneras el inventario puede ser utilizado mientras se sincroniza.

16. Apéndice B: Herramienta Configuración BD

16.1. Contexto

El inventario ofrece una herramienta de configuración que permite el seteo de los parámetros de conexión y la creación de una estructura de datos (tablas e índices) apropiada en la base de datos. Ésta debe ser utilizada previamente a la inicialización del inventario y ser configurada apropiadamente para su correcto funcionamiento.

Esta aplicación salva los valores de configuración en un archivo **"connections.xml"** que luego es cargado por el inventario al inicializarse. Se debe tener especial cuidado con respecto a los permisos de acceso de este archivo ya que contiene el password del usuario en forma plana, el cual es utilizado por el inventario para conectarse a la BD.

Se debe crear la Base de Datos vacía antes de comenzar a utilizar esta herramienta. También se deberá definir un usuario y asignarle los permisos correspondientes para esta base.

16.2. Utilización

La herramienta consta de una barra de menú y un tab control de tres lengüetas. En el menú de manejo del archivo de configuración se puede cargar la configuración, salvarla y cerrar el archivo. El archivo es único y no permite salvarlo con un nombre distinto (pues es el nombre con el cual lo busca el inventario). Solo se puede salvar la configuración o cerrar el archivo luego de haberlo cargado, para evitar sobrescribir configuraciones correctas en caso de una equivocación.

En todo momento se tiene una descripción del estado en el campo de status de la aplicación. Ver siguiente figura.

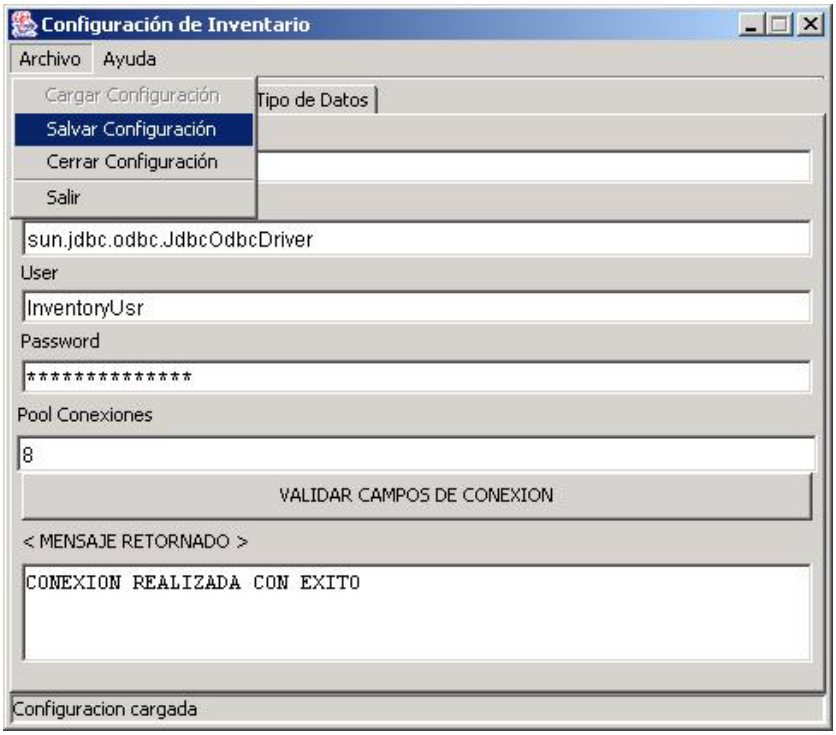


Figura 16-1 - Manejo Configuración Inventario

16.3. El archivo “connections.xml”

Este contiene información para la conexión con un manejador de base de datos relacional, así como permite añadir información de otras posibles conexiones como por ejemplo DBOO o archivos de texto plano. Si no se dispone de un archivo de configuración inicial puede ser creado como muestra la siguiente figura.

```
- <CONNECTIONS>
      - <SQL>
            <URL>jdbc:odbc:DB_Inventory</URL>
            <DRIVER>sun.jdbc.odbc.JdbcOdbcDriver</DRIVER>
            <USER />
            <PASSWORD />
            <POOL_CONN>10</POOL_CONN>
        </SQL>
      - <TYPES_SQL>
            <INT>INT</INT>
            <LONG>BIGINT</LONG>
            <DOUBLE>BIGINT</DOUBLE>
            <FLOAT>FLOAT</FLOAT>
            <VARCHAR>VARCHAR</VARCHAR>
            <VC10>30</VC10>
            <VC19>20</VC19>
            <VC20>20</VC20>
            <VC50>50</VC50>
            <VC250>250</VC250>
        </TYPES_SQL>
  </CONNECTIONS>
```

Figura 16-2 - El archivo connection.xml

El archivo se divide en dos partes principales:

- Conexiones: esta contiene todos los tipos de conexiones posibles y sus parámetros.
- Tipos SQL: Contiene los atributos de mapeo de tipos con el manejador de base de datos relacional utilizado.

16.4. Configurando la conexión

Como se puede ver en la siguiente figura, se tienen cinco campos que deben ser ingresados para determinar una conexión a una base de datos relacional, como son la **ruta** (URL) de la base, el **driver JDBC**, **usuario**, **contraseña** y **Pool Conexiones**.

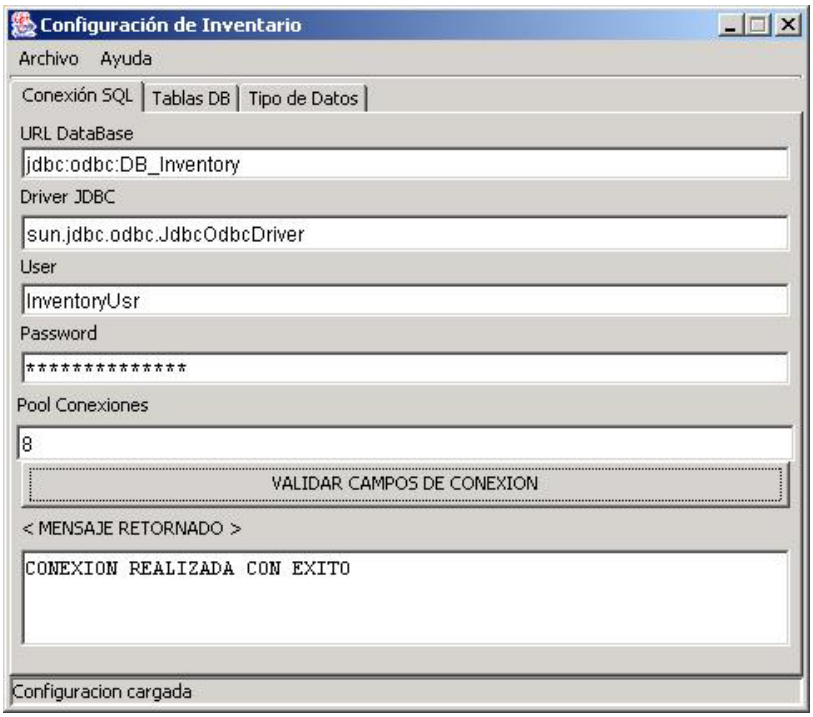


Figura 16-3 - Parámetros de Conexión

Los campos **ruta, driver, usuario** y **contraseña** son los que necesarios para JDBC pueda establecer conexión con la BD. El campo **Pool Conexiones** indica la cantidad de conexiones concurrentes que realizará el inventario con la base de datos.

Se tiene a disposición el botón "**VALIDAR CAMPOS DE CONEXIÓN**" que permite chequear si la conexión ha sido seteada correctamente.

Recuerde que el chequeo no implica el salvado de los datos, luego de chequeada la conexión puede o no salvar la nueva configuración !

El campo "MENSAJE RETORNADO" retorna el mensaje que puede provenir del manejador de BD o de la propia aplicación, dependiendo de cada caso. Esta información permite corregir posibles errores de configuración rápidamente.

16.5. Creación de la estructura de datos

La herramienta permite la creación de las tablas e índices en la BD como puede verse en la siguiente figura.

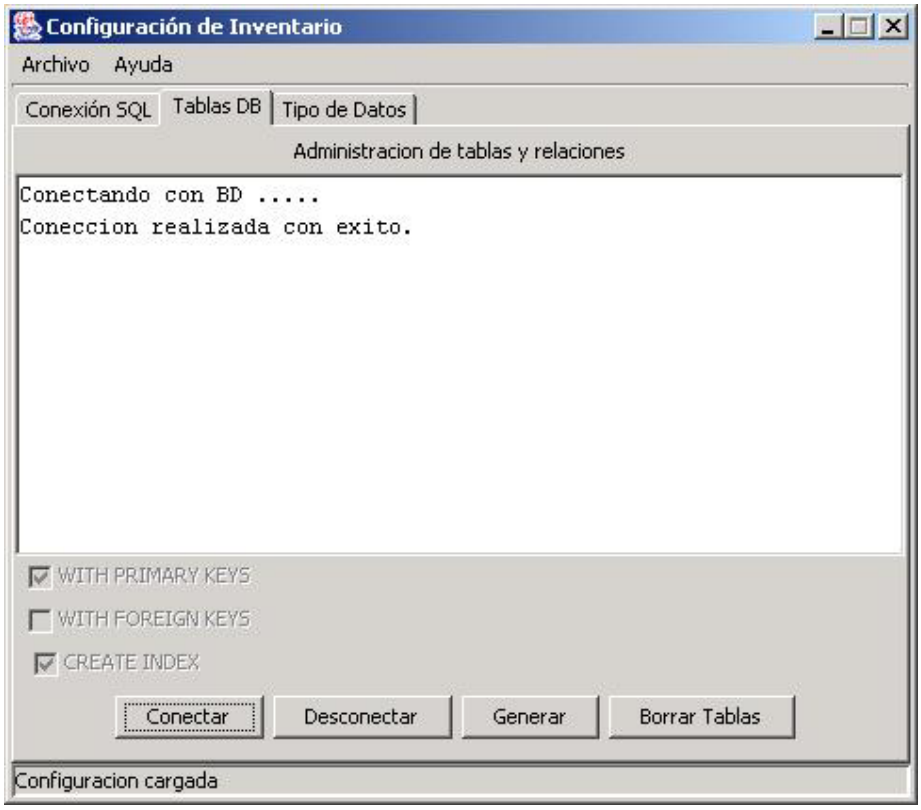


Figura 16-4 - Creación Tablas e Índices

Recuerde que para utilizar esta funcionalidad debe haber configurado una conexión exitosamente y haberla salvado.

Como primer paso debe establecer una conexión con la BD (botón Conectar) y posteriormente puede utilizar las funcionalidades de crear las tablas (botón Generar) o eliminarlas (botón Borrar Tablas).

En el momento de creación de las tablas, también se crean automáticamente las claves primarias y los índices necesarios para que las operación funcione apropiadamente.

Cuando sean creadas o eliminadas las tablas se deberá verificar en el área de texto principal si el proceso se ha concluido satisfactoriamente observando en la última línea.

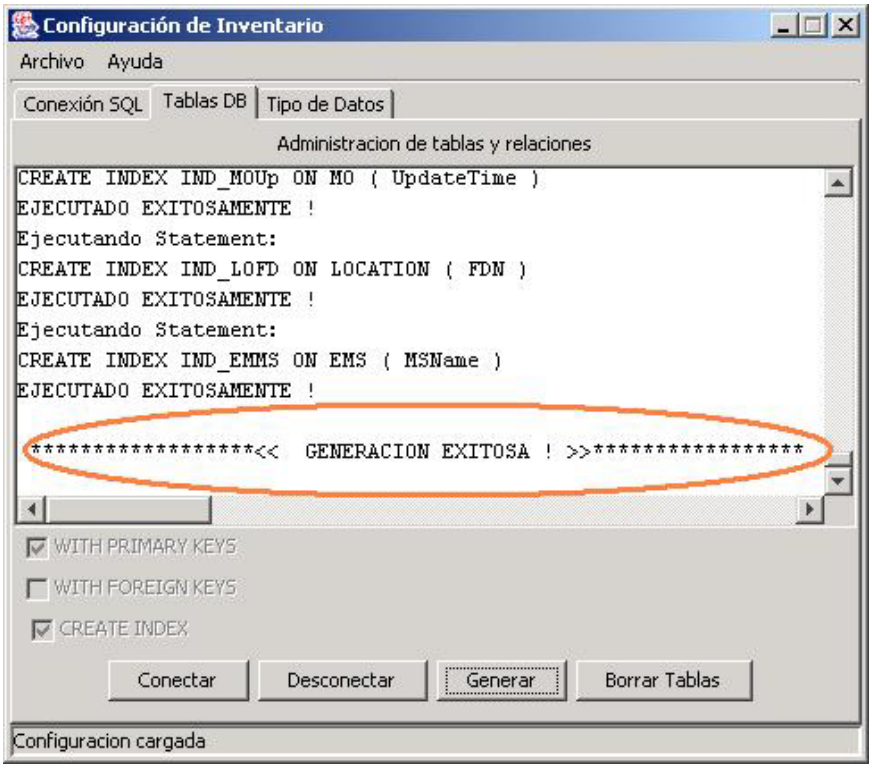


Figura 16-5 - Mensaje Exitoso de Creación de Tablas

En este caso podemos ver la generación exitosa de todo el proceso de creación de las tablas. Luego puede proceder a desconectarse y cerrar la aplicación.

17. Apéndice C: Navegator

El navegador es una aplicación para observar los objetos almacenados en el inventario.

Este se conecta a las fachadas IInventory y TreeViewObject del inventario para obtener la estructura jerárquica de objetos lógicos contenida en el mismo y los atributos de los objetos que se requiera.

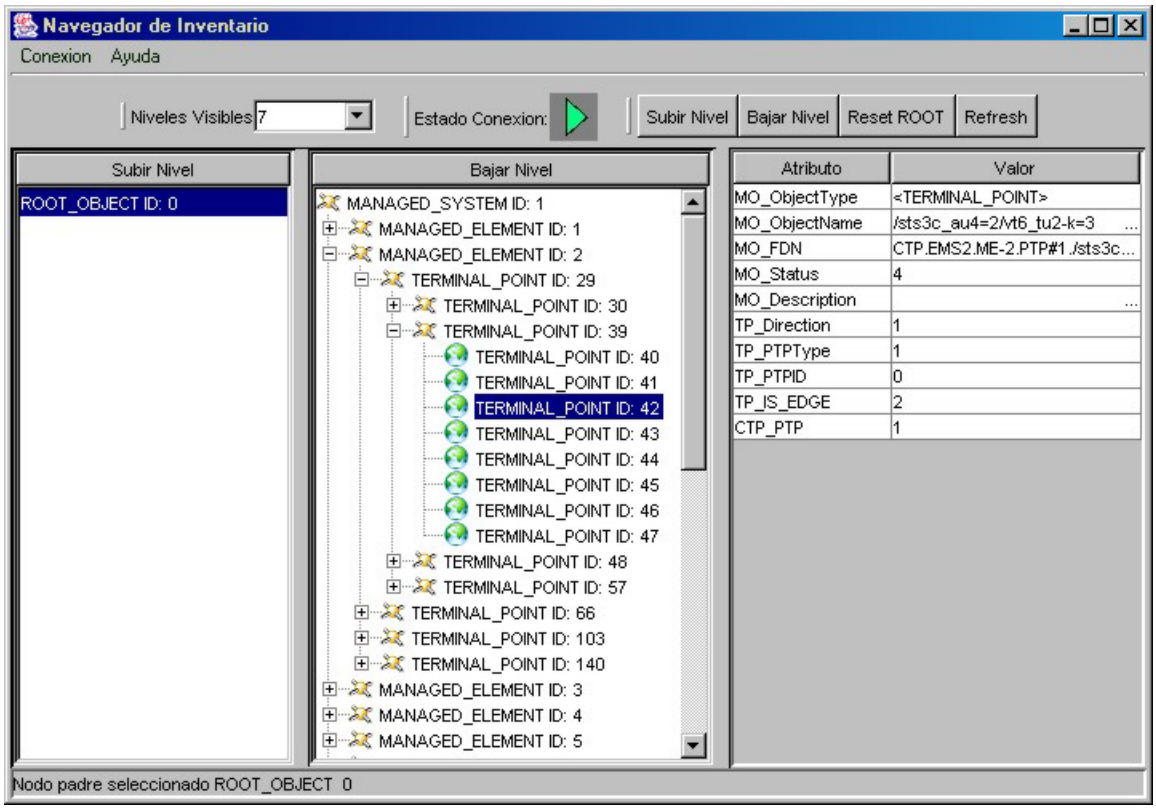


Figura 17-1 - Vista general del Navegador

La aplicación posee en el menú principal **Conexion** las opciones:

- **Establecer conexión:** conecta a las fachadas.
- **Desconectar:** desconecta de las fachadas.
- **Salir:** cierra la aplicación.

Se requiere que los servicios del inventario se encuentren corriendo para que funcione el Navegador !

En la parte superior encontramos una barra de herramientas que contiene:

- **Niveles visibles:** define los niveles visibles del árbol de elementos del inventario que van a ser desplegados.
- **Estado Conexión:** indica si la conexión no fue realizada, falló el intento de conexión o fue establecida con éxito.
- **Botones de navegación** en el árbol jerárquico de elementos que explicaremos posteriormente.

17.1. Navegación

El funcionamiento del navegador radica en la elección de un elemento específico del inventario, del cual se desplegaran los padres, los hijos y sus atributos. Cada uno de estos datos son desplegados en la parte central, en sus respectivos marcos.

Si se desea navegar a elementos descendientes se debe seleccionar alguno de los hijos del elemento y clickear en Bajar Nivel, en dicho caso el hijo seleccionado pasará a ser el elemento principal. Si se desea navegar a elementos superiores se selecciona el padre deseado y se clickear en Subir Nivel, o se puede subir a nivel del objeto raíz clickeado en Reset ROOT.

17.2. Marcos Centrales

En la parte izquierda se encuentra el área de exhibición de padres. En la misma se pueden observar los padres que tiene el elemento seleccionado. Debe recordarse que un elemento podía tener varios padres

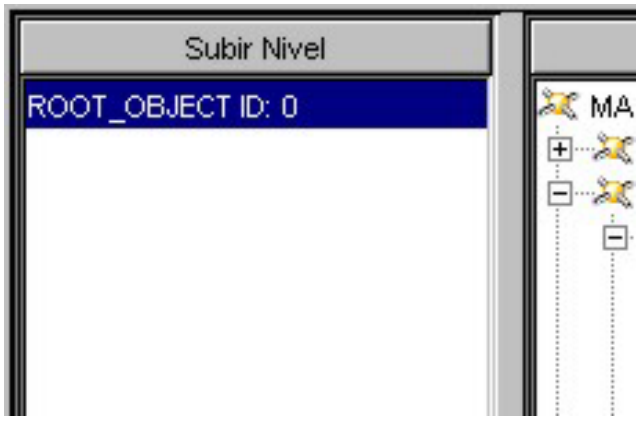


Figura 17-2 - Área de visualización de objetos Padre

En la parte central se encuentra el árbol de elementos hijos del elemento central de navegación. Así entonces, el MANAGED_SYSTEM con ID=1 es el analizado en el ejemplo, y se pueden observar tantos niveles jerárquicos de descendientes como seleccionados en el menú superior.

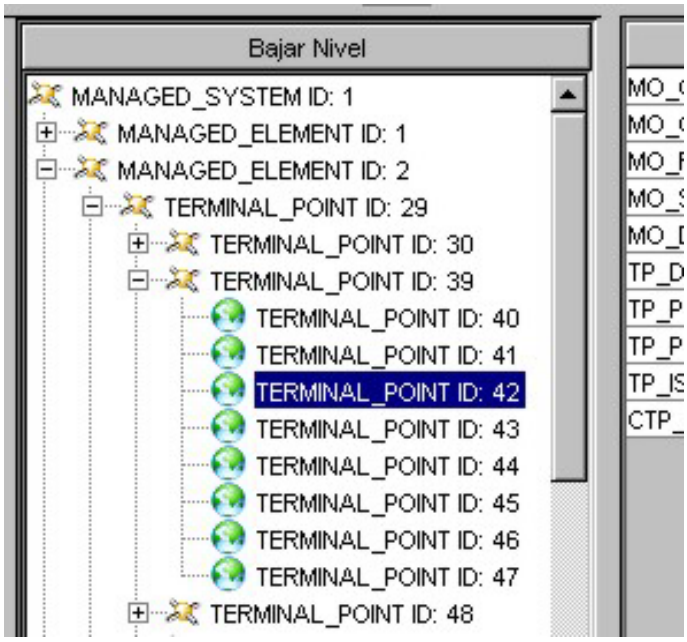
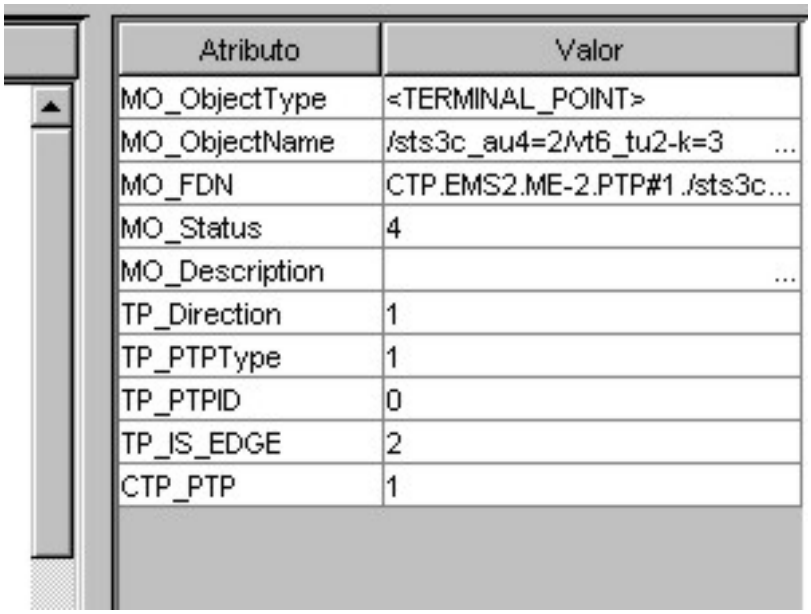


Figura 17-3 - Área de árbol de Hijos

A la derecha podemos observar los atributos del elemento seleccionado.



Atributo	Valor
MO_ObjectType	<TERMINAL_POINT>
MO_ObjectName	/sts3c_au4=2/vt6_tu2-k=3 ...
MO_FDN	CTP.EMS2.ME-2.PTP#1./sts3c...
MO_Status	4
MO_Description	...
TP_Direction	1
TP_PTPType	1
TP_PTPID	0
TP_IS_EDGE	2
CTP_PTP	1

Figura 17-4 - Visualización de Atributos

17.3. Seteo de tipo de datos

Dado que el inventario puede ser conectado a distintos tipos de manejadores de base de datos, puede ser necesario adaptar los tipos de datos establecidos en el inventario con los tipos de datos que soporta el manejador.

Los tipos de datos pueden ser configurados sin tener conocimiento del funcionamiento del inventario, solamente se deben conocer los tipos soportados por el manejador y definir el apropiado para cada tipo usado por el inventario.

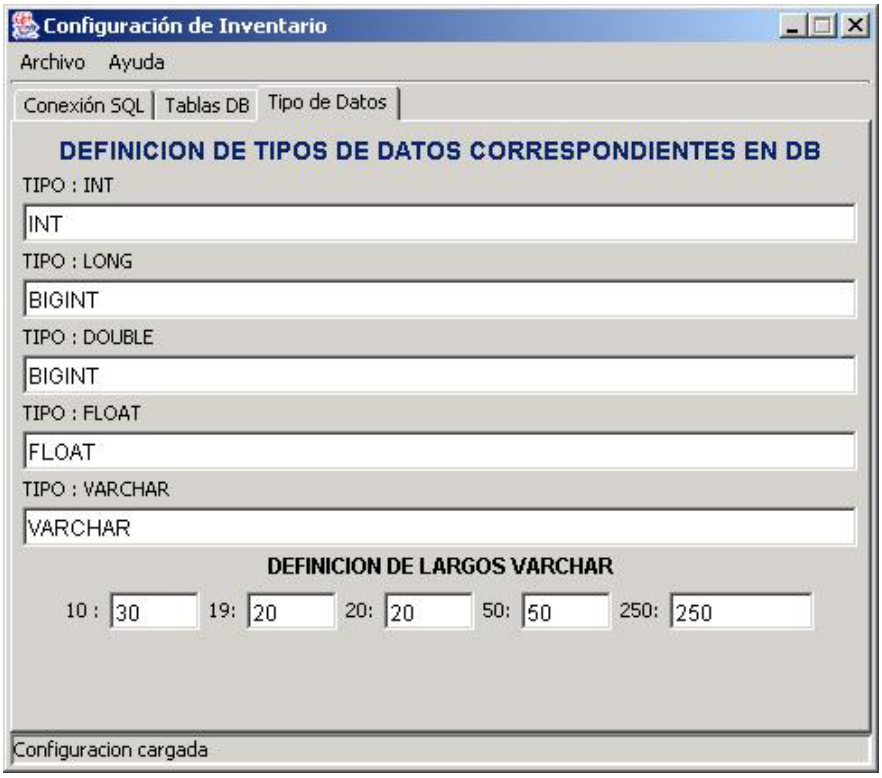


Figura 17-5 - Adaptación de Tipos de Datos

Con respecto a la **definición de largos varchar**, éstos mapean definiciones de largo provenientes de la estructura original de WINMAN y dependen de la configuración interna del inventario. Los valores mínimos para la implementación actual del inventario son los utilizados en la figura. Éstos pueden requerir una modificación en el caso de agregar nuevos objetos a ser manejados por el inventario que requieran largos mayores de algunos campos específicos de las tablas.

Recomendamos no modificar estos campos hasta no tener un conocimiento profundo del funcionamiento del inventario