Extensión 'network - aware' del GUI de Windows

Director:

Ing. Alejandro Martínez.

Alumnos:

Linares de la Cal, Héctor Fabián.
Ríos, Pablo Javier.
Salicioni, Néstor Marino.

Facultad de Ciencias Exactas - U.N.L.P.
Mayo de 1995

La Plata, Mayo de 1995.-

Trabajo de grado **"Extensión 'network - aware' del GUI de Windows"** correspondiente a propuesta entregada en Junio de 1994.

Director:

Ing. Alejandro Martínez

Alumnos:

Linares de la Cal, Héctor Fabián

Ríos, Pablo Javier

Salicioni, Néstor Marino

Indice General

Parte 1 - Introducción

1. 1. Objetivos Generales del Trabajo.

El objetivo es proveer un mecanismo que permita ejecutar cualquier aplicación Windows 3.1 existente en una computadora remota y que su Interfase Gráfica de Usuario (GUI) esté en la workstation local.

En otras palabras, un usuario debería poder interactuar con un programa Windows en una máquina (a la que nos referiremos indistintamente como máquina, computadora o nodo **local**), pero ese programa no necesariamente tendría que estar ejecutándose en la computadora en la que el usuario se encuentra, sino que podría estar ejecutándose en un nodo **remoto** conectado con el primero mediante alguna tecnología de red.

Para lograr esto, en el nodo local debería existir minimamente un **UIMS (User Interface Management System)**, que en principio no necesariamente debería ser el de Windows. Actualmente la mayoría de los sistemas operativos disponen de un entorno gráfico; por ejemplo el entorno gráfico de cualquier plataforma UNIX está basado en el sistema de ventanas X Window System, las computadoras Apple Macintosh tienen su propio sistema de ventanas, el UIMS de OS/2 es Presentation Manager, etc. Ademas, debido a que Windows no implementa un UIMS basado en red, es necesario definir un protocolo de comunicacion a nivel de aplicación para trasladar todos los elementos de la GUI desde el nodo remoto (donde se ejecuta la aplicación Windows) hacia el nodo local. Por último, en el nodo local deberían resolverse los requerimientos gráficos que transporta el protocolo antes mencionado y generar los requerimientos correspondientes sobre el UIMS local.

Por otro lado, como consecuencia de la separación de la GUI de las aplicaciones, hay que disponer de un mecanismo para que el usuario en el nodo local pueda iniciar la ejecución de aplicaciones en el nodo remoto, que presenten su interfase en el nodo local.

1. 1. 1. Consideraciones a tener en cuenta en la solución

- Deberían poder convivir en el nodo remoto aplicaciones Windows ejecutándose normalmente y aplicaciones Windows que estubieran siendo controladas remotamente.

- **Las aplicaciones Windows existentes deberían poder ser controladas remotamente sin modificación alguna.**

- Proponer **una aproximación basada en aplicaciones**. Es decir, trasladar solo los requerimientos gráficos a nivel de aplicación y no a nivel de desktop completo. Las aproximaciones a nivel de desktop completo trasladan todas las salida a pantalla sin diferenciar aplicaciones o el mismo sistema de ventanas (no hacen un 'mapping' de la administración de ventanas). La solución debería realizar una traslación más 'inteligente'.

- La solución debería **considerar diferentes UIMS existentes en el mercado** sobre el cual trasladar la interfase de las aplicaciones Windows, de manera de poder controlar aplicaciones Windows desde distintos entornos gráfico.

- Considerar las cuestiones de **performance** por el hecho de poder tener una red entre la aplicación y su interfase. La pérdida de performance obviamente depende de la tecnología de red existente. Se espera que la solución propuesta tenga una performance razonable si se trabaja con tecnologías de redes LAN. El hecho de poder tener una red WAN, llevaría a

agregar optimizaciones al protocolo de aplicación que si bién nunca se dejan de tener en cuenta, no hace a la escencia del proyecto.

- **No es requesito indispensable mantener el 'look' de una aplicación Windows**.

1. 1. 2. Problemas a solucionar

Basicamente para lograr nuestro objetivo se presentan los siguientes problemas.

- **Separar la interfase de usuario de una aplicación Windows.** Esto implica poder resolver la administración de ventanas y primitivas gráficas en un nodo (recordemos que lo llamaremos nodo local) distinto al nodo donde se ejecuta la aplicación. En términos mas específicos de Windows, será necesario **interceptar los llamados al API de Windows** relacionados con la GUI que haga una aplicación y en lugar de que éstos sean resueltos por Windows en el mismo nodo, trasladarlos hacia otro nodo para que ésto los resuelva.
  Aclaremos que los servicios de sistema (allocación de memoria, creación de archivos, etc.) que invoquen las aplicaciones seguirán siendo resueltos en el nodo remoto, por lo que las invocaciones al API de estos servicios no necesitarán ser interceptados.

- Definir un **protocolo de comunicación** que transporte los requerimientos gráficos desde el nodo remoto hacia el nodo local. Este protocolo gráfico debería definir básicamente los PDUs (Protocol Data Units)[1] que representan a las funciones del API de Windows, resultados de estas funciones, eventos generados desde la GUI, objetos gráficos entre otros.

- **Resolver los requerimientos gráficos** en el nodo local, provenientes desde el nodo remoto haciendo uso del UIMS local. Debido a que el UIMS destino no necesariamente va a ser el de Windows, es que habrá que realizar un **'mapping' de los requerimientos de GUI de Windows** provenientes de del nodo remoto para poder resolverlos sobre el UIMS local. Además será necesario enviarles a las aplicaciones en los nodos remotos las respuestas a sus requerimientos gráficos y los eventos de interfase de usuario producidos localmente.

- El problema de **comenzar la ejecución de aplicaciones existentes en nodos remotos** requiere básicamente de un mecanismo de 'browsing' de aplicaciones de manera de saber en que nodos existen aplicaciones y cuales son estas aplicaciones, más otro que permita iniciar la ejecución de estas. Las posibles soluciones surgen de la flexibilidad del mecanismo de browsing, que van desde no disponer de un mecanismo por lo que el usuario debe saber el nodo y la aplicación a ejecutar, hasta un mecanismo que presente los nodos y las aplicaciones existentes en cada nodo y así poder elegir el nodo/aplicación.

[1] Los PDUs en terminología OSI, son las unidades de datos que intercambian dos entidades del mismo nivel N; un PDU está formado por los datos de la entidad de nivel N+1 más información de control de la entidad N. PDU es una denominación genérica que aplica a entidades de cualquier nivel. Los PDUs intercambiados por dos entidades a nivel de transporte se denominan TPDUs (Transport Protocol Data Units).

1. 1. 3. Aproximaciones consideradas

A continuación se presentan las distintas aproximaciones consideradas durante el análisis del problema.

1. 1. 3. 1. WtW (Windows To Windows)

La primer idea de diseño frente a los problemas antes enunciados fue la de trasladar la interfase de Windows de un nodo sobre el UIMS de Windows en otro nodo.

De esta forma podrían ser controladas remotamente aplicaciones Windows desde nodos sobre los que se ejecuta Windows, logrando un resultado muy interesante si se diseña el protocolo de transporte de manera de funcionar sobre enlaces de ancho de banda reducido (por ejemplo lineas telefónicas), ya que se podría tener la interfase y el control de una aplicación Windows que se está ejecutando en un nodo remoto al cual no se tiene otra forma de acceso.

Esta solución en principio parece ser sencilla, pero requiere fundamentalmente de la resolución del problema de la separación del GUI de Windows para llevarlo a un esquema de red, lo cual no es obvio conociendo la intensa interacción existente en Windows entre las aplicaciones y el UIMS. En este esquema además es necesario definir el protocolo de aplicación propietario y la implementación de un server de gráficos que interprete los requerimientos gráficos provenientes desde el nodo remoto e invoque las funciones del API de Windows del nodo local. Es importante aclarar que en esta aproximación el 'mapping' se reduce considerablemente debido a que el UIMS destino es el mismo al origen, y consiste basicamente en trasladar directamente los PDUs del protocolo, orientado a requerimientos de Windows, a funciones del API en el nodo local.

1. 1. 3. 2. WExX (Solución basada en el X Window System)

El X Window System es un standard de la industria que nació en el MIT (Massachusets Institute of Technology) como parte del Proyecto Athena[2] y que cuenta con más de 10 años de investigación y desarrollo. Actualmente es una organización independiente del MIT formada por decenas de empresas, el X Consortium, la que continúa con la evolución del X Window System.

El X Window System es una **sistema de ventanas basado en un modelo Cliente/Servidor**. Un cliente X (**X client**) hace requerimientos gráficos a un servidor X (**X server**). El servidor maneja los dispositivos de entrada y salida (display, mouse, teclado, etc.) y envía los eventos de la interfase de usuario a los clientes. Los clientes y los servidores pueden estar en diferentes computadoras o nodos. Los requerimientos y eventos definen los PDUs que forman parte del **protocolo X**.

La idea sería utilizar el protocolo X para dar solución a la necesidad de un protocolo de comunicación que transporte los requerimientos gráficos, resultados de los requerimientos y eventos de interfase de usuario entre los nodos. Esto implica que ya no podrán ser transportados los requerimientos gráficos nativos de Windows sobre ese protocolo, ya que el protocolo X define los propios requerimientos gráficos. Por lo tanto habrá que resolver el 'mapping' de requerimientos gráficos de Windows hacia los definidos en el protocolo X del lado en el que se ejecuta la aplicación, para luego transportarlos sobre el protocolo X.

En el nodo local debería existir una implementación de un X server el cual resolvería los requerimientos realizados desde el nodo remoto, en el que se está ejecutando el programa Windows. O sea que el X server sería el UIMS necesario en el nodo local. Lo importante de este 'aproach' es

[2] Del Proyecto Athena se originaron múltiples desarrollos de la Industria de la Computación como ser el mecanismo de autenticación distribuída Kerberos.

que existen implementaciones de X servers para casi todas las plataformas comerciales existentes (UNIX, OpenVMS, OS/2, Next, Macintosh, DOS, Windows, NT, y otros).

1. 1. 3. 3. WSI (Window System Independence)

Una idea que siempre se tuvo en mente fue la independencia del UIMS sobre el cual se traslade la interfase de Windows.

A continuación se esquematiza una solución que logra este objetivo. Es importante aclarar que sólo se presentarán las ideas sin avanzar en su implementación.

Existe un 'trade-off" entre la flexibilidad y performance que se lograría con esta solución y con una solución no independiente del sistema de ventanas.

La arquitectura de esta solución estaría formada por tres componentes básicos:

- **WAP (Windows Application Protocol):** protocolo de aplicación propietario mediante el cual se transportan los requerimientos gráficos. Los PDUs de este protocolo incluye requerimientos Windows, resultados y errores de los requerimientos, mensajes Windows, recursos Windows, objetos gráficos, etc.

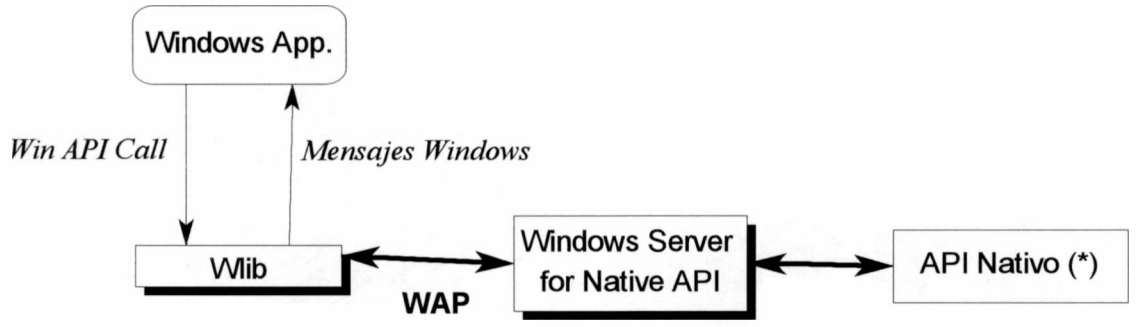
- **Wlib:** Módulo que intercepta los llamados a funciones del API de Windows, los mapea en requerimientos del protocolo WAP, y pasa los resultados de estas funciones y los mensajes Windows a las aplicaciones.

- **Windows Server:** Este módulo es el que recibe los requerimientos Windows y los mapea al API de Interfase de Usuario de la plataforma desde la cual se quiera controlar aplicaciones Windows. Conceptualmente es un **gateway de requerimientos gráficos**.

El Windows Server consistirá de un módulo que recibe los requerimientos sobre WAP y un módulo por cada UIMS al que mapee requerimientos, logrando así un gateway multi-UIMS que dá la verdadera independencia del UIMS destino.

Una característica también importante de este modelo es que el Windows Server puede ser implementado tanto en la plataforma origen (Windows) como en la plataforma destino (el entorno donde se ejecuta el sistema de ventanas destino), de manera de minimizar el tráfico de requerimientos gráficos a traves de la red, utilizando el protocolo de transporte de requerimientos gráficos mas adecuado.

Arquitectura de la solución independiente del sistema de ventanas (WSI)



(*) API Nativo: Win16, X Window System, OS/2 PM, etc.

Es fácil ver que con esta arquitectura podrían ser implementados cualquiera de los dos 'aproachs' anteriores.

1. 1. 4. Aproximación adoptada.

Se adoptó avanzar en la solucion llamada WExX ("Windows Extension X-aware") principalmente por los motivos detallados a continuación. De todas formas existe una gran relación entre las tres aproximaciones enunciadas, por lo que se estarán dando soluciones relacionadas con el objetivo inicial mas que con una solución en particular, pero basados en el X Window System de manera de poder obtener también una implementación funcional.

- **El X Window System es un standard de la industria**.

- **Las aplicaciones Windows se convertirán en verdaderos clientes X**, es decir que desde el momento que una aplicación 'hable' el protocolo X, podrá comunicarse con cualquier X server y este resolverá sus requerimientos gráficos. De esta forma se podrá tener la interfase de aplicaciones Windows e interactuar con ella desde cualquier plataforma para la que exista una implementación de un X server, que como antes se dijo es muy variada.

- **El X Window System fue específicamente diseñado para funcionar en red y con una estructura cliente/servidor**.

- **El protocolo X es un protocolo independiente del protocolo de transporte**. Lo que se le pide al transporte, es que provea un servicio de stream de bytes bidireccional y confiable. Es importante resaltar que requiriendo sólamente un transporte basado en un stream de bytes, bidireccional y confiable, hace que X sea usado en muchos entornos. Por ejemplo, el protocolo X puede ser usado arriba de TCP/IP, DECnet y IPX/SPX.

- **La filosofía del X Window System es ser 'policy-free'**, es decir, que el X Window System no impone políticas sino mecanismos para implementar cualquier GUI. Por lo tanto se puede hacer el 'mapping' de toda la funcionalidad provista por el UIMS de Windows.

A una aplicación que se desee controlar remotamente con este mecanismo la llamaremos indistintamente **'aplicación sobre WExX'**, **'aplicación WExX'** o **'aplicación Windows cliente X'.**

1. 1. 5. Solución al problema de comenzar la ejecución de aplicaciones Windows remotas.

Para solucionar el problema de iniciar la ejecución de aplicaciones Windows existentes en el nodo remoto se implementará, por un lado, un mecanismo de 'browsing' de aplicaciones el cual tiene como objetivo presentar las aplicaciones Windows existentes en la red, y por el otro lado, un mecanismo que permita iniciar la ejecución de la/las aplicaciones presentadas con el mecanismo anterior.

Para el mecanismo de 'browsing' se utilizará el protocolo **XDMCP (X Display Manager Control Protocol)** que es un protocolo entre un 'display manager' (el cual es un proceso que corre en los nodos donde corren los clientes X) y un X server que corre en la workstation en donde se encuentra el usuario. Este protocolo permite dinámicamente desde la workstation elegir un nodo o host con el cual establecer una sesión de login gráfica.

Para el mecanismo de arranque de aplicaciones se utilizará una solución propietaria.

1. 2. Temas a Estudiar

Microsoft Windows 3.1

- Estructura y funcionamiento interno de los módulos componentes de Windows:
  - Kernel (Servicios del sistema: administración de memoria, módulos y tareas).
  - User (Administración de ventanas, subsistema de mensajes).
  - GDI (Graphics Device Interface).
- Funciones del API de los tres módulos anteriores.
- Soluciones del mercado para ejecutar aplicaciones Windows en otras plataformas.

X Window System

- Protocolo X.
- Xlib. (Interfase "C" al protocolo X)
- Xt Intrinsics y Widget Sets.
- Window Managers (se estudiará el diseño del Window Manager twm - incluído en la distribución del código fuente del X Window System).
- Protocolo de comunicación entre clientes X. (ICCCM)

Interfases gráficas basadas en el X Window System

- Administración de interfases.
- Protocolo XDMCP.

Interfases de programación a los protocolos TCP/IP BSD Sockets y Windows Sockets.

1. 3. Resultados Finales Esperados

- **Estudio de Microsoft Windows 3.1 y del X Window System.**
- **Diseño de la extensión basada en red del UIMS de Windows basada en el X Window System.**
- **Implementación de un prototipo que pemita validar la idea del punto anterior.**

Parte 2 - Windows

2. 1. Introducción

Windows provee desde el punto de vista del usuario un **GUI** ('Graphical User Interface') con facilidades de multitarea. Desde el punto de vista de los desarrolladores provee **controles** como menúes, 'scroll bars', 'dialog boxes', para generar interfaces de usuario "amigables"; y de esta forma además mantener un GUI consistente.

En esta sección se explicarán brevemente las características principales de Windows.

2. 1. 1. Interface gráfica independiente de los dispositivos

Windows es una interface gráfica, y los programas Windows pueden hacer uso de gráficos y texto con formato tanto en el 'display' como en la impresora. Esta facilidad es conocida como WYSIWYG (What You See Is What You Get.).

Los programas escritos para Windows no acceden directamente al hardware de dispositivos de display, tales como el 'screen' y la impresora, sino que Windows incluye una API gráfica en el módulo GDI (Graphic Device Interface), el cual se comunica con drivers específicos de los distintos dispositivos. De esta forma Windows 'virtualiza' el hardware de display, es decir, que todo programa Windows podrá ejecutarse sobre cualquier placa de vídeo o impresora para la que exista un 'driver' para Windows.

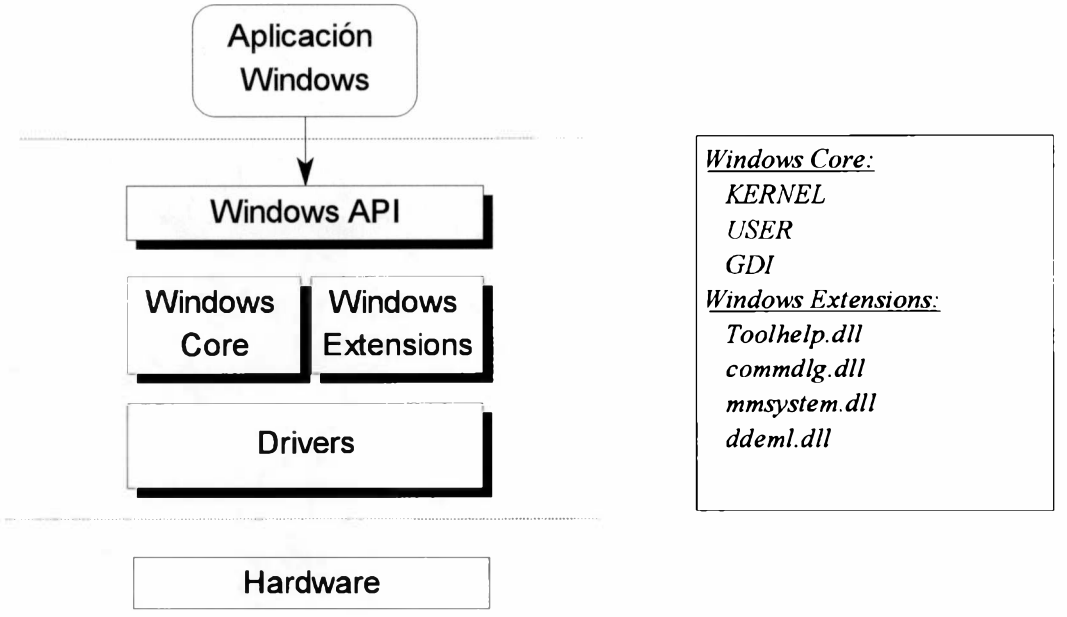
2. 1. 2. Arquitectura de Windows

Windows esta basado en tres módulos principales, que a pesar de sus nombres son librerías de enlace dinámico (DLLs) standards :

- **KERNEL**[1]: se encarga de la Administración de Módulos y Tareas, así como de la Administración de la Memoria del sistema.
- **USER.EXE**: implementa el Sistema de Ventanas y el Sistema de Mensajes.
- **GDI.EXE** (Graphic Device Interface): implementa toda la porción gráfica de Windows, proveyendo una interface genérica a todos los dispositivos gráficos.

[1] KERNEL es un nombre generico, ya que la DLL utilizada depende del hardware disponible : KRNL286.EXE en el caso de tener un procesador 80286 y KRNL386.EXE en el caso de que el procesador sea 80386/486.

Arquitectura general de Windows



2. 1. 3. Multitasking cooperativo

Una *Tarea* es la unidad de ejecución de Windows, es el equivalente a un proceso en otro sistema operativo.

Una vez que el scheduler de Windows selecciona una tarea para ser ejecutada, esta mantiene el control hasta que lo libera explícitamente mediante la invocación de funciones del KERNEL de Windows (Ej.: *GetMessage, PeekMessage*). Esto es conocido como *multitasking cooperativo.* Cuando el scheduler toma el control nuevamente debe seleccionar la próxima tarea a ejecutar. Para esto el scheduler mantiene las tareas en una lista encadenada de la cual selecciona la próxima tarea que tiene un *mensaje* sin procesar.

En caso de que una tarea no libere el control ninguna otra tarea podrá seguir ejecutándose.

2. 1. 4. Estructura de un programa Windows

La programación en un sistema de ventanas es inherentemente tediosa y Windows no es la excepción. Sumado a esto, está el hecho que la programación en este entorno está fuertemente ligada a su **compleja** estructura interna. Aunque aún no hemos detallado los componentes internos de Windows, haremos un breve comentario acerca de los pasos necesarios en Windows para poder dibujar una ventana e interactuar con el usuario frente a los eventos que este genera.

Si uno desea dibujar algún gráfico sobre la pantalla es necesario tener los que llama un **HDC** (handle to a device context). Para obtener un hdc se necesita un **HWND** (handle to a window) que es obtenido al **crear un ventana.** Luego, es necesario estar preparado para recibir **mensajes** que le son enviados a la ventana, y para recibir mensajes hay que tener un **procedimiento de ventana** (conocido como WndProc) asociado a la ventana. Este procedimiento de ventana es una función a la que Windows llama cuando necesita notificarle de algún evento en el sistema, y es la forma en la que Windows envía mensajes a las ventanas. En ese momento el procedimiento de ventana toma el control, realiza el procesamiento necesario según el tipo de mensaje recibido, y retorna el control a

Windows. Una ventana puede recibir mensajes que la informen sobre acciones directas del usuario, como el movimiento del mouse o el pulsado de una tecla del teclado; y de mensajes generados por el sistema, como el aviso de que una ventana cambió de tamaño o que necesita ser redibujada.

Toda ventana es creada en base a una **clase de ventana** (Window Class), la cual especifica, entre otros, el procedimiento de ventana que procesa los mensajes de la ventana, por lo que de esta forma las ventanas de una misma clase comparten el procedimiento de ventana.

Cuando un programa Windows comienza su ejecución, Windows crea una **cola de mensajes** para el programa, donde se almacenan los mensajes para todas las ventanas que la aplicación cree.

Finalmente, para el procesamiento de los mensajes, los programas Windows incluyen un **'loop' de mensajes** para obtener mensajes desde la cola de mensajes y despacharlos a la ventana correspondiente. Esta arquitectura de los programas es conocida como arquitectura **message-driven,** donde el flujo del programa es controlado por el usuario a través de la generación de eventos a los que el programa debe responder. Algunos **mensajes** son **enviados** por Windows a los procedimientos de ventanas sin pasar por la cola de mensajes.

2. 2. Administración de Memoria

La función fundamental de un sistema operativo es la administración de memoria, la cual es la base para todos los otros servicios del sistema operativo. Un sistema operativo no puede implementar 'multitasking' sin administrar eficientemente la memoria. A medida que los programas se ejecutan y terminan la memoria se fragmenta, y por lo tanto el S.O. debe implementar mecanismos para compactar esos espacios libres. En Windows, además, los programas ejecutándose pueden contener mas código que el que puede almacenarse en la memoria, por lo tanto se debe descartar código desde memoria y luego recargarlo cuando es necesario desde el archivo ejecutable, o sea que debe proveer mecanismos de memoria virtual.

La gran cantidad de código en el modulo KERNEL demuestra la complejidad del manejo de memoria segmentada en una arquitectura de CPU Intel 80x86. En este capítulo explicaremos muy brevemente la administración de la memoria que realiza Windows y otros módulos que conviven con Windows, ya que extenderse en la administración de memoria nos privaría de poder profundizar en temas que se ajustan mas directamente a nuestro objetivo.

2. 2. 1. Windows : mucho mas que un 'DOS extender'

Una cosa importante para notar de la administración de memoria de Windows es que este se comporta como un **'DOS extender'** con una interface de usuario gráfica. Durante el diseño de Windows 3.0, Microsoft y otras varias empresas, se propusieron elaborar una interface que permitiera que el 'DOS extender' de Windows, el KERNEL, coexistiera con otros 'DOS extenders'. Este acuerdo resultó en la especificación del **DPMI (DOS Protected Mode Interface)**. Un 'device driver' o programa que provee servicios de DPMI se lo conoce como un 'DPMI Server'. A los programas que utilizan los servicios de DPMI (como Windows) se los conoce como 'DPMI Clients'.

En Windows hay dos módulos que implementan los servicios de DPMI. Dependiendo el modo de ejecución de Windows se carga uno u otro. En modo standard el DPMI server' es el modulo DOSX, y en modo 'Enhaced' es el VMM ('Virtual Machine Manager') que está dentro del módulo WIN386. Luego de cargado el 'DPMI server' se carga el KERNEL (desde el archivo KRNL286.EXE o KRNL386.EXE según se ejecute Windows en modo 'Standard' o 'Enhanced') sobre él.

Es muy importante aclarar que la administración de memoria que hacen DOSX y WIN386 es completamente diferente a la administración de memoria del KERNEL de Windows. El KERNEL opera con memoria en términos de segmentos y bloques de memoria alocados usando los servicios de DPMI. WIN386 y DOSX proveen los servicios DPMI y trabajan con la memoria en términos de páginas. Usando el mecanismo de paginado que ofrecen las arquitecturas 386 o superiores, el VMM crea espacios de direcciones de memoria lineal que pueden o no estar mapeados en memoria física. Por lo tanto una dirección lineal con la que trabaja el KERNEL no necesariamente se corresponde con una dirección física. El VMM y la unidad de paginado de la CPU manejan la traslación de direcciones para proveer un bloque continuo de memoria lineal, no necesariamente física. El KERNEL aloca bloques de memoria lineal vía los servicios de DPMI que ofrece el VMM.

2. 2. 2. Dos clases de 'heaps'

El KERNEL utiliza los servicios de DPMI para alocar grandes regiones de memoria para utilizar como **'global heap'** accesibles por lo programas mediante selectores. Para cada 'global heap' que es obtenida desde el bloque inicial obtenido desde el 'DMPI server' le es asociado un selector, lo que genera un gran problema, sabiendo que en la arquitectura 80x86 hay 2^13 (8192) selectores

disponibles. Estos segmentos son obtenidos por los programas mediante las funciones **GlobalAlloc** y **GlobalFree** de la API del KERNEL, y un programa que hiciera muchos llamados agotaría los selectores.

Para remediar este problema en Windows un programa puede alocar pequeñas porciones de una 'global heap' sin alocar un selector cada vez. Esto es llamado **'local heap'**. El KERNEL provee un conjunto completo de funciones similares a los de manejo de la 'global heap' para manejar la local heap, entre las que se encuentran **LocalAlloc** y **LocalFree** para obtener y liberar bloques de memoria.

2. 2. 3. Atributos de Memoria

La arquitectura 80x86 permite especificar en los segmentos si contienen datos o código. De todas formas el KERNEL necesita mas información para manejar segmentos y bloques efectivamente. Así es que a los bloques que son alocados desde la 'global heap' o desde la 'local heap' les son asociados atributos adicionales.

Uno de los principales problemas en el esquema de heap es la fragmentación. Para reducir este efecto el KERNEL usa bloques **MOVEABLE**s. Cuando Windows era ejecutado en modo real, para acceder a bloques marcados como MOVEABLEs era necesario notificar a Windows, mediante la función GlobalLock o LocalLock, de manera que éste retornara un puntero a la dirección de memoria física donde el bloque se encontraba. A partir de los modos 'Standard' y 'Enhanced' esto ya no es necesario, y los programas pueden mantener los bloques 'locked' durante toda la ejecución.

El opuesto de un bloque MOVEABLE son los bloques **FIXED**. Los bloques marcados como FIXED no son movidos en la memoria lineal. El problema de la utilización de los bloques FIXED es que estos previenen la compactación de la heap. En general no hay razones para usar este tipo de bloques.

Otro atributo disponible para bloques de la heap es el **DISCARDABLE**. El uso mas común de estos bloques es en los segmentos de código y de recursos. Ya que típicamente estos bloques no son escritos, podrían ser recargados desde el EXE o DLL cuando sea necesario (por ejemplo los segmentos de código). El KERNEL utiliza la notificación de *segmento no presente* para detectar cuando es necesario recargar un segmento de código o recurso desde el archivo.

2. 3. Administración de Ventanas

Uno de los objetivos de Windows es abstraer a los programadores de los problemas de tratar con hardware específico para la interacción con el usuario (teclado, mouse, display, etc.), pero indudablementee el componente mas visible de Windows es el sistema de ventanas. La implementación del sistema de ventanas se encuentra en el módulo USER.EXE. A continuación comentaremos brevemente la información que éste módulo debe mantener para llevar a cabo la administración de la ventanas.

2. 3. 1. Clases de Ventanas

En vez de especificar todos los atributos de una ventana cada vez que se desea crear una, Windows provee de la capacidad de registrar clases de ventanas. Luego, cuando se desea crear una instancia de una clase de ventana, el USER obtiene muchos de los datos necesarios para llevar a cabo la creación de la nueva ventana desde la estructura de clase que debe haber sido previamente registrada.

Las clases de ventanas pueden ser locales a la aplicación o globales. Una aplicación generalmente registra **clases locales a la aplicación**, lo que significa que solo instancias de esa aplicación pueden utilizarlas. Si otro programa registra una clase utilizando el mismo nombre de clase, Windows crea una clase diferente, y son distinguidas por el handle del modulo que la creo. Las clases globales son visibles a todas las aplicaciones, y las clases standards de Windows son el mejor ejemplo de éstas. Cuando una aplicación especifica el nombre de una clase al crear una ventana, el USER primero busca por una clase local a la aplicación, y si no la encuentra busca dentro de las clases globales.

2. 3. 1. 1. Elementos de una clase de ventana

Los elementos de una clase de ventana definen el comportamiento 'default' de una ventana que es creada a partir de ésta. La función utilizada para la registración de las clases de ventanas es *RegisterClass* y los elementos mas importantes que una aplicación debe especificar al llamar a esta función son :

Nombre de la clase : es el nombre por el cual se distingue la clase de otras clases registradas.

Estilo de clase : define el comportamiento de las ventanas creadas a partir de la clase en el aspecto de como actualizar una ventana luego de un 'resizing' o movimiento de la ventana, como procesar los eventos de 'double clicks', y otros. Este elemento especifica el estilo de clase y no debe ser confundido con el estilo de ventana que se especifica en tiempo de creación de una ventana.

Dirección del procedimiento de ventana : es un puntero a la función que procesará todos los mensajes que le son enviados a las ventanas creadas a partir de la clase. Toda clase debe especificar un procedimiento de ventana, que es la función que Windows invoca cuando precisa que las ventanas realicen algunas tareas tales como repintar el área cliente o responder a eventos de entrada del usuario.

Cursor : especifica el cursor 'default' a ser mostrado cuando el mouse se encuentra sobre una ventana perteneciente a la clase.

Menu : define el menú a ser utilizado por las ventanas de la clase si no es especificado otro al crear la ventana. Un menú es una herramienta en una aplicación Windows que ofrece uno o mas ítems los cuales pueden ser seleccionados con el mouse o el teclado. Un ítem en una barra de menú puede desplegar un pop-up menú y cualquier ítem en un pop-up menu puede desplegar otro pop-up menu. Los menúes son generalmente referidos por el nombre y se correponden con un template en los

recursos de la aplicación. Las ventanas no requieren un menú, si una aplicación no define este campo, Windows asumirá que las ventanas en la clase no tienen barra de menú.

2. 3. 1. 2. Clases de ventanas predefinidas

A modo de ofrecer una funcionalidad básica, el módulo USER en tiempo de inicialización, registra clases de ventanas globales, también conocidas como clases de **controles**, las cuales cualquier aplicación puede utilizar para crear instancias. Estas clases son :

```
ComboBox
ScrollBar
ListBox
Edit
Static
Button
```

Estas clases definen controles especiales que llevan con sigo la tarea de permitir al usuario ingresar un texto (Edit), realizar una selección desde una lista (ListBox), o sencillamente la selección de una opción (Button) entre varias.

Windows también registra otras clases que no son utilizadas directamente como por ejemplo una clase para cuando se crea un dialog box o cuando se crea una ventana con el estilo MDI (ver mas adelante las características de dialog boxes y ventanas MDI).

2. 3. 2. Creación y manipulación de ventanas

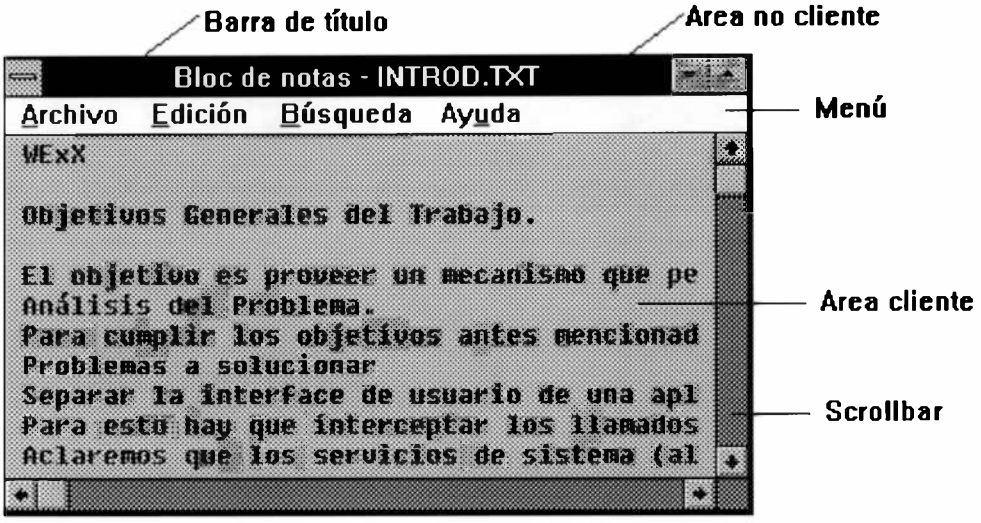
Las clases de ventanas definen características generales de una ventana, permitiendo que la misma clase de ventana sea usada para crear muchas ventanas diferentes. Cuando se crea una ventana, usando la función del API de Windows *CreateWindow*, se especifica, además de la clase de ventana a la que la nueva ventana va a pertenecer, información mas detallada acerca de una ventana particular como por ejemplo el estilo y la posición.

La función CreateWindow retorna un **HWND** ('handle to a window') que es el identificador que una aplicación utiliza para referirse a una ventana. Todas las funciones de manipulación de ventanas requieren como parámetro un HWND. Un handle es una referencia a una estructura WND en la heap local del USER (la estructura WND no esta documentada por Microsoft, aunque hay varios autores que la han descifrado y documentado casi completamente). Esta estructura es accedida por medio de las funciones *GetWindowWord*, *SetWindowWord*, *GetWindowLong* y *SetWindowLong*, que reciben como parámetro un HWND y un desplazamiento (aunque este no es el verdadero desplazamiento dentro de la estructura). Los datos almacenados en esta estructura son datos específicos de una ventana como por ejemplo:

- Rectángulo describiendo el área de la ventana.
- Cola de mensajes de la aplicación.
- Dirección del procedimiento de ventana.
- 'Flags' de estilo de la ventana.
- Handle de la ventana padre (parent window)
- Handle de la primera hija (child window).
- Handle a la próxima ventana (sibling window).

Es importante aclarar que el área de una ventana está compuesta por el **área no cliente,** formada por la barra de título, el menú y los scrollbars, y el **área cliente** que es la parte interior de la ventana, y que es la que las aplicaciones normalmente utilizan.

Componentes de una ventana



2. 3. 3. Estilos de Ventanas

Windows provee diferentes estilos de ventanas que pueden ser combinados para formar diferentes clases de ventanas. Estos estilos son utilizados en la función CreateWindow cuando la ventana es creada.

Overlapped Windows

Una ventana overlapped es siempre una ventana top-level, es decir que no tiene nunca una ventana padre. Estas tienen un area cliente, un borde, una barra de título y pueden tener un ícono asociado que se desplegará cuando la ventana es minimizada.

Una aplicación crea una overlapped window utilizando el estilo WS_OVERLAPPED o WS_OVERLAPPEDWINDOW en la función CreateWindow.

Pop-up Windows

Pop-up windows son otro tipo de ventana especial de overlapped window. La principal diferencia entre una pop-up window y una overlapped window es que una overlapped window siempre tiene una barra de título mientras que en las pop-up windows es opcional.

Desde el punto de vista de programación una pop-up window es creada utilizando el estilo WS_POPUP en la función CreateWindow.

Child Windows

Una child window es una ventana que esta confinada al área cliente de su ventana padre. Las child windows son utilizadas para dividir el área cliente de una ventana padre en diferentes áreas funcionales. *Toda child window debe tener una ventana padre,* la cual puede ser una overlapped window, una pop-up window u otra child window. Una child window tiene un área cliente, pero no tiene otras características a menos que éstas sean requeridas explicitamente, aunque no es permitido que tengan asociado un menú.

Una aplicación utiliza el estilo WS_CHILD en la función CreateWindow para crear una child window.

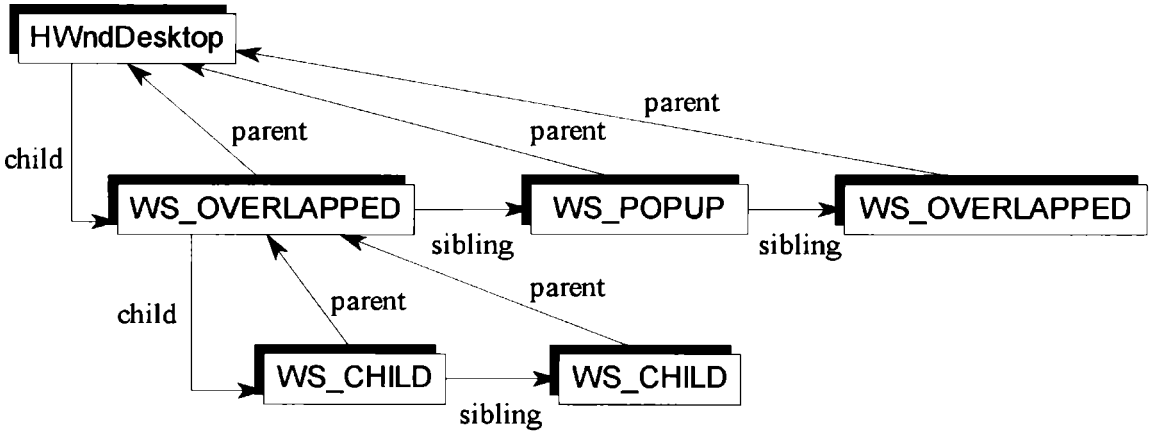
2. 3. 3. 2. Jerarquía de Ventanas

El sistema de ventanas de Windows puede ser visto como un árbol. Todas las ventanas caen en algún lugar de la jerarquía, con la ventana 'desktop' al tope del árbol. Cada ventana además tiene su área de datos para mantener el estado de la ventana.

Como ya vimos, cada ventana mantiene un handle a su padre, hija y a la siguiente ventana en el mismo nivel de la jerarquía, lo que permite recorrer la jerarquía en distintos sentidos.

La jerarquía que el USER crea es como la de la figura siguiente:

Jerarquías de ventanas creadas por el USER



Existe una ventana particular llamada 'desktop', que es la primera ventana creada por el USER. Es la raíz de la jerarquía, ocupa todo el 'screen', y se mantiene al final en el orden z, lo que significa que esta detrás de todas las demás ventanas.

2. 3. 4. Multiple Document Interface (MDI)

MDI es una especificación para aplicaciones que manejan múltiples documentos. La especificación describe una estructura de ventana y una interface que permite al usuario trabajar con múltiples documentos dentro de una aplicación.

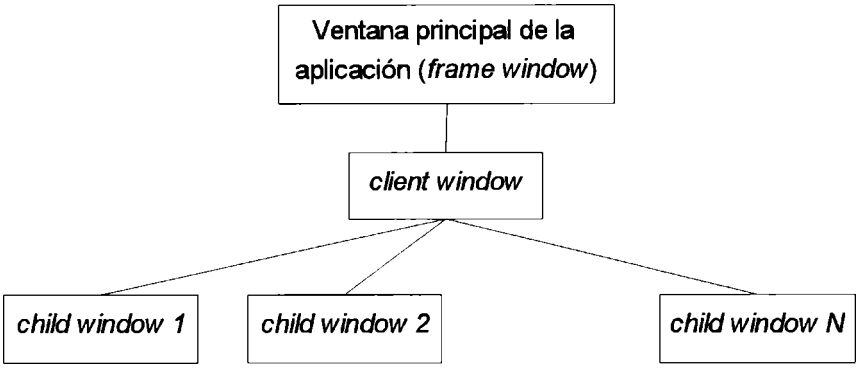
A partir de Windows 3.0 MDI es una clase de ventana global registrada por Windows. Además el programador dispone de cuatro funciones especificas para trabajar con MDIs, dos estructuras de datos, y once mensajes con el propósito específico de facilitar el uso de las MDI.

Componentes de una ventana MDI

La ventana principal, llamada 'frame window', es una ventana convencional, es decir con un título, un menú, botones para maximizar y minimizar la ventana, etc. Pero a diferencia de la ventanas convencionales, el área cliente, también llamada 'workspace' no es utilizada como tal, sino que exixte otra ventana hija de la 'frame window' y llamada 'client window', que ocupa todo el espacio del área cliente de la 'frame window', y es esta la ventana padre de todas las 'child windows'. Una

característica muy importante en la especificación de MDI es que todas las 'child windows' se mantienen dentro de su ventana padre ('client window') la cual a la vez esta restringida a la ventana principal ('frame window').

Jerarquía de ventanas MDI



2. 3. 5. Dialog Boxes

Un dialog box es una ventana temporaria que se crea en Windows para ingreso de datos de propósito especial y se destruye después de usarla.

La principal diferencia entre una ventana tradicional y un dialog box es que en el caso de una ventana tradicional la aplicación provee un procedimiento de ventana para determinar el comportamiento de la misma, y en el caso de un dialog box Windows crea una ventana de la clase DIALOG, la cual tiene asociado un procedimiento de ventana tambien definido en Windows que determina su comportamiento, y la aplicación solo provee un procedimiento que es invocado por el procedimiento de ventana del dialog box en determinadas oportunidades. Otra particularidad de los dialogs boxes es que son creados a partir de un 'dialog box template', el cual describe el estilo y contenido del dialog box.

El uso de los dialog boxes permite cierta independencia del dispositivo ya que se crean usando coordenadas lógicas. Un dialog es conveniente de usar debido a que el comportamiento de estos está predefinido, proveyendo una clase de ventana y un procedimiento de ventana.

Hay tres tipos de dialogs boxes, **modeless dialog box, modal dialog box** y **system-modal dialog box**.

2. 3. 5. 1. Tipos de Dialog Boxes

Modeless Dialog Box

Un modeless dialog box permite al usuario ingresar información en el dialog box y retornar a la tarea previa sin la necesidad de remover el dialog box. Por ejemplo un modeless dialog box es frecuentemente usado para una busqueda de texto en las aplicaciones de procesamiento de texto.

El procedimiento de un modeless dialog box (equivalente al procedimiento de ventana en una ventana normal) debe enviar una mensaje a la ventana padre cuando este tiene datos para ella.

Modal Dialog Box

Un modal dialog box requiere que el usuario responda a un requerimiento antes de que la aplicación continúe con el flujo de ejecucion. Estos son usados tipicamente cuando un comando elegido necesita información adicional antes de poder continuar. *Un modal dialog box deshabilita su ventana padre y crea su propio loop de mensajes, tomando temporariamente el control de la cola de mensajes de la aplicación.*

System-Modal Dialog Box

Un system-modal dialog box es identico a un modal dialog box excepto que todas las ventanas, no solo la ventana padre son deshabilitadas. Este estilo de dialog box debe ser utilizado con cuidado debido que detienen el flujo de ejecución de todo Windows. Efectivamente ninguna aplicación se continuará ejecutando mientras el system-modal dialog box no es cerrado.

2. 3. 5. 2. Creando un Dialog Box

Un dialog box es creado utilizando las funciones de la API del USER **CreateDialog** o **DialogBox**. La función DialogBox crea un modal dialog box, y la función CreateDialog un modeless dialog box. Ambas funciones cargan un 'dialog box template' desde el archivo ejecutable de la aplicación y luego crea una ventana pop-up que cumple con la especificación del 'template'.

Dialog Box Template

El dialog box template es una descripción del dialog box : su tamaño (alto y ancho), los controles que este contiene, su estilo, el tipo del borde, etc. Un template es parte de los recursos de una aplicación. Las dimensiones y coordenadas del dialog box y de los controles que pertenecen a él son 'device-independent'. Debido a que un dialog box sera desplegado en 'displays' que tienen una amplia variedad de resoluciones, sus dimensiones son especificadas en función del alto y ancho del caracter 'default' del sistema en vez de hacerlo en términos de pixels.

2. 4. Sistema de Mensajes

En Windows los mensajes son utilizados para muchos propósitos. El uso mas general es el de indicar cuando un mouse se mueve, o cuando un ítem de un menú es seleccionado; pero por otro lado dialog boxes, menúes, y otros controles hacen un uso intensivo de los mensajes para comunicarse entre si. Los mensajes son utilizados también como una forma de comunicación interprocesos.

El código para el Sistema de Mensajes esta en su mayoría implementado en el USER.EXE, pero también juega un rol vital en permitir el flujo de mensajes el 'scheduler' en el modulo KERNEL.

Windows mantiene una cola de mensajes llamada **System Message Queue** y además toda aplicación en Windows tiene asociada una única cola de mensajes llamada **Application Message Queue.**

Hay dos funciones con las que las aplicaciones envían mensajes a ventanas :

SendMessage : el mensaje es enviado a la ventana en forma sincrónica, es decir que la ventana (mediante su procedimiento de ventana) recibe y procesa el mensaje antes de que la función retorne. Nos referiremos a esta forma de envío de mensajes como **enviar** (Send) mensajes.

PostMessage : el mensaje es colocado en la cola de mensajes asociada a la ventana y la función retorna. El mensaje es procesado mas tarde cuando la aplicación toma mensajes de su cola de mensajes. Esta forma de envío de mensajes asincrónicos la referiremos como **poner** (Post) mensajes.

Durante la inicialización de un programa, el USER crea la cola de mensajes de la aplicación. Luego toda ventana creada por la aplicación tendrá una referencia a esta cola, y cuando se 'ponga' un mensaje a la ventana, el mensaje será agregado a la cola de mensajes a la que la ventana hace referencia.

2. 4. 1. Flujo de Mensajes

Los mensajes son la entrada a una aplicación. Ellos representan eventos, producidos generalmente por el usuario, a los que la aplicación necesita responder. Desde el punto de vista del programador, un mensaje es una estructura que contiene un identificador del mensaje y otros campos asociados al mensaje.

Windows genera un mensaje por cada evento que ocurre, como por ejemplo cuando el usuario mueve el mouse o cuando presiona una tecla, y los guarda en la cola del sistema. También Windows pone mensajes que pertenecen a una aplicación especifica en la cola de mensajes de aplicación. La aplicación luego lee los mensajes usando la función del API 'GetMessage' (que inspecciona la cola de la aplicación y del sistema) y los despacha a la ventana apropiada usando la función 'DispatchMessage'. También Windows envía algunos mensajes directamente a las ventanas en vez de ponerlos en la cola de la aplicación. Esto lo hace con mensajes de los que Windows necesita una respuesta inmediata, como por ejemplo el mensaje WM_CREATE indicándole al procedimiento de ventana que realice los pasos necesarios para la creación de una nueva ventana.

Aunque Windows genera la mayoría de los mensajes, una aplicación puede generar sus propios mensajes y ponerlos en su propia cola de mensajes o en el de alguna otra aplicación.

Una aplicación generalmente usa la función 'GetMessage' en un 'loop' dentro de su procedimiento principal (WinMain) para obtener mensajes desde la cola de mensajes de la aplicación. La función 'GetMessage' retorna el primer mensaje de la cola de mensajes de la aplicación, y si la cola esta vacía, espera hasta que sea puesto un mensaje en ella, dejando el control a Windows, y permitiendo que otra aplicación tome el control y procese sus propios mensajes.

La forma general del loop de mensajes de una aplicación es :

```
int PASCAL WinMain(HINSTANCE hinstCurrent, HINSTANCE hinstPrev,
      LPSTR lpszCmdLine, int nCmdShow)
{
      // Creacion de la ventana principal (tambien conocida como
      // TopLevel).
         .
         .
         .
      while (GetMessage(&msg, NULL, 0, 0))  /* Obtiene un mensaje de
                                               la cola de mensajes */
            DispatchMessage(&msg);           /* Despacha el mensaje a
                                            la ventana correspondiente */
    return((int)msg.wParam);
}
```

2. 4. 2. Dynamic Data Exchange

DDE es una de las tres formas de comunicación entre procesos soportadas por Windows. Las otras dos formas son el Clipboard[2] y la memoria compartida en las DLLs.

Dos programas participan en una conversación por DDE enviándose mensajes asincrónicos (PostMessage) y sincrónicos (SendMessage) y son conocidos como Server y Cliente. El Cliente envía requerimientos al Server y este responde a los pedidos del Cliente. La conversación es iniciada por el Cliente haciendo un Broadcast del mensaje WM_DDE_INITIATE especificando en los parámetros del mensaje el Server con el que se quiere comunicar y un Tópico sobre el cual requiere datos. Si un Server esta dispuesto a responder al mensaje de inicialización de DDE envía un mensaje WM_DDE_ACK aceptando la comunicación, y enviándole como parámetro del mensaje el hWnd de la ventana que esta respondiendo al Cliente, ya que este ultimo lo necesitara en futuros envíos de mensajes.

De esta forma un Cliente obtiene el hWnd de una ventana de otro proceso (el Server), con lo que se logra una comunicación entre procesos basada en el sistema de mensajes de Windows, utilizando las funciones SendMessage y PostMessage según se indique en la especificación de DDE para enviar los distintos mensajes de requerimientos y respuestas entre Server y Cliente.

Mensajes de DDE

WM_DDE_ACK

El mensaje WM_DDE_ACK notifica a una aplicación de la recepción de un mensaje de DDE.

WM_DDE_ADVISE

La aplicación Cliente envía (post) este mensaje al Server para indicarle que requiere que el Server le envíe una actualización de los datos cada vez que estos cambian.

[2] El Clipboard es una serie de funciones en el modulo USER que permiten el intercambio de bloques de memoria entre programas. Los datos en el Clipboard pueden tener distintos formatos : texto, bitmap, metafile, etc.

WM_DDE_DATA

Una aplicación Server envía (post) este mensaje a un Cliente para indicarle que tiene datos actualizados para enviarle.

WM_DDE_EXECUTE

Un Cliente envía (Post) al Server este mensaje especificando un string que deberá ser procesado por el server como una serie de comandos. El Server debería responder con un WM_DDE_ACK a este mensaje.

WM_DDE_INITIATE

Este mensaje es enviado (Post) por una aplicación Cliente que desea iniciar una conversación por DDE con un Server. Los parámetros del mensaje son un nombre de aplicación y un tópico, de manera que la aplicación Server que coincida con el nombre especificado y soporte el ítem requerido debería responder con un mensaje WM_DDE_ACK.

WM_DDE_POKE

Un Cliente usa este mensaje para enviarle datos al Server, que debería responder con un mensaje WM_DDE_ACK si los acepta.

WM_DDE_REQUEST

Un Cliente envía este mensaje a un Server para requerirle datos.

WM_DDE_TERMINATE

Este mensaje puede ser enviado por un Cliente o un Server para indicar que se desea finalizar con un conversación.

WM_DDE_UNADVISE

Un aplicación Cliente envía (Post) este mensaje para informa al Server que los datos ya no necesitan ser actualizados como había sido especificado por un anterior WM_DDE_ADVISE.

2. 4. 3. OLE (Object Linking and Embedding)

OLE es una forma de interacción entre aplicaciones mas que entre procesos. Una aplicación usando OLE puede cooperar con otras aplicaciones que soportan OLE para producir un documento conteniendo diferentes clases de datos. OLE 1.0 estaba soportada sobre la base de una comunicación entre procesos mediante DDE, pero la idea de Microsoft es realizar una arquitectura de interacción de objetos que desplace totalmente a DDE. Un primer paso hacia esta arquitectura es OLE 2.0 el cual ya no necesita el soporte de DDE.

Desde el punto de vista del usuario, una aplicación que soporta OLE le permite editar documentos insertando objetos desde otras aplicaciones soportando OLE. Las aplicaciones que soportan OLE le permite al usuario moverse desde un esquema de computación centrada en la aplicación en un esquema de **computación centrada en el documento**. En este tipo de entorno la herramienta utilizada para realizar una tarea no es una aplicación sino que el usuario puede combinar las ventajas

de diferentes herramientas para completar su trabajo. Estos tipos de documentos son llamados **'compound documents'**. Un 'compound document' usa las facilidades de diferentes aplicaciones OLE para manipular los distintos tipos de datos que este muestra.

2. 4. 3. 1. Objetos

Un objeto es cualquier dato que pueda ser presentado en un 'compound document' y manipulado por el usuario. Cualquier cosa desde una celda en una planilla de cálculos hasta un documento completo puede ser un objeto. Cuando un objeto es incorporado en un documento, este mantiene una asociación con la aplicación que lo provee. Esta asociación puede ser un **'link'** o el objeto puede esta **'embedded'** en el archivo.

Si el objeto es 'linked' el documento provee almacenamiento mínimo para los datos a los cuales el objeto se refiere, y el objeto puede ser actualizado automáticamente cuando cambian los datos en la aplicación original.

Si un objeto es 'embedded' todos los datos asociados con el son guardados como parte del archivo en el cual el objeto es embedded.

2. 5. Graphics Device Interface. (GDI)

2. 5. 1. Introducción

Un sistema operativo es un entorno que administra las tareas, maneja la memoria y posee 'loaders' que cargan inicialmente el sistema. Algunos sistemas operativos tienen una *Graphical User Interface* (GUI) como X-Window para UNIX o el USER y GDI para Windows.

El modulo GDI es el motor gráfico de Windows (GDI.EXE). Este modulo permite que una aplicación Windows no acceda directamente a los dispositivos gráficos de display logrando por ejemplo producir los mismos resultados con un plaqueta gráfica SuperVGA o con un plotter. Para el acceso del GDI al dispositivo gráfico se usan drivers que tienen distintas capacidades. Estas capacidades las conoce GDI, por ejemplo si un drivers conoce como dibujar una línea, el GDI le dice al driver "dibuja un línea", si el driver no sabe dibujar una línea pero si sabe dibujar puntos el GDI le dice "dibuja un punto en la posición (x,y)" tantas veces hasta formar la línea.

Existen dos clases de 'screen output', la primera es por medio del USER, por ejemplo dibujar ventanas, 'caption', iconos, etc, esto se realiza usando funciones como *CreateWindow*, *ShowWindow*(), *MoveWindow*. El USER guarda el contexto de la ventana como la posición, el tamaño, etc. El USER usa la función *BitBlt* para actualizar la pantalla si es necesario. La segunda clase es usando directamente las funciones del GDI por ejemplo *LineTo*, *TextOut* y otras mas.

Como ya explicamos una característica interesante es que es independiente al sistema, es decir que el GDI es capaz de descomponer una función que no entienda del driver en varias funciones que la maneje, por ese motivo el GDI es muy grande, también tiene funciones que posiblemente nunca se usen porque el driver actual maneja funciones de mas alto nivel.

Cuando se usa una función del GDI, no se sabe que dispositivo se esta usando, el GDI dinámicamente carga el driver del dispositivo que necesita, esto se realiza cuando empieza el sistema, el GDI lee de un archivo de configuración llamado SYSTEM.INI para saber que driver debe cargar, por ejemplo en el archivo debe existir una línea como display.drv=VGA.DRV, en el caso de una impresora se realiza de la misma forma pero usando otra entrada.

2. 5. 2. Objetos GDI

El GDI usa handles a objetos para cumplir con los requerimientos gráficos de las aplicaciones. Estos objetos se encuentran en el heap local del GDI en bloques de memoria movible (LMEM_MOVIBLE). El GDI posee una lista de los objetos que pueden ser:

Objeto	Identificador	Descripción
Pen	1	Lápiz para dibujar líneas, rectángulos, círculos, etc
Brush	2	Pincel para llenar figuras como círculos, rectángulos, polígonos, etc.
Font	3	Letra la cual se usa para mostrar texto.
Palette	4	Conjunto de colores.
Bitmap	5	Mapa de bits. (imagen)
Region	6	Región delimitada por 2 coordenadas
DC	7	Device Context, contiene objetos y atributos.
IC	8	Information Context, información del dispositivo sin crear un Device Context.

Dado un handle con estos valores se puede saber de que tipo es este objeto. Los objetos como se menciono anteriormente son movibles, es decir que sus handles no son offsets fijos en la heap local del GDI.

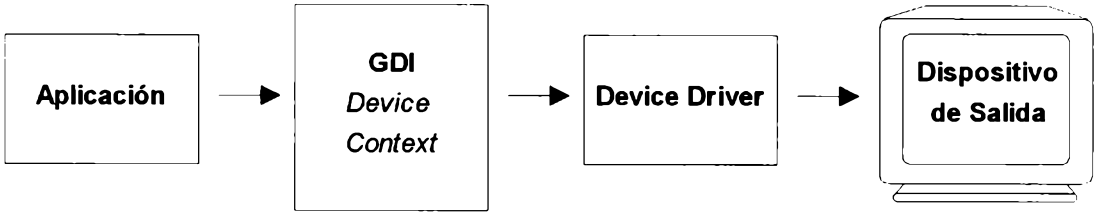
2. 5. 3. Device Contexts (DCs)

Es la estructura principal del GDI ya que con esta se resuelven muchas cosas. Pueden existir device context para impresoras o plotter además para un particular output. Cuando se quiere dibujar un gráfico sobre un dispositivo output (screen o impresora) primero se debe obtener un handle a un Device Context, el cual contiene atributos que indican como las funciones del GDI deben comportarse sobre ese dispositivo. Estos atributos permiten a las funciones conocer información sobre el dispositivo como la paleta de colores, las coordenadas de comienzo del brush o el pen, tamaño que Windows necesita para poder mostrar gráficos en la pantalla, etc.. Un ejemplo sencillo es la función *TextOut* la cual muestra un texto en pantalla, esta función tiene como parámetros el handle al DC, el comienzo de donde se va a mostrar el texto, el texto y el tamaño del mismo, mientras que el tipo de font a usar, el color del texto, el color del background y el espacio entre caracteres se toma del DC. Macintosh usa un concepto similar llamado GrafPort.

Casi todas las funciones del GDI usan un handle al DC, para obtenerlo se usa la función *BeginPaint* y cuando finaliza el uso del mismo hay que liberarlo usando la función *EndPaint*. También se puede obtener y liberar un handle usando las funciones *GetDC* y *ReleaseDC*. Los atributos mas usados del DC son handles a objetos del GDI como el pen actual, brush, font y bitmaps. Cuando se llama a una función que muestra texto o dibuja una línea, el GDI usa el objeto mas apropiado para realizarla. Para seleccionar otro objeto en el DC hay que llamar a otra función del GDI llamada *SelectObject*. Cuando un objeto ya no es utilizado se debe borrar con la función *DeleteObject*, que libera el toda la memoria asociada al objeto. Para liberar la memoria asociada al DC después de crearlo con la función *CreateDC* se usa las funciones *ReleaseDC* o *DeleteDC*.

Por ultimo, cuando Windows comienza su ejecución el USER crea cinco DCs para el display device, que son los que las aplicaciones obtienen mediante llamados a BeginPaint y GetDC, por lo que es recomendado liberarlos cuando la aplicación no los utiliza, para que el GDI pueda entregárselos a otras aplicaciones.

Mecánica de un requerimiento gráfico



Un Device Context realiza un link entre una aplicación Windows, un device driver y un dispositivo de salida como una impresora o un plotter. Windows mantiene un cache de 5 devices contexts. Se debe liberar un device context luego de usarlo. La idea es que una aplicación accede a un

dipositivo de salida por medio del GDI, luego este pasa el llamado al device driver, el cual traslada el llamado a operaciones dependientes del dispositivo de salida.

2. 5. 3. 1. Funciones

Un device context puede ser guardado y restaurado posteriormente (*SaveDC*, *RestoreDC*). También puede ser borrado usando la función *DeleteDC*, los recursos compartidos no son eliminados hasta que el ultimo device context es borrado. El device driver es un recurso compartido. Si una aplicación usa la función *GetDC* para obtener un device context, para liberarlo se usa la función *ReleaseDC*. La función *CreateIC* crea un information context para un dispositivo. Un information context es un device context con capacidades limitadas.

2. 5. 3. 2. Atributos

Los atributos del device context describen los objetos gráficos seleccionados (*pens y brushes*), *font* seleccionado y su *color*, el área disponible sobre el dispositivo (*clipping region*), e información importante. La estructura que contiene los atributos del device context es llamada data block. Otros atributos son: el *color del background*, el *bitmap* asociado, el origen del *brush* y del *pen*, la *paleta de colores*, el *modo de dibujo*, es decir el comportamiento al dibujar cualquier figura gráfica sobre el screen.

2. 5. 3. 2. 1. Regiones y rectángulos

Una región es un área del screen que es una combinación de rectángulos, polígonos y elipses. Windows tiene funciones que trabajan con regiones y rectángulos como *fillrect* que llena un rectángulo con un especificado brush, *frameRect* usa un brush para dibujar un frame, *InvertRect* invierte todos los pixels de un rectángulo.

La función *SetRect* actualiza las coordenadas en una estructura rectángulo (*REC*), Las funciones que manipulan a esta estructura son: *SetRectEmpty*, *CopyRect*, *IntersectRect*, *UnionRect*, *IsRectEmpty y PtInRect*.

Una región de clipping se usa para seleccionar una región dentro del device context. Para ampliar mas la idea una región de clipping es la zona donde se puede dibujar y pintar, los requerimientos gráficos fuera de ella no tendrá ningún tipo de efecto. La función *InvalidateRect* invalida un área rectangular del display y luego genera el mensaje WM_PAINT, se puede obtener las coordenadas de esta region con la funcion *GetUpdateRect* y la funcion *ValidateRect* valida el area.

2. 5. 4. Colores

Algunas funciones que crean pens y brushes requieren como parámetro un color. El color especificado puede ser un valor RGB o una entrada a una paleta lógica. Otro método es crear una paleta, un color RGB es un long integer que esta formado por valores de rojo, verde y azul. (**R**ed, **G**reen, **B**lue)

Cada valor en el color RGB indica la intensidad del color, 0 indica la intensidad mas baja, y 255 la mas alta. Cuando el GDI recibe un valor RGB como parámetro, este pasa el valor RGB directamente al device driver de salida, el cual selecciona valores del dispositivo. Si el dispositivo usa tecnología raster el valor RGB para un pen debe ser un color sólido, en cambio si el color es para un brush ese valor esta formado por combinaciones de colores.

2. 5. 5. Paleta de Colores

Existen dispositivos que son capaces de mostrar gráficos de diversos colores, el numero de colores a mostrar esta limitado. Por ejemplo un display puede producir mas de 262,000 colores, pero solo se puede mostrar 256 de estos colores a la vez por limitaciones de hardware.

Los colores de un display son guardados en una paleta de colores. Cuando una aplicación necesita un color que no esta en la paleta, el display lo agrega en la paleta, pero si el numero de colores sobrepasa la cantidad máxima de colores en la paleta, se reemplaza un color existente por este. Es decir una paleta es un buffer entre la intensidad de los colores y el sistema.

Las paletas de colores proveen un método para acceder a la capacidad de color que tiene un display. Para usar los colores en una aplicación Windows, primero se crea una paleta lógica con los colores a usar y luego se trasladan los colores a la paleta del sistema (la paleta por hardware). Una aplicación usa la paleta del sistema, la cual en realidad es usada por una o mas paletas lógicas. Cada entrada en esta paleta contiene un color especifico.

Cuando muchas aplicaciones usan paletas y la cantidad de colores excede la cantidad máxima de colores, Windows funciona como una mediador. La idea es pasar los valores de la paleta lógica a la del sistema, si el color esta en la paleta del sistema no se hace nada, en caso contrario se trata de guardar, si todas las entradas en la paleta del sistema están ocupadas y existen colores de la paleta lógica que no hacen matching, Windows hace el matching con los valores posibles.

Antes de dibujar en una display con los colores de la paleta, una aplicación primero debe crear una paleta usando la función *CreatePalette* y luego usar la función *SelectPalette* para seleccionar la paleta para el device context del display.

Sobre un dispositivo que soporta paletas se pueden crear paletas lógicas, seleccionar paletas en un device context y 'realize' la paleta. 'Realize' la paleta consiste en mapear los colores de la paleta lógica a la paleta del sistema (hardware) y se realiza usando la función *RealizePalette*. Para saber si el dispositivo es capaz de soportar estas funciones y el tamano de la paleta del sistema se usa la función *GetDeviceCaps*.

La diferencia entre la paleta del sistema y la paleta lógica es que la primera es la que actualmente esta en uso (los colores en el screen son colores de la paleta del sistema). Algunas aplicaciones no crean paletas lógicas, usan la paleta por default, es decir usan los colores predefinidos en la paleta del sistema. Las paletas lógicas son las que la aplicación crea, la aplicación también especifica que colores contiene. Usando las funciones de GDI la aplicación temporariamente reemplaza la paleta del sistema por la paleta lógica, la cual permite que la aplicación use los colores de la paleta lógica. Windows reserva algunas entradas en la paleta para si mismo, están no se pueden reemplazar.

2. 5. 6. Fonts

Cuando se llama a las funciones *TexOut*, *TabbedTextOut*, *ExtTextOut*, o *DrawText* para escribir un text, Windows usa el *font* seleccionado en el device context. El GDI ofrece fonts predefinidos, para obtener un handle a un font se usa la función *GetStockObject* y luego para seleccionarlo en un device context se usa la función *SelectObject*. Los seis fonts predefinidos por el GDI son:

Stock Font	Descripción
SYSTEM_FONT	Identifica al *system font*. Windows usa este font para texto en los menúes, dialog boxes, message boxes y window caption bars.
SYSTEM_FIXED_FONT	Identifica al *fixed-pitch ANSI FONT* compatible con system font, y usada en versiones anteriores al 3.0
OEM_FIXED_FONT	Identifica al *terminal font*. Windows usa este font para las ventanas DOS en modo carácter.
ANSI_FIXED_FONT	Identifica al *Courier font*, la cual es mas pequeña que el system font y el terminal font.
ANSI_VAR_FONT	Identifica a los font con ancho variable de los caracteres como *Helvética* y *Times Roman font.*
DEVICE_DEFAULT_FONT	Identifica al *font default* del dispositivo de salida.

2. 5. 6. 1. Tipos de Fonts

Windows soporta dos categorías de fonts llamadas *GDI fonts* y *device font.* Las primeras son guardadas en archivos con extensión FON, llamados *font resource files.* Estos archivos contiene un directorio de fonts y los fonts. Los device fonts internos al dispositivo de salida. son muy comunes en las impresoras, por ejemplo, al imprimir un GDI fonts Windows envía un conjunto de pixels a la impresora, al imprimir un device font solo se le envía el código ASCII.

Las aplicaciones Windows pueden usar tres tipos de GDI fonts para mostrar en un display o imprimir, estos fonts son:

Raster.
Vector.
TrueType.

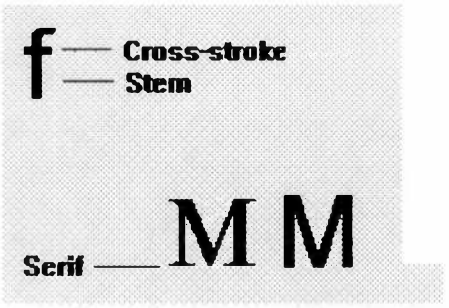
Los font llamados *raster* son guardados como bitmaps. cada raster font son diseñados para un aspecto y tamaño especifico. Window puede crear caracteres mas largos de un raster font simplemente duplicando columnas y filas de los pixels, el problemas es que al cambiar de tamaño el carácter es mas difuso. Una ventaja es que una aplicación usa una cantidad muy grande de font raster y al manejarlos como bitmaps es mas rápido para su uso.

Los font *vectorizados* son definidos como series de segmentos de lineas conectados y guardados como colecciones de vectores. Estos son 'scalables', es decir se pueden generar distintos tamaños y aspectos del font, y son usados en dispositivos como plotters donde los bitmaps no se pueden usar.

Los font *TrueType* son guardados como colecciones de *splines* (funciones poligonales) y algoritmos que generan el bitmap y manejan los contornos para mejorar la apariencia de este (algoritmos de *rendering* y *antialiasing*).

Los GDI fonts pueden ser pueden ser *italic*, **Bold,** underlining, ~~strikethrough~~.

Windows define formas (*typefaces*) las cuales son: Courier, Helvética, Times Roman, Gothic y Palatino y las agrupa en 5 familias basadas sobre la apariencia del tipo. Las familias son *Modern, Swiss, Roman, Script,* y *Decorative* y están compuesta por fonts que tiene características de diseño en común. Las familias se distinguen por el ancho de un stroke (línea horizontal o vertical) y por las características de serif. Un stroke horizontal es llamado crodd-stroke y las líneas verticales ítem.



Los serifs son pequeñas líneas dibujadas al final de la letra.

2. 5. 7. Gráficos

2. 5. 7. 1. Línea

Para dibujar una línea el GDI usan las coordenadas lógicas, las cuales son transformadas a coordenadas dependientes del dispositivo por el device driver. Es decir que el GDI mapea esta línea desde un espacio lógico a un uno de pixels sobre el dispositivo. Windows al dibujar una línea usa el pen seleccionado en el device context. El GDI tiene un pen por default, este pen es negro y con ancho de 1 pixel. La aplicación puede crear un nuevo pen de diferente ancho, estilo y color usando la función *CreatePen*. Los estilos pueden ser 'solid', 'dotted', 'dashed', o combinados (dotted y dashed). Una vez que la aplicacion crea un pen, hay que seleccionarlo en el device context usando la función *SelectObject*.

2. 5. 7. 2. Elipses y Polígonos

Igual que al dibujar una línea, dibujar un polígono o un elipse también se usan las coordenadas lógicas, Al usar coordenadas lógicas nos aseguramos de la independencia de dispositivo de output. El GDI mapea un objeto desde un espacio lógico a pixels en el dispositivo.

2. 5. 7. 2. 1. Rectángulos

La función *Rectangle* dibuja un rectángulo, usando el pen actual. la función *RoundRect* dibuja un rectángulo con las esquinas redondeadas. Cuando el GDI dibuja un rectángulo necesita 4 parámetros. La coordenada de la esquina superior izquierda y la coordenada de la esquina inferior derecha.

2. 5. 7. 2. 2. Elipse, Chord y Pie

Las funciones *Chord*, *Ellipse*, y *Pie* usan un rectángulo limitado (bounding rectangle), en vez del radio para definir el tamaño de la figura a dibujar. El bounding rectangle esta oculto, es decir, el GDI lo usa solo para saber el tamaño y la localización de la figura.

2. 5. 7. 3. Funciones

Chord	Dibuja un chord (una figura formada por un arco y una línea que une ambos extremos).
Ellipse	Dibuja un elipse.
Pie	Dibuja un pie (una figura formada por un arco y dos líneas unidas de los extremos hacia en centro).

Polygon	Dibuja un polígono.
PolyPolygon	Dibuja una serie de polígonos.
Rectangle	Dibuja un rectángulo.
RoundRect	Dibuja un rectángulo redondeado.

2. 5. 8. Cursores

Las funciones de manipulación de cursores en Windows son:

CreateCursor	Crea un cursor con dimensiones especificas.
DestroyCursor	Destruye un cursor creados por las funciones CreateCursor o LoadCursor.
GetCursorPos	Retorna la posición actual del cursor.
LoadCursor	Carga un cursor de un recurso.
SetCursor	Cambia el cursor del mouse por el cursor especificado.
SetCursorPos	Setea la posición el cursor del mouse con las coordenadas indicadas.
ShowCursor	Oculta o Muestra el cursor del mouse.

2. 6. Módulos y Tareas

La principal función de un sistema operativo es el control de la ejecución de los programas conocidos como *tareas* o *procesos*. Esto incluye la creación de tareas y la finalización normal o anormal de la ejecución de las mismas. Un sistema operativo puede manejar la ejecución de múltiples programas y también diferentes instancias del mismo programa las cuales comparten los segmentos de código que son de solo lectura. Primero explicaremos los distintos tipos de segmentos de un programa Windows.

2. 6. 1. Segmentos de Programa

Bajo un modelo de memoria segmentada cuando se enlaza un programa, *código* y *datos* se separan y organizan en segmentos. El numero de segmentos de código y datos asignable depende del modelo de memoria usado por el compilador y el linker. Luego el compilador de recursos agrega al final del EXE los *recursos*. El comienzo del programa tiene un a cabecera que contiene información de los segmentos de código, datos y recursos para el dynamic linking.

El numero total de segmento de cualquier programa Windows no puede exceder de 254. También cada segmento tienen un nivel de protección, por ejemplo los segmentos de código solo pueden ser de ejecución y de lectura., los segmentos de datos de lectura y escritura, y los segmentos de recursos de lectura.

2. 6. 1. 1. Segmentos de código

Los segmentos de programa representan las secciones de espacio de direccionamiento que contiene la totalidad o parte de las instrucciones de un programa. Los segmentos de código como los otros segmentos se mapean en secciones del espacio lineal de direcciones. Un segmento de código debe tener funciones completas, es decir una función no puede estar distribuida en distintos segmentos. La distribución de las funciones en los segmentos de código puede ser indicado por el linker o por el programador. Windows para gestionar memoria usa LRU (*last recently used*) y el VMM se encarga del mecanismo de paginado del 80386 para memoria virtual.

Las funciones siempre se agrupan en los segmentos por orden alfabético excepto si la función es muy grande o cuando la ubicación de la función ha sido alterada. Los segmentos de código están marcados normalmente DISCARDABLE o MOVEABLE, los segmentos de código de los drivers son FIXED para evitar que sean descargados, el tamaño de los segmentos de código ideal debe estar por debajo de los 4K.

2. 6. 1. 2. Segmentos de datos

Los segmentos de datos son secciones de lectura y escritura del espacio de direcciones que contienen la totalidad o una parte de datos de un programa. Estos segmentos no tienen nivel de protección, Windows solo protege a nivel de segmento. Los segmentos de datos generalmente están marcados como MOVEABLE ya que ser DISCARDABLE puede ocasionar mas problemas que ventajas. En Windows todos los programas contienen al menos un segmento de datos, este es conocido como DGROUP (*automatic data segment*) donde se pueden encontrar variables estáticas, la pila, variables dinámicas (están basadas en la pila), el espacio local de memoria y la tabla local de átomos, por este motivo el segmento no es DISCARDABLE. El código de carga crea la pila y la memoria local, indicando en la cabeza del segmento (16 bytes) donde situar los objetos dentro del

segmento. Los segmentos de datos deben ser de tamaño superior a 4K u 8K, deben ser movibles y al usar Windows modo protegido de 16 bits los segmentos no pueden exceder los 64K.

2. 6. 1. 3. Segmentos de recursos

Los segmentos de recursos contiene parte de los datos de los programas. A diferencia de los segmentos de datos los segmentos de recursos son de solo lectura. pueden contener bitmap, iconos, cursores, menú, etc. son generalmente MOVEABLE, DISCARDABLE.

2. 6. 1. 4. Segmentos de tarea

Los *segmentos de tarea* o *segmentos de datos asignados dinámicamente* representa secciones e lectura y escritura de direcciones que contienen datos del programa asignados globalmente no necesariamente compartidos. La principal diferencia esta en su tamaño y la forma en que la memoria es reservada o liberada. Se pueden crear dinámicamente usando la función *GlobalAlloc*. Estos segmentos se crean para guardar matrices, array o cualquier tipo de datos y pueden crecer hasta 16M (están formados en segmentos de 64K). Los segmentos de tarea se liberan automáticamente cuando la aplicación termina, existen dos excepciones, la primera es cuando una librería necesita reservar memoria global, la necesita para ella y no para la tarea, entonces usando los atributos GMEM_SHARE y GMEM_DDESHARE permite a la librería retener la memoria hasta que la libere explícitamente o hasta que acabe su ejecución. Cuando una aplicación reserva memoria usando GMEM_SHARE, esta memoria es compartida por varias instancias de una aplicación, por eso la memoria solo se libera cuando la ultima instancia de la aplicacion termina.

2. 6. 2. Módulos

En Windows hace referencia a todo el código, datos y recursos en un archivo particular, puede ser un programa o una librería de enlace dinámico (*DLL*). Un modulo esta formado por información que existe en un archivo en el disco.

Una DLL contiene lo mismos componentes (código, datos, recursos) que un EXE por eso no es necesario que una librería tenga la extensión DLL, un ejemplo común son los archivos de Font los cuales tienen extensión FON o FOT, los drivers que tienen extensión DRV son también DLL, los archivos de Windows USER.EXE, KRNLx86 y GDI.EXE son DLL, este formato de archivo es un estándar llamado New Executable (NE).

Cuando Windows necesita información de un archivo NE (cuando se ejecuta un programa o se carga una DLL) el KERNEL lee el header del NE y varias tablas guardándolas en una tabla llamada tabla de modulo o base de datos de módulos (*Module Database MDB*). Cuando se quiere cargar una segunda instancia de un programa Windows usa la misma tabla, porque entre las instancias solo difieren en los segmentos de datos, pero los segmentos de códigos son iguales, con esta tabla Windows maneja el compartimiento de los segmentos de código. La MDB contiene información aplicable a todas las instancias activas del programa. La mayor parte de esta información esta relacionada con los segmentos de código y recursos que son unidades compartidas, también contiene un contador de instancias, este campo se usa para saber la cantidad de instancias activas del modulo.

Cuando la ultima instancia de un programa termina se libera la MDB con los segmentos y recursos que este apunta, similarmente si el numero de programas o DLLs usando una DLL es cero el código, recursos y datos asociados con esta DLL son liberados por el KERNEL.

Windows usa un método de compartir código entre programas muy simple comparado con otros sistemas operativos. Windows supone que los segmentos de código nunca serán modificados. Este es

un problema para los debugging porque al poner un breakpoint lo esta haciendo en un segmento de código que esta siendo compartido, en Windows no existe ese problema porque una instancia puede interceptar excepciones destinadas a otra instancia, con eso un debugger de Windows puede chequear si la tarea que recibió la excepción del breakpoint es la misma que comenzó el debugger.

Una restricción 'NO popular' en Windows surge debido a la suposición fundamental que los segmentos de código son compartidos por múltiples instancias del mismo programa. Si un programa tiene mas de un segmento de datos solo se podrá ejecutar una instancia del mismo. En un programa común, el cual solo tiene un segmento de datos, el segmento de código no necesita tener referencias al DGROUP, porque el selector que apunta a este esta en el registro DS. En un programa con múltiples segmentos de datos, cuando el loader carga los segmentos de código tiene que resolver las referencias a los segmentos de datos distintos al DGROUP, por este motivo al ser el segmento de código compartido entre las instancias de un programa, cualquier instancia del mismo haría referencia a los segmentos de datos (distintos del DGROUP) de la primera instancia.

Esta restricción hace que ciertos programas como ser Microsoft Word 2.0 estén limitados a correr sólo una instancia ya que este programa tiene más de un segmento de datos (no ocurre lo mismo con la versión 6.0 que sólo tiene 1 segmento de datos).

Por lo tanto, como hago para poder editar dos documentos al mismo tiempo ?. Claramente la solución más simple es correr dos instancias de Microsoft Word, pero por lo anterior esto no es posible. Otra solución es que Word me permita editar los dos documentos. **Sí, he aquí el porqué del concepto de MDI que Microsoft introdujo con Windows**[3] .

Como sabemos MDI (Multiple Document Interface), es un concepto de Interface de Usuario que permite trabajar con múltiples documentos (denominación genérica a los documentos Word, planillas Excel, drives del File Manager, etc.) al mismo tiempo salvando de esta manera la importante restricción de que un programa que tenga más de un segmento de datos pueda correr sólo una instancia. Honestamente no quedó tan mal ...

Windows mantiene una lista de todos los módulos cargados en el sistema. Cada tabla de modulo tiene un selector a la próxima tabla de modulo en una lista. El handle de un modulo es un selector a códigos, datos y recursos. Un handle de una instancia es un selector a un segmento que contiene el segmento de datos de un EXE o DLL.

En conclusión un modulo es un conjunto de información de un archivo que puede ser compartido entre varias instancias del modulo y las instancias de módulos es la información que no se puede compartir como es el segmento de datos, entonces puede haber un modulo con varias tareas.

2. 6. 3. Tareas

Una *tarea* es una unidad activa, es decir tiene un flujo de control ejecutándose en el segmento de código de su modulo. Como se explico antes los segmentos de código son puros, permitiendo que mas de una tarea use el mismo segmento de código y segmentos de solo lectura. Los módulos tienen referencias a los segmentos de código y recursos que fueron cargados por el loader del archivo en disco que tiene formato NE. Una tarea es el la ejecución del código del modulo. Si lo comparamos a la programación orientada a objetos un modulo es el conjunto de miembros y métodos, es decir la clase y una tarea es un objeto. Las tareas asociadas a un mismo modulo usan los mismos segmentos de código, pero se diferencian entre ellos usando distintos punteros al DGROUP.

[3] Es importante reiterar que el concepto de MDI no existe en ningún otro sistema de ventanas.

Cada tarea en Windows tiene una estructura que contiene datos propios. La estructura es llamada base de datos de tarea (*Task Database TDB*), y contiene información especifica a la invocación del programa, incluyendo el handle a su modulo, el handle al segmento de datos (DGROUP). Windows ejecuta una tarea a la vez, todas las demás tareas están en estado de espera.

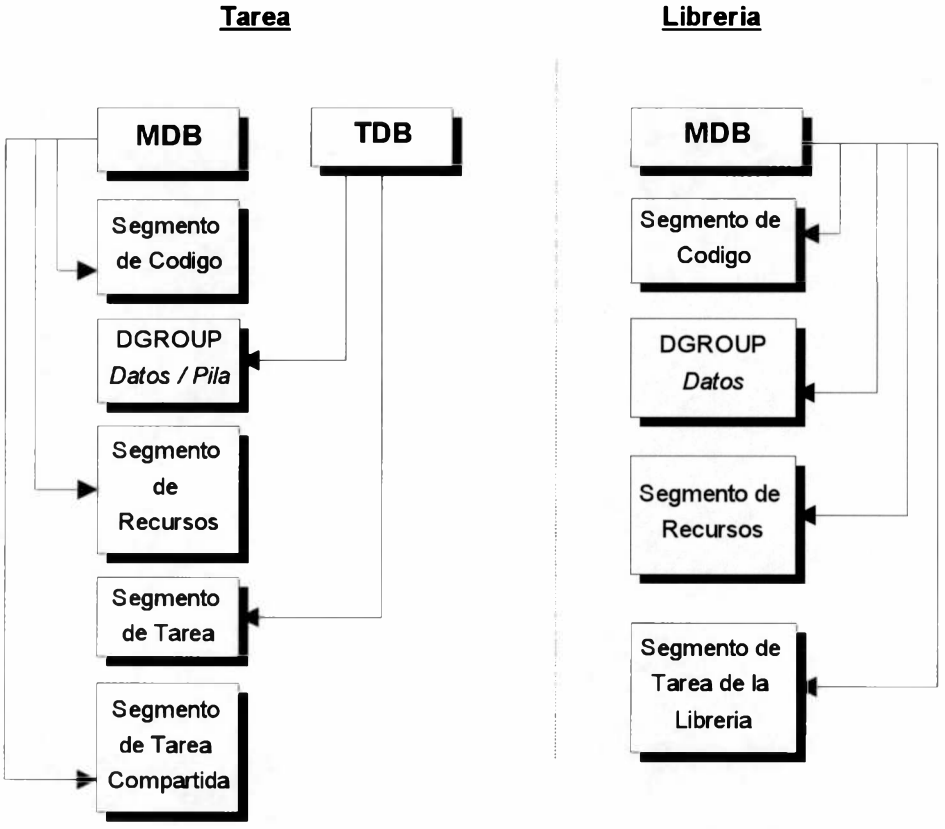
Toda la información que necesita una tarea para ejecutarse se toma información del TDB. Como la tabla de módulos, las tabla de tareas están unidas en una lista. Existe una función llamada *GetCurrentTask* que devuelve el handle de la tarea que esta activa actualmente.

2. 6. 4. Diferencias entre las Tareas y las Librerías

Las *Librerías* se diferencian de las *Tareas* en varios conceptos: como son creadas, como son manejadas por Windows, como se cargan y las funciones que pueden o no contener. La TDB contiene todo lo que hace diferente a una tarea con respecto a una librería, porque una librería no tiene asociada ninguna TDB. Una tarea contiene un segmento de pila, la cual define el flujo de control para la ejecución de la misma. Los segmentos de recursos, código y datos son compartidas. Esto quiere decir que todas las ventanas creadas por la DLL están asociadas a la tarea porque la librería carece de TDB y por lo tanto no tiene cola de mensajes para procesar los mensajes que reciba.

Las librerías no contienen segmento de pila, lo que provoca que SS!=DS, es decir el segmento de pila esta en el segmento de datos de la tarea y no en el segmento de datos de la librería. Una librería se define como una unidad pasiva que usa el flujo de control de las tareas que la llaman y las tareas son unidades activas. Para poder diferenciar una tarea de una DLL en una MDB existe un bit en el offset 0C (tarea=0, DLL=1).

Segmentos de programa en tareas y librerias



2. 7. Dynamic Linking

El termino *Dynamic Linking* se refiere al proceso que usa Windows para resolver la referencia a un llamado a una función de una librería desde un modulo. *Static Linking* sucede cuando se ejecuta el linker para formar un EXE crea una única y sola entidad con todos los componentes, es decir el código del programa y las librerías con las funciones que este referencia.

Windows usa el *Dynamic Linking,* es decir el linking en tiempo de ejecución. El código de las librerías nunca están cuando se enlaza un programa, por ese motivo el linker no puede resolver las referencias a las funciones de la librería, estas referencias las resuelve el loader.

Con esta característica Windows es mas potente y posee una mayor flexibilidad, un ejemplo común es el uso de los drivers por parte Windows. El mismo corazón de Windows (*GDI, USER, KERNEL*) trabaja con diferentes tipos de videos, mouse, keyboard, sound, puertas comunicaciones, y driver para impresión. El Dynamic Linking permite calcular las direcciones a las funciones en las librerías a ultimo momento, mientras un programa se este ejecutando. Cuando Windows comienza, el KERNEL usa el archivo SYSTEM.INI para saber que driver debe cargar. Windows no fue el primer sistema operativo con esta característica, también la tiene OS/2 y UNIX con unas librerías compartidas.

Cuando Windows carga un programa en memoria para ejecutarlo, el loader debe resolver las referencias a las funciones externas en este. Estas referencias están en la tabla de reubicación en el archivo EXE o DLL organizada por segmentos donde las entradas están dadas por el nombre del modulo y el ordinal de la función que exporta el modulo. Sabiendo el nombre del modulo se debe buscar en la MDB la tabla de entradas, en caso de no existir la MDB asociada a la librería, el loader carga en memoria los segmentos de código y datos y crea una MDB para esta.

En la Tabla de entradas se debe acceder a la función referenciada donde se informa el segmento lógico y el offset de la función, luego se accede a la tabla de segmentos donde esta el valor del selector que carga el loader cuando cargo el modulo de la librería. con el selector y el offset se reemplaza la referencia a la función.

Cuando se habla de Dynamic Linking se asocia con el concepto de Dynamic Link Libraries (DLL) que no son programas ejecutables, y no reciben mensajes. Estas librerías son archivos que contienen funciones que pueden ser llamadas por programas u otras librerías para realizar ciertas trabajos, es decir que varios programas pueden realizar llamados a funciones a un misma librería.

El linking de funciones externas de una aplicación se puede realizar el tiempo de carga de una aplicación (*Load Time linking*) o en tiempo de ejecución (*Run Time Linking*).

2. 7. 1. Load Time Linking

Cuando se cargan las librerías se traen a memoria antes que se cargue y ejecute el modulo cliente que hace referencia a ella. Si este cliente es una aplicación previamente cargada, el punto de entrada de la librería no podrá hacer llamado a las funciones que invoquen al sistema de mensajes, porque la cola de mensajes no ha sido creada.

Una librería solo se carga si el programa hace al menos una referencia a alguna de las funciones de la librería. Si una librería es cargada para el uso de las clases de ventanas registradas debería utilizar la función *LoadLibrary* para forzar una referencia. La razón es simple: el loader carga la librería solo si no existe la MDB asociada a esta para resolver alguna referencia a una función.

Como se explico una librería esta referenciada solo por el uso funcional y no por el empleo de las clases de ventanas asociadas con la sentencias edit, listbox o algo similar. Para liberar estas librerías se usa la función *FreeLibrary* al final. Para evitar todo esto se podría forzar a una referencia a una función de tipo InitListBox en la librería.

2. 7. 2. Run Time Linking

Una librería que se carga dinámicamente usando la función *LoadLibrary*. Las referencias en este tipo de librerías se resuelven en ejecución (*Run Time Linking*). Una diferencia es que cuando falla la carga de una librería en ejecución le devuelve al programador NULL, mientras que en tiempo de carga el mensaje es recibido por el usuario final.

```
---- declaraciones ----

hInst=LoadLibrary("USER.EXE");
MessageBeeper = (MSGBEEP) GetProcAddress(hInst, "MessageBeep");
(*MessageBeeper)(1);
FreeLibrary(hInst);
```

También en vez de usar llamadas a *LoadLibrary* y *FreeLibrary* se podría haber usado la función *GetModuleHandle* ya que la función USER esta cargada.

La diferencia entre los dos tipos de linking de DLL es que en *Load Time Linking* se cargan primero las librerías y luego el modulo padre, esto sucede porque el modulo padre necesita las direcciones a las funciones de la librería. En la carga dinámica para saber las direcciones se usa la función *GetProcAddress*.

2. 8. VMM (Virtual Machine Manager)

El Virtual Machine Manager es un **sistema operativo manejado por eventos, con 'preemptive multitasking', 'single-threaded'**. Este sistema operativo o kernel es también conocido como "WIN386" o "VMM".

Siendo estrictos, el VMM no forma parte de Windows; sino, que es un kernel con multitasking que administra (creación, destrucción y scheduling) de múltiples máquinas virtuales (VMs). Una vez que el VMM se ha inicializado, Windows (KRNL386.EXE, USER.EXE y GDI.EXE), junto con sus drivers (los .DRV) son cargados en la VM del Sistema (la primera máquina virtual creada en la inicialización de VMM). Sin embargo, VMM podría cargar el COMMAND.COM en la VM del Sistema y con la asistencia de ciertos VxDs más algunas 'helper hot-keys' (como el task switcher de Windows) y así disponer de un DOS con multitasking.

La VM del Sistema interacciona con el VxD SHELL del VMM para crear nuevas máquinas virtuales o lo que es lo mismo "cajas DOS". Cada nueva VM se ejecuta en el modo Virtual 8086 (V86). Es importante notar que la VM del Sistema se ejecuta en modo protegido de 16 bits, el KERNEL de Windows es un extender de 16 bits.

2. 8. 1. VxDs (Virtual Device Drivers).

Un VxD está compuesto de código y datos que corre en el anillo (o nivel de privilegio) 0 en el modelo flat de 32 bits como parte del Virtual Machine Manager. En efecto, el VMM está formado básicamente por un conjunto de VxDs standards, todos dentro del mismo archivo WIN386.EXE. Los VxDs sólo operan cuando Windows está corriendo en modo Enhanced. No existe diferencia entre estos VxDs standard y un VxD provisto por 3ras partes.

Debido a que los VxDs operan en el anillo 0, el cual es el nivel de protección del sistema operativo, la CPU permite que el código ejecute cualquier instrucción del 386. En niveles de anillo superiores (1, 2 y 3), el acceso a direcciones de memoria y a ports de I/O (por parte de las VMs) puede restringirse, permitiendo que el VMM o un VxD procese la excepción como desee. Por supuesto, ciertas instrucciones ejecutadas por una VM causan siempre una excepción del procesador, pero un VxD puede simular la funcionalidad de la instrucción al VM, permitiendo a ésta que opere como si tuviera los privilegios suficientes.

Un VxD es el único programa que no tiene restricciones para acceder al hardware. Si un VxD realiza una E/S a un port, se comunica directamente con el port físico, sin restricción alguna. Si un VxD es dueño de una interrupción por hard, el VxD recibe la IRQ directamente desde el Driver Virtual del Controlador de Interrupciones Programables (VPICD), sin transiciones de anillo. Por ejemplo, una ISR (Interrupt Service Routine) en una VM ve una interrupción virtualizada vía eventos administrados por el VPICD, mientras que un VxD tiene un camino más directo para el servicio de las interrupciones. Mientras que el código para la comunicación con el hardware en una VM puede restringirse o ser más lento debido a transiciones de anillo o por verificaciones de permisos de acceso, un VxD no tiene restricciones y es extremadamente rápido.

Un VxD puede interceptar y/o generar interrupciones (de hardware y software), hacer trapping de los ports de I/O, y aún restringir el acceso a ciertas área de memoria que haga una VM. Un VxD determina si permite el acceso, simularlo, terminar la ejecución de la VM o ignorar el acceso.

Una aplicación DOS que se comporte mal usualmente producirá un crash de la máquina virtual DOS. Una aplicación Windows que se comporte mal afectará la operación de otras aplicaciones Windows. Sin embargo un VxD que se comporte mal producirá un crash de todo Windows. Debido a que un VxD es parte del kernel VMM, el VxD está activo durante el procesamiento crítico del Windows.

Los VxDs fueron originariamente diseñados para manejar la **contención** en el acceso a dispositivos de hardware entre múltiples procesos y para trasladar o hacer buffering de datos trasmitidos por las VMs a los dispositivos de hardware. Cuando dos o más programas intentan acceder al mismo dispositivo, se debe usar algún método de contención. Es posible usar un VxD para permitir que un proceso actúe como si tuviera acceso exclusivo al hardware. Por ejemplo, un Virtual Printer Device proveerá al proceso con un port de impresión virtual, y los caracteres escritos al port serán enviados a un spooler de impresión. El VxD enviará luego el trabajo a la impresora cuando este esté libre. Existen varios métodos de contención.

Recientemente el uso de VxDs se ha expandido a incluir la comunicación entre procesos y a la implementación de dispositivos virtuales verdaderos, proveyendo los mecanismos necesarios para que una VM vea un dispositivo que realmente no exista en hardware.

2. 8. 2. Scheduling de Máquinas Virtuales.

Como se mencionó anteriormente, el VMM provee el scheduling de ejecución de las máquinas virutales. El VMM provee dos schedulers: un **scheduler primario** (basado en prioridades de ejecución), y un **scheduler secundario**, o time-slice scheduler. El scheduler primario selecciona la VM que estará activa basándose en la prioridad de ejecución más alta de la VMs que no están suspendidas. Una VM permanecerá activa hasta que se encuentre en la cola una VM con mayor prioridad. Estas prioridades son variables y las maneja el VMM.

El scheduler secundario ajusta la prioridad de ejecución de las VMs basándose en las prioridades de foreground y background que tienen configuradas cada VM.

2. 8. 3. Administración de Memoria.

El VMM implementa dos administradores de memoria. El Memory Manager (MMGR) provee servicios tales como administración de las tablas local y global de descriptores (LDT y GDT), administración de memoria física, administración de páginas y traslación de direcciones entre el modo V86 y el modo protegido de 32 bits.

El V86MMGR provee simplemente una interface a los VxDs para mapear buffers de datos desde modo protegido a modo V86 y viceversa.

Sintetizando el Virtual Machine Manager provee los siguientes servicios:

- VxDs (Virtual Devices).
- Scheduling de Máquinas Virtuales (VMs).
- Administración de Memoria.

Parte 3 - X Window System

3. 1. Introducción.

El X Window System, o X, a diferencia de la mayoría de sistemas de ventanas, está definido por un *protocolo de red,* es decir, la forma tradicional de llamadas a procedimientos o invocación al kernel es reemplazada por una comunicación entre procesos asincrónica basada en streams.

El sistema de ventanas provee gráficos de alta performance a una jerarquía de ventanas 'resizable'. X no define una interfase de usuario en particular, provee primitivas para soportar varias políticas y estilos. Un importante objetivo de diseño fue entonces hacer que X sea 'policy-free'.

Una aplicación puede utilizar ventanas en cualquier display de la red de una manera independiente al dispositivo y transparente a la red. Interponer una conexión de red mejora la utilidad del sistema de ventanas, sin afectar la performance significativamente. La performance de implementaciones de X actuales es comparable a la de sistema de ventanas actuales y, en general, está limitada por el hardware de display y no por la comunicación de red.

3. 1. 1. Requerimientos.

Un sistema de ventanas contiene muchas interfases. Una *interfase de programación* es una librería de rutinas y tipos provistas en un lenguaje de programación para interactuar con el sistema de ventanas. Típicamente se proveen interfases de bajo nivel (ej.: dibujo de líneas) y de alto nivel (ej.: menúes).

Una *interfase de aplicación* es la interacción mecánica con el usuario y la apariencia visual que es específica a la aplicación. Una *interfase de administración* es la interacción con el usuario que trata con el control completo del desktop y los dispositivos de entrada. La interfase de administración define cual es el 'layout' de las aplicaciones en la pantalla, y como el usuario cambia entre aplicaciones; una interfase de aplicación particular define como es presentada y manipulada la información en esa aplicación. La *interfase de usuario* es la suma total de todas las interfases de aplicación y de la interfase de administración.

Además de aplicaciones, distinguimos tres componentes principales en un sistema de ventanas. El *window manager* implementa la porción desktop de la interfase de administración; controla el tamaño y ubicación de las ventanas de aplicación, y los atributos de estas, como títulos y bordes. El *input manager* el resto de la interfase de administración; controla que aplicaciones ven el input desde que dispositivos (ej.: teclado y mouse). El *sistema de ventanas base* es el núcleo a partir del cual se construyen las aplicaciones, windows managers e input managers.

Los siguientes son los requerimientos que se le pidieron al sistema de ventanas base durante el diseño de X:

1) El sistema debe ser implementado en una variedad de displays.

2) Las aplicaciones deben ser independientes de los dispositivos: No debe ser necesario reescribir, recompilar o aún relinquear una aplicación para cada display. También, toda función gráfica definida por el sistema debe soportarse en todo display soportado.

3) El sistema debe ser transparente a la red: Una aplicación corriendo en una máquina debe poder utilizar el display en cualquier otra máquina; no debe ser necesario que ambas máquinas tengan la misma arquitectura o sistema operativo.

4) El sistema debe soportar múltiples aplicaciones dibujando en pantalla concurrentemente.

5) El sistema debe soportar múltiples interfases de aplicación y de administración: Para lograr esto, el sistema debe proveer mecanismos más que políticas. Por ejemplo, debido a que las semánticas y estilos de menúes varían dramáticamente entre diferentes interfases de usuario, el sistema de ventanas base debe proveer primitivas a partir de las cuales pueden construirse menúes, en lugar de proveer una facilidad de menú particular. El sistema debe ser diseñado de tal manera que sea posible implementar políticas de administración de una manera que sea externa al sistema de ventanas base y externa a las aplicaciones. Las aplicaciones deben ser independientes de mecanismos y políticas de administración; las aplicaciones deben reaccionar a decisiones de administración en lugar de dirigir esas decisiones. Por ejemplo, una aplicación necesita ser informada cuando una de sus ventanas es 'resized' y debe reaccionar reformateando la información que está mostrando, pero no debe ser necesario que participe la aplicación para que el usuario cambie el tamaño. Hacer que las aplicaciones sean independientes de la administración e independientes de los dispositivos, facilita que se compartan las aplicaciones entre diversas culturas.

6) El sistema debe soportar ventanas que se solapen, incluyendo dibujar en ventanas parcialmente visibles.

7) El sistema debe soportar una jerarquía de ventanas 'resizable', y una aplicación debe poder usar varias ventanas al mismo tiempo: Las subventanas proveen un mecanismo claro y poderoso para exportar a la aplicación mucha de la maquinaria del sistema base y que esta pueda usuarla directamente. En principio cada aplicación define su propia noción de ventana; por ejemplo, algunas aplicaciones implementan lo que escencialmente es otro sistema de ventanas dentro del sistema de ventanas real. Es importante soportar niveles arbitrarios de anidamiento. Lo que es visto como una única ventana en un nivel de abstracción, puede requerir múltiples ventanas en un nivel de abstracción de menor nivel. Al proveer una jerarquía de ventanas verdadera, las ventanas de las aplicaciones pueden ser implementadas como verdaderas ventanas dentro del sistema, liberando a las aplicaciones de la necesidad de duplicar la maquinaria como ser control de clipping e input.

8) El sistema debe proveer texto, gráficos en 2-D e 'imaging' de alta calidad y alta performance: El sistema de ventanas base debe proveer gráficos "inmediatos", es decir, primitivas gráficas de bajo nivel. No se soportan modelos de alto nivel, donde la aplicación describe que quiere en términos de objetos abstractos y el sistema determina la mejor manera de generar la imagen. Es extremamente importante proveer modelos de alto nivel, pero estos deben ser construidos arriba del sistema de ventanas base. No es un requerimiento soporte a gráficos en 3-D.

9) El sistema debe ser extensible: Por ejemplo, el sistema base puede no soportar gráficos en 3-D, pero debe ser posible extender al sistema para este soporte. El mecanismo de extensión provisto debe permitir extender el sistema en forma no cooperativa, y aún permitir que tales extensiones independientes puedan ser unidas consistentemente.

3. 1. 2. Arquitectura del Sistema. El modelo Cliente / Servidor.

El X Window System está basado en el modelo cliente/servidor; esto modelo surge a partir de los puntos 2 y 3 anteriores. Las aplicaciones X son los clientes que usan los servicios de red del sistema

de ventana. Un programa que corre en la máquina con el hardware de display provee estos servicios y es llamado el X server. Para cada display, existe un X server que lo controla. El X server actúa como un intermediario entre las aplicaciones y el display, manejando la salida que producen al display los clientes y enviando la entrada (desde un teclado o mouse) hacia los clientes apropiados para su procesamiento.

Clientes y servidores se comunican sobre un stream de bytes bidireccional y confiable. Si el cliente y el server están en la misma máquina, se utiliza un algún mecanismo local de comunicación entre procesos; de otra forma, se establece una conexión de red entre ambos. Existe un protocolo arriba de este stream de bytes basado en un stream de bloques. Este protocolo es el que usan clientes y servidores para intercambiar información, y el que define la sintaxis y semántica de esta conversación. Este protocolo es la base del X Window System.

Es importante resaltar que requiriendo sólamente un transporte basado en un stream de bytes, bidireccional y confiable, hace que X sea usado en muchos entornos. Por ejemplo, el protocolo puede ser usado arriba de TCP/IP, DECnet, Chaos y IPX/SPX.

Múltiples clientes pueden tener conexiones abiertas a un mismo server simultáneamente y un cliente puede tener conexiones abiertas con múltiples servers simultáneamente.

Los clientes X usan el protocolo para enviar requerimientos al X server para crear y manipular ventanas, generar texto y gráficos, controlar el input producido por el usuario y para comunicarse con otros clientes. El X server debe multiplexar requerimientos desde los clientes al display, y demultiplexar entradas producidas por teclado y mouse hacia los clientes apropiados.

El X server encapsula el sistema de ventanas base. Provee los mecanismos y recursos fundamentales requeridos para implementar varias interfases de usuario. Todas las dependencias de los dispositivos están encapsuladas en el server; el protocolo de comunicación entre clientes y el server es independiente de los dispositivos. Ubicando todas las dependencias de los dispositivos en un extremo de la conexión de red, las aplicaciones son efectivamente independientes de los dispositivos. Obviamente, el server mismo está formado por dos capas de software, la de más alto nivel es independiente de los dispositivos y está arriba del código dependiente de los dispositivos.

Debido a que un 'round-trip' en la red es una operación costosa comparada con la ejecución de un requerimiento, el protocolo es asincrónico, y los datos viajan en ambas direcciones (del cliente al server y del server al cliente) simultánemente. Después de generar un requerimiento, un cliente típicamente no espera que el server ejecute el requerimiento antes de generar el próximo requerimiento. En lugar de esto, el cliente genera un conjunto de requerimientos que son recibidos por el server y ejecutados. El server no envía un 'acknowledge' ante la recepción de un requerimiento y, en la mayoría de los casos, no envía un 'acknowledge' despueés de ejecutar un requerimiento. (Esto es posible ya que la capa de transporte es confiable).

El protocolo es diseñado explícitamente para minimizar la necesidad de consultar información al sistema de ventanas. Los clientes no deben depender del server para obtener información que ya han suministrado. Además, los clientes usan requerimientos para registrar interés en varios eventos, y el server envía notificaciones de los eventos en forma asincrónica. La operación asincrónica es tal vez una de las diferencias más significantes entre X y otros sistemas de ventanas.

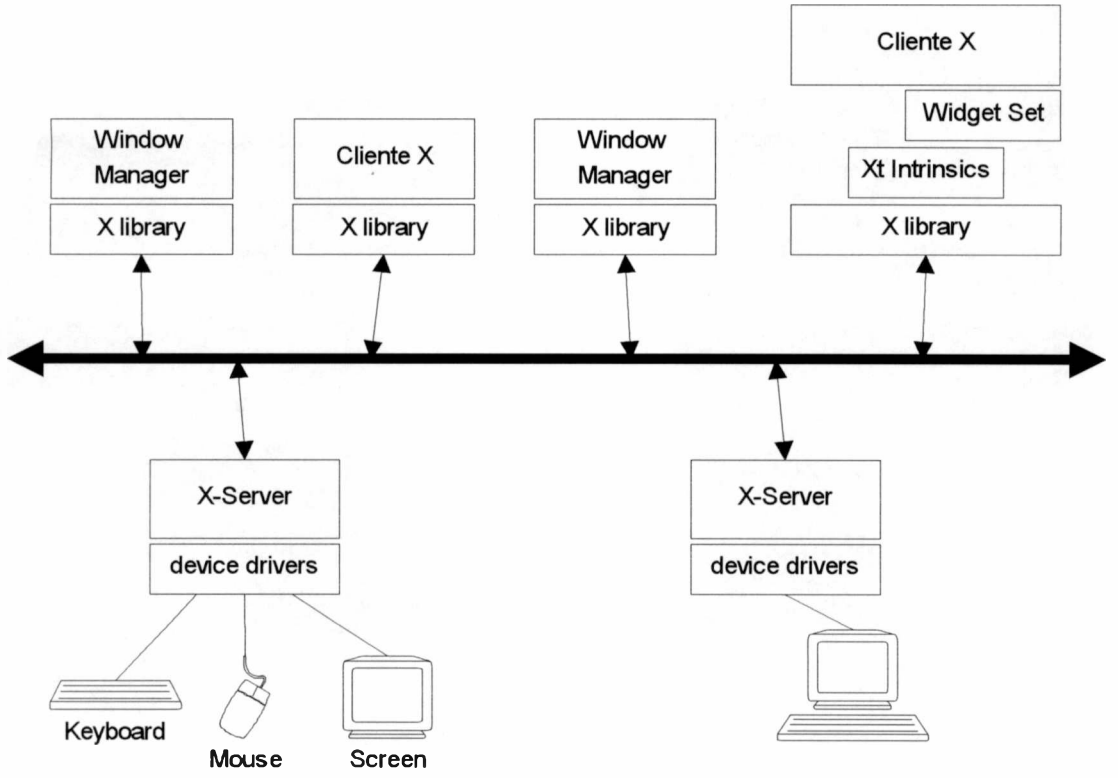
El protocolo X fue diseñado para que sea independiente del sistema operativo, lenguaje de programación y tipo hardware de procesamiento. De esta manera, un mismo display puede mostrar simultáneamente aplicaciones escritas en múltiples lenguajes de programación, bajo múltiples sistemas operativos sobre múltiples arquitecturas de hardware.

3. 1. 3. Interfase al protocolo X.

Aunque X está basado fundamentalmente por un protocolo de red, existe una librería de interfase a este protocolo. Esta librería provee una interfase procedural que abstrae los detalles de la codificación del protocolo y de las interacciones con la capa de transporte y maneja automáticamente el buffering de los requerimientos para que el transporte hacia el server sea eficiente. La librería también provee varias funciones auxiliares que no están directamente relacionadas con el protocolo pero que son fundamentales para construir aplicaciones. La interfase exacta para esta librería difiere para cada lenguaje de programación. Xlib es la librería para el lenguaje de programación C.

En X, muchas de las facilidades que están construídas dentro de otros sistemas de ventanas son provistas por librerías clientes. El protocolo y la librería X evitan imponer políticas de interfase de usuario tanto como se pueda y deben ser vista como un kit de desarrollo que provee un rico conjunto de mecanismos que pueden implementar una gran variedad de políticas de interfase de usuario; X no especifica menúes, scrool bars, y dialog boxes, o como debe responder una aplicación a la entrada del usuario. Toolkits (librerías de alto nivel que implementan características de interfase de usuario como menúes, botones, scrool bars, etc.), y UIMS ('user interface management systems') pueden ser implementados arriba de la librería X. Si bién la librería X provee los mecanismos de base, la idea es que las aplicaciones sean escritas usando estas facilidades de alto nivel en conjunción con las facilidades de la librería X, y no sólamente con esta librería.

Arquitectura del X Window System



3. 2. Conceptos de X.

3. 2. 1. Displays y Screens.

X es un sistema de ventanas para operar sobre un display gráfico de bitmap[1]. Una característica inusual es que un *display* está formado por un conjunto de dispositivos de entrada (teclado, mouse, etc.), y *una o más screens*. Múltiples screens pueden trabajar juntas, y se permite que el movimiento del mouse cruce los límites físicos de los screens.

3. 2. 2. Window Management.

Recordemos que una interfase de usuario está conceptualmente dividida en dos interfases, una *interfase de aplicación* que es *interna* a esta y una *interfase de administración* que es *externa* a la aplicación. Referente a la interfase de administración, los clientes X no deben controlar cosas tales como donde debe aparecer una ventana o cual es su tamaño. En lugar de esto, un cliente provee *hints* acerca del lugar y tamaño donde desearía que se displaye una ventana. La apariencia y el layout del screen y el estilo de interacción del usuario con el sistema es controlado por un único cliente X especial, llamado window manager.

Recordemos que el protocolo X no define una interfase de usuario externa, sino que provee mecanismos a partir de los cuales pueden construirse una gran variedad de interfases de usuario externa. Estos mecanismos están diseñados para que este único cliente pueda proveer esta interfase independiente de todos los otros clientes.

Un window manager puede forzar una *política de layout de ventanas* estricta si lo desea (por ejemplo, hacer un 'tile' del screen de manera que las ventanas no se solapen), así también como automáticamente proveer:

- Barras de títulos, bordes, y otras decoraciones de ventanas para cada aplicación.
- Íconos uniformes para las aplicaciones.
- Medios uniformes para mover y 'resize' ventanas.
- Una interfase uniforme para switchear el teclado entre las aplicaciones.
- El control de la entrada desde teclado o mouse es típicamente provisto por otro cliente especial llamado *input manager*, el cual, usualmente, es parte del window manager.

3. 2. 3. Xlib, la interfase 'C' al protocolo X.

3. 2. 3. 1. El protocolo X.

El protocolo X define cuatro tipos de PDUs: *requerimientos*, *respuestas*, *eventos* y *errores*.

Un *requerimiento* es generado por Xlib y enviado al server. Un requerimiento puede llevar una amplia variedad de información, como ser la especificación para dibujar una línea, cambiar el valor del color de una celda en un colormap, o consultar el tamaño actual de una ventana. La mayoría de la

[1] En gráficos de bitmap, cada punto de la pantalla (llamado *pixel*) se corresponde con uno o más bits en memoria. El término gráfico de bitmap es equivalente a gráfico mapeado en memoria.

rutinas de Xlib generan requerimientos. Las excepciones son rutinas que solo afectan estructuras de datos locales a Xlib y no afectan al server.

Una *respuesta* es enviada desde el server a Xlib como respuesta a ciertos requerimientos. No todos los requerimientos tienen respuesta, sólo aquellos que consultan información. Una rutina Xlib que requiere una respuesta es llamada requerimiento *'round-trip'*. Los requerimientos 'round-trip' tienen que ser minimizados en los clientes ya que bajan la performance cuando hay retardos en la red.

Un *evento* es enviado del server a Xlib y contiene información acerca de la acción de un dispositivo o información que es consecuencia de un requerimiento anterior. Los datos de los eventos son muy variados, debido a que son el único medio mediante el cual los clientes obtienen información. Los eventos son mantenidos en un cola en Xlib y son leídos de a uno por vez por los clientes. El rango de tipos de eventos que el server envía a un cliente es especificado por el cliente.

Un *error* indica al cliente que un requerimiento previo fue inválido. Xlib maneja los errores levemente diferente a los eventos; no pueden ser leídos vía las funciones que leen eventos, y son enviados a un manejador de errores de Xlib. El manejador de errores default puede ser reemplazado por una rutina específica a un cliente.

3. 2. 3. 2. Buffering.

Xlib hace buffering de requerimientos en lugar de enviarlos al server inmediatamente, de manera que el cliente puede seguir corriendo en lugar de esperar acceder a la red después de cada llamado a Xlib. Esto es posible debido a que la mayoría de los llamados a Xlib no requieren acción inmediata por parte del X server.

Xlib envía el buffer lleno de requerimientos al server bajo tres condiciones. La más común es cuando una aplicación llama a una rutina de Xlib para esperar por un evento, pero no existe ningún evento que haga matching en la cola de Xlib. Dado que, en este caso, la aplicación debe igualmente esperar por tal evento, tiene sentido hacer un flush del buffer de requerimientos. Segundo, los llamados a Xlib que obtienen información del X server requieren una respuesta inmediata, y por lo tanto, el buffer de requerimientos es enviado. Tercero, el cliente podría explícitamente hacer un flush del buffer en situaciones donde no espera eventos del usuario o ya no tiene requerimientos para enviar al server.

Xlib mantiene una cola de eventos para cada server al cual un cliente ha establecido una conexión. Siempre que llegue un evento del server, este es encolado hasta que el cliente lo lee de la cola.

El X server envía los eventos a Xlib inmediatamente después que se producen y tan pronto como lo permita la red (el server no encola o agrupa eventos, execpto bajo condiciones especiales).

3. 2. 3. 3. Recursos.

Los recursos básicos provistos por el X server son ventanas, fonts, pixmaps, cursores, colormaps, imágenes 'offscreen', contextos gráficos y otros. Los clientes requieren la creación de un recurso especificando ciertos parámetros (por ejemplo el nombre del font); el server almacena el recurso y retorna un identificador único para referenciarlo. El uso e interpretación de este identificador de recurso es independiente de la conexión de red. Cualquier cliente que conozca el identificador de un recurso, puede manipular el recurso libremente, aún si fue creado por otro cliente. Esto último es muy importante ya que permite que los windows managers sean escritos independiente de las aplicaciones; esto también permite que las aplicaciones compartan recursos. El tiempo máximo de

vida de un recurso está asociado a la conexión sobre la cual el recurso fue creado. Así, cuando un cliente termina, todos los recursos que ha creado son destruídos automáticamente.

Todas las operaciones (funciones de Xlib) sobre un recurso, explícitamente contienen el identificador del recurso como parámetro. De esta manera, una aplicación puede multiplexar el uso de muchas ventanas sobre una única conexión de red. Este multiplexado facilita que los clientes controlen el orden en el que se actualizen múltiples ventanas; ésto no sería posible si se usara un stream por ventana. De la misma manera, cada evento de entrada generado por el X server contiene el identificador de la ventana en la cual se produjo el evento. El multiplexado sobre un único stream permite que los clientes actúen sobre eventos de múltiples ventanas en el orden correcto; nuevamente, el uso de un stream por ventana no permitiría este orden.

El uso de identificadores de recursos permite reducir el tráfico en la red; en lugar de enviar por la red una estructura de datos, sólo se envía un entero que referencia a la estructura. Por ejemplo, un requerimiento para mostrar texto en una ventana consiste de un código de operación, los identificadores para la ventana y el font (parámetros de longitud fija) y el string de caracteres (parámetro de longitud variable).

3. 2. 3. 4. Propiedades y átomos.

Una propiedad es una colección de datos con nombre y con tipo. Las propiedades se asocian a ventanas y están disponibles a todos los clientes que corren bajo el mismo X server; proveen un mecanismo de comunicación de información entre clientes. El sistema de ventanas tiene un conjunto de propiedades predefinidas (por ejemplo, el nombre de una ventana), además los clientes pueden definir cualquier otra información arbitraria y asociarla a ventanas. Cada propiedad tiene un nombre el cual es un string. Para cada propiedad con nombre, se asocia un identificador único llamado átomo. Una propiedad tiene también un tipo. Estos tipos también se indican con átomos, con lo que se pueden definir nuevos tipos. Sólamente datos de un tipo pueden asociarse a una propiedad. Los clientes pueden almacenar y obtener propiedades asociadas a ventanas. Una aplicación obtiene el átomo de una propiedad llamando a la función XInternAtom(), a la que se le pasa como parámetro el nombre de la propiedad. A partir de este punto las aplicaciones referencian la propiedad usando el átomo.

Para las propiedades predefinidas se tienen átomos predefinidos. Estos átomos están disponibles vía constantes simbólicas. El protocolo no define ninguna semántica para estas propiedades, pero existen otros standards relacionados con X que especifican la semántica, como ser ICCCM.

3. 2. 4. Qué son las ventanas X.

Un X Server controla un screen de 'bitmapped'. Para poder ver y controlar distintas tareas al mismo tiempo, el screen puede ser dividido en áreas más pequeñas llamadas *ventanas*. Una ventana es un área rectangular que trabaja como un pequeño screen. Las ventanas en el screen pueden ser ubicarse de manera que estén todas visibles o de que se cubran entre sí en forma completa o parcial. Una ventana puede tener cualquier tamaño.

3. 2. 4. 1. Características de una ventana

1) Una ventana siempre tiene una ventana padre, la cual es asignada cuando ésta es creada. Cada ventana está contenida dentro de los límites de su ventana padre. La primer ventana, la única que no tiene padre, es llamada la ventana *root* que cubre el screen completo. La ventana root es creada por el X Server cuando éste arranca.

2) Cada ventana tiene su propio sistema de coordenadas.

3) Una ventana tiene una *configuración* que incluye lo siguiente:
   - Alto y ancho en pixels, sin incluir el borde.
   - Borde, el cual puede variar en ancho; cero hace que el borde sea invisible.
   - Posición (x,y) en el screen en pixels, medida desde el origen de la ventana padre (vértice superior izquierdo, dentro del borde) al vértice superior izquierdo de la ventana en cuestión, fuera del borde.
   - Debido a que varias ventanas pueden tener el mismo padre, una ventana debe tener un *stacking order* entre estas ventanas para determinar cual será visible si éstas se solapan.

   El ancho, alto y posición se denominan colectivamente *geometría*.

4) Una ventana tiene características referidas a *profundidad* (*depth*) y *visual*, las que juntas determinan su color. La profundidad es el número de bits que tiene cada pixel para representar el color (o escala de grises). El visual representa la manera en que los valores de pixel son trasladados para producir el color en el monitor.

5) Una ventana tiene una clase que puede ser `InputOutput` (pueden recibir E/ de dispositivos y pueden usarse para producir S/) o `InputOnly` (son sólo usadas para E/).

6) Una ventana tiene un conjunto de *atributos*. Los atributos de ventana controlan muchos aspectos relacionados con la apariencia y respuesta de la ventana.
   - Color y 'pattern' para el borde y el 'background' de la ventana.
   - Cuando se salva el contenido de la ventana cuando parte de ésta (o toda) se vuelve visible o *exposed* ?
   - Como se se reubica el contenido parcial de una ventana durante el 'resizing' ?
   - Que tipos de eventos son recibidos, y cual se descartan (no se propagan a ventanas ancestros) ?
   - Se debe permitir que la ventana se mueva o cambie de tamaño sin comunicar esto al window manager ?
   - Que colormap se utiliza para interpretar los valores de pixels dibujados en la ventana ?
   - Que cursor se debe utilizar cuando el puntero del mouse está en la ventana ?

Para terminar de comprender la escencia de las ventanas X, a continuación se describe uno de los conceptos más importantes del X Window System, el concepto de jerarquía de ventanas.

3. 2. 5. Jerarquía de ventanas.

El X server soporta una jerarquía arbitraria de ventanas rectangulares. Al tope se encuentra la ventana *root*, que cubre todo el screen. Las ventanas *top-level* de las aplicaciones son creadas como subventanas de la ventana root. Para una ventana dada, sus subventanas pueden estar apiladas en cualquier orden y solaparse arbitrariamente. Cuando la ventana V1 cubre parcial o completamente la ventana V2, decimos que V1 *oculta* a V2. Esta relación no está restringida a ventanas hijas de un mismo padre; si la ventana V1 oculta a la ventana V2, entonces V1 puede ocultar a subventanas de V2. Una ventana también oculta a su padre. Además si las ventanas V1 y V2 son hijas del mismo

padre y V1 oculta a V2, entonces las subventanas de V2 nunca ocultarán a V1 o a subventanas de V1. Una ventana no está restringida en tamaño o ubicación por los límites de su ventana padre, pero una ventana es siempre 'clipped' por su padre, es decir, las partes de una ventana que se extienden más allá de los límites de la ventana padre nunca se muestran y no ocultan a otras ventanas. Finalmente una ventana puede estar *mapeada* o *no mapeada*. Una ventana no mapeada nunca es visible en el screen; una ventana mapeada puede estar visible si todas sus ventanas ancestras están mapeadas.

Las salidas hechas a una ventana sin subventanas están restringidas a las partes visibles de la ventana; dibujar en tales ventanas nunca dibuja en ventanas que la ocultan. Las salidas realizadas a una ventana que contiene subventanas pueden ser realizada en dos modos: en modo *'clipped'* la salida está restringida por todas las ventanas que la ocultan (incluyendo subventanas), pero en modo *'draw-through'* la salida no es restringida por por subventanas. Por ejemplo, este último modo es usado sobre la ventana root para dibujar el contorno de una ventana cuando esta es movida o 'resized'.

El sistema de coordenadas esta definido con el eje X horizontal y el eje Y como vertical. Cada ventana tiene su propio sistema de coordenadas, con el origen en la esquina superior izquierda de la ventana. El sistema de coordenadas es discreto.

Las ventanas descriptas hasta el momento son *ventanas opacas*. X también provee *ventanas transparentes*. Una ventana transparente es invisible en el screen y nunca oculta la salida hecha a, o la visibilidad de, otras ventanas. La salida realizada a ventanas transparentes es restringida a esa ventana, pero se dibuja sobre la ventana padre. Así, para salidas, una ventana transparente es simplemente un rectángulo que restringe la salida a una ventana (que es siempre la padre).

El X server está diseñado explícitamente para que las ventanas sean un recurso muy económico. El objetivo del X Window System fue hacer razonable el uso de ventanas para cosas tales como items de menúes, botones, las flechas de scrolling de un scrollbar, el scrollbar mismo, etc.

3. 2. 6. Qué son los eventos.

Mover el mouse o presionar una tecla causa que ocurra un evento de entrada. Un evento es un paquete de información generado por el X server cuando ocurren ciertas acciones, y es encolado por el cliente para posterior procesamiento. Los eventos encolados pueden ser ser leídos en cualquier momento y en cualquier orden, pero usualmente son leídos y procesados en el orden que están en la cola que se corresponde con el orden en que se produjeron.

Los eventos tienen distintos usos:

- Indicar la entrada desde los dispositivos y control de la interfase de usuario(mover el mouse, presionar una tecla o un botón del mouse, el cruce del mouse por los límites de una ventana, etc.).
- Efectos laterales de operaciones sobre ventanas (cuando una ventana se vuelve visible después de haber sido ocultada, cuando una ventana pasa a estar mapeada o no mapeada, etc.).
- Permitir la comunicación entre clientes (por ejemplo, cuando el window manager hacer el 'reparent' de una ventana top-level de una aplicación para agregarle el frame a esta, el cliente recibe un evento; cuando un cliente cambia una propiedad de una ventana, etc.)

Existen tres pasos importantes que deben cumplir las aplicaciones con respecto al manejo de eventos:

- Seleccionar los eventos deseados para cada ventana.
- Mapear las ventanas.

- Proveer un loop de eventos en el cual se leen los eventos de la cola de eventos a medida que estos se producen, y se procesan cada uno de estos.

Como ejemplo se presenta la estructura del evento Expose, el cual se produce cuando la ventana completa o una parte previamente no visible de la ventana se vuelve visible:

```
typedef struct {
   int type;                        /* tipo del evento; para este evento
                                    vale la constante Expose */
   unsigned long serial;            /* # del último requerimiento
                                    procesado por el server */
   Bool send_event;                 /* TRUE si se originó vía el
                                    requerimiento SendEvent */
   Display * display;               /* Display desde el cual proviene el
                                    evento */
   Window window;                   /* ventana que recibe el evento */
   int x, y;
   int width, height;
   int count;                       /* cantidad de eventos Expose que
                                    restan por enviarse; este campo
                                    sirve para hacer más eficiente el
                                    refresco de la ventana.*/
} XExposeEvent;
```

3. 2. 7. Input.

Una ventana se dice que *contiene* el mouse si el 'hot-stop' del cursor del mouse está dentro de la porción visible de la ventana o de alguna de sus subventanas. El mouse se dice que *está* en una ventana si la ventana, pero ninguna de sus subventanas, contiene el mouse.

La entrada está asociada a ventanas. La entrada producida en una ventana dada está controlada por un único cliente, que no necesita ser el cliente que creó la ventana. Los eventos se clasifican en varios tipos, y el cliente que los controla selecciona los tipos de eventos sobre los cuales tiene interés (vía la función XSelectInput pasándole como parámetro una *máscara de eventos* en la que se especifican los tipos de eventos en los que está interesado). El X server envía al cliente sólo aquellos eventos que hagan matching con los seleccionados por el cliente. Cuando se produce un evento de entrada en una ventana y el cliente que la controla no ha seleccionado este tipo de eventos, el server *propaga* el evento a la ventana ancestro más próxima para la cual algún cliente ha seleccionado este tipo de eventos, y envía el evento a este último cliente. La ventana en la cual se genera el evento se llama *ventana fuente*, y la ventana que tiene el tipo de evento seleccionado se llama *ventana evento*.

3. 2. 7. 1. El teclado.

El teclado está siempre asociado ('attached') a una ventana (típicamente la ventana root a una ventana top-level); llamamos a esta ventana la ventana *foco*. Puede usarse un requerimiento (usualmente usado por el input manager) para asociar el teclado a alguna ventana. La ventana que recibe la entrada de teclado depende de la posición del mouse de la ventana foco. Si el cursor del mouse está en alguna ventana descendiente de la ventana foco, aquella ventana recibe la entrada. Si el cursor del mouse no está en una ventana descendiente de la ventana foco, entonces la ventana foco recibe la entrada.

3. 2. 7. 2. El mouse.

Una aplicación puede recibir, selectivamente, eventos de presionado o liberación de cada botón. Una aplicación puede también decidir seleccionar eventos de movimiento de mouse, ya sea siempre que el mouse esté en la ventana o sólo cuando ciertos botones están presionados. La aplicación no puede controlar la granularidad de reporte de estos eventos, y tampoco se garantiza una granularidad mínima. Las implementaciones típicas de servers compactan eventos de movimiento de mouse con el propósito de minimizar el overhead del sistema.

Aún con el compactado, un server puede generar un número considerable de estos eventos. Si una aplicación intenta responder en tiempo real a cada evento, puede perder fácilmente la posición actual del cursor del mouse. En lugar de esto, muchas aplicaciones tratan a los eventos de movimiento de mouse como 'hints' o pistas; para esto el cliente debe especificar que desea recibir hints de movimiento de mouse. De esta manera el X server es libre de enviar sólo un evento de movimiento de mouse hasta que cambie el estado de las teclas o de los botones. Cuando se recibe el movimiento de mouse, el evento simplemente se desecha, y el cliente explícitamente consulta al server la posición actual del mouse (vía un función de la interfase a X). Mientras espera por la respuesta del server, pueden ser recibidos nuevos eventos de movimiento de mouse, los cuales serán posteriormente descartados.

Los clientes pueden también recibir un evento cada vez que el mouse entra a o sale de una ventana. Esto es particularmente útil para implementar menúes.

3. 2. 8. 'Exposures'.

Dado que es posible dibujar en ventanas ocultas, debe tratarse el tema de la *exposición* o *'exposure'* de las ventanas. Cuando parte de una ventana oculta nuevamente se vuelve visible, quién es el responsable de de restaurar el contenido de la ventana ?. En X, la responsabilidad es del cliente. Cuando una región de una ventana se vuelve expuesta, el server envía un evento en forma asincrónica al cliente especificando la ventana y la región de esta que ha sido expuesta; el resto depende de la aplicación.

El cliente es responsable porque X no impone ninguna estructura o relación entre las operaciónes gráficas que el cliente ha requerido. Si quisiéramos que sea el server el responsable de restaurar el contenido de una ventana existen dos mecanismos básicos: que mantenga una lista de requerimientos gráficos generados o mantener imágenes 'off-screen'. Ninguna de las aproximaciones basadas en el server es aceptable por razones claras.

El argumento es que las aplicaciones pueden tomar ventajas de sus propias estructuras de información para facilitar el rápido redibujado.

Delegar al cliente la responsabilidad de refrescar las ventanas es consecuencia de la filosofía de administración de ventanas. La justificación es que las aplicaciones no pueden ser escritas con tamaños de ventanas fijos, sino que deben operar correctamente con cualquier tamaño de ventana a medida que el tamaño de estas cambien. El asunto es que la mayoría de las aplicaciones deben tener el código para refrescar completamente las ventanas. Este no es un argumento para que el server nunca deba mantener el contenido de las ventanas, sólo que no se requiere que el server lo mantenga. Igualmente las ventanas X tienen varios atributos para indicar al server cuando y como deben ser mantenidos los contenidos de estas.

3. 2. 9. Introducción a los gráficos en X

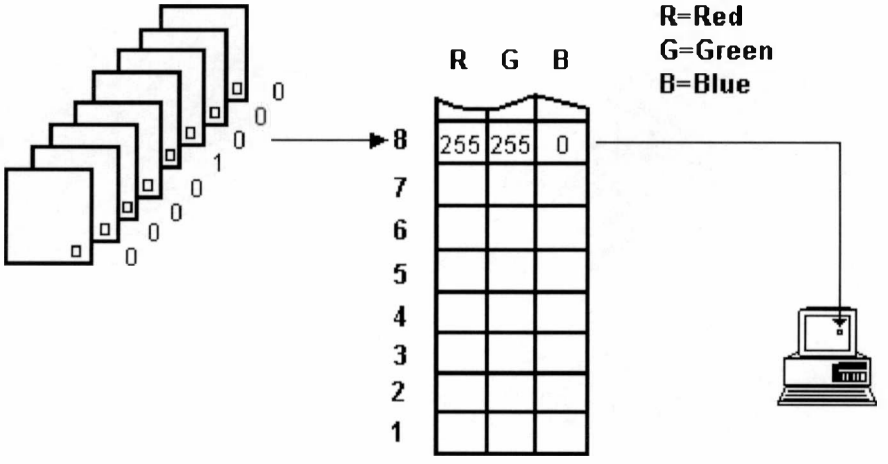
3. 2. 9. 1. Pixels y Colores

X Window System soporta display gráficos de bitmaps. En un display de 2 colores un pixel es representado por un bit, en otros tipos de display el pixel puede estar representado por multiples bits. El estado de estos bits indican el color del pixel en forma indirecta, ese valor es un indice en una tabla llamada *colormap*. En un display de color, un pixel consiste en un indice al colormap, el cual tiene un valor compuesto por valores de rojo, verde y azul (**R**ed, **G**reen, **B**lue) que indica la intensidad de cada color.

Cada miembro del colormap es llamado *colorcell*. En un screen monocromo el colormap solo tiene dos colorcells.

Cada ventana X puede potencialmente especificar un colormap diferente, un colormap es un atributo de ventana.

Mapeo de un valor de pixel a un color vía un 'colormap'



3. 2. 9. 2. Pixels y Planos

El numero de bits por pixel también se refiere al numero de *planos* en el display grafico. X11 soporta hasta 32 planos.

Todas las operaciones gráficas son realizadas con valores de pixels antes de ser trasladados a RGB. El valor de pixel de *origen* y el *viejo* valor de pixel de *destino* especificados en un requerimiento gráfico son combinados de acuerdo a la *máscara de plano*, *máscara de clipping* y *función lógica* para llegar al *nuevo* valor de pixel *destino*. La máscara de plano, máscara de clipping y función lógica son aspectos de una estructura llamada **graphics context (GC)** descripta más adelante.

3. 2. 9. 3. Pixmaps y Drawables

No solo en una ventana se pueden hacer requerimientos graficos, tambien en los *pixmap*. Un pixmap es un parte de memoria fuera del screen (*off-screen memory*) en el X-Server. Las ventanas y los pixmaps se denominan *drawables*.

Un pixmap es un 'array' de valores de pixels. Tiene una profundidad como tiene una ventana. No tiene, sin embargo, una posición relativa a alguna otra ventana o pixmap, y no tiene atributos de ventana como ser un colormap; todas estas características afectan al pixmap cuando éste es copiado en la ventana. Un pixmap es visible sólo cuando es copiado en una ventana.

Las ventanas tienen la desventaja que, cuando no están visibles, dibujar sobre ellas no tiene sentido, dado que sus contenidos no son mantenidos cuando son cubiertas por otras ventanas o pasan a estar *unmapped* (a menos que la característica de *backing store* esté disponible, lo que permite justamente mantener los contenidos de las ventanas para que el X Server lo redibuje). Un pixmap, que representa un área en el screen, reside en memoria y puede ser escrito en cualquier momento. Desafortunadamente, un pixmap debe ser copiado a una ventana visible para que éste pueda ser visto.

Para que un pixmap pueda ser copiado a una ventana, el pixmap debe tener la misma profundidad que la ventana, esa operacion se realiza con la funcion *XCopyArea*. Una vez que el pixmap es copiado el colormap asociado a la ventana es usado para trasladar los valores de pixels desde el pixmap a colores. Si se quiere copiar solo un plano del pixmap se usa la funcion *XCopyPlane*.

Un pixmap de profundidad 1 se conoce como *bitmap*, aunque no existe un tipo de recurso llamado Bitmap. Un bitmap es un 'array' de bits 2-D usados para varios propósitos incluyendo definiciones de cursores y fonts.

3. 2. 9. 4. Graphics Contexts (GCs) y dibujo de gráficos.

El X Window System tiene funciones para dibujar pixels, lineas, rectangulos, poligonos, arcos, texto, etc. Las funciones que dibujan graficos se llaman genéricamente *primitivas graficas*. En X, estas funciones no contienen toda la informacion necesaria para realizar un gráfico en paritcular. El recurso del server que tiene esa informacion restante (como ser los colores, el ancho de una linea, patrones de rellenado etc.) se llama *graphic context* (GC). El ID (identificador de recurso del X Server) de un GC es especificado como un argumento de la rutina de dibujo.

El Graphic Context debe ser creado por el cliente antes de algun requerimiento grafico. El graphic context es un recurso X almacenado en el X Server (como cualquier otro recurso), de manera que cuando se llama a una funcion grafica no es necesario pasar toda la informaicón que contiene, solo se pasa el ID del GC. Con esto se logra reducir el trafico sobre la conexión X entre el Cliente y el Server. Todo la información de un GC aplica a todos los gráficos dibujados usando este GC.

3. 2. 9. 5. Tiles y Stipples

Cuando un pixmap es usado como *pattern* (patron) en un area, como ser para el background de una ventana o en un graphic context, los pixmaps son denominados *tiles* o *stipples*.

Un tile es un pixmap con la misma profundiad del drawable sobre el cual se quiere aplicar el 'patterning'. Puede aplicarce un tile sobre las áreas en las que se usan la rutinas gráficas especificando ciertos valores en el GC. También puede aplicarse un tile en el background y el borde de un ventana especificando un pixmap en los atributos de la ventana.

Un stipple es un pixmap de profundidad 1. Un stipple es usado en conjunción con un pixel de foreground y algunas veces también con un pixel de background para aplicar un pattern a un área de manera similar a un tile. Existen 2 maneras de hacer 'stippling' las que se especifican en el GC.

3. 2. 10. Administración de las preferencias del usuario: el Administrador de Recursos

Las aplicaciones deben poder ser personalizadas por el usuario. Toda aplicación debe proveer opciones de línea de comando para los elementos configurables más importantes de la aplicación, como colores del background de ventanas, colores de foreground para dibujar, geometría de la aplicación, fonts, etc. Además los usuarios podrían querer personalizar muchos aspectos del programa y necesitar una forma conveniente de establecer defaults. Para este propósito existe el *resource manager* o *administrador de recursos*.

Un recurso es una opción de programa configurable (completamente diferente de un recurso que lleva el X server como una ventana). Las especificaciones de recursos están usualmente guardadas en *archivos de recursos* y en *propiedades* en el X server. Cuando todos los archivos de recursos y propiedades son combinadas por Xlib en una única base de datos, lo que resulta es una *base de datos de recursos*. Las rutinas y estructuras provistas por Xlib para administrar preferencias de usuarios se denominan en conjunto como el *resource manager*.

Citemos un ejemplo muy conocido: es posible personalizar la ventana de login presentada por el entorno gráfico de una workstation Unix la cual permite ingresar un nombre de usuario y password para establecer el 'login' con la máquina. Es posible configurar la posición, ancho y alto de la ventana en el screen; el color y el font usado en el nombre de usuario que ingresa el usuario; el string usado para el prompt del nombre de usuario, cuyo default es *"Login:"*; el string usado para el prompt de la password, cuyo default es *"Password:"*; el color y el font para ambos prompts; el string, color y font que se utiliza en el mensaje que se muestra cuando falla la autenticación, cuyo string default es *"Login incorrect"*, etc.

3. 3. ICCCM (Protocolo de comunicación entre clientes X).

3. 3. 1. Introducción.

Unos de los objetivos de diseño más importantes de X fue especificar mecanismos y no políticas. Como resultado, un cliente que conversa con el server usando el protocolo X puede operar correctamente en forma aislada, pero podría no coexistir adecuadamente con otros clientes que compartan el mismo server.

Para que los clientes coexistan apropiadamente, X provee mecanismos de comunicación que aseguran, por ejemplo, una integración adecuada con el window manager y que se compartan ordenadamente los recursos del sistema. Además la comunicación permite a los clientes el intercambio de datos.

Como se mencionó anteriormente, la comunicación ocurre vía propiedades que se asocian a las ventanas.

3. 3. 2. Comunicación 'peer-to-peer' vía selecciones.

Las selecciones son el mecanismo básico que define X para el intercambio de información entre clientes. Pueden existir un número arbitrario de selecciones (cada una identificada por un átomo) y estas son globales al server. Cada selección es poseída por un cliente y está asociada a una ventana.

La comunicación se establece entre un *owner* (dueño) y un *requestor* (requeridor). El dueño posee los datos que representan el valor de su selección, y el requeridor los recibe. Un requeridor que desee obtener el valor de una selección debe proveer lo siguiente:

- Ventana del owner.
- Nombre de la selección.
- Nombre de la propiedad.
- Ventana del requeridor
- El átomo que representa el tipo de datos requerido.

Si el dueño posee actualmente la selección, este recibe el evento SelectionRequest con la información anteriormente especificada, y se espera que haga lo siguiente:

- Convertir el contenido de la selección al tipo requerido.
- Poner este dato en la propiedad en la ventana del requisidor.
- Enviar el evento SelectionNotify al requisidor para informarle que la propiedad está disponible; este evento es enviado vía la función XSendEvent provista por Xlib la que genera el requerimiento SendEvent.

Si bién nuetra solución no hace uso de este mecanismo, consideramos importante presentar este tema por la importancia que tiene en X.

3. 3. 3. Comunicación entre clientes y el Window Manager.

Para que un window manager pueda administrar el espacio en el screen, los clientes que están siendo administrados deben adherirse a ciertas convenciones. Estas convenciones especifican cosas que deben hacer los clientes, cosas que deberían o que pueden hacer si lo desean, y cosas que no deben hacer.

En general, se definen convenciones que aplican a todos los paradigmas de administración de ventanas. Aquellos clientes que son diseñados para correr con un window manager particular, pueden definir protocolos privados que se agregan a estas convenciones, pero deben tener en cuenta que los usuarios podrían elegir cualquier otro window manager.

Un principio fundamental de estas convenciones es que todo cliente no debería saber o preocuparse cual window manager está corriendo o, si alguno está corriendo. La elección del window manager depende del usuario o del administrador del sistema y no del cliente.

3. 3. 3. 1. Acciones de los clientes.

En general, el objetivo del diseño de X es que los clientes deben, tanto como puedan, hacer exactamente lo que harían en la ausencia de un window manager, excepto en lo siguiente:

- Proveer hints al window manager acerca de los recursos que desean usar.
- Cooperar con el window manager aceptando los recursos que se les asignan, aún si no son exactamente los que se pidieron.
- Estar preparados a que la asignación de recursos cambie en cualquier momento.

Los clientes crean una o más ventanas que son hijas de una o más de las ventanas root. Todas estas ventanas que se crean son las ventanas top-level. Son éstas las ventanas controladas por el window manager, y son en éstas ventanas en las que los clientes setean las propiedades como hints al window manager.

3. 3. 3. 1. 1. Propiedades de los clientes.

Las cosas más básicas que se espera que hagan los clientes es darle valor a ciertas propiedades de ventana de manera que el window manager tenga información a partir de la cual basar sus decisiones.

El hecho que los window managers requieran información de los clientes que están administrando, que estos implementen políticas de administración particulares y que pueden no estar corriendo, surge el concepto de *hint*. Un hint es una sugerencia que se hace al window manager acerca de una preferencia de la aplicación asignando valor a una propiedad. El window manager debe tratar de cumplirlas tanto como pueda, pero no es obligación que las cumpla. Por lo tanto, la aplicación no debe depender de que sus hints sean cumplidos; debe ser capaz de poder operar cuando alguno de sus hints son ignorados o negados.

Una vez que el cliente haya creado una o más ventanas top-level, pero antes de mapearlas, debe darle valor a propiedades en estas ventanas para informar al window manager del comportamiento que desea el cliente.

El window manager examina el contenido de estas propiedades cuando la ventana hace la transición desde el estado *withdrawn* (no está visible la ventana top-level ni la ventana ícono) y monitorea algunas propiedades ante cambios mientras la ventana este en el estado *normal* (la ventana está visible) o *iconizada* (el significado de este estado depende del window manager).

Algunas propiedades son: WM_NAME es un string que desea el cliente que muestre el window manager en asociación con la ventana (ej.: en la barra de título de esta); WM_NORMAL_HINTS especifica las preferencias de geometría de la ventana, como ser, estado y posición inicial, incrementos de tamaño y razón ancho/alto que el window manager debe respetar cuando el usuario cambia el tamaño de la ventana; WM_HINTS que especifica el *modelo de input* adoptado por el cliente (referido a la asignación de la ventana foco), *estado de la ventana* cuando se mapea, ventana y pixmap para el ícono, posición incial del ícono.

3. 3. 3. 1. 2. Propiedades del window manager.

Las propiedades anteriores son aquellas que los clientes X mantienen en sus ventanas top-level. El window manager setea propiedades en las ventanas top-level de los clientes y en la ventana root.

Por ejemplo, la propiedad WM_STATE puesta en las ventanas (top-level) de los clientes sirve para comunicación entre el session manager y el window manager.

3. 3. 3. 1. 3. Configuración de ventanas.

Los clientes pueden resizear y reposicionar sus ventanas top-level usando el requerimiento *ConfigureWindow*. Los atributos de la ventana que pueden alterarse con este requerimiento son la posición, ancho y alto, y la posición en el stack. ***El sistema de coordenadas en que se expresa la posición es el de la ventana root, independientemente que haya ocurrido algún 'reparenting'.***

Los requerimientos de configuración son interpretados por el window manager de la misma manera que cuando se mapea la ventana. Los clientes deben tener en cuenta que el window manager puede no allocar el tamaño y posición requerida y deben estar preparados para aceptar cualquier tamaño y posición.

El requerimiento ConfigureWindow enviado por un cliente, es recibido por el X server pero es ignorado, en su lugar se envía el evento *ConfigureRequest* al window manager, el cual puede decidir no cambiar el tamaño y posición de la ventana, mover la ventana sin cambiar el tamaño o sólamente cambiar el tamaño. Cualquiera sea la acción tomada por el window manager, éste envía al cliente el evento *ConfigureNotify* vía el requerimiento *SendEvent* a través del X server (la función de Xlib que genera este requerimiento es *XSendEvent()*).

Las coordenadas en eventos ConfigureNotify reales (aquellos enviados por el X server) están en el espacio de la ventana padre; en eventos sintéticos (aquellos enviados por el window manager), están en el espaci de la ventana root.

3. 3. 3. 2. Notificaciones a clientes de las acciones del Window Manager.

El window manager realiza operaciones sobre recursos de los clientes, principalmente en las ventanas top-level de estos. Los clientes no deben negarse a estas decisiones, pero pueden decidir recibir notificación de las operacione del window manager.

3. 3. 3. 2. 1. Reparenting.

Los clientes deben considerar que algunos window managers harán un *reparent* de las ventanas top-level de los clientes que no tengan el atributo override-redirect en true, por lo tanto, una ventana que es creada como hija de la ventana root, se mostrará como hija de alguna otra ventana que creó el window manager.

El propósito básico del reparenting es la decoración (frame de resizing, barra de título, controles de maximizado y minimizado) de las ventanas top-level de los clientes.

Los clientes que quieren ser notificados cuando son reparented pueden seleccionar los eventos asociados a la máscara *StructureNotify* sobre sus ventanas top-level. Ellos recibirán el evento *ReparetNotify* si y cuando se haga el reparenting.

Si el window manager reparents una ventana de un cliente, esta ventana se agregará al save-set de la ventana padre. Esto significa que la ventana reparented no será destruída si termina el window manager, y será mapeada si estaba no mapeada.

Cuando el window manager deja el control sobre ventanas top-level de los clientes, este pondrá a las ventanas top-level como hijas de la ventana root.

3. 3. 3. 2. 2. Redirección de operaciones.

Los clientes deben tener en cuenta que el window manager puede indicar al X server que los requerimientos de los clientes sean interceptados y redirigidos a ellos. Los requerimientos que se redirigen no son ejecutados por el X server, sino que resultan en que el X server envíe eventos al window manager, el cual puede decidir hacer nada, alterar los argumentos, o ejecutar el requerimiento en nombre de los clientes.

El hecho que un requerimiento sea redirigido al window manager significa que el cliente no debe suponer que el requerimiento se realiza una vez que es recibido por el X server o que este se realiza. Por ejemplo, lo siguiente es incorrecto ya que el requerimiento MapWindow puede ser interceptado y que la salida de PolyLine sea hecha a una ventana no mapeada.

```
MapWindow V1
PolySegment V1 GC <punto> <punto> ...
```

El cliente debe esperar el evento Expose antes de dibujar en la ventana.

El siguiente ejemplo, incorrectamente asume que el requerimiento ConfigureWindow es realizado con los argumentos que se le pasan:

```
ConfigureWindow width=N height=M
<se hace una salida asumiendo que la ventana es de N por M>
```

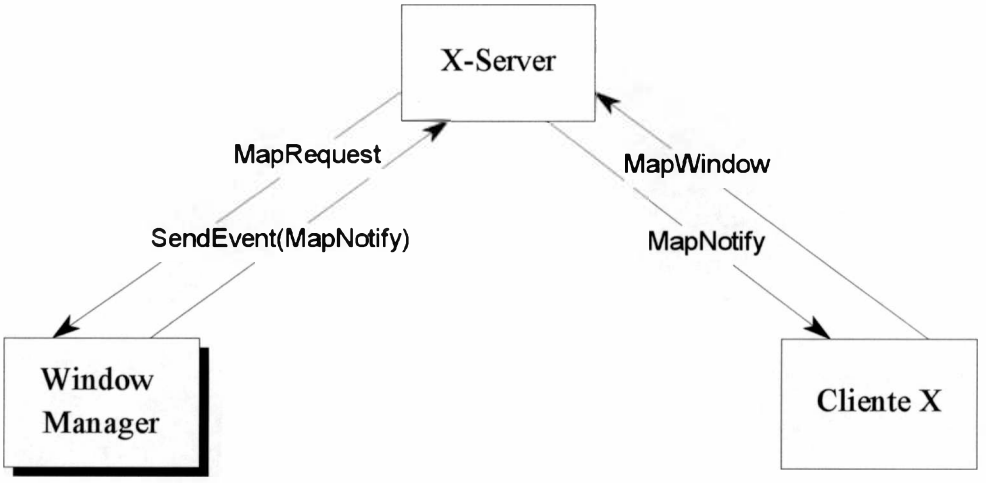
El cliente debe esperar el evento ConfigureNotify.

Los requerimientos que pueden ser redirigidos son:

- MapWindow.
- ConfigureWindow.
- CirculateWindow.

Aquellos clientes que mapeen ventanas top-level con el atributo override-redirect en true, están asumiendo la responsabilidad del estado del desktop. En efecto, esta clase de clientes están actuando como window managers temporarios. Esto no está aconsejado ya que producirá conflictos con la política de interfase de usuario que intenta mantener el window manager.

Interacción entre el Window Manager, el X-Server y un Cliente X normal



3. 4. Window Management (Windows Managers).

Un window manager es un cliente X que controla el layout de las ventanas en el screen, respondiendo a requerimientos de usuarios de mover, cambiar de tamaño, cambiar el orden de stacking e iconificar ventanas. El window manager tiene definida una política de layout de ventanas. Es importante notar que estas ventanas son las ventanas top-level de los clientes, y que la configuración de las ventanas descendientes de estas está bajo el control de la aplicación.

Las características y rutinas provistas por Xlib para permitir que un window manager controle el layout de las ventanas son las siguientes:

- *Substructure redirection*: permite que el window manager intercepte requerimientos que cambién el layout del screen.
- *Repareting*: permite al window manager construir un frame o decoración al rededor de cada ventana top-level.
- El *save-set* que asegura que las ventanas que el window manager iconifica o 'reparents' son retornadas a sus estados originales si el window manager termina inesperadamente[2].

A continuación se describirá sólamente el concepto de substructure redirection.

3. 4. 1. Substructure redirection.

El window manager asegura su política de layout de ventanas vía *substructure redirection*. Esto consiste en setear la máscara de eventos *SubstructureRedirectMask* a una ventana (generalmente la ventana root y las ventanas que creó el window manager para hacer el reparenting) lo que produce que cualquier requerimiento de cambio de estructura en alguna de las subventanas directas de la ventana con dicha máscara no se realice y que el X server envíe un evento al cliente que pidio redirección (generalemente el window manager) indicando el cambio de configuración requerido por el cliente. Esto es así, siempre que el cliente no haya seteado el atributo *override-redirect* en True en la ventana que se reconfigura. En función a los cambios pedidos que se indican en este evento, el window manager decide aceptarlos, cambiarlos o rechazarlos, informándole al cliente enviándole el evento *ConfigureNotify*.

Sólo un cliente puede haber seleccionado SubstructureRedirectMask sobre una ventana.

El término *estructura* se refiere a la posición, tamaño, stacking order y mapeo de una ventana. El término *subestructura* son estos mismos parámetros pero aplican a las ventanas hijas. *Redirección* significa que el requerimiento no se hace, y en su lugar se envía un evento al cliente que pidió redirección.

3. 4. 1. 1. Selección de eventos relacionados con la estructura de una ventana:

Máscara de evento	Evento
ResizeRedirectMask	ResizeRequest
StructureNotifyMask	CirculateNotify ConfigureNotify DestroyNotify

[2] Una causa de la terminación anormal es un bug en el window manager.

3. 5. Interface Gráfica (X Graphics)

3. 5. 1. Graphics Context

Las rutinas que dibujan gráficos se llamas primitivas gráficas. Una primitiva gráfica no contiene toda la información requerida para dibujar un gráfico en particular. Existe un recurso llamado *graphics context* que especifica varias variables que aplican a cada primitiva gráfica. El graphics context controla la apariencia de todas los graficos que se dibujen en una ventana excepto el borde y el background de la misma (estos datos son atributos de la ventana). Lo que se dibuja dentro de un pixmap tambien es controlado por el graphics context usado en la/las primitivas gráficas usadas.

Existen dos razones relacionadas con la performance para usar graphics contexts. La primera es reducir el trafico entre Xlib y el server ya que la informacion del graphics context es mantenida en el server y sólo es necesario enviarla una vez antes del primer requerimeinto gráfico. Todas las demas primitivas graficas que especifiquen el mismo GC usaran los mismos valores. Cuando se necesite cambiar unos pocos valores del GC, sólo es necesario enviar esos pocos valores y no todo el GC.

Segundo, uno puede crear varios GCs y simplemente especificar el GC que se usará en cada primitiva gráfica . Esto tiene grandes beneficios de performance si el X Server puede hacer 'catching' de múltiples GC.

Uno debe pensar en una primitiva gráfica como una manera de especificar que pixels en un *rectángulo origen* seran dibujados, y en que lugar debe ubicarse este rectángulo en el *drawable de destino*. El rectángulo origen no se corresponde a alguna area del screen (excepto en las funciones *XCopyArea()*, *XCopyPlane()*). El rectángulo origen es una entidad interna usada como una etapa intermedia en el proceso de dibujado. El rectángulo origen es un bitmap generado por las primitivas gráficas con bits en 1 para los pixels que se van a dibujar. El rectángulo origen es transformado cierto número de veces antes que aparezca en el screen. Una primitiva gráfica no escribe en el screen como si fuera una hoja blanca de papel. Recordemos que se tienen múltiples bits por pixel (múltiples planos); se puede usar un máscara de planos para especificar que planos serán afectados por la primitiva gráfica. Además, los bits del screen ya tiene valores de pixels; aunque es posible simplemente sobreescribir estos valores con nuevos valores, existe un gran número de *funciones lógicas* que pueden aplicarse para combinar los valores de pixels generados por la primitiva gráfica con lo que ya existen en pantalla. Es también posible especificar una máscara de 'clipping' que limite el efecto de un requerimiento gráfico a un área particular o a pixels particulares en la ventana. Y esto es sólo parte de las posibilidades que pueden controlarse vía un GC.

Las siguiente lista son algunas de las caracteristicas controladas por el graphics context. Algunas de estas caracteristicas son especificadas por más de un miembro del graphics context:

- Pueden combinarse los valores de pixel origen y destino para computar los nuevos valores de pixels destino de acuerdo a una *función lógica*.
- En todas las primitivas, el foreground determina el valor de pixel de los bits en 1 del rectángulo fuente. En algunas pocas primitivas, el background especifica el valor de pixel para los bits en 0 del origen.
- Las características de línea determinan el ancho, 'dot pattern' y tipo de finalización de las líneas.
- Pueden seleccionarse los eventios `GraphicsExpose` y `NoExpose` para indicar cuando cuestiones de visibilidad afectan a los requerimientos `XCopyArea` y `XCopyPlane`.

	GravityNotify MapNotify ReparentNotify UnmapNotify
SubstructureNotifyMask	CirculateNotify ConfigureNotify CreateNotify DestroyNotify GravityNotify MapNotify ReparentNotify UnmapNotify
SubstructureRedirectMask	CirculateRequest ConfigureRequest MapRequest

Luego de establecer una conexión con el X-Server (vía XOpenDisplay()), el window manager selecciona los eventos que tiene interés sobre la ventana root vía la función XSelectInput():

```
XSelectInput(dpy, RootWindow (dpy, scrnum),
      ColormapChangeMask | EnterWindowMask | PropertyChangeMask |
      SubstructureRedirectMask | KeyPressMask |
      ButtonPressMask | ButtonReleaseMask);
```

3. 4. 1. 2. Eventos seleccionados por las máscaras anteriores:

Máscara de evento	**Evento**
ColormapChangeMask	ColormapNotify
EnterWindowMask	EnterNotify
PropertyChangeMap	PropertyNotify
SubstructureRedirectMask	*CirculateRequest* *ConfigureRequest* *MapRequest*
KeyPressMask	KeyPress
ButtonPressMask	ButtonPress
ButtonReleaseMask	ButtonRelease

3. 4. 2. Modelo de Input.

Los window managers también controlan cual es la ventana que recibe la entrada de teclado seteando la ventana foco de teclado a una ventana top-level de la aplicación. La distribución de eventos según el foco de teclado actual es manejado por el X Server. Algunos window managers, como twm, usan el modelo de entrada de teclado *pointer-following*; la ventana que contiene el puntero del mouse es la que recibe la entrada de teclado. Seteando el foco de teclado una vez a la ventana root implementa este modelo. Otros window managers usan el modelo *click-to-type*, como ser Macintosh, Windows o *mwm* (Motif Window Manager) el cual requiere que el usuario clickee la ventana a la cual debe ir la entrada de teclado. El window manager setea el foco de teclado a la ventana que clickea el usuario.

- Las características de rellenado determinan el 'tile' y estilo de rellenado para lineas y áreas.
- Puede especificarse el font usado en lo requerimientos que escriben texto.
- Las subventanas pueden o no ocultar los gráficos dibujados en la ventana padre.

Creación de un GC.

Para crear un graphics context se usa la funcion *XCreateGC()* con 4 parametros: el display (conexión con el X Server), drawable (ID de una ventana o un pixmap. Este ID indica realmente el screen con el que se asociará el GC y las profundidades de ventana con las que se podrá usar el GC. Un GC puede ser usado sobre cualquier ventana o pixmap de la misma profundidad y screen del drawable especificado.), valores (especificados en la estructura XGCValues), y máscara de valores (que miembros son considerados de la estructura XGCValues).

Miembros de la estructura XGCValues.

Estructura de datos para asignar valores a un Graphics Context

```
typedef struct {
   int function;            /* función lógica */
   unsigned long            /* máscara de planos */
plane_mask;
   unsigned long            /* valor pixel de foreground */
foreground;
   unsigned long            /* valor pixel de background */
background;
   int line_width;
   int line_style;          /* que secciones de una línea son dibujadas
                            y con que valores de pixel*/
   int cap_style;           /* como se dibujan los puntos finales de
                            una línea */
   int join_style;          /* como se dibujan las esquinas de una
                            línea */
   int fill_style;          /* controla si los gráficos son dibujados
                            con color sólido, con un 'tile', o con uno
                            de los 2 estilos de stipple */
   int fill_rule;           /* define cuales son los pixels que se
                            dibujan;    usado    en    requerimientos
                            XFillPolygon*/
   int arc_mode;            /* usado en requerimientos XFillArc y
                            XFillArcs */
   Pixmap tile;             /* para operaciones de 'tiling' */
   Pixmap stipple;          /* Pixmap de profundidad 1 */
   int ts_x_origin;         /* offset para operaciones de 'tiling' o
                            'stippling' */
   int ts_y_origin;
   Font font;               /* Font para operaciones de texto */
   int subwindow_mode;      /* controla si las subventanas ocultan a la
                            ventana cuando se dibuja sobre la ventana
                            padre*/
   Bool graphics_exposure; /* Usado en los requerimientos XCopyArea y
                            XCopyPlane (para copiar datos de un
                            drawable a otro). Es posible que ciertas
```

```
                        porciones de la region origen estén
                        ocultas, 'unmapped' o no disponibles, en
                        cuyo caso puede ser deseable generar un
                        evento para avisar al cliente que no pueden
                        copiarse una o más areas en la ventana
                        destino y por lo tanto deberán redibujarse
                        de otra manera */
  int clip_x_origin;    /* origen para operaciones de clipping */
  int clip_y_origin;
  Pixmap clip_mask;     /* Bitmap para operaciones de 'clipping' */
  int dash_offset;      /* Información de 'patterning' para el
                        dibujo de líneas */
  char dashes;
} XGCValues;
```

El GC provee una manera flexible de controlar exactamente que pixels y que planos serán afectados por los requerimientos gráficos y cómo serán usados los valores de pixel origen y destino para computar los nuevos valores de pixel de destino. Como se mencionó anteriormente, en función del bitmap generado por la primitiva gráfica (el rectángulo origen) se aplican los valores de pixels de los miembros *foreground* y *background* del GC. Posteriormente se aplica la función lógica entre estos valores de pixel y los valores de pixel existentes (el rectángulo destino), la máscara de planos y la máscara de 'clipping'.

Los pixels origen y destino son combinados aplicando una función lógica en los bits correspondientes a cada pixel. La máscara de planos restringe la operación a un subconjunto de planos, de manera de poder excluir del cómputo un subconjunto de bits. La máscara de 'clipping' restringe la operación a un subconjunto de pixels del display.

Los pixels de origen, destino, la máscara de 'clipping', y la máscara de planos son combinadas usando el algoritmo que se describe a continuación para obtener los nuevos valores de pixel de destino. Para cada bit en cada pixel del drawable de destino, si el bit correspondiente en la máscara de 'clipping' está en 1, la siguiente expresión define si este bit estará en 1:

```
((origen FUNC destino) AND mascara_planos ) OR (destino AND (NOT
mascara_planos))
```

Es decir, si el bit de la máscara de planos `mascara_planos` está en 1, los bits de origen `origen` y de destino `destino` son combinados usando la función lógica `FUNC`. Si el bit de la máscara de planos `mascara_planos` no está en 1, el bit existente en destino `destino` no se cambia.

3. 5. 2. Gráficos.

En esta parte se describen las primitivas gráficas para dibujar lineas, rectangulos, y arcos; uso de bitmaps y dibujo de texto. Se pueden dibujar pixels con las funciones *XDrawPoint* o varios pixels con la funcion *XDrawPoints* indicando el array de pixels a dibujar. La función *XDrawLine* es similar al *XDrawPoint* pero requiere 2 puntos para dibujar una linea entre ellos; tambien con la funcion *XDrawLines* se pueden dibujar varias lineas entre puntos consecutivos. Existen otras funciones como *XDrawSegments* (no necesariamente líneas consecutivas), *XDrawRectangle* y *XDrawArc*.

Existen funciones que permiten pintar distintos tipos de figuras graficas como *XFillArc, XFillArcs, XFillPolygon, XFillRectangle*, y *XFillRectangles*.

Los bitmaps, tiles, y stipples son todos tipos de pixmaps. Las aplicaciones necesitan a menudo

crear pixmaps para patrones de íconos, cursores y tiles. Xlib provee funciones para manipular pixmaps.

3. 5. 3. Fonts y Texto.

En X Window un *font* es un conjunto de bitmaps que representan texto, un conjunto de formas de cursor, y quizás algún otro conjunto de formas usado para otro propósito. Los fonts son usados para incrementar la eficiencia siempre que sea necesario usar en forma repetitiva un conjunto de patrones pequeños y del mismo tamaño.

El X Server carga los fonts siempre que un cliente requiera un nuevo font (por ej. vía la función *XLoadFont* la cual retorna el ID del font). El Server puede hacer 'catching' de los fonts para mayor eficiencia en el acceso. Los fonts son globales a lo largo de todas las screen que controla el Server.

La administración de fonts en el X Window System (cuestiones tales como carga, acceso y liberación de fonts) tiene asociada muchas implicancias de performance que hacen que se implemente en forma separada al X Server en un proceso comunmente conocido como **Font Server**.

Toda función X que dibuja texto tiene dos versiones, una que maneja fonts de 1 byte (8-bits) y la otra de 2 bytes (16-bits). La diferencia entre estos es que un font de 1 sólo byte esta limitado a 256 caracteres, mientras que un font de 2 caracteres puede soportar hasta 65536 caracteres. Se necesita un gran número de caracteres para los lenguajes Orientales.

Para poder usar un font es necesario que el X Server lo cargue. Si uno o más clientes X usan el mismo font, ellos comparten la misma copia en el Server, pero ambos requieren que el font sea cargado, aunque sólo sea para obtener el ID. Los fonts disponibles estan almacenados en una base de datos (mantenida por el Server) que es accedida por funciones tales como *XListFonts* y *XListFontsWithInfo*. La función *XSetFont* sirve para asociar un font con el GC. Los clientes deben liberar los fonts que usan, para esto Xlib provee la función *XFreeFont*.

3. 5. 4. Regiones

En X una region es un conjunto de pixels en el screen. Usualmente una región es un área rectangular, varias áreas rectangulares adyacentes o que se solapan, o un polígono. Las regiones se usan principalmente para asignar una máscara de 'clipping' a un GC.

La forma más común de asignar una región al área de 'clipping' es combinar los rectángulos de múltiples eventos Expose contiguos y de una misma ventana. Esto mejora la performance en la mayoría de las situaciones.

A continuación se presenta el código C de una aproximación para el tratamiento del evento Expose, el cual sirve para ilustrar diversas funciones de manipulación de regiones. En general se evita redibujar el contenido completo de la ventana por razones obvias. Una aproximación buena es trabajar con la máscara de 'clipping' del GC para permitir que el X Server elimine los requerimientos innecesarios (aquellos que tocan áreas de la ventana fuera de ésta área).

```
Region region;
XRectangle rect;
   ...
region = XCreateRegion();
while (1) {
   XNextEvent(display, &report);
   switch (report.type) {
```

```
   case Expose:
      /* Asignar el rectángulo al área expuesta definida por
report.xexpose.x,
      report.xexpose.y, report.xexpose.width, report.xexpose.height */

      /* Unir el rectángulo a la region */
      XUnionRectWithRegion(&rect, region, region);

      if (report.xexpose.count == 0) {
         /* Asignar el área de 'clipping' al GC con el que se redibujará
*/
         XSetRegion(display, gc, region);
         /* Limpiar la región */
         XDestroyRegion(region);

         /* Redibujar en la ventana */
      }
      break;
```

3. 5. 5. Imágenes

Xlib provee una estructura capaz de almacenar toda la información correspondiente a un área del screen o pixmap. La principal diferencia entre una imágen y un pixmap es que una imágen es una estructura del lado del cliente, de manera que su contenido puede ser manipulado directamente por el cliente, en lugar de únicamente vía requerimeintos X. Xlib provee las rutinas *XGetImage* y *XPutImage* que usan el protocolo X para transferir el contenido de una ventana o pixmap a una estructura de imágen y para escribir el contenido de una estructura de imágen a una ventana o pixmap.

3. 5. 6. Cursores

Cada ventana en X Window tiene asociado un cursor definido en sus atributos usando la funcion *XDefineCursor*. Siempre que el puntero esta en una ventana visible, el cursor es seteado con el cursor de dicha ventana. Si el cursor de la ventana no esta definido toma el cursor asociado con la ventana padre. En X el cursor consiste de la forma del cursor, una mascara, el color de background y foreground, y el 'hotspot'.

El bitmap del cursor define la forma del cursor, el bitmap de máscara define que pixels sobre el screen que seran modificados por el cursor. Los pixels en el bitmap del cursor en 1 seran pintados con el color de foreground y los que están 0 seran pintados con el color del background. El hotspot define el punto del cursor que se indicara al producirse un evento de mouse.

Existen funciones para la manipulacion de los cursores. La funcion *XCreateFontCursor* crea un cursor usando el font de cursores estandar de X. La funcion *XDefineCursor* asocia el cursor con la ventana. La funcion *XUndefineCursor* realiza la accion contaria a XDefineCursor, de manera que la ventana usará el cursor de la ventana padre. *XCreateGlyphCursor* permite hacer lo mismo que se hace con los cursores estandard pero usando caracteres de fonts. *XCreatePixmapCursor* permite crear un cursor a partir de un pixmap que define la máscara, un pixmap de máscara y valores de pixels de foreground y background.

3. 5. 7. Colores

Cada ventana X tiene asociada un mapa de colores (colormap), que provee un nivel de indirección entre valores de pixels y colores mostrados en el screen. Xlib provee funciones que se pueden usar para manipular un colormap. El protocolo X define colores usando valores en el espacio de colores RGB. El espacio de colores RGB es dependiente del dispositivo; hacer el 'rendering' de un valor RGB en diferentes displays generalmente resulta en en colores diferentes. Xlib también provee mecanismos para que los clientes puedan especificar colores usando espacios de colores independientes de los dispositivos para obtener resultados consistentes en distintos displays. Xlib soporta espacios de colores independientes de los dispositivos derivables del espacio CIE XYZ. Estos espacios incluyen CIE XYZ, xyY, L*u*v, y L*a*b como también el espacio TekHVC[3].

Diferencias en el hardware de display: Visuals.

Un visual describe las capacidades de color que tiene el display, como ser el numero de colorcells en el colormap. Un visual describe las características de un colormap virtual para ser usado en un screen en particular. Debe especificarse un visual cuando se crea una ventana o un colormap, y el visual utilizado en la creación de la ventana debe ser el mismo que se utilizó para crear el colormap que se asocia a la ventana.

Al crear una ventana se puede usar el visual de la ventana root conocido como el 'default visual', el cual naturalmente describe tambien al 'default colormap'. Si se quiere crear una ventana con el default visual y el default colormap se llama a la funcion *XCreateSimpleWindow*.

Existen 6 clases de visual correspondientes a diferentes tipos de hardware de display los cuales son
`StaticGray`, `GrayScale`, `StaticColor`, `PseudoColor`, `TrueColor` y `DirectColor`.

Las clases de visuals distinguen entre color o monocromo, si el colormap es 'read/write' o 'read-only', o si un valor de pixel provee un índice único al colormap o se descompone en índices separados para los valores RGB.

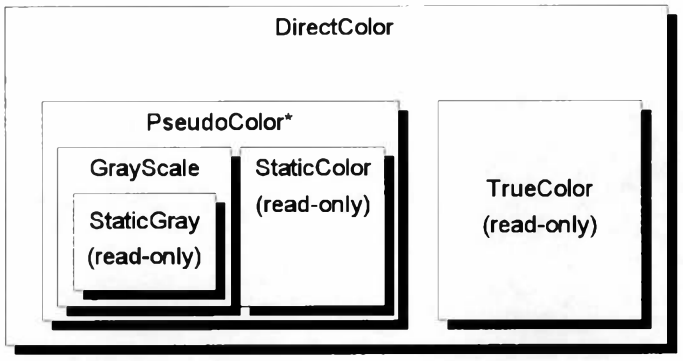
Clases de visuals

Tipo de colormap	Read/Write	Read-Only
Monocromo/Gris	`GrayScale`	`StaticGray`
Unico índice para RGB	`PseudoColor`	`StaticColor`
Indices separados para RGB	`DirectColor`	`TrueColor`

A continuación se representa esquemáticamente las clases de visuals que pueden ser soportadas (al menos en teoría) por cada tipo de hardware de screen. Por ejemplo, el gráfico muestra que el screen que puede soportar `DirectColor` puede soportar cualquiera de las otras 6 clases de visuals.

[3] Se mencionan los espacios de colores a modo ilustrativo ya que no se serán descriptos.

Jerarquía de clases de 'visuals'



*** PseudoColor puede también simular a DirectColor, pero sólo co un pequeño número de colorcells.**

Un screen puede tener disponible o soportar varios visuals. El X server especifica cuales clases de visual estan disponibles y define un visual por default. Los visuals 'read/write' permiten a una aplicacion agregar colores o instalar su propio colormap.

Muchos displays tiene solo un colormap por hardware. X implementa multiples colormaps virtuales. El window manager es el que se encarga de hacer el 'swapping' entre los colormaps virtuales y el colormap por hardware. Cuando el server es inicializado crea un colormap virtual llamado 'default colormap' con el que inicializa y instala el colormap por hardware.

El colormap por hardware es compartido por todos los clientes X. Si un cliente modifica el colormap default o cambia el colormap por uno propio, algunas aplicaciones mostraran colores no deseados dado que los valores de pixels que ahora usan apuntan a un colormap de otro cliente. Cada ventana X tiene un colormap especificado en una propiedad; la politica adoptada por el window manager es instalar el colormap asociado con la ventana que tiene el foco. La administración de colormaps forma parte de ICCCM (convenciones de comunicación entre clientes X).

Los displays de color de alta performance, por ejemplo 24 o 32 planos (24 o 32 bits por pixels), tienen múltiples colormaps por hardware, así pueden estar instalados al mismo tiempo múltiples colormaps virtuales.

Para obtener informacion sobre los tipos de visuals soportados en un screen en particular se puede usar la funcion *XGetVisualInfo*, esta funcion retorna una lista de estructuras de tipo `XVisualInfo` que a continuación se describe:

```
typedef struct
{
  Visual *visual;
  VisualID *visualid;
  int screen;
  unsigned int depth;
  int class;
  unsigned long red_mask;
  unsigned long green_mask;
  unsigned long blue_mask;
  int colormap_size;
  int bits_per_rgb;
}XVisualInfo;
```

3. 6. Xt Intrinsics y Widgets Sets.

3. 6. 1. Intrinsics y Widgets.

El propósito de Xt Intrinsics es proveer una aproximación simplificada a la programación de interfases gráficas de usuario teniendo como base un sistema de ventanas de red, específicamente el X Window System. Sin embargo, para adherir a la filosofía de X de *mecanismos* y no *políticas*, Xt Intrinsics no provee un conjunto fijo de componentes con un look and feel predefinido. En su lugar, provee un mecanismo general para producir componentes de interfase de usuario reusables de forma tal que estos componentes puedan ser construídos por los programadores de aplicaciones o por *programadores de objetos* de más bajo nivel para coleccionarlos en librerías y luego poder ser usados por las aplicaciones.

El corazón de Xt Intrinsics es una librería en C que usa las funciones provistas por Xlib. Xt provee rutinas para crear y usar componentes de interfase de usuario llamados *widgets*. Una aplicación típica usa Xt más una librería de widgets predefinidos (un *widget set*) como es Athena[4].
Este widget set incluye abstracciones como menúes, dialog boxes, scrollbars, botones, etc.

Xt provee un estilo de programación orientado a objetos, donde éstos objetos son los widgets. Sin embargo, como las librerías de Xt están escritas en C, un lenguaje que no provee soporte para OOP, Xt depende en convenciones de programación y en disciplinas que debe mantener el programador.

Un widget es una combinación de una ventana X más su semántica de display e input, y contiene información de estado. Algunos widgets muestran información (por ejemplo, texto o gráficos), y otros son simples contenedores de otros widgets (por ejemplo un menú). Algunos widgets son de sólo salida y no reaccionan a la entrada de mouse o teclado, y otros cambian su apariencia en respuesta a la entrada y pueden invocar funciones que les asociaron las aplicaciones.
Un widget set define *clases* de widgets. Todo widget pertenece a exactamente una *clase*, para la cual se reserva espacio y se inicializa en forma estática. Cada vez que se crea un widget, se crea una *instancia* de alguna de las clases.

Una clase widget define ciertas características fijas comunes a todas las intancias de la clase (*class record*), y ciertas características que dependen de cada instancia (*instance record*).

Una clase widget contiene atributos que son fijos y atributos configurables. Cada atributo configurable es un *recurso*. Los atributos pueden *heredarse* de otras clases de widgets más básicas; las clases widgets están organizadas en un jerarquía de herencia simple. La *jerarquía de clases* es completamente diferente de la relación padre-hijo existente entre instancias de widgets que se crean en una aplicación. La *jerarquía de instancias* define que widgets contiene otro widget en pantalla.

[4] Athena es el widget set provisto por el MIT junto con la distribución del X Window System. Dos widget sets comerciales fácilmente disponibles y muy completos son Motif de OSF y OPEN LOOK Intrinsics Toolkit (OLIT) de AT&T. Además estos Toolkits proveen sus propios window managers cuyo look and feel es consistente con el provisto por el Toolkit.

3. 6. 1. 1. Configuración de widgets vía recursos.

Un recurso es un campo del widget con la correspondiente entrada de recurso en la lista de recursos del del widget (*XtResourceList resources*) o de alguno de sus superclases.

Esto significa que el campo es seteable vía XtCreateWidget, o después de la creación usando XtSetValues; algunos son valores default especificados por el widget.

Aún antes de esto, a momentos de arranque de la aplicación, parte de Xlib llamado el *resource manager* lee valores de configuración definidos por el usuario y/o el programador de archivos ASCII, y Xt usa esta información para configurar los widgets de la aplicación.

La colección de recursos (pares nombre/valor) existentes en los distintos archivos de recursos se conoce como *base de datos de recursos*.

3. 6. 1. 2. Características de los widgets.

Existe cierta independencia entre un widget y la aplicación. Xt hace el dispatching de eventos a un widget, el cual realiza las acciones definidas en la clase, sin intervención de la aplicación. Por ejemplo, un widget se redibuja automáticamente cuando se hace visible después de haber estado oculto por otros widgets. Los widget también controlan las consecuencias de recibir cambios de sus recursos por parte de la aplicación (XtSetValues).

El propósito de los widgets es permitir que el usuario controle la aplicación. Por lo tanto, los widgets tienen la habilidad de invocar ciertas secciones de código deseadas por la aplicación. Una de las maneras para que un widget invoque código de la aplicación es registrando funciones con Xt. Xt invocará estas funciones como respuesta a cierta ocurrencia en el widget.

Existen tres mecanismos básicos usados para linkear widgets con funciones de la aplicación: callbacks, acciones y manejadores de eventos.

Esta arquitectura provista por Xt Intrinsics permite mantener en forma separada el código que implementa la interfase de usuario del código de la aplicación. Esto es una ventaja ya que permite modificar cualquiera de las dos partes sin afectar a la otra. Estas dos partes de código tienen que comunicarse y lo hacen vía los tres mecanismos mencionados anteriormente y que se describirán posteriormente.

Las clases widget base provistas por Xt Intrinsics son Core, Composite, Constraint y Shell.

La clase widget *Core* contiene la definición de campos comunes a todos los widgets. Todas las clases widgets son subclases de la clase Core.

La clase *Composite* es subclase de Core. El propósito de los widgets Composite es que sean contenedores de otros widgets. En general estos widgets manejan la posición (y posiblemente el tamaño) de los widgets *hijos* que contienen.

La clase *Constraint* es una subclase de la clase Composite. Los widgets de esta clase mantienen datos de estado adicional para cada widget hijo; por ejemplo, restricciones definidas por el cliente asociadas con la geometría de estos.

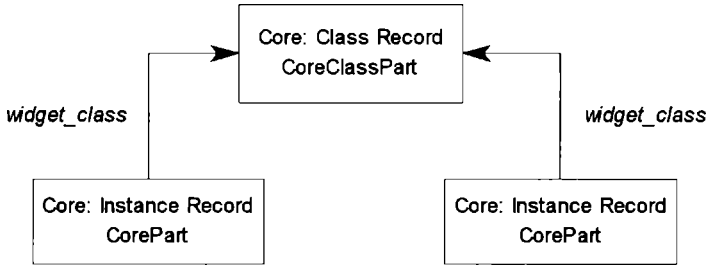
En cualquier aplicación Xt Intrinsics, se crea un widget de la clase *Shell* cuando se invoca a la función de inicialización de Xt. Este widget es usado como padre de todos los otros widget de la aplicación (con excepción de los popups, los cuales reciben sus propios widget Shell transientes cómo

padres), e incluye una funcionalidad especial que permite interactuar con el window manager. Este widget es invisible, ya que es cubierto por el widget principal de la aplicación (generalmente un widget de la clase Composite), que tiene el mismo tamaño.

La clase widget Core está definida por dos estructuras de datos, una estructura de clase llamada *Class Record* que contiene datos de clase y punteros a *métodos*, y una estructura de instancia llamada *Instance Record* que contiene datos propios de la instancia. Todas las clases e instancias de widget heredan estas estructuras.

El instance record de un widget tiene el campo *widget_class* que apunta a su estructura de clase, que contiene información que es constante para todos los widgets de esa clase. Los métodos disponibles en la estructura de clase son invocados por Xt Intrinsics cuando ocurren eventos o cuando una aplicación llama a funciones de Xt Intrinsics.

Arquitectura de clase de un widget



CoreClassPart y CorePart son nombres de estructuras 'C'.
widget_class es un atributo del Instance Record.

3. 6. 1. 3. Herencia.

Muchos lenguajes orientados a objetos proveen herencia como un constructor del lenguaje. Sin embargo, Xt Intrinsics, está escrito en el lenguaje C, el que no soporta directamente la programación orientada a objetos. En Xt Intrinsics, la herencia es implementada incluyendo los componentes de la class record e instance record de cada una de las superclases widget en la class record e instance record de la nueva clase widget.

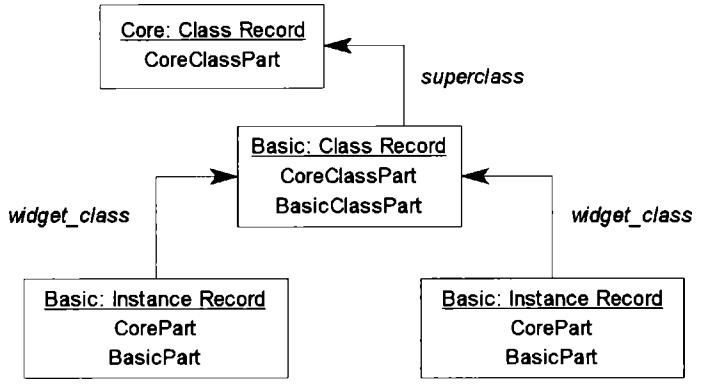
Xt Intrinsics provee también un mecanismo para heredar los métodos definidos por una superclase. Esto es hecho de dos maneras. El primer mecanismo es *chaining* o encadenamiento. Cuando un método es encadenado, los Intrinsics invocan primero los métodos definidos por cada una de las superclases del widget, antes de invocar el método del widget. Un ejemplo es el método Initialize() de la clase widget Core.

Xt Intrinsics provee también un mecanismo para heredar métodos que no se encadenan. Esto es hecho usuando símbolos especiales para especificar los métodos en la class record del widget. Cada símbolo es definido por la superclase que agrega el método a la class record del widget. Por ejemplo, la clase widget Core provee el símbolo XtInheritExpose. Los simbolos definidos por una clase, pueden ser usados por cualquier subclase.

Cuando una clase widget especifica algún símbolo en el 'class record', Xt Intrinsics copia el correspondiente método usado por la superclase en la estructura de clase a momento de inicialización

dinámica de la clase (además de la inicialización estática de la class record, algunas clases deben inicializarse dinámicamente cuando se crea el primer widget perteneciente a la clase).

Herencia desde la clase widget Core



CoreClassPart, BasicClassPart, CorePart y BasicPart son nombres de estructuras 'C'.
widget_class, y *superclass* son atributos.

3. 6. 2. Instanciación de los Widgets.

Una jerarquía de instancias de widgets constituye un *arbol de widgets*. El widget de la clase Shell retornado por XtAppCreateShell es la raíz del arbol de instancias de widgets. Un widget con uno o más hijos constituyen nodos internos de este árbol, y widgets sin hijos son las hojas del árbol de widgets. A excepción de hijos *pop-up*, este árbol de instancias de widgets define el árbol de ventanas X asociadas.

Los widgets pueden ser *compuestos* (instancias de la clase widget Composite) o *primitivos*. Ambas clases de widgets pueden contener hijos, pero Xt Intrinsics provee un conjunto de mecanismos de administración para la construcción e interacción entre widgets compuestos, sus hijos y otros clientes.

Un árbol de widgets es manipulado por varias funciones de widgets provistas por los Intrinsics. Por ejemplo, XtRealizeWidget recorre el árbol hacia abajo y 'realizes' en forma recursiva todos los widgets pop-up y los hijos de los widgets compuestos. La misma idea aplica a la función XtDestroyWidget.

3. 6. 2. 1. Inicialización de Xt Intrinsics.

Antes que una aplicación invoque a cualquier función, debe inicializar a Xt Intrinsics usando la función XtToolkitInitialize, que inicializa la parte interna de los Intrinsics, XtCreateApplicationContext, que inicializa el estado por aplicación, XtDisplayInitialize o XtOpenDisplay que inicializa el estado por display, XtAppCreateShell que crea la raíz del árbol de instancias de widgets.

Xt Intrinsics soporta la noción de *contexto de aplicación*. Un contexto de aplicación permite que múltiples aplicaciones lógicas o instancias de aplicación cohexistan en un único espacio de direcciones. Cada contexto corresponde a una aplicación lógica que tiene su propio loop de eventos y conexión con el X server. Este uso está limitado a sistemas operativos que soportan múltiples threads

dentro de un proceso. Además una instancia de aplicación podría requerir conexiones con varios displays. Desde el punto de vista de la aplicación, estas múltiples conexiones son tratadas como una única unidad para propósitos de dispatching de eventos.

El componente principal de un contexto de aplicación es una lista de X Displays de la aplicación. Xt Intrinsics maneja las conexiones con todos los displays de un contexto de aplicación en forma simultánea, administrando eventos con una política de round-robin.

Toda aplicación debe tener al menos un contexto de aplicación.

Para destruir un contexto de aplicación y cerrar las conexiones con todos los displays se usa la función XtDestroyApplicationContext.

Para abrir un X Display, inicializarlo y agregarlo al contexto de aplicación, se usa la función XtOpenDisplay. XtOpenDisplay invoca a la función XOpenDisplay, entre otras.

3. 6. 2. 2. Carga desde la base de datos de recursos.

La función XtDisplayInitialize crea una base de datos de recursos (Xlib provee el tipo base de datos de recursos XrmDatabase) la cual se asocia al screen default del display combinando las siguientes fuentes: línea de comandos de la aplicación, archivo de recursos del ambiente del usuario (archivo especificado por la variable de entorno XENVIRONMENT), especificaciones de recursos de la propiedad SCREEN_RESOURCES (de la ventana root) del X server, especificaciones de recursos de la propiedad RESOURCE_MANAGER (de la ventana root) del X server, archivo de preferencias del usuario y el archivo de recursos específicos de la aplicación.

De la combinación de los recursos de las fuentes anteriores, resulta una *base de datos de recursos de un screen*.

3. 6. 2. 3. Creación de Widgets.

La creación de instancias de widgets es un proceso de tres fases. Los widgets son allocados e inicializados con los recursos y son opcionalmente agregados al conjunto de administración del widget padre. Los widgets crean ventanas X, las que luego son mapeadas.

Para comenzar la primera fase, la aplicación invoca XtCreateWidget para todos sus widgets y los agrega a los respectivos conjuntos de administración de los padres. llamando a la función XtManageChild. Durante esta fase, no se notifica a los widgets padres de que ha cambiado su conjunto de widgets a administrar.

Luego que se crearon todos los widgets, la aplicación debe llamar a XtRealizeWidget con el widget top-level para realizar las fases dos y tres.

XtRealizeWidget recorre recursivamente en forma botton-up el árbol de widgets notificando a los widgets compuestos de los hijos que tienen que administrar invocando el procedimiento especificado en el atributo de la class record change_managed. Esta notificación implica establecer la geometría de los widgets hijos y posiblemente una negociación de esta. Un padre puede forzar que algunos de sus hijos cambien de tamaño y posición o hacer requerimientos de geometría a su propio padre para acomodar mejor a sus hijos.

De esta manera, en estas dos etapas no se crean ventanas X ya que es muy probable que tengan que moverse después de ser creadas. Esto evita tener que enviar requerimientos innecesarios al X server.

Finalmente, esta función comienza la tercer etapa haciendo un recorrido top-down del árbol de widgets, creando una ventana X para cada widget vía el procedimiento realize y finalmente mapeando los widgets que son administrados.

3. 6. 2. 4. Destrucción de Widgets.

Xt Intrinsics provee soporte para destruir todos los hijos pop-up del widget que está siendo destruído y destruir todos los hijos de widgets compuestos, remover el widget de su padre, invocar a las callbacks que han sido registradas para ser invocadas cuando el widget es destruído, minimizar el número de cosas que un widget debe deallocar cuando se destruye y minimizar el número de invocaciones a XDestroyWindow (función de Xlib para destruir una ventana X) cuando se destruye un árbol de widgets.

3. 6. 2. 5. Terminación de una aplicación.

Todas las aplicaciones basadas en Xt Intrinsics deben terminar invocando la función XtDestroyApplicationContext y luego salir invocando la correspondiente system call provista por el sistema operativo.

3. 6. 3. Composite Widgets.

Los widgets compuestos (widgets cuyas clases son subclases de la clase Composite) *administran* a sus hijos, lo que significa que son responsable de:

- Determinar el 'layout' (administración de la geometría) del subconjunto de widgets hijos que administra (es decir, los hijos que son displayables).
- Liberar la memoria usada por todos los hijos cuando el widget Composite es destruído. Cuando un widget Composite es destruído, primero se destruyen todos sus hijos.
- 'Map' y 'unmap' un subconjunto de los hijos que administra. Normalmente, un widget está mapeado cuando el widget es administrado y no mapeado cuando no está siendo administrado.

3. 6. 4. Shell Widgets.

La clase de widgets Shell es una subclase de Composite, pero sólo puede tener un hijo.

Los widgets de la clase Shell son contenedores de los widgets top-level de las aplicaciones, para permitir que estas se comuniquen con el window manager.

La clase Shell ha sido diseñada para ser tan invisible como se pueda; los clientes deben crear widgets de esta clase, pero nunca deben preocuparse acerca de su tamaño.

Un widget Shell negocia la geometría de las ventanas top-level de una aplicación con el window manger. Si un widget Shell es resizeado por el window manager, el widget Shell también resizea automáticamente el widget que contiene. Similarmente, si el widget hijo del shell necesita cambiar el tamaño, puede hacer un requerimiento de geometría al shell, y el shell negocia el cambio de tamaño con el window manager. Los widget hijos nunca deben cambiar el tamaño de sus shells.

Además un widget Shell setea las propiedades requeridas por el window manager, y en general maneja el protocolo con el window manager sin que las aplicaciones tengan que hacerlo. Esto es importante ya que cualquier cliente X escrito arriba de Xt Intrinsics *adhiere* a las convenciones definidas en ICCCM.

Existen cuatro clases de shells (subclases de la clase Shell) que son públicas (disponibles para las aplicaciones):

- *OverrideShell*: Usados para ventanas que bypassean completamente al window manager, por ejemplo menúes pop-up.
- *TransientShell*: Un widget de esta clase tendrá la ventana X asociada con la propiedad WM_TRANSIENT_SHELL seteada. El efecto de esta propiedad depende del window manager, pero en general es una indicación a éste que debe decorar la ventana mínimamente. Esta propiedad es usada en aquellas ventanas de una aplicación que son temporarias. Este clase podría ser usada para menúes o dialog boxes.
- *TopLevelShell*: Usada para ventanas top-level normales, por ejemplo toda ventana top-level adicional que requiera una aplicación.
- *ApplicationShell*: Usada para la ventana top-level principal de una aplicación. La ventana X asociada a un widget de esta clase es la que interactúa con el session manager.

Un widget negocia su tamaño y posición con su widget padre, es decir, el widget que directamente lo contiene. Las ventanas X de los widgets al tope del árbol de widgets son subventanas directas de la ventana root, y por lo tanto, deben interactuar con el window manager. Para esto, cada widget top-level es encapsulado por un widget especial de la clase Shell.

3. 6. 5. Pop-Up Widgets.

Los widgets pop-ups son usados para crear ventanas fuera de la jerarquía de ventanas definida por el árbol de widgets. Todo widget pop-up tiene una ventana que es descendiente de la ventana root.

Todos los widgets pop-up rompen la correspondencia entre la jerarquía de ventanas X y la jerarquía de widgets. Los widget pop-ups no están restringidos en geometría por el widget padre.

Por lo tanto, los widget pop-ups se crean y se attachan a sus widgets padres de manera distinta que los widgets hijos normales.

Un padre de un widget pop-up no administra sus hijos pop-ups.

Todas las clases de widget pop-up son responsables de la comunicación con el window manager y por lo tanto son subclases de alguna de las clases widget Shell.

3. 6. 6. Administración de eventos.

Mientras que Xlib permite leer y procesar eventos en cualquier parte de una aplicación, los widgets de un toolkit no leen directamente eventos. Los widgets registran procedimientos que serán invocados cuando un evento o una clase de eventos ocurren en el widget.

Una aplicación típica consiste de sección de código de preparación seguida de un loop de eventos que lee y hace el dispatching de estos invocando a los procedimientos que han registrado los widgets. El loop de eventos default provisto por Xt Intrinsics es XtAppMainLoop.

El event manager es una colección de funciones para realizar las siguientes tareas:

- Agregar y eliminar procedimientos a ser llamados cuando ocurre un evento para un widget en particular.
- Consultar el estado de la cola de eventos de Xlib.

- Habilitar y deshabilitar el dispatching de eventos iniciados por el usuario (eventos de teclado y mouse) para un widget en particular.
- Restringir el dispatching de eventos a una cascada de widgets pop-ups.
- Registrar procedimientos a ser invocados ante la llegada de eventos específicos.

La mayoría de los widgets no requieren invocar explícitamente ninguna de las funciones de manejo de eventos. La interfase normal a los eventos X es vía el *translation manager*, que mapea secuencias de eventos X, con modificadores, en llamados a procedimientos.

3. 6. 6. 1. Dispatching de eventos.

Xt Intrinsics provee funciones para hacer el dispatching de eventos a widgets. Todo cliente X interesado en eventos X sobre un widget usa XtAddEventHandler para registrar los eventos en los que está interesado y un procedimiento (event handler) a ser llamado cuando ocurre un evento en la ventana X asociada al widget.

El translation manager registra automáticamente manejadores de eventos para los widgets que usan tablas de traslación.

Para procesar toda la entrada de una aplicación en un loop contínuo se usa el función XtAppMainLoop:

```
void XtAppMainLoop(app_context)
XtAppContext app_context;
```

app_context especifica el contexto de aplicación que identifica a la aplicación

La función XtAppMainLoop lee el próximo evento X llamando a XtAppNextEvent y luego hace el dispatching del evento al procedimiento registrado llamando a XtDispatchEvent.

Esto constituye el loop principal de la aplicaciones Xt. Se espera que las aplicaciones terminen como respuesta a alguna acción del usuario en una función callback.

No hay nada especial acerca de XtAppMainLoop; es simplemente un loop infinito que llama a XtAppNextEvent y luego a XtDispatchEvent.

Las aplicaciones pueden proveer su propia versión de este loop.

XtDispatchEvent invoca a los event handlers apropiados y les pasa el widget, el evento y datos específicos del cliente registrados con cada procedimiento.

Si no hay registrado ningún manejador para ese evento, el evento es ignorado y el dispatcher retorna.

```
Boolean XtDispatchEvent(event)
XEvent * event;
```

event especifica un puntero a la estructura evento a ser dispatchada al event handler apropiado.

3. 6. 6. 2. Filtros de eventos X

El manejador de eventos provee filtros que pueden aplicarse a eventos X específicos.

Los filtros, que descartan eventos que son redundantes o temporalmente no deseados, manejan los eventos MotionNotify, Enter/LeaveNotify y Exposure.

3. 6. 6. 2. 1. Compresión del movimiento del mouse.

El tiempo de procesamiento por parte de un widget de un flujo de eventos de movimiento de mouse es muy alto. Además usualmente no interesa cada uno de estos eventos. Para descartar eventos de movimiento de mouse redundante, el campo de la clase compress_motion debe valer true. Si el próximo evento de la cola de eventos es un evento de movimiento de mouse, Xt Intrinsics chequea si existen más eventos de este tipo para el mismo widget inmediatamente después del actual, si es así, los descarte y se queda con el último.

3. 6. 6. 2. 2. Compresión enter/leave.

La clase widget provee el atributo compress_enterleave para especificar esta opción. Los pares de eventos Enter/LeaveNotify contiguos, Intrinsics no los envía al cliente.

3. 6. 6. 2. 3. Compresión de exposure.

Muchos widgets prefieren procesar una serie de eventos de exposición como una única región de exposición en lugar de considerar rectángulos individuales. Aquellos widgets con displays complejos deben usar la región de exposición como areas de clipping en contextos gráficos, y widgets con displays simples ignoran completamente la región y hacen el redisplay de la ventana completa o calculan el bounding box de la unión de todas las regiones y sólo redibujan aquel rectángulo.

Aquellos widgets no interesados en la información de eventos de exposición parciales tienen el campo compress_exposure en la clase para especificar el tipo y número de eventos de exposición que serán dispatchados al procedimiento de expose del widget.

Si el campo compress_exposure en la estructura de clase del widget no especifica XtExposeNoCompress, el event manager invoca al procedimiento expose del widget sólo una vez para una serie de eventos de exposición. En este caso, todos los eventos Expose o GraphicsExpose son acumulados en una región. Cuando se recibe el evento final, el event manager reemplaza el rectángulo en el evento con el bounding box de dicha región e invoca al procedimiento de expose del widget pasándole el evento de exposure modificado y la región.

La definición del tipo del procedimiento de expose es la siguiente:

```
typedef void (*XtExposeProc)(Widget, XEvent*, Region);
```

El valor de tipo Region especifica la unión de todos los rectángulos de la serie de eventos de exposición.

3. 6. 6. 3. Manejadores de eventos X.

Los manejadores de eventos son procedimientos llamados cuando ocurren eventos específicos en un widget. La mayoría de los widgets no necesitan usar explícitamente manejadores de eventos. En su lugar, usan el translation maneger de Xt Intrinsics.

Los punteros a los manejadores de eventos son del tipo XtEventHandler:

```
typedef void (*XtEventHandler)(Widget, XtPointer, XEvent*, Boolean*);
```

```
Widget w;
XtPointer client_data;
XEvent * event;
Boolean * continue_to_dispatch;
```

donde:
w especifica el widget para el cual llegó el evento,
client_data especifica cualquier información específica del cliente registrada con el event handler,
event especifica el evento producido y
continue_to_dispatch especifica si deben ser invocados los manejadores de eventos restantes registrados para el evento actual.

Para registrar un manejador de eventos con el mecanismo de dispatching hay que usar la función XtAddEventHandler:

```
void XtAddEventHandler(w, event_mask, nonmaskable, proc, client_data)
Widget w;
EventMask event_mask;
Boolean nonmaskable;
XtEventHandler proc;
XtPointer client_data;
```

XtAddEventHandler registra el procedimiento proc que será llamado cuando ocurra un evento en w que haga matching con la máscara event_mask.
Todo widget tiene una lista de manejadores de eventos.
XtAddEventHandler llama a XSelectInput.

3. 6. 7. Callbacks.

El concepto de callback es sumamente importante ya que en el está subyacente el *"modelo de programación"* de la mayoría de los toolkits X, en particular Motif.

Las callbacks son procedimientos asociados a un widget que son invocados ante ciertas condiciones preespecificadas, no necesariamente como respuesta a eventos X. Por ejemplo, cuando un widget es destruído, se invocan a todos los procedimientos en la lista de callbacks del widget destroy_callbacks. Es decir, un widget puede llamar a estas rutinas en puntos arbitrarios de su código, en los cuales quiera proveer un 'hook' para interactuar con la aplicación.
Todo widget tiene un recurso lista de callbacks XtNdestroyCallbacks. Los widgets pueden definir listas de callbacks adicionales. La aplicación agrega funciones a estas listas de callbacks.

Los punteros a callbacks son del tipo XtCallbackProc:

```
typedef void (*XtCallbackProc)(Widget, XtPointer, XtPointer);
Widget w;
XtPointer client_data;
XtPointer call_data;
```

w especifica el widget que posee la lista en la cual se registra la callback,
client_data especifica datos adicionales dados por el cliente cuando se registra el procedimiento,

call_data especifica datos específicos de la callback que pasa el widget al cliente. Por ejemplo, cuando un Scrollbar ejecuta su lista de callbacks XtNthumbChanged, pasa la nueva posición del thumb.

Parte 4 - X Display Manager Control Protocol (XDMCP)

4. 1. Propósito y Objetivos.

El propósito de XDMCP es proveer un mecanismo para que un display 'autónomo' acceda al servicio de login de un host remoto. Por autónomo se entiende que el display consiste de hardware y procesos que son independientes de cualquier host del cual se requiera el servicio de login. Una 'X Terminal' (screen, teclado, mouse, procesador y tarjeta de red) es un ejemplo de un display autónomo.

Desde el punto de vista del usuario, se desea que un display autónomo sea tan fácil de usar que una terminal de caracteres tradicional. Específicamente, cuando se enciende una terminal se le presenta al usuario el prompt de login. Esto mismo debería ser posible con un display autónomo. Sin embargo, en un ambiente en red con múltiples hosts, el usuario podría elegir los hosts a los cuales conectarse. En un entorno con múltiples hosts y múltiples display se debería ser posible asociar un conjunto particular de hosts con un conjunto particular de displays.

Se desean soportar las siguientes situaciones:

1) El display tiene asociado ('hardwired') un host específico al cual se debe conectar. Debe ser posible encender el display y recibir un prompt de login sin intervención alguna por parte del usuario.
2) Cualquiera de varios host de la (sub)red debe poder aceptar pedidos de login desde un display. Debe ser posible que el display haga un 'broadcast' para poder encontrar tales hosts y que el display elija automáticamente el host, o que se le presente los hosts para que el usuario elija.
3) Un display tiene que tener un conjunto fijo 'wired-in' de hosts a los cuales puede conectarse. También debe ser posible proveer una facilidad de administración centralizada mediante la cual se pueda mantener conjuntos de hosts para un conjunto grande de displays, sin tener que interactuar (físicamente o electrónicamente) con algún display. Debe permitirse que cualquier host pueda no aceptar el servicio de login pedido por algún display, basándose el cualquier criterio local que desee.

El protocolo de control debe diseñarse de manera que pueda ser usado arriba de una variedad razonable de protocolos de transporte. Es deseable que cualquier familia de protocolos de red que soporte el protocolo X soporte XDMCP, dado que el resultado final de una negociación XDMCP serán unaconexiones X con el display. Debido a que el número de display por host puede ser muy grande se prefiere un protocol sin conexión a un protocolo con conexión.

Es deseable que no se requiera que los display mantengan estado permanente alguno (entre ciclos de encendido) para propósitos del protocolo de control (por ej.: saber cuando un paquete es recibido fuera de secuencia.), y es deseable mantener al mínimo el estado requerido mientras el display esté encendido.

Consideraciones de seguridad.

Se deben considerar ciertos criterios los cuales deben ser parte integral del diseño. Los objetivos de seguridad en el contexto de XDMCP son:

1) Debe ser posible que el display pueda verificar que se está comunicando con servicio de login de host legítimo, ya que el usuario presenta passwords a este servicio.
2) Posibilidad de negociación entre el display y el servicio de login del *mecanismo de autorización* a ser usuado para el protocolo X.

3) Debe ser posible proveer el mismo nivel de seguridad en la verificación del servicio de login al que es provisto por el mecanismo de autorización negociado.

4. 2. Overview del protocolo.

4. 2. 1. Introducción.

La idea de XDMCP es proveer acceso autenticado a servicios de administración de display para display remotos. Un nuevo servidor de red, llamado Display Manager administrará una colección de displays X que pueden estar en el host local o en nodos remotos. XDMCP define la interacción entre X Servers y el display manager. Este usará XDMCP para comunicarse con displays para negociar el arranque de sesiones X. El protocolo permite que el display autentique al manager. También permite que la mayoría de la información de configuración esté centralizada con el manager para facilitar tareas de administración en una red con un gran número de displays.

XDMCP implementa además el método de Control de Acceso a nivel de Usuario para controlar que clientes tienen permitido el acceso al display.

4. 2. 2. Breve descripción de un Display Manager.

Normalmente el display manager corre como un daemon (un proceso background a nivel de usuario) en sistemas UNIX. Básicamente provee servicios similares a los provistos por init, getty y login en terminales de caracteres, es decir: presentar un prompt para ingresar el nombre de login y password, autenticar al usuario y arrancar una 'sesión' (correr el login shell que tiene configurado el usuario).

Una sesión está definida por el tiempo de vida de un proceso en particular; en el mundo de las terminales basadas en caracteres es el login shell del usuario. En el contexto del display manager es un Session Manager arbitrario. Cuando no se dispone de un session manager verdadero, el window manager es usado como session manager, lo que significa que cuando termine este proceso, termina la sesión del usuario.

4. 3. Descripción general del protocolo.

4. 3. 1. Descripción de los PDUs que conforman el protocolo.

Query

BroadcastQuery

IndirectQuery

Display ⇒ Manager

Nombres de Autenticación: Es una lista de métodos de autenticación que soporta el display, con los cuales espera que el display manager se autentique. El display manager eligirá uno de estos y se lo retornará en el paquete Willing.

Semántica de los paquetes:
Un paquete *Query* es enviado por el display a un host específico para preguntarle si ese host desea proveer servicios de administración a este display. El host debe responder con Willing y desea servir al display o Unwilling si no.

Un paquete *BroadcastQuery* es similar al paquete Query excepto que la intención es que sea recibido por todos los hosts de la subred. A diferencia de Query, aquellos hosts que no desean servir al display deben simplemente ignorar el paquete BroadcastQuery.

Un paquete *IndirectQuery* es enviado a un display manager conocido el cual reenvía el requerimiento a un conjunto de display managers secundarios usando paquetes FordwardQuery. El manager primario puede también responder con un paquete Willing.
Se espera que la lista de managers secundarios sea mantenida en forma centralizada.

ForwardQuery

Primary Manager ⇒ Secondary Manager

Client Address: Dirección de red del display cliente.

Client Port: Identificación del proceso X Server en el host del display.

Nombres de Autenticación: Duplicado de la lista de nombres de autenticación que fue recibida en el paquete IndirectQuery.

Semántica del paquete:
Cuando un manager primario recibe un paquete IndirectQuery, es responsable de enviar paquetes FordwardQuery a una lista apropiada de managers los cuales pueden proveer servicio de login a displays usando el mismo tipo de red desde el cual fue recibido el paquete IndirectQuery.
La dirección de red y el port son usados por los managers secundarios para contactar al display. Cada manager secundario envía un paquete Willing si desea proveer servicios al display.

FordwardQuery es igual a BroadcastQuery en el hecho que aquellos managers que no desean proveer servicios a displays específicos no deben responder con Unwilling.

Willing

Manager ⇒ Display

Nombre de Autenticación: Especifica el método de autenticación elegido por el display manager de la lista ofrecida en el paquete Query, BroadcastQuery o IndirectQuery.
El display puede ignorar a aquellos managers que hayan elegido un nivel insuficiente de autenticación.

Hostname: String que identifica al host desde el cual este paquete es enviado.

Status: String leíble por un humano el cual describe el "estado" del host. El protocol no especifica ninguna interpretación de los datos de este campo.

Semántica del paquete:
La recepción de este paquete no implica un compromiso del display manager con el display de proveerle servicios (puede luego decidir que ya ha aceptado suficientes conexiones).

Unwilling

Manager ⇒ Display

Los mismos campos que el paquete Willing.
El campo Status debe indicar al usuario la razón por la que se niega a dar el servicio.

Semántica del paquete:
Un paquete Unwilling es enviado por managers como respuesta a paquetes Query si este no acepta requerimientos de administración. Este paquete es típicamente enviado por managers que sólo desean proveer servicios a display particulares o que manejan simultáneamente un número limitado de displays.

Request

Display ⇒ Manager

Display Number: Especifica el # de display, o X Server. Cada display en el sistema se le asigna un :display-number. Si sólo un X Server está corriendo en el sistema, su :display-number es ":0"; este es el caso para la mayoría de displays autónomos.

Tipos de Conexión: Arreglo que indica las capas de transporte aceptados por el display.
Ej.: FamilyInternet, FamilyChaos, FamilyDecnet, etc.

Direcciones de conexión: Para cada tipo de conexión en el arreglo previo, la entrada correspondiente en este arreglo indica la dirección de red del display.

Nombre de Autenticación: Especifica el protocolo de autenticación con el cual el display espera que el display manager se autentique.

Datos de Autenticación: Contiene ciertos datos que el display manager interpretará, modificará y usará para autenticarse con el display.

Nombres de Autorización: Tipos de autorización que soporta el display. El manager puede decidir rechazar displays con los cuales no pueda realizar autorización.

Manufacturer Display ID: Campo usado por el display manager para determinar como desencriptar los Datos de Autenticación de este paquete.

Semántica del paquete:
Un paquete Request es enviado por un display a un hosts específico para requerir un ID de sesión como preparativo para establecer una conexión. Si el manager desea abrir una conexión con el display, debe retornar un paquete Accept con un ID de sesión y debe estar listo para recibir un requerimiento Manage. Caso contrario debe retornar un paquete Decline.

Accept

Manager ⇒ Display

ID de la Sesión: Identifica la sesión que puede ser comenzada por el manager.

Nombre de Autenticación: Protocolo de autenticación exigido por el display (es una copia del mismo campo en el paquete request).

Datos de Autenticación: Datos retornados al display para autenticar al manager. Si estos datos no son los esperados por el display, debe inmediatamente terminar la conversación con el manager y mostrar un error al usuario.

Nombre de Autorización: Indican el tipo de autorización que el manager usará en el primer requerimiento XOpenDisplay luego que reciba el paquete Manage.

Datos de Autorización: Datos que tendrán que enviar los clientes X a momentos de establecer la conexión X con el X Server para poder empezar a correr.

Semantica del paquete:
Un paquete Accept es enviado por un manager como respuesta al paquete Request si el manager desea establecer una conexión X con el display. El ID de sesión es usado para identificar esta conexión de conexiones anteriores y será usada por el display en su posterior paquete Manage. El ID de sesión es incrementado cada vez que se envía un paquete Accept.

Decline

Manager ⇒ Display

Status: String que indica la razón por la que no se acepta administrar el display.

Nombre de Autenticación:

Datos de Autenticación: Estos datos son retornados al display para autenticar al manager. Si estos datos no son los que espera el display, este debe terminar el protocolo inmediatamente y mostrar un error al usuario.

Semántica del paquete:
Un paquete Decline es enviado por el manager como respuesta al paquete Request si el manager no desea establecer una conexión para el display. Esto es aún permitido si el manager ha respondido con un paquete Willing a un requerimiento anterior.

Manage

Display ⇒ Manager

ID de la Sesión: El valor de este campo debe ser igual al valor del campo ID de la Sesión retornado el paquete Accept.

Número de Display: Este campo debe hacer matching con el valor enviado en el paquete Request previo.

Clase del Display: Especifica la clase del display.

Semántica del paquete:
Un paquete Manage es enviado por un display para decirle al manager que comience una sesión con el display. Si el ID de sesión es correcto el manager debe abrir una conexión X, caso contrario debe responder con un paquete Refuse o Failed, a menos que el ID de sesión haga matching con una sesión ya en curso o una sesión que aún no ha podido ser abierta y que aún el display está intentando abrirla.

Respuestas válidas: Conexión X con información de autorización correcta, Refuse o Failed.

Refuse

Manager ⇒ Display

ID de la sesión: Campo que debe ser seteado al ID de la sesión recibido en el paquete Manage.

Semántica del paquete:
Este paquete es enviado por un manager cuando el # de sesión recibido en el paquete Manage no hace matching con el # de sesión actual. El display debe en este caso reenviar el paquete Request.

Failed

Manager ⇒ Display

ID de la Sesión: Este campo debe ser seteado al ID de sesión recibido en el paquete Manage.

Status: String que indica la razón de la falla.

Semántica del paquete:
Enviado por el manager cuando tiene problemas para abrir la conexión X inicial en respuesta al paquete Manage.

KeepAlive

Display ⇒ Manager

Número de Display.

ID de la Sesión: Este campo debe estar seteado al ID de Sesión recibido en el paquete Manage durante la negociación de la sesión actual.

Semántica del paquete:
Un paquete KeepAlive puede ser enviado en cualquier momento durante la sesión por un display para saber si el display manager aún sigue corriendo. El display manager debe responder con un paquete Alive siempre que reciba este tipo de paquete. No se requiere que un display envíe paquetes KeepAlive, y, el protocolo no especifica ninguna acción que debe tomar un display ante la no recepción de un paquete Alive.
El uso esperado de este paquete es terminar un sesión activa cuando el host del display manager o la red se caen. El display debería llevar el tiempo transcurrido desde la última recepción de algún paquete desde el host del display manager y enviar un paquete KeepAlive cuando ha transcurrido un tiempo substancial desde la recepción del paquete más reciente.

Alive

Manager ⇒ Display

Estado de la Sesión: Este campo indica si la sesión indicada en ID de Sesión esta actualmente activa. El valor es 0 si no hay ninguna sesión activa, 1 si hay una sesión activa.

ID de Session: Identifica la sesión que actualmente esta en curso, si es que existe. Cuando no hay ninguna sesión activa, este campo debe ser 0.

Semántica del paquete:
Un paquete Alive es enviado como respuesta al paquete KeepAlive. Si hay una sesión activa con el display, el display manager incluye el ID de la sesión en el paquete. El display puede usar esta información para determinar el estado del display manager.

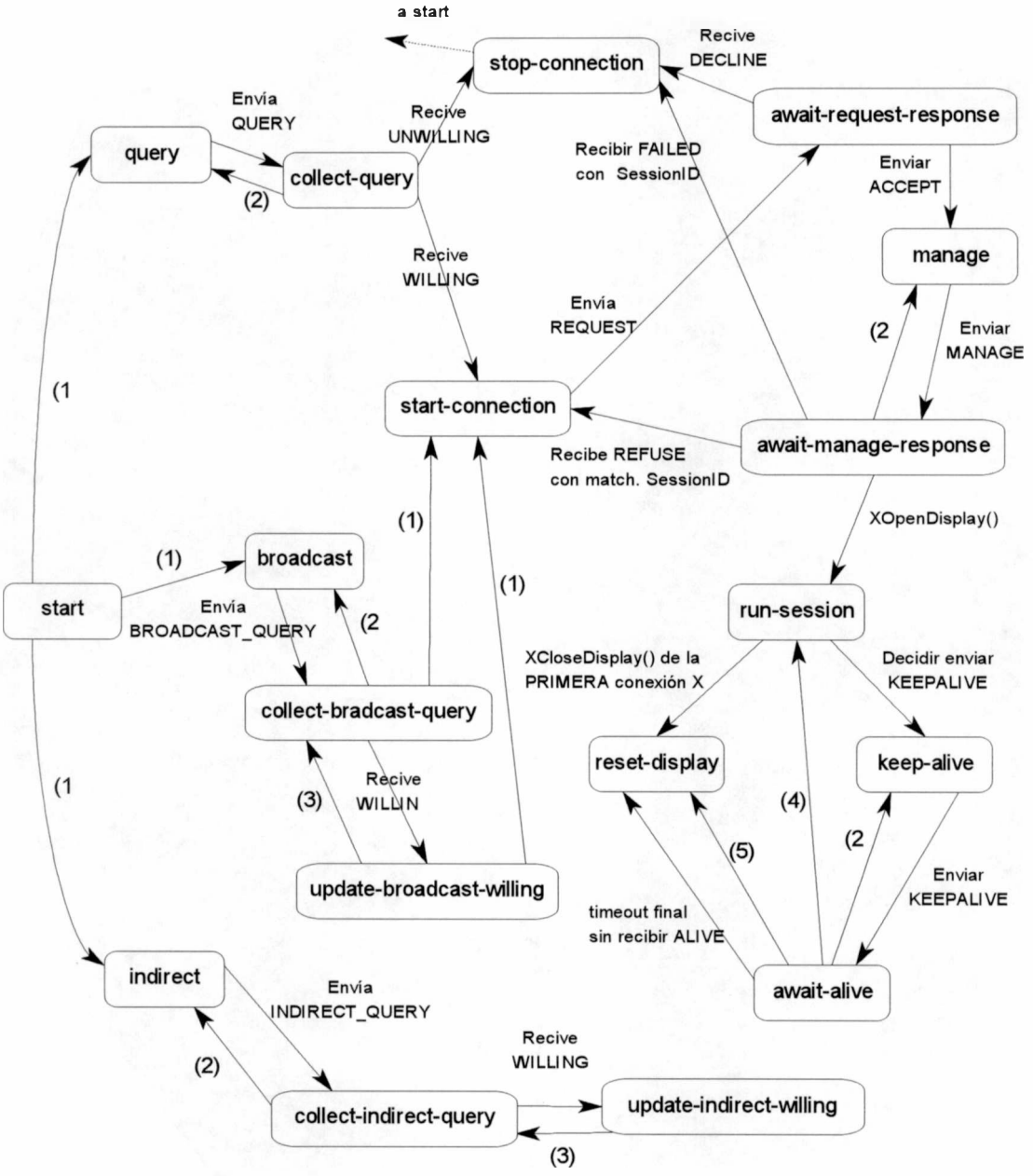
4. 3. 2. Terminación de una Sesión.

Cuando ha terminado una sesión, la conexión X inicial con el display (la que fue respuesta al paquete *MANAGE*) será cerrada por el display manager (vía XCloseDisplay()). Si sólo una sesión estaba activa con el display, todas las otras conexiones X deben ser cerradas por el display y el display debe ser rearrancado. Si están activas múltiples sesiones, y el display puede identificar cuales conexiones pertenecen a la sesión que termina, aquellas conexiones deben ser cerradas. De otra manera, todas las conexiones deben ser cerradas y el display rearrancado sólo cuando todas las sesiones han terminado, es decir, cuando todas las conexiones X iniciales han sido cerradas.

La sesión puede también ser terminada en cualquier momento por el display si el host que lo está administrando no responde a paquetes *KEEPALIVE*.

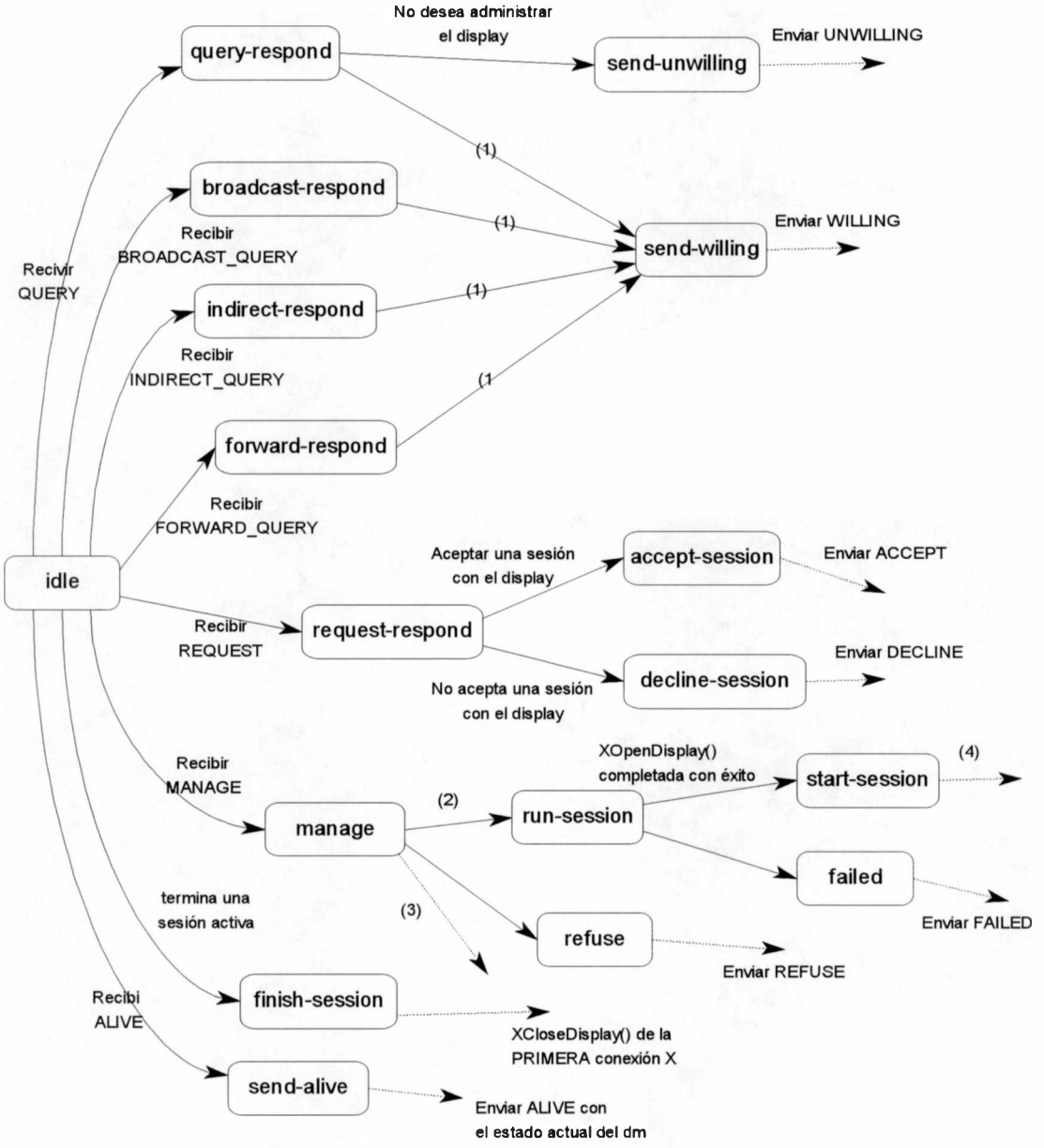
Después que el display manager pasa el control de la sesión al session manager, el display manager (un subproceso creado por este) se queda esperando a que finalice la sesión. El session manager le avisa al display manager de la terminación de la sesión en cuyo caso el display manager hace un XCloseDisplay() de la conexión X inicial (básicamente es una conexión de control), evento con el cual se espera que el X Server se resetee. Por último el display manager corre el script *Xreset*.

Diagrama de estados del display (X-Server)



(1) El usario pide conectarse a un host.
(2) Timeout.
(3) Agregar host a la lista, (donde corre un dm).
(4) Recibe ALIVE con matching SessionID.
(5) Recibe ALIVE y no hace matching el SessionID o flag que no hay sesión.

Diagrama de estados del display manager.



(1) El dm desea administrar el display.
(2) SessionID del paquete hace matching con SessionID almacenado.
(3) SessionID hace matching con SessionID de la sesión en proceso de arranque o con la sesión actualmente activa.
(4) Arranca todas las Apps Windows Clientes X configuradas en el nodo.
Transición a idle.

4. 4. Seguridad en XDMCP.

4. 4. 1. Autenticación en XDMCP - XDM-AUTHENTICATION-1.

En un ambiente donde no se requiere autenticación, XDMCP puede deshabilitar la autenticación enviando el display el paquete Request con los campos *nombre de autenticación* y *datos de atenticación* vacíos. En este caso, el display manager no intentará autenticarse. Otros protocolos de autenticación pueden se desarrollados, dependiendo de las necesidades locales.

En un ambiente inseguro, el display debe ser capaz de verificar que la fuente de los paquetes XDMCP es un display manager en el cual confiar. Estos paquetes contendrán información de autenticación. Un sistema de autenticación es XDM-AUTHENTICATION-1, el cual se describe en los siguientes párrafos.

Para que el display manager se autentique con el display, el display y el manager comparten una clave privada. El manager, entonces, debe ser capaz de descubrir que clave usar para un display en particular. El manager usa esta clave para desencriptar los datos de autenticación de este paquete. El método de encriptado usado por este sistema de autenticación es DES (Data Encryption Standard).

Típicamente, el manager contiene el mapeo entre el *Manufacturer Display ID* del paquete *Request* y la clave en el archivo (valor del recurso *DisplayManager.keyFile* del display manager). Se espera que este campo sea único por display.

4. 4. 2. Autorización provista por XDMCP.

XDMCP provee un mecanismo de Control de Acceso por Usuario, el cual puede ser usado para restringir el acceso al X Server por cliente. El paquete *Accept* enviado por el display manager al display, contiene *datos de autorización*. Cuando un cliente X, el cual debe ser parte de la sesión de administración de display subsecuentemente contacta al X Server, debe pasar los mismos datos para que pueda correr. El host en el cual corre el cliente autorizado no es relevante. Las aplicaciones X debe compilarse para usar datos de autorización. Las aplicaciones no compiladas con las librerías correctas no podrán correr.

XDMCP soporta 2 niveles de Contro de Acceso por Usuario: MIT-MAGIC-COOKIE-1 y XDM-AUTHORIZATION-1 (Los strings "MIT-MAGIC-COOKIE-1" y "XDM-AUTHORIZATION-1" son los posibles valores del campo *nombre de autorización* de los paquetes XDMCP).

Cuando se usa MIT-MAGIC-COOKIE-1, los datos de autorización viajan como un string de caracteres. Este esquema es peor que un mecanismo de control de acceso basado en host en entornos de baja seguridad, ya que permite que se conecte cualquier host una vez que se descubre la clave. Pero en muchos entornos, este nivel de seguridad es mejor que el basado en host, ya que controla el acceso a nivel de usuario.

El mecanismo XDM-AUTHORIZATION-1 es más seguro ya que encripta estos datos antes de transmitirlos. El encriptado y el desencriptado se realiza usando los valores del ID del Display y una Clave y también se usa DES (idem XDM-AUTHENTICATION-1). Este esquema de autorización puede usarse con o sin XDM-AUTHENTICATION-1.

Si no se soporta XDMCP, aún se puede usar Control de Acceso a nivel de Usuario creando un archivo de autorización local.

Usando un archivo de Autorización Local.

Para esto hay que crear un archivo en el host del display manager y en el host del display. Este archivo contiene datos de autorización necesarios para permitir que los clientes corran en el server. Por default, el archivo se llama *.Xauthority* en el directorio $HOME de los usuarios. Si un cliente pasa un dato que está incluído en este archivo, entonces el X Server permitirá su acceso.

La seguridad basada en XDMCP usa un archivo .Xauthority temporario en el host. Este es creado dinámicamente cuando corren clientes que usan XDMCP.

4. 5. xdm - X Display Manager con soporte para XDMCP.

A continuación se describe el display manager *xdm* que forma parte de la distribución del X Window System del MIT. Los display managers comerciales (por ejemplo scologin de SCO) son recompilaciones de xdm más extensiones y pequeñas modificaciones para integrarse al entorno gráfico correspondiente.

Archivo de configuración de xdm, (usualmente) */usr/lib/X11/xdm/xdm-config*, el cual define el comportamiento del display manager. Este tiene el formato de la especificación de recursos en X.
Ejemplo de una configuración:

```
DisplayManager.servers: /usr/lib/X11/xdm/Xservers
DisplayManager.errorLogFile: /usr/lib/X11/xdm/xdm-errors
DisplayManager*resources: /usr/lib/X11/xdm/Xresources
DisplayManager*startup: /usr/lib/X11/xdm/Xstartup
DisplayManager*session: /usr/lib/X11/xdm/Xsession
DisplayManager._0.authorize: true
DisplayManager*authorize: false
```

Notar que algunos de los recursos son especificados con "*". Estos recursos pueden ser únicos para cada display, reemplazando "*" por el display-name.
(display-name es de la forma: [hostname]:display-number[.screen-number])

El primer archivo, */usr/lib/X11/xdm/Xservers* contiene la lista de displays a administrar que no están usuando XDMCP. Cada entrada del archivo es de la forma:

display-name [display-class] display-type [comando de arranque]

Cuando el display a administrar es remoto, se debe arrancar el X Server en el screen a administrar en el nodo del display de forma tal que el display manager pueda alcanzar el display (comenzar una sesión X) con este.

El archivo */usr/lib/X11/xdm/xdm-errors* contendrá mensajes de error de xdm y cualquier salida a stderr hecha por Xsetup, Xstartup, Xsession o Xreset.

4. 5. 1. Recursos del display manager.

La siguiente entrada de configuración, */usr/lib/X11/xdm/Xresources*, es cargada en el display como una base de datos de recursos usando el comando *xrdb*.

Algunos recursos modifican el comportamiento de xdm en todos los displays, mientras que otros lo hacen en un único display. Cuando las acciones se refieren a un display específico, el nombre del display es insertado en el nombre del recurso entre "DisplayManager" y el último segmento del nombre del recurso. Por ej.: DisplayManager.disp_0.startup es el nombre del recurso que define del archivo de shell de startup para el display "disp:0".

DisplayManager.servers
Ver Especificación de Servers más adelante.

DisplayManager.requestPort
Especifica el # de port UDP el cual xdm usa para escuchar los requerimientos XDMCP.
El valor default es 177.

DisplayManager.daemonMode
Normalmente, xdm se vuelve un proceso daemon no asociado con ninguna terminal. Cuando esto no es deseado (en situaciones de debugging), se puede deshabilitar esta característica especificando este recurso en 'false'.

En cuanto a seguridad se tienen estos recursos:

DisplayManager.keyFile
Si el método de autenticación XDMCP que se usa es *XDM-AUTHENTICATION-1*, se requiere que una clave privada sea compartida entre xdm y el display. Este recurso especifica el archivo que contiene aquellos valores. Cada entrada en el archivo contiene el nombre del display y la clave compartida. Por default, xdm no incluye soporte para XDM-AUTHENTICATION-1.

DisplayManager.accessFile
Para prevenir acceso no autorizado a los servicios provistos por XDMCP y para permitir hacer reenvío de requerimientos IndirectQuery, este archivo contiene nombres de hosts los cuales tienen permitido el acceso a esta máquina o tienen una lista de hosts a los cuales se les debe reenviar el requerimiento. *Ver Control de Acceso usando XDMCP.*

DisplayManager.DISPLAY.resources
Este recurso especifica el nombre del archivo a ser cargado por *xrdb* como la base de datos de recursos en la ventana raíz del screen 0 del display (la base de datos de recursos se guarda en la propiedad *RESOURCE_MANAGER*). El programa Xsetup, el widget Login y el programa chooser usarán los recursos definidos en este archivo. Esta base de datos de recursos es cargada inmediatamente antes que comience el procedimiento de *autenticación del usuario* de manera que pueda controlar la apariencia de la ventana de login. El nombre default es /usr/lib/X11/xdm/Xresources.

DisplayManager.DISPLAY.chooser
Especifica el programa a correr que ofrecerá el menú de hosts para requerimientos IndirectQuery redirigidos al nombre especial de host CHOOSER. El default es /usr/lib/X11/xdm/chooser. *Ver Control de Acceso usando XDMCP y Chooser.*

DisplayManager.DISPLAY.setup
Especifica un programa que se ejecuta (como root) antes de que se muestre la ventana de Login. Puede ser usado para cambiar la apariencia del screen o displayar otras ventanas junto a la de Login. El nombre convencional del archivo es *Xsetup*. Por default no se corre ningún programa. *Ver Programa Setup.*

DisplayManager.DISPLAY.startup
Especifica un programa que es corrido (como root) después que haya finalizado exitosamente el proceso de autenticación del usuario. El nombre convencional del archivo es *Xstartup*. Por default no se corren ningún programa. *Ver Programa Startup.*

DisplayManager.DISPLAY.session

Después de ejecutar el script Xstartup, el display manager busca un script que defina la sesión X del usuario. Primero busca el archivo *$HOME/.xsession* (en el directorio de trabajo del usuario). Si no existe tal archivo, se busca el archivo */usr/lib/X11/xdm/Xsession* el cual recorre los archivos *.profile* y *.login* del directorio $HOME del usuario y se setean las variables de entorno que estén definidas en estos. Luego el display manager arranca el *Session Manager* pasando de esta manera la responsabilidad de la administración de la sesión. *Ver Programa Sesión.*

DisplayManager.DISPLAY.reset

Especifica un programa que se ejecuta (como root) después que termina la sesión. Nuevamente, por default no se corre ningún programa. El nombre convencional es */usr/lib/X11/xdm/Xreset. Ver Programa Reset.*

DisplayManager.DISPLAY.authorize
DisplayManager.DISPLAY.authName

authorize es un recurso booleano que controla si xdm genera y usa autorización para las conexiones locales con el X Server. Si se usa autorización, authName es una lista de mecanismos o protocolos de autorización a usar.

Las conexiones XDMCP dinámicamente especifican que mecanismos de autorización son soportados, por lo tanto authName es ignorado en este caso. Cuando authorize es seteado para lun display y la autorización no está disponible (en el paquete Request - del display al manager- no se especificó ningún mecanismo de autorización), se le informa al usuario vía un mensaje mostrado en la ventana de Login. Por default, authorize es true; authName es "MIT-MAGIC-COOKIE-1".

DisplayManager.DISPLAY.authFile

Este archivo es usado para comunicar los datos de autorización del xdm al server.

4. 5. 2. Control de acceso usando XDMCP.

Cuando el X Server arranca, hace un requerimiento XDMCP para preguntar que hosts pueden proveer servicios de administración a este display. La respuesta enviada por el display manager está determinada por el archivo de control de acceso XDMCP en el host, especificado en el recurso DisplayManager.accessFile, cuyo nombre default es */usr/lib/X11/xdm/Xaccess.*

El archivo Xaccess provee información que el xdm usa para controlar el acceso de display que requieren servicos XDMCP. Este archivo contiene tres tipos de entradas: entradas que controlan la respuesta a consultas Directas (Query) y de BroadCast (BroadcastQuery), entradas que controlan la respuesta a consultas Indirectas (IndirectQuery) y definiciones de macros.

El formato de las entradas para consultas Query/BroadcastQuery es tanto un nombre de host o un pattern las cuales son comparadas con el nombre de host del diplay.

Prefijar un nombre de host o un pattern con el caracter '!' causa que los hosts que hagan matching con esta entrada sean excluídos.

Una entrada para una consulta IndirectQuery contiene también el nombre de un host o un pattern pero sigue una lista de nombres de hosts o macros a los cuales deben reenviarse las consultas Indirectas.

Una definición de macro contiene un nombre de macro y una lista de nombres de hosts

Las entradas Indirectas pueden también especificar que xdm debe correr el programa *Chooser* para ofrecer un menú con los posibles hosts a conectarse. *Ver Chooser.*

Cuando se chequea el acceso de un host de un display en particular, se recorre cada entrada y la primer entrada que haga matching determina la respuesta. El acceso no es permitido si no se encuentra una entrada que haga matching con el nombre de host del display.

4. 5. 3. Chooser.

Para las terminales X que no ofrecen un menú de hosts para consultas de Broadcast o Indirectas, el programa chooser hará esto si en el archivo Xaccess se especifica la palabra CHOOSER como la primer entrada en la lista de hosts de una entrada Indirecta.

Esto produce que chooser envíe requerimientos Query a cada uno de los hosts de la lista y ofrecerá un menú con aquellos hosts que respondieron.

Si la palabra CHOOSER es seguida de la palabra BROADCAST, chooser enviara en cambio un requerimiento de Broadcast y nuevamente ofrecerá un menú con los hosts que respondieron.

4. 5. 4. Especificación de X-Servers.

El recurso DisplayManager.servers especifica servers o nombres de archivos que contienen especificaciones de servers.

Cada especificación indica un display que debe ser administrado por el Display Manager y que no está usando XDMCP. Cada una consiste de al menos 3 partes: un nombre de display, una clase de display un tipo de display, y para servers locales un comando para arrancar el server. Si el tipo de server es remoto, entonces xdm debe abrir una conexión X con él server que está controlando el display. Para esto, el X Server ya debe estar corriendo en el host remoto. Básicamente, luego que arranque el display manager, este presentará la ventana de Login en cada uno de los X Servers especificados en este archivo.

4. 5. 5. Programas (scripts) del display manager.

A continuación se describen los programas o scripts que son corridos por el display manager antes de comenzar la sesión con el entorno gráfico y al finalizar la sesión.

4. 5. 5. 1. El programa Setup.

El archivo Xsetup es corrido después que el X Server es reseteado, pero antes que sea mostrada la ventana de Login. El archivo es típicamente un shell script. Corre como root. Este es el lugar para cambiar el background de la ventana Root o mostrar otras ventanas que deben aparecer junto a la ventana de Login.

4. 5. 5. 2. El programa Startup.

El archivo Xstartup es típicamente un shell script. Corre como root y es el lugar para poner comandos para montar directorios desde file servers (vía NFS por ej.), abortar la sesión si no se permite el login, etc.

Xdm espera hasta que termine este script antes de comenzar la sesión de usuario. Si el script no termina correctamente, xdm discontinúa la sesión y inicia otro ciclo de autenticación de usuario.

4. 5. 5. 3. El programa Session.

El programa Xsession es el comando que es corrido como la sesión del usuario. Este se corre con los permisos del usuario autorizado.

En la mayoría de las instalaciones, Xsession debe buscar un archivo .xsession en $HOME el cual contiene comandos que cada usuario quiere usar como la sesión.

Xsession puede también implementar una sesión default de sistema si no existe alguna sesión especificada por el usuario.

4. 5. 5. 4. El programa Reset.

Simétrico a Xstartup, el script *Xreset* es corrido luego que la sesión del usuario ha terminado, es decir cuando el usuario hace un log out del sistema. Corre como root y debe contener comando que deshagan el efecto de los comandos en Xstartup tales como desmontar directorios desde un file server que fueron montados cuando comenzó la sesión, etc.

Cuando termina la sesión con el entorno gráfico, el display manager resetea al X Server (vía la system call kill() si se trata de un server local o usando la función *XKillClient()* definida en Xlib si se trata de un server remoto que no usa XDMCP. Para el último caso, la idea es resetear al display matando a todos los clientes X que hayan creado ventanas X. Este procedimiento reseteará al X Server la mayoría de las veces, a menos que queden clientes X conectados sin ventanas.) y redespliega la ventana de Login. En caso que el X Server use XDMCP, el display manager hace un *XCloseDisplay()* de la conexión X inicial.

4. 5. 6. Control del X-Server.

Xdm controla a X Servers locales usando señales POSIX. Se espera que la señal SIGHUP resetee al server, cerrando todas las conexiones X de los clientes y realizando otras tareas de 'clean-up'. La señal SIGTERM termina el proceso server. Para controlar terminales remotas que no usan XDMCP, xdm recorre la jerarquía de ventanas del display y usa el requerimiento X *KillClient* (vía la función *XKillClient()*) con la idea de preparar la terminal para la próxima sesión. Como se dijo anteriormente esto no matará todos los clientes ya que serán notificados sólo aquellos que crearon ventanas.

XDMCP provee un mecanismo más robusto; cuando el display manager cierra la conexión inicial (vía *XCloseDisplay()*), la sesión termina y se requiere que el X-Server cierre las otras conexiones. Básicamente, la sesión X inicial que establece el display manager con el X Server (como respuesta al paquete *MANAGE* de XDMCP) es usada para mostrar la ventana de Login, sirve para terminar una sesión (XDMCP o no) y también para descubrir cuando el display remoto ha desaparecido.

Para esto último, el display manager ocasionalmente pollea a los displays usando la función *XSync()* de Xlib y esperando que la terminal responda a los requerimientos. Si la terminal no responde, la sesión es declarada muerta y es terminada.

4. 6. Session Managers.

Un session manager es un cliente X especial responsable del startup y shutdown de sesiones con el X Server. Un session manager debe ser capaz de:

- Arrancar una colección de clientes.
- Recordar el estado de una colección de clientes de manera que estos puedan ser rearrancados en el mismo estado.
- Terminar una colección de clientes de una manera controlada.

Las convenciones de comunicación entre clientes (ICCCM) define también la comunicación entre el session manager y clientes X standards. La nueva versión de ICCCM (incluída en X11R6) definirá también la comunicación entre el session manager y el window manager. Aclaremos que no se describirán las convenciones de ICCCM referentes al session manager.

Varios son los archivos que determinan su comportamiento, a continuación serán descriptos.

4. 6. 1. Archivos de configuración.

Archivos de configuración de un session manager:

- *startup*: (Ver Arranque del Session Manager).
- *static*: define los clientes X que se ejecutarán como parte de la sesión default del sistema.
- *shutdown*: (Ver Finalización del Session Manager).
- *xrdbcomp*: compara los recursos cargados por xrdb al comenzar la sesión con los recursos agregados a la base de datos de recursos (en el X Server) en la sesión actual y guarda los cambios de forma que puedan ser usados en sesiones futuras.

El session manager también guarda información de las sesiones de cada usuario en archivos ubicados en subdirectorios del directorio de trabajo de estos.

Archivos de configuración de sesión por usuario:

- *dynamic*: contiene los clientes que son salvados de una sesión previa. Estos clientes comenzarán a ejecutarse si el usuario elige retomar la sesión previa.
- *static*: contiene los clientes que constituyen la sesión default del usuario. En caso que el usuario no haya salvado una sesión default, se usa la sesión default del sistema que tiene configurada el session manager.
- *xrdb.save*: contiene los valores de los recursos de la base de datos de recursos que existen al finalizar la sesión del usuario. Estos recursos son cargados en la base de datos de recursos la próxima vez que se retome la sesión.

4. 6. 2. Arranque del Session Manager.

Cuando el session manager comienza su ejecución, lee el script *startup*. Este archivo prepara la sesión con el entorno gráfico. Las tareas principales que realiza son las siguientes:

- Carga los recursos ubicados en el archivo */usr/lib/X11/xdm/Xresources* en la base de datos de recursos. El script también carga los recursos almacenados en el archivo xrdb.save del usuario. Estos recursos residen en el X Server y determinan la apariencia básica y el comportamiento de muchos de los clientes X que son arrancados como parte de la sesión.
- Restaura cualquier información de estado de la sesión previa, incluyendo aceleración del mouse, intervalo del double-click del mouse, etc. Esta información reside en otros archivos en el directorio de trabajo del usuario.
- Lee el archivo dynamic de algún subdirectorio del directorio de trabajo del usuario si el usuario decide retomar la sesión anterior o el archivo static si elige la sesión default y arranca todos los clientes X especificados. Si no existe ninguno de estos archivos, el session manager corre los clientes indicados en su archivo de configuración del sistema static.
- Estos archivos no sólo indican los clientes a correr, sino cualquier opción de línea de comando usada para arrancar a las aplicaciones.
- Arranca el cliente *Window Manager* especificado por el recurso *session-manager*windowManager*.

4. 6. 3. Finalización del Session Manager.

Cuando finaliza la sesión, el session manager corre el script *shutdown*, el cual corre el script *xrdbcomp*. Estos scripts hacen lo siguiente:

- Remueve la base de datos de recursos de la propiedad RESOURCE_MANAGER de la ventana Root. Todo recurso que fue agregado en la base de datos de recursos durante la sesión es almacenado en el archivo xrdb.save de un subdirectorio de $HOME.
- Guarda la información de estado de los clientes que estaban aún corriendo cuando finalizó la sesión y guarda la información el el archivo dynamic de un subdirectorio de $HOME. Estos clientes son arrancados en el mismo estado en la próxima sesión si el usuario decide retomar la sesión previa.
- Guarda toda la información de estado en los archivos apropiados en un subdirectorio de $HOME.
- Termina todos los clientes X que están corriendo, incluyendo al Window Manager (*ICCCM* define, aunque en forma rudimentaria, la comunicación entre clientes y el session manager para situaciones tales como la terminación de una sesión).

Parte 5 - WExX (Windows Extension X-aware)

5. 1. Especificación

Recordemos cuales son los problemas a solucionar:

- Traslación o 'mapping' del subconjunto del API de Windows referente a interfase de usuario al X Window System, lo que incluye básicamente un 'mapping' bidireccional.
- 'Browsing' y ejecución de las aplicaciones Windows existentes en la red desde otro nodo de esta.

Aclaremos que para disponer de las librerias de programación sobre el X Window System para Windows fue necesario recompilar los fuentes del X Window System, que como ya se dijo son de dominio público (Ver *Apéndice A* en el cual se comentan los problemas encontrados en la recompilación, entre otras cosas).

5. 1. 1. Que se le pide a nuestra solución.

Adherirse a todas las convenciones de X que gobiernan la comunicación entre clientes, haciendo énfasis en el área Window Managers, debido a que esta es una característica que debe cumplir cualquier Cliente X. Una aplicación deberá utilizar las convenciones definidas en ICCCM para comunicarse con el Window Manager de forma tal de acceder al frame de una ventana; es decir, se ignora todo aquello que modifique el 'look' del frame de una ventana y que no se pueda soportar con las convenciones anteriormente mencionadas, como modificar el tamaño de la barra de título de una ventana, agregar items de menú al menú del sistema de una ventana, etc.

Con respecto al 'look' de las aplicaciones es importante aclarar que si bien no se hace énfasis en preservar la semántica de display de una aplicación, se siguen manteniendo todos los elementos de interfase de usuario que provee Windows como scrollbars, botones, listboxes, menúes, dialog boxes modales y no modales, soporte de MDI, etc.

5. 1. 2. Separación de la interfase de usuario de Windows

A continuación se hará un análisis de las funciones del API de Windows que deberán interceptarse para lograr la separación del GUI de Windows y permitir trasladarlos a requerimientos y sobre el API del X Window System

5. 1. 2. 1. Problemas que impiden una separación clara entre los módulos.

1) Existe una fuerte dependencia interna entre los 3 módulos básicos de Windows, lo que hace difícil una separación clara de los componentes de interfase de usuario y el resto.
2) La interfase entre los módulos de Windows no es pública (no está documentada).

Uno de los ejemplos más claros de dependencia etre módulos es el sistema de mensajes el cual está implementado en el módulo USER. En Windows, los mensajes son usados para múltiples propósitos. El uso más claro es para la interfase de usuario, pero también son usados para comunicación entre tareas, por el módulo KERNEL para consultar a todas las aplicaciones si es posible hacer el 'shutdown' de Windows y fundamentalmente, el flujo de mensajes está íntimamente ligado al scheduling de aplicaciones Windows, funcionalidad provista por el KERNEL.

Referente a la interfase entre los módulos de Windows, ocurre que cuando una aplicación invoca a rutinas del USER tales como CreateWindow(), MoveWindow(), ShowWindow() y otras, éste invoca a rutinas del GDI para dibujar los frames de las ventanas, iconos, titles bars y otros. Es de esperar que la funciones del GDI que son invocadas por el USER fueran las mismas que forman parte del API provisto a las aplicaciones, pero no siempre es así, ya que existe un gran número de funciones exportadas por el GDI no documentadas, que son usadas por el USER y otros módulos de Windows.

Como se describió anteriormente, los módulos que implementan el UIMS de Windows son el USER y el GDI. El módulo USER implementa el Sistema de Administración de Ventanas y el Sistema de mensajes, mientras que el GDI implementa las operaciones gráficas. La funciónalidad de estos módulos es la que debemos separar de aquellas aplicaciones a ser controladas en forma remota.

5. 1. 2. 2. Criterio de separación inicial.

Nuestra primera aproximación se basó en el criterio de reusar el código del USER y separar solamente el sistema gráfico de Windows, es decir la funciónalidad del GDI.

Para funciones del USER, la idea era "interceptarlas y dejarlas pasar a Windows", mientras que para las funciones del GDI "interceptarlas y no dejarlas pasar a Windows" trasladándolas a las funciones gráficas de X.

La ventaja de esta solución era que se iba a reusar todo el código de los procedimientos de ventana de las clases de ventana de Windows. De esta manera, evitaríamos redefinir todo el comportamiento default de las ventanas y de los controles.

Sin embargo esta aproximación nos trajo muchos problemas, que trataremos de explicar a continuación.

Si bien las ventanas de una aplicación no aparecerían en el desktop de Windows debido a que las operaciones gráficas a las que le USER invoca para dibujarlas serían interceptadas, para el USER las ventanas seguirían existiendo, ya que este mantiene internamente información sobre las ventanas que fueron creadas, como su posición y tamaño, si tiene el foco, área de clipping etc.. Luego, las ventanas seguirían recibiendo mensajes cuando por ejemplo, el usuario mueve el mouse en el área donde estas se encuentran (aunque no se vean). Además, el USER enviaría a redibujar solamente las áreas visibles de éstas ventanas en caso que estubieran tapadas por otra de una aplicación Windows standard, y el hecho de ocultar las ventanas a momento de su creación empeora las cosas, ya que el USER directamente no dibuja las ventanas que están ocultas. Una posibilidad para engañar al USER para que dibuje las ventanas ocultas era modificar, en la estructura de ventana interna que mantiene, la región visible de ésta, la cual es nula si la ventana está oculta. Forzando a esta región a que sea no nula a momento de dibujar la ventana podría ayudar, pero el gran problema de esto es tener que tratar con estructuras de Windows que no están documentadas.

Otro problema es mantener la misma información asociada a una ventana en los dos UIMS, Windows y X. Por ejemplo si el usuario mueve una ventana (en el Desktop de X), tendríamos que reflejar esto en las estructuras que lleva el USER. Aún pudiendo manipular estas estructuras, no es posible que el estado de una ventana sea el mismo en ambos desktops debido a que en Windows podrían existir ventanas de aplicaciones Windows standards y en X ventanas de clientes X standards.

Como consecuencia de lo expuesto anteriormente no es posible que Windows administre las ventanas de nuestras aplicaciones, paralelamente con el X Server, y realizar sólamente el mapeo de los llamados al GDI.

5. 1. 2. 3. Criterio de separación adoptado.

Los problemas enunciados anteriormente nos llevaron a pensar en una solución en la cual las aplicaciones WExX no utilizaran código del GDI ni del USER. Para esto fue necesario entonces interceptar todas las invocaciones al API referente con el UIMS de Windows, que hicieran las aplicaciones Windows que se fueran a controlar remotamente.

Esta aproximación nos lleva a tener que implementar la funcionalidad de todos los elementos de la GUI existentes en Windows, utilizando los provistos por el X Window System.

5. 1. 3. 'Mapping' al X Window System.

Este punto implica el mapeo de :

- funciones del UIMS de Windows a X
- objetos del USER y del GDI a recursos y estructuras de X
- eventos X a mensajes Windows
- mensajes Windows a requerimientos de X

La necesidad del mapeo de mensajes Windows a requerimientos de X resulta de los llamados a funciones del USER que generen el envío de mensajes, como MoveWindow() (que produce el envío de WM_MOVE) o explícitamente la invocación de la función SendMessage().

Debido a que la interfase de programación al X Window System esta basada en capas, hay que determinar la o las capas a las que se mapearán la funcionalidad de interfase de usuario de Windows. A continuación se hará un análisis de las diferentes alternativas de mapeo considerando las siguientes pautas:

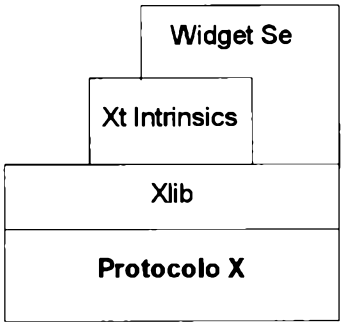
- Funcionalidad de las aplicaciones WExX.
- 'Look and feel' de las aplicaciones WExX.
- Flexibilidad. Estar directamente arriba de la interfase al protocolo X implica tener mayor control de este sistema de ventanas.
- Performance. Cuanto más arriba estemos en los niveles de abstracción al protocolo X, la performance decrece.
- Cantidad de código a portar.

5. 1. 3. 1. Alternativas de mapeo.

Existen básicamente 2 alternativas de mapeo (ver grafico *Interfases de programación al X Window System*).

1) Utilizar exclusivamente Xlib.
2) Trabajar con Xt Intrinsics + algún Widget Set.

Interfases de programacion al X Window System



Las ventajas de adoptar la alternativa 1) es que siendo Xlib la interfase a X de mas bajo nivel permite tener mayor control sobre este sistema de ventanas, como por ejemplo el procesamiento más flexible de eventos X. Por otro lado, como **el API de interfase de usuario que provee Windows es de mayor nivel que el de Xlib**, el utilizar Xlib como interfase de programación implica disponer solamante de ventanas como objetos gráficos del lado de X Window, por lo que todos los demás elementos de interfase de Windows deberán ser implementados sobre la funcionalidad de una ventana.

La alternativa 2) tiene la ventaja de disponer con elementos de interfase de usuario (widgets) de un nivel de abstracción similar a los provistos por Windows. Debido a que todo widget set define una política de interfase de usuario (recordemos que un widget es una combinación de una ventana X y su semántica de display e input), estaríamos perdiendo el 'look and feel' de una aplicación Windows.

En cuanto a la administración de eventos y al manejo de callbacks que provee Xt Intrinsics, siendo este mecanismo más general que el dispatching de mensajes de Windows es posible poder implementar este manejo.

Una desventaja de esta alternativa es la necesidad de tener que portar una mayor cantidad de código del X Window System que en la otra alternativa.

Se decidió elegir la alternativa de trabajar con Xt Intrinsics + algún Widget Set y en algunas situaciones acceder directamente a las funciones que operan sobre la cola de eventos de Xlib por lo siguiente:

- Se dispone de elementos de interfase de usuario de similar nivel a los provistos por el USER de Windows.
- En cuanto al manejo de eventos X, si bién la generalidad provista por Xt puede agregar cierto overhead innecesario para simular el manejo que hace Windows, el hecho de estar arriba de Xt no nos impide poder invocar funciones de Xlib. Invocar a funciones del widget set, Xt Intrinsics y de Xlib es una práctica usual cuando se programa en X.
- La alternativa de utilizar Xlib tiene la desventaja de tener que definir algún 'look and feel' de la UI ya que esto no es parte de la filosofía de X.

Aclaremos que los objetos graficos del GDI y las funciones del API asociadas a estos se mapearán directamente a estructuras y funciones de Xlib, debido a que los widgets sets no agregan ninguna funcionalidad para esta tarea.

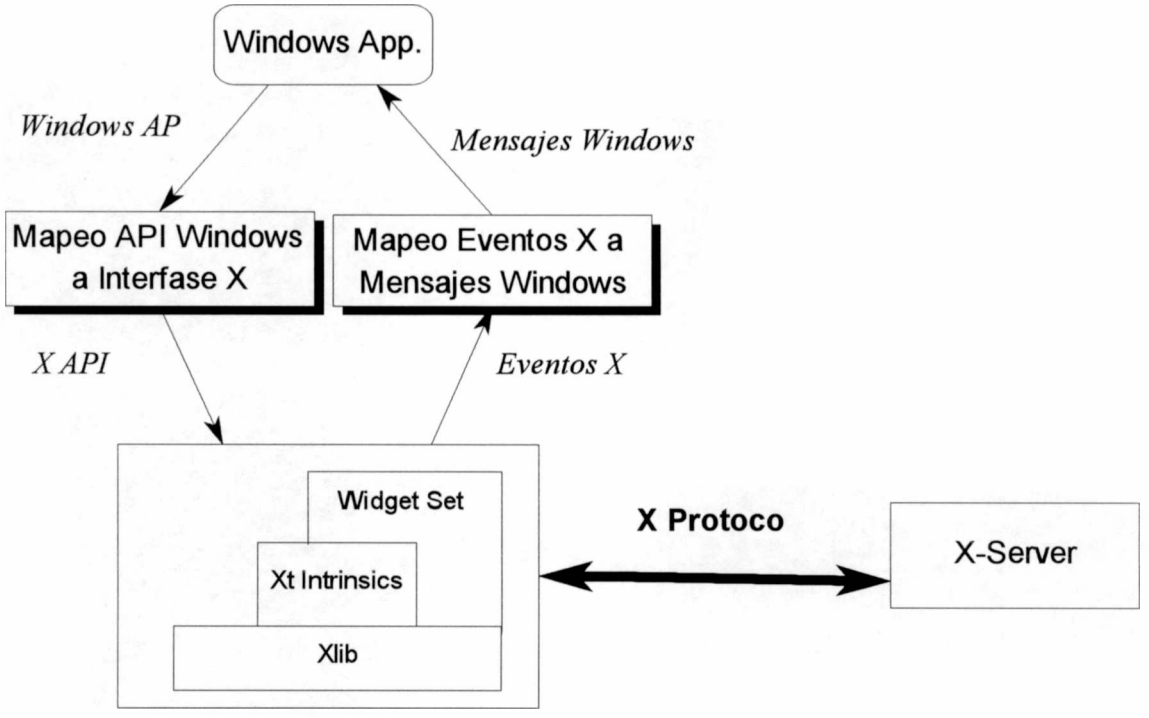
5. 1. 4. Arquitectura de WExX

Cuando una aplicación Windows sobre WExX llame a una función de alguno de los módulos del UIMS de Windows, el llamado será interceptado y se invocará una función de WExX. Esta función de WExX invocará a un conjunto de funciones del API de X (Xlib + Xt Intrinsics + Widget Set) las cuales generarán requerimientos al X-Server.

En caso que la función del API de Windows tenga algún valor de retorno y/o parámetros de salida, la X library esperará las respuestas desde el X-Server y las funciones del API de X devolverán los valores de retorno y/o parámetros de salida, y por último la función de WExX retornará los resultados a la aplicación Windows.

Por otro lado cuando se reciba un evento X desde el X server, este deberá ser mapeado a mensajes Windows y ser enviado al procedimiento de ventana correspondiente en la aplicación Windows.

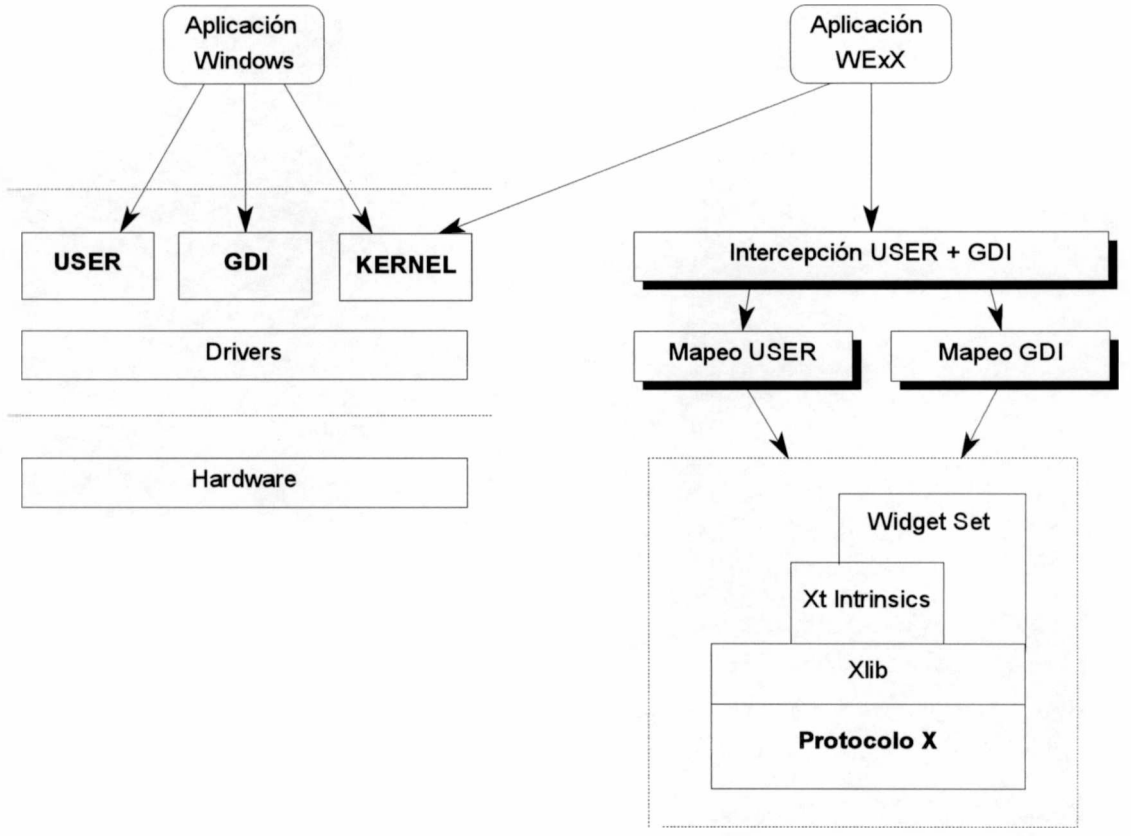
Arquitectura general de WExX



Las aplicaciones WExX siguen siendo desde el punto de vista del KERNEL de Windows, aplicaciones Windows normales. Es decir, se instancian de la estructura MDB, tienen asociada una TDB y tambien tienen asociada una cola de mensajes creada por el USER a tiempo de arranque de estas. Esta cola sigue siendo necesaria ya que como se mencionó anteriormente, tienen otros usos además de ser usados en la UI.

Como se señaló anteriormente las aplicaciones WExX 'hablan' el protocolo X, tienen ventanas en un desktop X y se adhieren a todas las convenciones que gobiernan la comunicación entre clientes, debido a que deben cohexistir adecuadamente con otros clientes X que comparten el mismo X-Server. Asi **las aplicaciones WExX son verdaderos clientes X**.

Relación entre una App. Windows y una App. WExX



5. 1. 5. 'Browsing' y ejecución de las aplicaciones WExX.

Recordemos que nuestra solución al problema de 'browsing' y ejecución de aplicaciones Windows consiste en proveer los siguientes dos mecanismos:

- Un mecanismo de 'browsing' que permita identificar las aplicaciones WExX existentes en la red.
- Un mecanismo que permita lanzar la ejecución de las aplicaciones WExX identificadas.

En la Parte 4 se presentó la solución al problema de establecer una sesión gráfica con un nodo de la red elegido en forma dinámica desde cualquier otro nodo de la red. Recordemos que en la solución apareció un nuevo server en la red, el X Display Manager, y que se definió un protocolo de comunicación entre este proceso y el X Server, el protocolo XDMCP.

También se presentaron otros problemas como consecuencias de esta solución, como es el problema de la seguridad en lo que hace a la **autenticación** del display manager y al mecanismo de **autorización** que el display manager exige al X Server que soporte para autorizar a los clientes X a establecer conexiones X con el X Server.

Para la implementación del mecanismo de 'browsing' se portará a Windows la máquina de estados del display manager xdm que implementa XDMCP[1]. No se tendrán en cuenta las consideraciones de seguridad contempladas en el diseño de xdm.

Recordemos que con XDMCP "se llega hasta el nodo", por lo tanto para poder elegir la/las aplicaciones WExX a correr ("llegar hasta una aplicación") hay que agregar una interacción más entre el X Server y el módulo que implementa XDMCP en Windows.

[1] El código fuente del display manager xdm es de dominio público (se incluye en la distribución del MIT del X Window System)

5. 2. Diseño Preliminar

Debido a que los programas en X Window, ya sea utilizando Xlib o algun Widget Set, tienden a ser muy extensos y complejos, es que decidimos realizar un diseño orientado a objetos que encapsulara las caracteristicas particulares de este sistema de ventanas.

Se definieron objetos que encapsulan todas las entidades lógicas de Windows, tanto a nivel de componentes de interfase de usuario (ventanas) como a nivel de aplicación (modulos, tareas). Estos objetos están organizados en una jerarquia de clases (a la que llamaremos jerarquia de clases de WExX). Esta jerarquía es parte del módulo WXUSER. Para la implementación de estas clases hemos utilizado Xlib como interfase de programación sobre el X Window System. No utilizamos Xt Intrinsics y alguna implementación de un Widget Set ya que nuestro objetivo era lograr una implementación que validara nuestra idea y diseño.

Por otro lado los objetos gráficos y las funciones del API del módulo GDI de Windows fueron encapsulados en objetos propios del módulo WXGDI. Nos basamos directamente en la funciónalidad provista por Xlib para hacer el mapeo del GDI de Windows; recordemos que Xt Intrinsics abstrae principalmente objetos de interfase de usuario.

Esta jerarquia de clases fue diseñada e implementada por nosotros, para luego utilizarlas para implementar las funciones del API de Windows que son necesarias interceptar para portar cualquier programa Windows hacia el X Window System. No obstante **la jerarquia de clases de WExX podria ser utilizada como un 'application framework' para el desarrollo de aplicaciones X en Windows y DOS**.

5. 2. 1. Mapeo del Sistema de Mensajes

En el X Window System los eventos son generados por el X server a partir de acciones del usuario o como notificación a requerimientos del cliente. Por otro lado, Windows envía mensajes a las aplicaciones también por acciones del usuario (mover el mouse, pulsar una tecla), y luego de llamados a funciones del API de Windows que producen un cambio en la configuración de una ventana.

Existe una cierta relación entre algunos eventos en X y los mensajes en Windows. Aquí explicaremos como será el mapeo de eventos X en mensajes Windows hecho por WExX, y como cada evento X trae asociado información adicional, se comentarán los campos mas relevantes asociados a cada evento, y como serán utilizados para dar valores a los parámetros en los mensajes Windows correspondientes que éstos generen.

5. 2. 1. 1. ButtonPress y ButtonRelease.

Estos eventos generados por el X-Server cuando el usuario presiona o suelta un boton del mouse.

Miembros mas relevantes de la estructura de evento.

subwindow : si la ventana fuente es hija de la ventana que esta recibiendo el evento, este miembro tiene el ID de esta child.

time : hora en la que ocurrio el evento en milisegundos (contados a partir de la inicialización del Xserver). Solo será utilizado en el caso que la aplicación Windows requiera de información adicional del mensaje mediante la función GetMessageTime.

x, y : coordenadas relativas al origen de la ventana que recibe el evento

x_root, y_root : coordenadas relativas a la ventana root a la que pertenece la ventana que recibe el evento.

state : estado de los botones y teclas modificadoras antes de ocurrido el evento.

button : un valor indicando cual boton produjo el evento.

Mensajes Windows correspondientes.

WM_LBUTTONDOWN	El boton izquierdo del mouse fue presionado.
WM_LBUTTONUP	El boton izquierdo del mouse fue soltado.
WM_MBUTTONDOWN	El boton medio del mouse fue presionado.
WM_MBUTTONUP	El boton medio del mouse fue soltado.
WM_RBUTTONDOWN	El boton derecho medio del mouse fue presionado.
WM_RBUTTONUP	El boton derecho del mouse fue soltado.

Parámetros.

fwKeys : Indica que teclas estaban presionadas en el momento de enviado el mensaje. Será asignado con los valores del campo "state" mapeados a Key Flags de Windows.

xPos : posición horizontal del cursor. Sera asignado con el parametro "x" del evento.

yPos : posición vertical del cursor. Sera asignado con el parametro "y" del evento.

Observaciones.

En Windows existen ademas los mensajes de "doble click" que son generados al presionar dos veces el mouse dentro de un intervalo espacial y temporal configurado en Windows. Estos seran generados de la misma forma en WExX a partir de los eventos de mouse de X. Los mensajes relacionados con "doble click" en Windows son :

WM_LBUTTONDBLCLK Indica double-click del boton izquierdo del mouse.
WM_MBUTTONDBLCLK Indica double-click del boton medio del mouse.
WM_RBUTTONDBLCLK Indica double-click del boton derecho mouse.

Los parámetros en los mensajes de "doble click" serán configurados de la misma manera que los mensajes de mouse individuales.

5. 2. 1. 2. ConfigureNotify.

Este evento es recibido cuando hubieron cambios en la configuración de la ventana (size, position, border, stacking order).

Miembros relevantes de la estructura de evento.

window : la ventana cuya configuración ha cambiado.
x, y : coordenadas de la ventana cambiada relativas al padre.
width, height : width y height en pixeles de la ventana luego de la reconfiguración.

Mensajes Windows correspondientes.

WM_SIZE

Parámetros.

fwSizeType : especifica el tipo de resizeo realizado. Siempre tendrá el valor SIZE_RESOTRED.
cx, cy : tamaño de la ventana luego de la reconfiguración. Les serán asignados los valores x e y de la estructura de evento respectivamente.

WM_MOVE

Parámetros.

cx, cy : posición de la ventana luego de la reconfiguración. Les serán asignados los valores width y height de la estructura de evento respectivamente.

Observaciones.

El evento **ConfigureRequest** no será seleccionado por las ventanas creadas por WExX, ya que este evento es generalmente interceptado por el Window Manager. Esta decisión fue tomada para no violar las normas del X Window System con respecto al control de la interfase externa de la aplicación, y afectara a aquellos programas Windows que intenten modificar el aspecto externo de una aplicación.

En consecuencia el mensaje **WM_WINDOWPOSCHANGING** no será nunca enviado a una aplicación que corra sobre WExX.

5. 2. 1. 3. CreateNotify.

Una aplicación recibe este evento luego de que una ventana X ha sido creada con exito.

Miembros relevantes de la estructura de evento.

parent : ID del padre de la ventana creada.
window : ID de la ventana creada.
x, y : coordenadas de la ventana.
width, height : width y height en pixeles de la ventana.

Mensajes Windows correspondientes.

WM_ACTIVATE
Es enviado cuando la ventana principal de una aplicación es activada o desactivada.
Este mensaje será enviado a la aplicación una vez cuando esta cree su ventana principal indicando que la aplicación esta activa, y otra vez cuando la ventana principal es destruida indicando que la aplicación ya no este activa, por lo que durante todo el tiempo de vida de la aplicación su ventana principal estara activa, ya que en X no existe el concepto de ventana activa o desactiva.

Parámetros.

fActive : Indica si la ventana es activada o desactivada. Los valores posibles son :
WA_ACTIVATE, WA_INACTIVATE, WA_CLICKACTIVATE.
Le será asignado el valor WA_ACTIVATE cuando la ventana principal es creada y con el valor WA_INACTIVATE cuando la ventana principal es destruida.
fMinimized : si es distinto de cero indica que la ventana esta minimizada. Le será asignado el valor 0 o 1 dependiendo si el estilo de la ventana principal incluye el estilo WS_MINIMIZED o no.
hwnd : identifica la ventana que esta siendo activada o desactivada. Se corresponde con el hWnd de la ventana principal de la aplicación.

Observaciones.

El mensaje **WM_CREATE,** que en Windows es enviado al procedimiento de ventana cuando se esta creando una ventana, no será enviado en este momento ya que este mensaje debe ser recibido (segun especificado por Microsoft) antes del retorno del llamado a función CreateWindow o CreateWindowEx, por lo que será enviado por estas funciones luego de hacer el pedido a XWindow de la creación de la ventana. Si la ventana X no puede ser creada en el X-Server, seran enviados a la aplicación los mensajes de destrucción correspondientes.
Si una ventana es activada con un click del mouse entonces tambien recibe el mensaje **WM_MOUSEACTIVATE**, pero en WExX este mensaje no será generado, ya que el mensaje WM_ACTIVATE será enviado en tiempo de creación de la ventana principal, por lo que no habra oportunidad de activar una ventana con el mouse.

5. 2. 1. 4. EnterNotify y LeaveNotify.

EnterNotify y LeaveNotify son utilizados para notificar a un cliente que el mouse entro o salio de una de sus ventanas. Cuando el mouse pasa el borde de una ventana, un evento "LeaveNotify" es enviado a ventana de la que sale el mouse y un "EnterNotify" a la ventana

donde ingresa. Si el cliente recibe un LeaveNotify en su top-level window, luego el cliente no recibira mas eventos de teclado y mouse hasta que reciba un "EnterNotify"; salvo que el teclado y/o el mouse hayan sido grabeados. Ademas son enviados "EnterNotify" y "LeaveNotify" a todas las ventanas que son "virualmente cruzadas", que son aquellas que estan entre las primeras dos en la jerarquia.

Mensajes Windows correspondientes.

WM_SETCURSOR

Es enviado a una ventana cuando el mouse se encuentra dentro de la ventana y produce movimientos del cursor.

Solo será generado este mensaje una vez al recibir el evento EnterNotify para darle la oportunidad a las ventanas de poner sus cursores particulares, aunque el cursor registrado en la clase de ventana será el utilizado al crear la ventana en el X Server por lo que el cambio de cursor por defecto lo hara el X Server sin mediación de la aplicación. Pero en algunas oportunidades el padre de una ventana desea indicar el cursor de sus hijas (por ejemplo los "Dialog Boxes"). En consecuencia, el procedimiento de ventana default (DefWindowProc) enviará el mensaje WM_SETCURSOR al padre.

Parámetros.

hwndCursor : Indica la ventana sobre la cual se está moviendo el mouse.

nHitTest : especifica el área donde la configuración del cursor es requerido. En WExX a este miembro será asignado siempre con el valor HTCLIENT, indicando el área cliente, que es sobre el unico área que la aplicación tendrá control (recordar diferencias entre la interfase interna de la aplicación - área cliente de una ventana - con la interfase externa - área no cliente o frame -).

wMouseMsg : es el número de mensaje de mouse.

Observaciones.

El evento LeaveNotify no será mapeado a ningun mensaje de Windows.

5. 2. 1. 5. DestroyNotify.

Reporta que una ventana ha sido destruida.

Miembros relevantes de la estructura de evento

window : ID de la ventana destruida.

Mensaje Windows Correspondiente

WM_DESTROY

El mensaje WM_DESTROY es enviado al procedimiento de ventana luego de que la ventana es removida del screen. Este mensaje es enviado primero a la ventana que esta siendo destruida y luego a sus hijas a medida que son destruidas.

5. 2. 1. 6. Expose.

Un evento "Expose" le avisa a un cliente que una ventana o parte de una ventana paso a estar visible y necesita ser "repintada".

Miembros relevantes de la estructura de evento.

x, y : coordenadas relativas al origen de la ventana.
width, height : el width y height en pixels de la región expuesta.
count : el numero aproximado de "Expose" remanentes que fueron generados como resultado de un solo llamado a función. El server garantiza enviar contiguamente a todos los "Expose" provocados por un movimiento de una ventana o por el llamado a una función

Mensaje Windows correspondiente.

WM_PAINT
Indica que una ventana necesita ser repintada.

Observaciones.

Este mensaje es enviado a una aplicación cuando no hay ningun otro mensaje en la cola de mensajes de la aplicación.

5. 2. 1. 7. FocusIn y FocusOut.

Estos eventos ocurren cuando el foco del teclado es cambiado, y es enviado a las ventanas que hayan seleccionado FocusChangeMask (estan muy relacionados con los eventos EnterNotify y LeaveNotify). Una ventana recibe el evento "FocusOut" cuando es la ventana que tenia el foco del teclado o cuando pertenece, en la jerarquia de ventanas X, a la rama en la que esta la ventana que perdio el foco; y recibe un evento "FocusIn" si es la nueva ventana foco o esta en la rama de la nueva ventana foco en la jerarquia de ventanas X.

Mensajes Windows correspondientes.

WM_SETFOCUS
WM_KILLFOCUS
Estos mensajes son enviados cuando una ventana gana o pierde el foco respectivamente.
Primero es enviado WM_KILLFOCUS a la ventana que pierde el foco y luego WM_SETFOCUS a la que lo gana.

Parámetros.

hwnd : handle de la ventana que gana o pierde el foco.

WM_ACTIVATE
Es enviado cuando la ventana principal de una aplicación es activada o desactivada.
Este mensaje será enviado a la aplicación una vez cuando esta cree su ventana principal indicando que la aplicación esta activa, y otra vez cuando la ventana principal es destruida indicando que la aplicación ya no este activa, por lo que durante todo el tiempo de vida de la aplicación su ventana principal estara activa, ya que en X no existe el concepto de ventana activa o desactiva.

Parámetros.

fActive : Indica si la ventana es activada o desactivada. Tendra los valores siguientes :
WA_ACTIVATE en caso de un evento FocusIn.
WA_INACTIVATE en caso de un evento FocusOut
fMinimized : si es distinto de cero indica que la ventana esta minimizada. Le será asignado el valor 0 o 1 dependiendo del estado de la ventana, que es mantenido en las ventanas WExX.
hwnd : identifica la ventana que esta siendo activada o desactivada. Tendrá siempre el valor NULL.

Observaciones.

Si una ventana es activada con un click del mouse el miembro fActive deberia ser **WA_CLICKACTIVATE** y la aplicación deberia recibir tambien el mensaje **WM_MOUSEACTIVATE**, pero en WExX esto no sucedera, ya que el evento de MapNotify no provee la información necesaria.

5. 2. 1. 8. GravityNotify

Este evento reporta que una ventana a sido movida debido a un cambio en el tamaño de su padre.

Observaciones.

Este evento no será seleccionado sobre las ventanas X creadas desde WExX, ya que luego de este evento se recibe el evento configurenotify.

5. 2. 1. 9. KeyPress y KeyRelease

Estos eventos son producidos para todas las teclas.

Miembros de la estructura de evento.

subwindow : si la ventana fuente es child de la ventana que recibe el evento, este miembro tiene el window ID de esa child.
time : hora del evento en milisegundos.
x, y : si la ventana esta en el mismo screen que la root, entonces estos miembros son las coordenadas del mouse relativas a la ventana origen.
state : el estado de los botones del mouse y las teclas modificadoras en el momento del evento.
keycode : codigo server-dependent de la tecla presionada. Este codigo debe ser trasladado a un simbolo portable llamado "keysym" con funciones provistas por X Window.

Mensajes Windows correspondientes.

WM_KEYDOWN
WM_KEYUP

Parámetros.

wvKey : identifica el virtual-key code. Correspondera al valor resultante del mapeo del keysym en un vitual key de Windows.

dwKeyData : Especifica el "repeat count", "scan code", "extended key", "context code", y "key-transition state". Le será asignado el valor del miembro state de la estructura de evento.

Observaciones.

En Windows tambien se genera el mensaje **WM_CHAR** (que no tiene equivalente en el sistema de mensajes de X Window), luego de llamar a la función TranslateMessage con un mensaje WM_KEYDOWN obtenido con GetMessage. En el mensaje **WM_CHAR,** el parametro lParam tiene el mismo valor que el parametro lParam del mensaje WM_KEYDOWN, y el parametro wParam contiene el codigo ASCII del caracter presionado. Por lo tanto si la aplicación llama a la función TranslateMessage con un mensaje WM_KEYDOWN, será puesto un mensaje WM_CHAR asociado a este indicando el ASCII que es obtenido a partir del "virtual-key code" y las teclas modificadoras que estaban presionadas como es indicado en el campo dwKeyData.

5. 2. 1. 10. MapNotify y UnmapNotify

Estos eventos los genera el XServer cuando una ventana cambia de estado desde mapeada a no mapeada y viceversa.

Miembros relevantes de la estructura de evento

window : la ventana que fue mapeada o desmapeada.

Mensaje Windows Correspondiente

WM_SIZE
Es enviado cuando una ventana ha cambiado de tamano.

Parámetros.

fwSizeType : Indica si la ventana esta siendo maximizada, minimizada o restaurada.Tendra los valores siguientes :

SIZE_MAXIMIZED en caso de un evento MapNotify

SIZE_MINIMIZED en caso de un evento UnmapNotify

nWidth, nHeight : tamano de la ventana luego de la reconfiguración. Les serán asignados los valores width y height de la ventana respectivamente.

5. 2. 1. 11. MotionNotify

Este evento es recibido cuando el usuario mueve el mouse.

Miembros relevantes de la estructura de evento.

time : hora en la que ocurrio el evento en milisegundos (contados a partir de la inicialización del XServer). Sera utilizado solo en el caso que la aplicación Windows requiera de información adicional del mensaje mediante la función GetMessageTime.

x, y : coordenadas relativas al origen de la ventana que recibe el evento.

x_root, y_root : coordenadas relativas a la ventana root.

state : estado de los botones y teclas modificadoras antes de ocurrido el evento.

Mensajes Windows correspondientes.

WM_MOUSEMOVE

Parámetros.

fwKeys : flags de teclas. Le será asignado el valor del campo "state" mapeado a Key Flags de Windows.

xPos : posición horizontal del cursor. Sera asignado con el parámetro "x" del evento.

yPos : posición vertical del cursor. Sera asignado con el parámetro "y" del evento.

5. 2. 1. 12. VisibilityNotify.

Este evento reporta algun cambio en la visibilidad de la ventana especificada. Es generado solo para ventanas InputOutput.

Miembros relevantes de la estructura de evento.

state : un valor indicando el status de la ventana. El valor puede ser uno de los siguientes : VisibilityFullyObscured, VisibilityPartiallyObscured, VisibilityUnobscured.

Mensajes Windows correspondientes.

WM_SHOWWINDOW

Este mensaje es enviado a una ventana cuando va a pasar a estar visible o invisible.

Parámetros.

fShow : TRUE o FALSE si la ventana esta siendo mostrada u ocultada respectivamente. Le será asignado TRUE si el miembro state tiene el valor VisibilityUnobscured y FALSE si tiene el valor VisibilityFullyObscured.

fnStatus : status de la ventana que esta siendo mostrada. Es cero si el mensaje fue producido por un llamado a la función ShowWindow, de lo contrario puede tener los valores :

SW_PARENTCLOSINGla ventana padre esta siendo minimizada.

SW_PARENTOPENING la ventana padre esta abriéndose. A este parámetro le será asignado cero, ya que no puede ser obtenida la información necesaria para asignarle el valor adecuado.

5. 2. 2. Mapeo del Sistema de Ventanas

En el capitulo anterior se hablo del mapeo de eventos X a mensajes Windows. Los eventos X pueden ser generados, al igual que los mensajes Windows, de dos formas : debido a una acción del usuario, o debido a requerimientos desde programa para realizar un cambio en una ventana. En este capítulo describiremos como serán trasladadas las funciones de manejo de ventanas de Windows en requerimientos al X Server, y como los eventos de notificación, provenientes del X Server, serán procesados y mapeados a mensajes Windows. Generalmente los requerimientos estarán asociados a la invocación de funciones de la API de Windows que producen alguna modificación en la posición, tamaño o algunos otros atributos de ventanas.

En esta parte de la traslación de Windows sobre X Window surge una diferencia principal entre los dos sistemas de ventanas. Esta gran diferencia es la imposibilidad, en X Window, de sincronizar un requerimiento al X Server con una notificación desde el X Server. Por ejemplo, cuando se envia un requerimiento al X Server de mover una ventana, el X Server procesa el requerimiento, mueve la ventana en el display, y luego envia un evento de notificación a la aplicación que lo requirió. Por otro lado, en Windows, cuando es invocada la función MoveWindow para mover y/o redimensionar una ventana, los mensajes de notificación WM_MOVE y WM_SIZE correspondientes son enviados al procedimiento de la ventana que se esta modificando **antes** de que la función MoveWindow termine. Esta diferencia entre el "modelo sincrónico" de Windows y el "modelo asincrónico" de X Window con respecto al envío de requerimientos y recepción de notificaciones, debe ser tratado con mucho cuidado ya que pueden existir aplicaciones Windows que dependan fuertemente de que los mensajes sean enviados en los momentos y orden en que Windows los envía, por lo que podrían cambiar impredeciblemente su comportamiento.

Es importante aclarar que no serán mapeados todos los elementos de interfase gráfica de Windows (menúes, dialog boxes, listboxes, etc.) ya que a efectos de lograr una validación de nuestra idea era suficiente con mapear el comportamiento completo de una ventana, que es la base de los demás elementos de interfase. Mapear el comportamiento completo de una ventana implica tanto trasladar las funciones de creación y manipulación de ventanas en requerimientos equivalentes hacia el X server, como generar mensajes Windows ante las notificaciones del X server. Para poder soportar también la creación y manipulación de los demás elementos de interfase, como por ejemplo un dialog box, solo sería necesario agregar el comportamiento específico de ese elemento de interfase utilizando un objeto de un 'widget set' que lo implemente.

A continuación se detalla el mapeo de las funciones del API de Windows que generarán requerimientos al X Server, y como son tratados los eventos de notificación enviados por el X Server.

5. 2. 2. 1. Funciones de Creación de Ventanas

5. 2. 2. 1. 1. CreateWindow

Esta función, además de alocar los datos necesarios para la administración de una ventana, tiene la responsabilidad de enviar los mensajes **WM_CREATE** y, si la ventana fue creada con el estilo WS_VISIBLE, **WM_SHOWWINDOW**.

Para una aplicación ejecutándose sobre WExX, el mensaje WM_ CREATE será enviado al procedimiento de ventana de la nueva ventana luego de hacer el pedido al X server de creación del widget asociado a la nueva ventana Windows.

Si el estilo incluye el flag WS_VISIBLE, antes de retornar de la función CreateWindow se llamará a la función de Xlib XMapWindow, y recién al recibir la notificación MapNotify desde el X

Server se enviará el mensaje WM_SHOWWINDOW al procedimiento de ventana de la nueva ventana.

5. 2. 2. 1. 2. DestroyWindow

Cuando una aplicación Windows llama a esta función para destruir una ventana, el mensaje WM_DESTROY es enviado al procedimiento de la ventana que se desea destruir para permitir la liberación, por parte de la aplicación, de datos asociados a la ventana.

Cuando la aplicación se ejecute sobre WExX, la función DestroyWindow solo generará el pedido al X Server de destrucción del widget asociado a la ventana que se esta destruyendo, pero el mensaje WM_DESTROY será enviado al procedimiento de ventana de la aplicación cuando llegue la notificación DestroyNotify desde el X Server.

Este es otro claro ejemplo de la diferencia entre el modelo sincrónico de Windows contra el modelo de requerimientos y notificaciones asincrónico de X Window debido a la estructura client-server en la que está basado.

5. 2. 2. 1. 3. ShowWindow

Esta función, cuando es invocada por una aplicación Windows, envia un mensaje WM_SHOWWINDOW antes de retornar.

Cuando la aplicación se ejecute sobre WExX la función solo generará un requerimiento al X Server de mapear el widget asociado a la ventana Windows, y el mensaje WM_SHOWWINDOW será enviado al procedimiento de ventana de la aplicación cuando sea recibido desde el X server el evento de notificación MapNotify o UnmapNotify.

5. 2. 2. 2. Funciones de Configuración de Ventanas

5. 2. 2. 2. 1. SetWindowPos

Esta función tiene una aplicación mas general que el solo hecho de modificar el z-order[2] de una ventana : cambia el tamaño, posición y/o z-order de una ventana.

Al querer modicar el tamaño, posición y/o z-order de una ventana Windows se enviará en primer lugar el mensaje **WM_WINDOWPOSCHANGING** al procedimiento de ventana para notificarle del pedido de cambio de la ventana. Si el mensaje es pasado al procedimiento default de la ventana (DefWindowProc), este enviará a la misma ventana el mensaje **WM_GETMINMAXINFO** para validar el nuevo tamaño y posición de la ventana. Al retorno del envío del mensaje WM_GETMINMAXINFO, se invocarán funciones de X lib que hagan los requerimientos al X Server de cambios para el widget asociado. Por último, cuando el X server envíe la notificación ConfigureNotify, será enviado el mensaje **WM_WINDOWPOSCHANGED**, que si no es procesado por la aplicación y es pasado al DefWindowProc genera el envío de los mensajes **WM_SIZE** y **WM_MOVE**.

5. 2. 2. 2. 2. MoveWindow

Esta función cambia la posición y/o el tamaño de una ventana, y será tratada como un caso especial de la función SetWindowPos.

---

[2] Recordemos que las ventanas están ordenadas de acuerdo a su apariencia en la pantalla, la ventana al tope es la primer ventana en el z-order.

5. 2. 2. 3. Solución a MDI (Multiple Document Interface).

Los widget set (Athena, Motif u Open Look, etc.) no soportan ninguna de las propiedades de MDI de Windows; no es posible tener una ventana hija de una ventana top-level que pueda ser movida iconizada o cambiada de tamaño por el usuario.

Esto es así ya que el concepto de MDI no adhiere a la definición de interfase de usuario la cual está compuesta por una interfase que es externa a la aplicación (recordemos que es responsabilidad del window manager), y otra que es interna a la aplicación de la cual es responsable ésta. El frame (barra de título, system menu, botones de minimizar y maximizar) de las child windows en MDI lo provee Windows; bajo X es responsabilidad de la aplicación.

Se proponen dos soluciones para soportar el concepto de MDI bajo el X Window System.

5. 2. 2. 3. 1. Grupo de ventanas.

Se tiene una ventana top-level para la frame window y una ventana top-level para cada child window. Además tiene que existir una relación entre estas ventanas, todas estas ventanas conforman un *grupo*. Debe diferenciarse la ventana que en Windows es la frame window o *líder de grupo* para nuestra solución. Debido a que estas ventanas son ventanas top-level, esta funciónalidad debe ser soportada por el window manager. El window manager debe tratar a la frame window de manera diferente para permitir que cuando se minimize esta ventana se minimizen las child windows, cerrar todas las ventanas cuando se cierre la frame window, etc.

Inclusive el window manager podría decidir decorar a la frame window de una manera, y todas las child window de otra.

Todo esto hay que comunicárselo al window manager vía *HINTS* (recordar estructura WM_HINTS definida en ICCCM usada cuando se crea una ventana top-level). Para esto hay que extender el window manager para que soporte estas propiedades.

Básicamente habría que poder indicar cual es la frame window y todas las child window *apuntar* a esta.

Esta idea no soluciona el hecho que las child window de MDI están restringidas a la client window.

5. 2. 2. 3. 2. Sub window manager.

La siguiente aproximación soluciona la principal característica de MDI de restringir las child window a la client window.

La idea es muy simple: proveer un window manager que administre las ventanas child window de una ventana top-level (un *sub window manager*). Para cada app. Windows cliente X existirá otro cliente X que es este window manager.

Ahora, la ventana sobre la que el window manager selecciona eventos de redirección de estructura no es la ventana root sino la frame window. Así este window manager proveerá un frame de decoración para las child window.

La restricción que pusimos a las apps. Windows en lo referente a la interfase externa de la aplicación, que es responsabilidad del window manager, también aplica a cada una de las child window; es decir que no permitimos que una app. Windows manipule el frame de estas ventanas.

Resumiendo, el mapeo de la funciónalidad de MDI de Windows a X consiste en proveer un window manager con las características antes mencionadas.

5. 2. 2. 4. Soporte de DDE (Dynamic Data Exchange)

Se registraran en Windows dos clases de ventanas que procesaran los mensajes de DDE :

1) **WExXDDEInitiateClass** : procesa solamente el mensaje WM_DDE_INITIATE.
2) **WExXDDEClass** : procesa los restantes mensajes de DDE.

Cuando una aplicación comienza a ejecutarse sobre WExX, se creara una ventana de la clase WExXDDEInitiateClass, que llamaremos WExXDDEInitiateWindow.

Cuando un mensaje WM_DDE_INITIATE es recibido por esta ventana, luego de que algun cliente envio un broadcast requiriendo que algun server responda sobre algun topico, la ventana WExXDDEInitiateWindow enviara el mensaje a cada una de las ventanas de la aplicación, soportando asi la posibilidad de que una ventana sobre WExX que implemente algun server de DDE pueda recibir el mensaje de inicialización desde un cliente.

Si una ventana WExX responde con un WM_DDE_ACK a un WM_DDE_INITIATE, se creara una ventana de la clase WExXDDEClass que llamaremos WExXDDEWindow, que estara asociada a la ventana que esta respondiendo. Luego el parámetro wParam del mensaje WM_DDE_ACK será asignado con el hWnd de la ventana WExXDDEWindow, de manera que esta actuará para la aplicación cliente (y para Windows) como la verdadera ventana server. De esta forma cuando el cliente envie un mensaje pidiendo datos o enviando datos al server con los mensajes WM_DDE_REQUEST, WM_DDE_POKE, etc.; la ventana WExXDDEWindow los recibira y pasara a la ventana WExX que tiene asociada, dando asi soporte completo a una aplicación sobre WExX a funciónar como server de DDE.

Para el caso en el que una aplicación sobre WExX inicia una conversación DDE, o sea que se comporta como cliente de DDE, el mecanismo será similar a cuando una aplicación responde con un WM_DDE_ACK a un WM_DDE_INITIATE que fue explicado anteriormente. Al enviar el mensaje WM_DDE_INITIATE, será creada una ventana WExXDDEWindow a la que se le asociara la ventana que inicia la conversación. El parametro wParam del mensaje WM_DDE_INITIATE será asignado con el hWnd de la ventana WExXDDEWindow con lo que todo mesnaje asociado a este WM_DDE_INITIATE será enviado a la ventana WExXDDEWindow y luego esta lo enviara a la ventana asociada, de la misma manera que lo hace cuando se comporta como server.

Asi, una aplicación sobre WExX puede tambien iniciar una conversación de DDE, o sea comportarse como cliente; por lo que se brinda soporte total de DDE a una aplicación que se ejecute sobre WExX.

5. 2. 3. Mapeo de GDI

La idea es mapear todos los objetos y funciones del GDI de Windows usando Xlib, para este objetivo se creara un modulo llamado WXGDI similar al GDI de Windows formado por objetos propios con las mismas funcionalidades de los objetos de Windows pero usando los recursos y funciones Xlib. Usamos Xlib por ser la capa que contiene las funciones gráficas de mas bajo nivel y el mapeo es mas directo hacia Windows, en otras capas como Xt Intrinsics o Widgets solo permiten desarrollar elementos gráficos mas complejos, usados en la interfase de usuario como son los scrollbars, botones, menúes ,etc.

5. 2. 3. 1. Objetos Gráficos

En Windows existen objetos que permiten responder a los requerimientos gráficos de una aplicación. Estos objetos son creados y guardados por el GDI y son:

```
Pen
Brush
Font
Palette
Bitmap
Device Context
Cursor
Region
```

En X Window existen recursos que cumplen con algunas funcionalidades de los objetos de Windows. Estos recursos pertenecen a Xlib y son:

```
Grahics Context
Colormap
Pixmap
Cursor
Font
```

Los objetos de WXGDI estan enlazados entre si formando una lista de objetos, cada uno de ellos tiene un header el cual indica el tipo de objeto y el HANDLE al próximo objeto.

```
typedef struct
{
  WORD   nextObject;
  WORD   typeObject;
} WXObject;
```

Al ser X Window un sistema de ventanas basado en red todos estos recursos están en el server para reducir el trafico sobre la conexión X entre el cliente y el server.

5. 2. 3. 1. 1. Device Context

Cuando se quiere realiza cualquier requerimiento gráfico sobre un determinado dispositivo gráfico en Windows primero se debe obtener un HANDLE a un *Device Context* (DC), este DC contiene atributos que determinan como se comporta esta acción sobre el dispositivo.

Los atributos que mapeamos de un DC son:

```
Background color
Bitmap
Brush
Brush origin
Clipping region
Color palette
Current pen position
Drawing mode
Font
Pen
Polygon-filling mode
Text color
```

Definimos un objeto llamado *WXDC* que contiene toda la información asociada a un DC. Este objeto tiene los siguientes atributos:

```
typedef struct
{
   WXObject  header;
   HBRUSH    hWXBrush;
   HBITMAP   hWXBitmap;
   HPEN      hWXPen;
   HPALETTE  hWXPalette;
   POINT     PenPosition;
   POINT     BrushPosition;
   WORD      kMode;
   DWORD     TextColor;
   FONT      hWXFont;
   GC        WXGC;
   HANDLE    hWXWin;
   Window    XWin;
   Display   *dpy;
} WXDC;
```

El atributo *WXObject* es el header del objeto, el cual entre otras cosas indica el tipo del objeto. también tiene atributos que asocian este objeto a otros como *WXBrush, WXBitmap, WXPen,WXPalette, WXFont,* los cuales seran explicados posteriormente.

Con el atributo *BrushPosition* mapeamos la posición del brush, de igual forma usamos el atributo *PenBrush* para el pen. Con el GC asociado al *WXDC* mediante el atributo *WXGC* mapeamos los campos: Background color, Clipping region, Drawing mode, Polygon-filling mode del Device Context.

Para que el cliente le pase un requerimiento gráfico al server se necesita conocer el identificador de la conexión entre ellos. Este identificador esta guardado en el atributo *dpy* del *WXDC*. El atributo *Xwin* indica la ventana X asociada al requerimiento gráfico, por ejemplo para dibujar un rectángulo en Windows se usa la función *Rectangle* la cual se le pasa como parámetros el HANDLE a un DC y dos coordenadas, en X además se necesita pasar la ventana X donde se dibujara el rectángulo y el identificador de la conexión.

5. 2. 3. 1. 2. Brush

Los *Brushs* (Pinceles) son objetos los cuales se usan generalmente para llenar con un determinado color o pattern (patrón de bits) una figura gráfica. (circulo, rectángulo, etc). Definimos un objeto llamado *WXBrush*, el cual contiene información para realizar el mapeo con el objeto Brush de Windows. En X no existe el objeto brush, pero existen atributos en el GC que cumplen con la misma funcionalidad.

En Windows existen seis tipos de brush por eso definimos un header para poder clasificarlos ya que actúan de distintas maneras. La estructura de un objeto *WXBrush* es:

```
typedef struct
{
  WXObject  header;
  WORD      typeBrush;
} WXBrush;
```

El *header* indica el tipo de objeto, el atributo *typeBrush* indica el tipo de brush que puede ser:

```
BS_DIBPATTERN, BS_HATCHED, BS_HOLLOW, BS_PATTERN, BS_NULL, BS_SOLID
```

Los seis tipos de brush estan definidos por estructuras que mantienen información referente a cada uno de ellos, el mapeo a Xlib se realiza en el código de la función que usan el brush.

Para cada función de creación de un brush se creara un objeto *WXBrush* con un determinado estilo en el cual tendrá toda la información necesaria para su uso, luego en el momento de usarlo por medio de la función *WXSelectObject* del WXGDI la información del brush se modificara en los atributos del GC asociado al WXDC. Esto sucede porque en Windows se pueden crear varios Brush y luego seleccionar cualquiera de ellos mientras que en X Window solo se puede seleccionar uno a la vez.

5. 2. 3. 1. 3. Pen

Los Pens son objetos que se usan para dibujar líneas, rectángulos, círculos, etc. En WXGDI definimos el objeto *WXPen*, el cual contiene información para realizar el mapeo con Xlib. En Xlib no existe el objeto pen, en cambio existen atributos en el recurso GC que producen la misma funcionalidad que los pens en Windows. La estructura del WXPen es:

```
typedef struct
{
  WXObject      header;
  int           width;
  int           style;
  unsigned long color;
  unsigned char *dash_list;
  int           dash_list_length;
} WXPen;
```

El atributo *header* especifica el tipo de objeto, un pen esta formado por ancho, color y estilo. El atributo *width* indica el ancho, el atributo *color* indica el color del pen y el con los tres últimos atributos se define el estilo del pen, es decir con el atributo *style* indicamos el estilo del pen el cual puede ser:

```
PS_SOLID, PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT, PS_NULL,
PS_INSIDEFRAME
```

Los atributos **dash_list, dash_list_length* se usan para generar los distintos tipos de pen porque en Xlib solo existen tres estilos, que son LineSolid, LineOnOffDash, LineDoubleDash. pero usando una función de X llamada *XSetDashes* nos permite generar los otros tipos de pen.

El mapeo se realiza en el código de la función *WXSelectObject* del WXGDI, la cual selecciona un pen en el WXDC para luego usarlo. No se modifica directamente el GC cuando se crea el pen porque en Windows se pueden crear varios pens y luego usarlos seleccionando alguno en el DC, en X no sucede lo mismo porque un GC solo tiene un atributo para el color del pen y otro para el ancho, lo que ocasiona que guardemos los atributos de los otros pens que creamos en algún otro lugar para luego al realizar algún requerimiento gráfico se modifiquen los atributos del GC con estos valores. Para solucionar esto se creo el objeto WXPen que almacenan estos valores para luego usarlos en la selección del objeto WXPen en el WXDC.

5. 2. 3. 1. 4. Font

En ambos sistemas de ventanas se pueden usar *Fonts*, Xlib tiene una base de fonts predefinidos, cada font es un bitmap, similar al tipo de font *raster* de Windows. Además Windows tiene dos tipos mas de font *vectorizados* y *true type*, los cuales no son contemplados por X. El mapeo se realiza usando los fonts predefinidos en los dos sistemas

```
typedef struct
{
  WXObject  header;
  LPCSTR    nameFont;
  Font      idFont
} WXFont;
```

El atributo *header* especifica el tipo de objeto, el atributo *nameFont* indica el nombre del font en Windows y en *idFont* es el identificados del font creado en X.

El GDI de Windows ofrece una variedad de stock fonts. Para poder crear un font se usa la función *WXGetStockObject*. Esta función devuelve el handle a un font, luego la aplicación selecciona el font en el WXDC. En X es similar, la función se llama *XLoadFont*. Cuando la aplicación llama a la función *WXDeleteObject*, esta llamara a la función *XFreeFont*.

5. 2. 3. 1. 5. Palette

Un objeto *Palette* (paleta) en Windows es un buffer de colores entre la aplicación y el sistema, en Windows se pueden crear paletas de colores y luego seleccionarla en la paleta por hardware. En X Window existe un concepto similar llamado *colormap* que esta formada por entradas llamadas *colorcell* que contienen la intensidad de cada color.

```
typedef struct
{
  WXObject    header;
  Colormap    colors;
} WXPalette;
```

El atributo *header* especifica el tipo de objeto y el atributo *colors* especifica un colormap. Windows para usar los colores maneja la paleta del sistema (*hardware*), la cual contiene los colores que son visualizados en el screen. Cuando una aplicación no crea ninguna paleta lógica, Windows usa la paleta por default, es decir usan los colores predefinidos en la paleta del sistema. Cuando la aplicación usa sus propios colores primero debe crear una paleta lógica con los colores a usar. La aplicación temporariamente reemplaza la paleta del sistema por la paleta lógica, la cual permite que la aplicación use los colores de la paleta lógica.

En Xlib existe el concepto de "colormap". El colormap contiene "colorcells" las cuales especifican el valor RGB, para acceder a cada entrada del colormap se usa como índice el valor del pixel. El numero de bits por pixel determina la cantidad de entradas disponibles en un colormap y el numero de colores que se pueden mostrar simultáneamente. X maneja los múltiples colormaps guardándolos en memoria. El proceso de instalación (*installing*) es el movimiento de un colormap virtual a un colormap por hardware. Cuando una aplicación crea un colormap virtual, este colormap debe estar indicado en los atributos de la ventana top-level para que el Window Manager pueda buscar el colormap a instalar.

5. 2. 3. 1. 6. Bitmap

Los *Bitmaps* de Windows son mapas de bits. En WXGDI definimos un objeto llamado *WXBitmap*, el cual contiene información del bitmap como el ancho. largo, profundidad y el conjunto de bytes que lo forman. En X usamos los *pixmap*, que son almacenados en el server. Un pixmap es un conjunto de pixels que no tienen asociado ningún colormap como los bitmaps en Windows, por eso el pixmap usa el colormap asociado con la ventana donde va a ser copiado.

```
typedef struct
{
   WXObject  header;
   char      *pBitmap;
   int       width;
   int       height;
   UINT      depth;
   Pixmap    idPixmap;
   int       flag;
} WXBitmap;
```

El atributo *header* especifica el tipo de objeto, el atributo *pBitmap* es un puntero al conjunto de bytes que forman el bitmap. El atributo *width*, *height* y *depth* indican el ancho, largo y profundidad del bitmap, el atributo *idPixmap* contiene el identificador del pixmap asociado al bitmap. El pixmap no es creado en la función *WXCreateBitmap* porque para crear un pixmap en Xlib necesitamos asociarle una ventana X y el identificador de la conexión entre el cliente y el server, los cuales estan en el WXDC. La función *WXCreateBitmap* no recibe como parámetro ningún WXDC, por lo que el pixmap se crea al ser invocadas las funciones *WXSelectObject* o *WXFillRect*. El atributo *flag* es el que indica si se creo el pixmap o no. Este atributo se modifica cuando se crea un pixmap en el server.

5. 2. 3. 1. 6. 1. 1. Conversión de un Bitmap a un Pixmap

Un bitmap tiene un formato diferente a un pixmap. En un bitmap existen dos headers, el primero es llamado *BitmapFileHeader* y especifica el tipo y tamaño del archivo, el segundo header denominado *BitmapInfoHeader* contiene información sobre el bitmap como las dimensiones y tipo de compresión. A continuación esta la tabla de colores del bitmap y después estan los bytes que forman

el bitmap. Estos datos consisten de un array de bytes formado por filas consecutivas denominadas *scan lines*. Cada fila consiste de bytes que representan pixels en la scan line. El numero de bytes de una *scan line* depende del formato de color y el ancho de los pixels del bitmap. Los bitmap estan almacenados de fin a principio eso significa que el primer bit del array es el ultimo de la imagen en pantalla.

Por ejemplo:

// Información asociada al archivo

```
BITMAPFILEHEADER
   Type           19778
   Size           3118
   Reserved1      0
   Reserved2      0
   OffsetBits     118
```

// Información asociada al bitmap

```
BITMAPINFOHEADER
   Size           40
   Width          80
   Height         75
   Planes         1
   BitCount       4
   Compression    0
   SizeImage      3000
   XPelsPerMeter  0
   YPelsPerMeter  0
   ColorUsed      16
   ColorImportant 16
```

// Colores del bitmap

```
COLORTABLE
                 Blue    Green     Red     Unused
[00000000]       84      252       84      0
[00000001]       252     252       84      0
[00000002]       84      84        252     0
[00000003]       252     84        252     0
[00000004]       84      252       252     0
[00000005]       252     252       252     0
[00000006]       0       0         0       0
[00000007]       168     0         0       0
[00000008]       0       168       0       0
[00000009]       168     168       0       0
[0000000A]       0       0         168     0
[0000000B]       168     0         168     0
[0000000C]       0       168       168     0
[0000000D]       168     168       168     0
[0000000E]       84      84        84      0
[0000000F]       252     84        84      0
```

// Bytes que forman la Imagen

```
IMAGEN
   .
   .
```

El formato de un pixmap es mas simple, esta dado por un conjunto de bytes y sus dimensiones. Por Ejemplo:

```
// Dimensiones del pixmap
#define pixmap_width    40
#define pixmap_height   40
#define pixmap_depth    1

// Bytes que forman el pixmap
static char pixmap_bits [] = {
   0xc3, 0xc3, 0x7f, 0x00, 0x78, 0x00, 0x00, 0x00, 0x00, 0xc0, 0x00, 0x00,
0x80, 0x00, 0x40, 0x00, 0x24, 0x00, 0x00, 0x00, 0x00, 0x00,
   .
   .
   .
   0x03, 0x00, 0x70, 0x00, 0x00};

Leer headerInfoFile
// Verificar si el archivo es un Bitmap
if (headerInfoFile.bfType = BM) then
   Leer headerInfoBitmap.
      // Leo el offset desde el header a los datos
      principioBitmap = headerInfoFile.bfOffsetBits
      // Leo el ancho en pixels
      anchoBitmap = headerInfoBitmap.biWidth
      // Leo el largo en pixels
      largoBitmap = headerInfoBitmap.biHeight
      // Leo el numero de bits que forman un pixel
      tamanoPixelBitmap = headerInfoBitmap.biBitCount
   Calcular el ancho de una linea en bytes.
   Leer la tabla de colores.
   Leer todas las lineas de datos.
else
   Este Archivo no es un BITMAP.
end if
```

Para realizar la conversion de un bitmap a un recurso de Xlib se deben tener en cuenta los colores, la imagen y el tamaño.

Un bitmap tiene asociado colores, en un pixmap no sucede lo mismo, un pixmap usa los colores asociados a la ventana donde va a ser copiado. A partir del bitmap se crea un colormap con los colores y cuando el pixmap sea dibujado en una ventana, se asocia a esta el colormap creado.

Los bytes de la imagen de un bitmap estan guardados de fin a principio, en un pixmap los bytes son guardados de principio a fin con la salvedad de que cada byte esta guardado en forma inversa (*bit-order*).

Un pixmap en X Window es guardado en el server, por ese motivo no hay que abusar de la creación de pixmaps. Una solución a este problema es crear una *imagen* en Xlib que es capaz de almacenar un pixmap en la memoria del cliente y no en el server. Xlib provee funciones permiten transferir un pixmap a una imagen y viceversa.

5. 2. 3. 1. 7. Cursor

En Windows existe el objeto cursor el cual nos permite usar distintos cursores en una aplicación, Xlib tiene un recurso llamado de la misma manera (cursor), la diferencia entre los cursores de Windows con los de X, es que en el primer caso los cursores están asociados a una aplicación, con la función *SetCursor* se asocia un cursor a una aplicación Windows, en X es diferente, los cursores están asociados a nivel ventana en el server.

```
typedef struct
{
  WXObject      header;
  LPCSTR        nameCursor;
  unsigned int  shapeCursor;
  Cursor        cursor;
} WXCursor;
```

El atributo *header* especifica el tipo del objeto, el nombre del cursor en Windows es indicado por el atributo *nameCursor*. El atributo *shapeCursor* identifica la forma del cursor en Xlib y el atributo *cursor* contiene el identificador del cursor en Xlib. Mapeamos todos los cursores predefinidos de Windows con los predefinidos en Xlib.

5. 2. 3. 1. 8. Región

El concepto de *región* en Xlib y en Windows es el mismo, en el WXGDI se definió el objeto *WXRegion*. En ambos sistemas de ventanas se usan para modificar la región de clipping en el WXDC.

Una región es un área del screen que es una combinación de rectángulos, polígonos y elipses. Una región de clipping se usa para seleccionar una región dentro del Device Context, la cual indica la zona donde se puede dibujar o pintar. Los requerimientos gráficos fuera de ella no tendrán ningún tipo de efecto.

```
typedef struct
{
  WXObject  header;
  HRGN      hRgn;
  Region    idRgn;
} WXRegion;
```

El atributo *header* identifica el tipo del objeto, el atributo *hRgn* contiene un HANDLE a la región en Windows y el atributo *idRgn* contiene la identificación a la región en Xlib.

5. 2. 4. Intercepción del API de Windows.

A continuación se presentan 2 posibles soluciones al problema de interceptar los llamados de funciones del API de Windows.

5. 2. 4. 1. INT3

Esta aproximación consiste en patchear los segmentos de código (en memoria) de los módulos de Windows que exportan las funciones a interceptar escribiendo el código de operación INT3 (0xCC) en el primer byte del código de la función, proveyendo un manejador para esta interrupción. Una vez que el manejador toma el control, hay que invocar a nuestra función correspondiente.

Con esta aproximación se interceptarán los llamados que haga cualquier aplicación, no sólo las apps. WExX; por lo tanto, si la función fue invocada por una app. Windows standard, ejecutar la función de Windows.

En esta solución se presentan los siguientes problemas:

1) Como saber la función del API que se invocó.
2) Determinar si la tarea que invocó la función es una aplicación Windows standard o una aplicación nuestra.
3) Escribir un segmento de código desde modo protegido.
4) Los segmentos de código en Windows tienen el atributo DISCARDABLE; son estos los segmentos que el KERNEL descarta de memoria (ver *Administración de memoria en Windows en modo Enhanced*) para cargar nuevos segmentos (de código, datos, recursos, etc.). Posteriormente cuando sea necesario el segmento descartado, el KERNEL lo recrea del archivo (.EXE, .DLL, etc.), por lo tanto nuevamente hay que escribir el código de operación INT3.

Soluciones a los problemas anteriores:

Cuando el manejador de la interrupción 3 toma el control, en el stack se encuentra la dirección de retorno de la interrupción la cual fue apilada por el procesador. Esta dirección es la dirección de la función del API de Windows más 1 byte (el código de operación de la INT3 ocupa 1 byte). Esta dirección hay que compararla con las entradas de la **tabla de direcciones de funciones del API** que lleva el objeto **WXSystem** de WExX. Esta tabla es creada y cargada a momentos de creación del objeto WXSystem. Las entradas de esta tabla tienen la forma:

```
(módulo, dirección de la función)
```

La dirección de la función se obtiene con la función del KERNEL **GetProcAddress()**.

Para determinar el tipo de la tarea que invocó la función se utiliza la función del KERNEL **GetCurrentTask()** y se compara el hTask retornado por esta función con los **hTask** - o **hInstance** - de la tareas que corren bajo WExX)

A continuación se presenta la función para escribir un byte en una posición de memoria perteneciente a un segmento de código:

```
BYTE ponerByte(FARPROC lpfn, BYTE byInst)
{
```

```
    BYTE byPrev;
    HANDLE hData, hDataRaw;

    /* Simplemente para segurarse que el segmento que contiene la dirección lpfn esté en  meoria. Si el
    segmento no está en  memoria, el KERNEL lo carga*/
    GetCodeHandle(lpfn);

    /* Para hacer que el segmento permanezca en la misma dirección  de memoria física */
    GlobalPageLock(FP_SEG(lpfn));

    /* Alocar un selector para leer/escribir un segmento de código
    AllocCStoDSAlias es una función NO DOCUMENTADA del KERNEL de Windows*/
    hDataRaw = AllocCStoDSAlias(FP_SEG(lpfn));

    /* Ajustar el nivel de privilegio del selector */
    hData = (hDataRaw & 0xfffc) | 3;

    /* Escribir byInst */
    byPrev = *((LPBYTE) MK_FP(hData, FP_OFF(lpfn)));
    *((LPBYTE) MK_FP(hData, FP_OFF(lpfn))) = byInst;

    /* Unlock el segmento */
    GlobalUnlock(FP_SEG(lpfn));

    /* Liberar el alias del selector */
    FreeSelector(hDataRaw);

    return ( byPrev );
}
```

Para la solución del problema 4 hay que saber el momento en que se carga un segmento en memoria y ver si este segmento pertenece a alguno de los módulos de Windows en cuestión y si en el segmento hay alguna función del API.

Windows (específicamente una DLL que extiende a Windows, TOOLHELP.DLL) provee **Notification Handlers** que son básicamente funciones callbacks que registra una aplicación Windows y que Windows invoca cuando se producen eventos muy especiales como ser la carga de un segmento de un módulo desde disco (NFY_LOADSEG). Esta misma notificación es usada por los Debuggers de Windows para poder reinsertar breakpoints que se perdieron cuando se descartó el segmento.

Otros tipos de eventos incluyen el comienzo de ejecución de una aplicación o tarea (NFY_STARTTASK), la terminación de una tarea (NFY_EXITTASK) etc.

La función callback que registra la aplicación vía la función **NotifyRegister()** es invocada por el KERNEL de Windows y le pasa como parámetro un puntero a una estructura del siguiente tipo:

```
#include <toolhelp.h>

typedef struct tagNFYLOADSEG {  /* nfyls */
    DWORD   dwSize;
    WORD    wSelector;
    WORD    wSegNum;
    WORD    wType;
```

```
    WORD    wcInstance;
    LPCSTR  lpstrModuleName;
} NFYLOADSEG;
```

5. 2. 4. 2. Resolución de las referencias que deja el linker.

Esta aproximación consiste en resolver las referencias que dejó el linker en el .EXE o .DLL, de la misma manera que lo hace el loader de Windows.

Estas referencias están en la **tabla de reubicación** (en .EXE o .DLL), la cual está organizada por segmentos.

Sea E1 = (**off1**, **mod1**, **#f1**) una entrada de esta tabla para el segmento lógico **s1**.

donde, **off1** es un offset dentro de **s1**

mod1 es un nombre de módulo

#f1 es el ordinal de la función que exporta **mod1**.

El segmento lógico **s1** tiene asociado un selector **sel1** que le asignó el loader de Windows.

A partir del nombre de módulo **mod1** buscar la **MDB** del módulo (las **MDBs** están en una lista encadenada en memoria).

Con el ordinal de la función **#f1** acceder a la **tabla de entradas** (entry table) del módulo.

Sea E2 = (**#f1**, **s2**, **off2**) una entrada de esta tabla.

donde, **#f1** es el ordinal de la función que exporta **mod1**

s2 es el # de segmento lógico en el que se encuentra **#f1**

off2 es el offset de **#f1** en **s2**

Con el # de segmento lógico **s2** acceder a la **tabla de segmentos** del módulo.

Sea **E3** = (**s2**, **sel2**) una entrada de esta tabla,

donde, **s2** es el # de segmento lógico en el que se encuentra **#f1**

sel2 es el valor del selector que le asignó el loader al segmento cuando cargó el módulo.

Con el valor de selector **sel2** y el offset **off2** (**sel2:off2**) patchear el segmento lógico **s1** en el offset **off1**, es decir la dirección de memoria (**sel1:off1**), dirección que no pudo resolver el linker.

Lo mismo hay que hacer para aquellos módulos ejecutables (generalmente .DLLs) con los que se linkeen nuestras aplicaciones (estas DLLs también hacen llamados a funciones del API de Windows), ya sea DLLs cargadas implícitamente (inmediatamente antes de comenzar la ejecución de la aplicación) o dinámicamente (en cualquier momento durante la ejecución de la aplicación), vía la función GetProcAddress().

Observación: Las aplicaciones Windows standards que también se linkeen con estas .DLLs sufrirán el overhead de la intercepción.

El 'patching' hay que hacerlo inmediatamente antes que comience la ejecución de una aplicación; para esto utilizaremos un 'notification handler' del tipo **NFY_STARTTASK**.

5. 2. 5. 'Browsing' y ejecución de las aplicaciones WExX.

Como mencionamos en *5. 1. 5*, para la implementación del mecanismo de 'browsing' se portará a Windows la máquina de estados del display manager xdm que implementa XDMCP. No se tendrán en cuenta las considereaciones de seguridad contempladas en el diseño de xdm.

No se tendrán en cuenta las consideraciones de seguridad contempladas en el diseño de xdm en lo que hace a la autorización, fundamentalmente por que no forma parte de los objetivos de la solución y porque su tratamiento es inherentemente complejo. Debido a que es el display manager el que decide si se usará algún mecanismo de autorización, y que mecanismo en caso que decida utilizar alguno, es posible implementar la máquina de estados de XDMCP del lado del display manager sin mecanismos de autorización.

Tampoco se considerará la autenticación de XDMCP por las mismas razones. Esto sin embargo, a diferencia de la autorización, es una limitación, ya que el mecanismo de autenticación es exigido por el X Server, y este puede decidir no ser administrado por un display manager que no implemente algún mecanismo de autenticación[3].

Por lo tanto, en cada nodo de la red en el cual existan aplicaciones WExX a ejecutar, existirá un este módulo que implemente XDMCP.

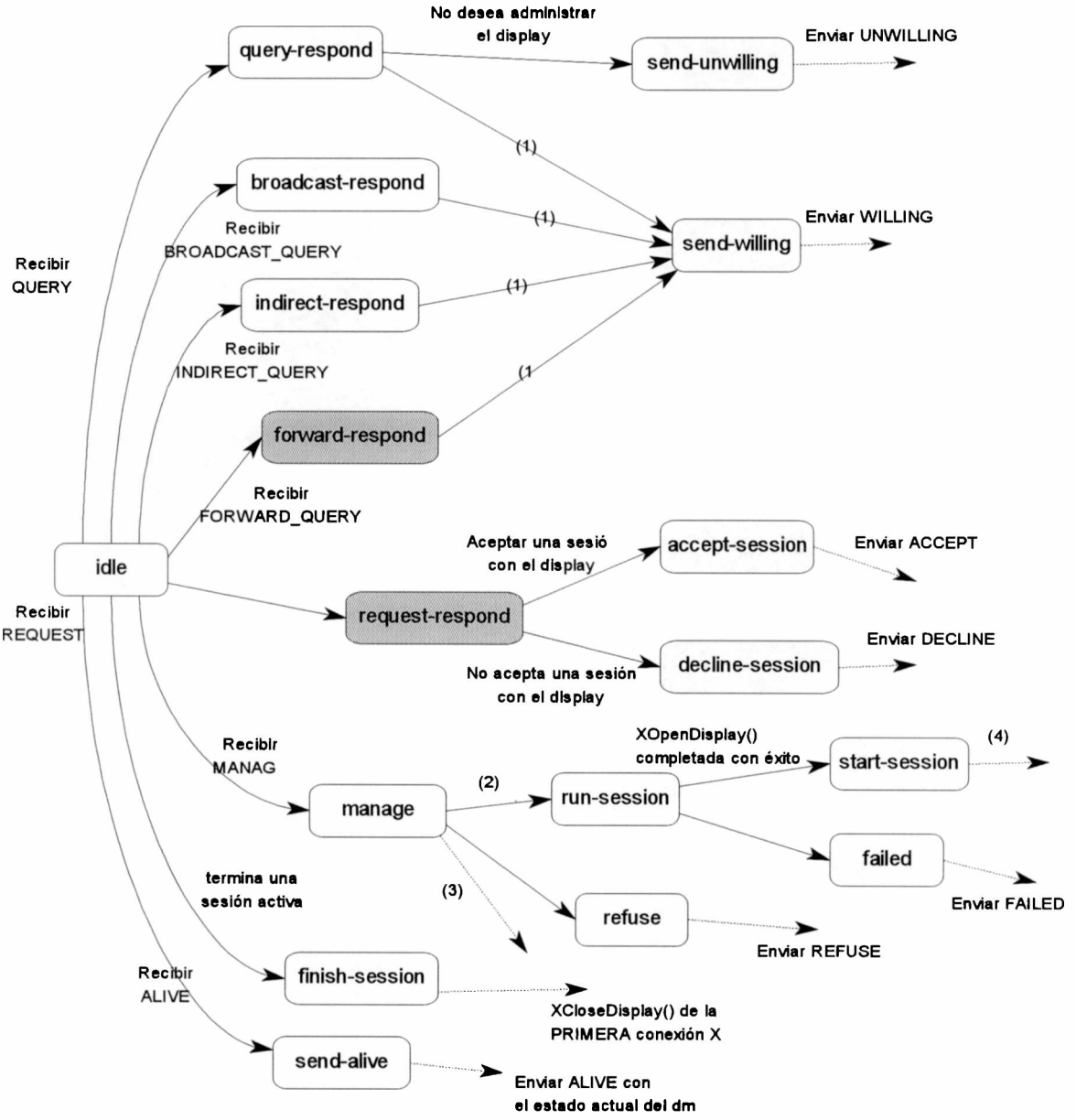
Como también señalamos en *5. 1. 5*, con XDMCP "se llega sólo hasta el nodo", por lo tanto para poder elegir la/las aplicaciones WExX a correr ("llegar hasta una aplicación") hay que agregar una interacción más entre el X Server y el componente de software. La idea es que nuestro módulo que implementa XDMCP termine ejecutando en Windows un cliente X verdadero el cual establece una conexión X con el X Server (la cual será el 'acknowledge' del paquete *Manage* que envía el X Server) y que por esta conexión presente una ventana X con la lista de las aplicaciones WExX configuradas en el nodo en la cual se podrá elegir la/las aplicaciones WExX a correr. Esta ventana X estará presente durante toda la sesión.

Una manera más simple es establecer una conexión X con el X Server que simplemente sea el 'acknoledge' al paquete Manager y lanzar a correr todas las aplicaciones WExX configuradas en el nodo.

A continuación presentamos la parte de la máquina de estados de XDMCP que se recompiló. Lo único que no se recompiló es la resolución de las consultas *IndirectQuery*.

[3] Dos de los X Servers para Windows que hemos probado soportan XDMCP, y ambos, durante la conversación XDMCP, sugieren la utilización del mecanismo XDM-AUTHENTICATION-1, pero no terminan la conversación cuando nuestro display manager le indica que no soporta este mecanismo. Lo mismo ocurre con el X Server de SCO y con el de Linux.

Máquina de estados que implementa XDMCP en Windows.



(1) El dm desea administrar el display.

(2) SessionID del paquete hace matching con SessionID almacenado.

(3) SessionID hace matching con SessionID de la sesión en proceso de arranque o con la sesión actualmente activa.

(4) Arranca todas las Apps Windows Clientes X configuradas en el nodo.

Transición al idle.

Estados no implementados.

5. 2. 6. 'Start-up' de una aplicación WExX.

Es necesario saber el momento en que comienza la ejecución de una aplicación WExX para poder realizar las tareas de 'start-up' necesarias como ser la creación de un objeto de la clase WXClient entre otras.

Para resolver el problema de conocer cuando comienza una aplicación sobre WExX utilizaremos **notifications handlers** (funciones callbacks), herramienta provista por la librería *Toolhelp* de Windows. En la inicialización de WExX (en la clase WXSystem) se registra un handler de notificación del tipo **NFY_STARTTASK**. Este handler de notificación será invocado por Windows antes que se ejecute la primer instrucción de toda aplicación Windows[4].

Lo primero que hay que hacer en el handler de notificación es determinar si la aplicación se ejecutará sobre WExX. Para esto hay que ver si entre los argumentos de la línea de comando que arrancó la aplicación se encuentra el switch **-wx**, el cual indica que la aplicación deberá correrse sobre WExX.

Si el esquema utilizado es **intercepción a nivel de aplicación**, hay que hacer 'patching' de los segmentos de código de la aplicación, ya cargados por el loader, y de los segmentos de las DLLs que referencia la aplicación y que ya están cargadas en memoria, en caso que sea necesario. Para hacer 'patching' del código de las DLLs que aún no han sido cargadas se utiliza un handler de notificación **NFY_STARTDLL**, el cual es invocado por Windows antes del código de 'start-up' de la DLL.

Si el esquema utilizado es **intercepción a nivel Windows**, entonces a momento de inicialización de WExX si hizo el 'patching' de los segmentos de código de los módulos de Windows. Por lo tanto, cuando una aplicación invoque a una función del API de Windows y WExX tome el control, hay que determinar si la aplicación invocante es una aplicación que está corriendo sobre WExX o no. Para esto se utiliza la función **GetCurrentTask()**.

En la Parte 5 - *Intercepción del API de Windows* se describe en profundidad los dos esquemas de intercepción del API antes mencionados.

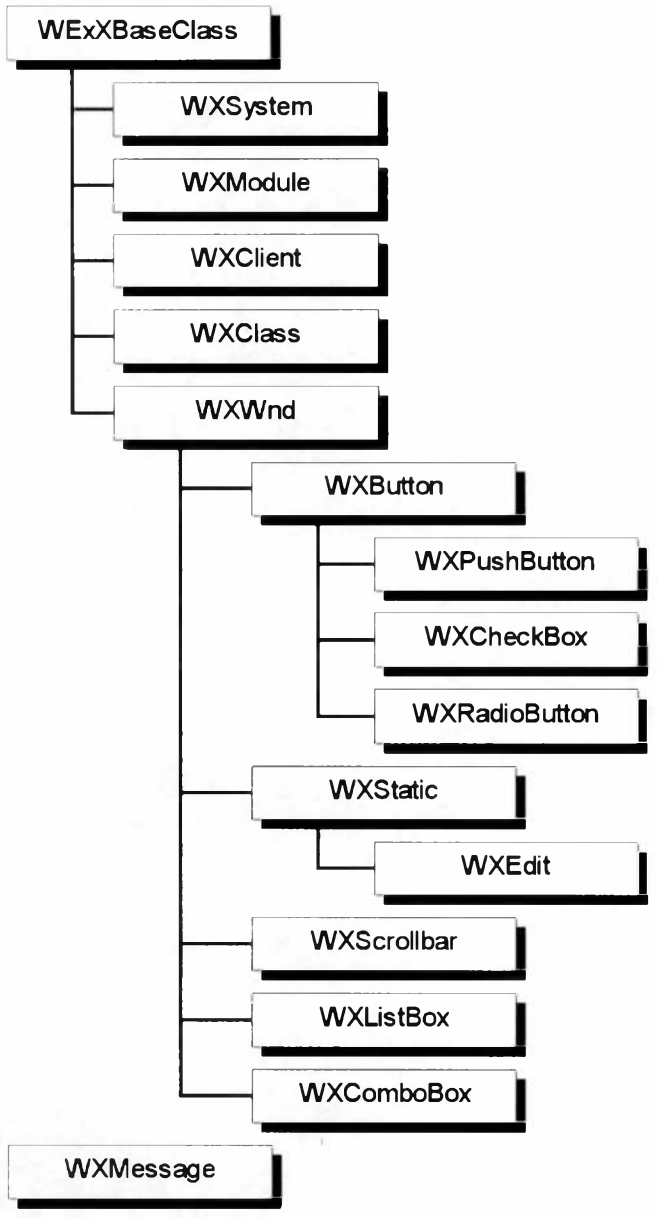
[4] Los debuggers usan esta notificación para poner un breakpoint para obtener el control en la primera instrucción del programa. Este handler de notificación es invocado por la función de Windows **LoadModule()**.

5. 3. Diseño Detallado

En este capítulo daremos el diseño detallado de WExX el cual comprende la jerarquía de clases implementando la traslación del sistema de ventanas y mensajes, y la implementación de los objetos y funciones gráficas.

Para cada clase se dará un comentario de su funciónalidad, es decir con que entidad de Windows se relaciona y que tareas debe realizar para lograr la traslación del sistema de ventanas y mensajes, y luego la definición con un breve comentario acerca de cada miembro y método de la clase. También se dará la especificación de los métodos mas importantes de cada clase, y cuando parezca necesario la implementación de los mismos.

5. 3. 1. Jerarquía de clases de WExX



5. 3. 1. 1. Clase WXSystem

La clase **WXSystem** realiza toda la tarea de inicialización de la libreria WExX. Existirá solo un objeto de ésta clase y será creado en el momento de carga de la libreria (libmain). Esta clase inicializa principalmente Windows Sockets para permitir luego su utilización por las tareas que se ejecuten sobre WExX. Además contendrá un arreglo de los módulos que se ejecutan sobre WExX, y un arreglo de clases de ventanas gloabales (ya sean las registradas por Windows como las registradas por las aplicaciones con la opción de clase global).

```
class WXSystem
{
  // Data Members.
  protected :
    // Modulos cargados.
    PArray          wxModules;
    // Aplicaciones ejecutandose.
    PArray          wxInstances;
    // Clases defaults de Windows + Clases registradas por aplicaciones con
    // el estilo CS_GLOBALCLASS.
    PArray          wxGlobalClasses;

  // Member Functions.
  public :
    // Constructor y Destructor.
    WXSystem();
    ~WXSystem();

    // Registración de Clases Globales

    // Registrar una clase.
    WXError     WXRegisterClass( const WNDCLASS FAR* lpwc );
    // Deregistrar una clase.
    WXError     WXUnRegisterClass( const WNDCLASS FAR* lpwc );

  protected :
    // Agrega una clase al container de clases globales.
    WXError     WXAddGlobalClass( PWXClass newClass );
    // Desregistrar una clase.
    WXError         WXDelGlobalClass( PWXClass );
    // Obtener un objeto WXClass a partir del nombre de una clase.
    PWXClass        WXGetGlobalClass( char * className );

    // Modulos
  public :
    // Agregar un Modulo.
    WXError         WXAddModule( PWXModule newModule );
    // Sacar un Modulo.
    WXError         WXDelModule( PWXModule );
    // Obtener un objeto WXModule a partir del hModule.
    PWXModule       WXGetModule( HMODULE hModule );
};

// Constructor y Destructor.
```

```
WXSystem::WXSystem()
{
   /*
   Inicialización de Windows Sockets mediante la función WSAStartup de la API de Windows
   Sockets contenida en la DLL WinSock.
   Inicialización de miembros "containers".
   Registrar las clases predefinidas de Windows como clases globales :
   BUTTON, COMBOBOX, EDIT, LISTBOX, SCROLLBAR y STATIC.
   */
}

WXSystem::~WXSystem()
{
   // Deinicializar Windows Sockets (WSACleanup).
   // Depuración.
}

// Registrar una clase.
ATOM WXSystem::WXRegisterClass( const WNDCLASS FAR* lpwc )
{
   /*
   Verificar que la clase no haya sido registrada.
   Crear Objeto WXClass.
   Si la clase es APLICATIONGLOBAL
      Agregarla dentro de las clases globales.
   sino
      Hallar modulo que esta registrando la clase.
      Decirle al modulo correspondiente que registre la clase.

   Retornar un handle que identifique univocamente la clase.
   */
}

// Deregistrar una clase.
// Antes de llamar a esta función la aplicación deberia destruir todas las ventanas creadas con la
// clase especificada.
// Obs.: las clases predefinidas de Windows no pueden ser deregistradas.
BOOL WXSystem::WXUnRegisterClass(LPCSTR classname,HINSTANCE hinst)
{
   /*
   Obtener el puntero a la clase (ya sea desde el container de clases globales u obteniendolo por
      medio de los módulos).
   Verificar que no haya ventanas creadas de esa clase (WXGet).
   Borrar la clase.
   Retornar TRUE o FALSE segun se haya podido borrar la clase o no.
   */
}
```

5. 3. 1. 2. Clase WXModule

La clase **WXModule** es la equivalente al modulo en Windows, y mantiene un arreglo de las intancias que se estan ejecutando del modulo, y un arreglo de clases de ventanas registradas por ellas (recordemos que las clases son registradas una sola vez por la primera instancia de un modulo y luego son compartidas por todas las instancias de este)

```
class WXModule
{
  // Data Members
  protected :
    // HANDLE que identifica al modulo.
    HMODULE         hModule;
    // Instancias ejecutandose.
    PArray          wxInstances;
    // Clases particulares de un modulo.
    PArray          wxClasses;

  // Member Functions
  public :
    // Constructor y Destructor.
    WXModule( HMODULE hModule );
    ~WXModule();

    // Manejo de registración de Clases.

    // Registrar una clase.
    WXError         WXAddClass(PWXClass newClass);
    // Desregistrar una clase.
    WXError         WXDelClass(PWXClass);
    // Obtener un objeto WXClass a partir del nombre de una clase.
    PWXClass        WXGetClass(char * className);

    // Administración de Aplicaciones

    // Agregar una Aplicación
    WXError         WXAddInstance(PWXClient);
    // Sacar un Aplicación.
    WXError         WXDelInstance(PWXClient);
    // Obtener un objeto WXClient a partir del hInstance.
    PWXClient       WXGetInstance(HANDLE hInstance);
};
```

5. 3. 1. 3. Clase WXClient

La clase **WXClient** es una de las clases mas importantes de la jerarquia de ventanas de WExX. Es la encargada de mantener toda la información referida a una instancia de una aplicación y fue llamada WXClient por el hecho de que toda instancia de una aplicación Windows se convierte sobre WExX, en un cliente en el sentido de X Window.

Entre los miembros mas relevantes, la clase WXClient mantiene la conexion con el X Server (dpy), que nos permite rutear al X Server correcto los requerimientos de la aplicación. Tambien implementa la cola de mensajes de la aplicación, ya que como dijimos, los mensajes ya no seran puestos en la cola de mensajes de Windows sino que seran procesados totalmente por WExX. Separadamente se tiene una cola de mensajes de repintado (WM_PAINT) ya que estos mensajes no son puestos en la cola de la aplicación sino que son retornados en un GetMessage luego de que no hay mensajes pendientes de mayor prioridad. Esta ultima cola de mensajes es actualizada al recibir el evento Expose desde el X Server como fue explicado en el mapeo de eventos de X Window al sistema de mensajes de Windows.

```
class WXClient
{
  // Data Members.
  protected :
    // Handle que identifica al WXClient.
    HANDLE          hInstance;
    // Conexion con el XServer.
    Display         xDpy;
    // Tabla de relaciones entre ids. de ventanas X Window y ventanas WExX.
    PArray          wxTable;
    // Modulo al que pertenece la Aplicación.
    PWXModule       wxModule;
    // TopLevels y Poupus de la Aplicación.
    PArray          wxTopLevels;
    // Mensajes posteados ( PostMessage() ) por la misma aplicación.
    PQueue          wxPostedMsgQueue;
    // Ventanas de la aplicación que tienen pendientes mensajes WM_PAINT.
    PQueue          wxNeedPaint;
    // Miembro indicando algun tipo de error, por ejemplo al intentar comunicarse con el
    // XServer.
    WXError         wxError;

  // Member Functions.
  public :
    // Constructor y Destructor
    WXClient(HINSTANCE intance, PWXModule aModule, char *displayName);
    ~WXClient();

    // Manejo de Ventanas.

    // Agregar una ventana al container de ventanas TopLevel de la aplicación.
    BOOL        WXAddTopLevel( PWXWnd newWindow );
    // Borrar una ventana TopLevel.
    BOOL        WXDelTopLevel( PWXWnd newWindow );
    // Obtener un objeto WXWnd a partir de un identificador unico de este
    // (utiliza la tabla de mapeo XId - PWXWnd).
```

```
    PWXWnd      WXGetWindow( XID XWindow );

    // Administración de mensajes

    // Agregar ( "postear" ) un mensaje en la cola de mensajes de la aplicación.
    BOOL        WXAddMessage(MSG * newMessage);
    // Retornar un mensaje desde la cola de mensajes de la aplicación.
    BOOL        WXGetMessage(MSG *, PWXWnd, UINT uMsgFilterMin,
  UINT uMsgFilterMax);
    // Mantener las ventanas que necesitan ser actualizadas
    BOOL        WXAddWindowNeedingPaint(PWXWnd aWindow);
    PWXWnd      WXGetWindowNeedingPaint();
    // Acceso a miembros.
    HINSTANCE   WXGetHInstance(){return(hInstance);}
};

// Constructor.
WXClient::WXClient(char * displayName)
{
  // Inicialización de miembros.
  // Agregarse a la lista de instancias del modulo.
  // Abrir una conexion con el X Server. El nombre del display (X Server) es la dirección IP del
  // X Server.
  dpy = XOpenDisplay(displayName);
  if(!xDpy)
    wxError = X_ERROR;
}

// Cerrar la conversación con el XServer cuando se cierra la Aplicación.
// Retorna TRUE si la operación fue exitosa y FALSE en caso contrario.
BOOL  WXClient::WXDisconnect()
{
  return(XCloseDisplay(dpy));
}
```

5. 3. 1. 4. Clase WXClass

La clase **WXClass** se relaciona uno a uno con las clases de ventanas en Windows. Es creado un objeto perteneciente a esta clase cada vez que una aplicación registra una clase de ventana con la función RegisterClass. Luego, esta clase mantendra toda la información necesaria para crear una ventana que pertenezca a ella.

```
class WXClass
{
  // Data Members.
  protected :
    // Estructura provista por el ususario ( la aplicación ) al registrar la clase.
    WNDCLASS        lpwc;
    // Atomos para el nombre del Menu y de la Clase.
    ATOM            aMenu, aClassName;
    // Bytes Extras de la Clase ( cantidad y ptr. a los datos ).
    int             cbClsExtra;
    void *          pExtra;
    // Si la clase tiene el estilo CS_GLOBALCLASS, se crea un objeto WXDC para ser utilizado
    // por todas las ventanas de la clase.
    PWXDC           wxDC;
    // Cantidad de ventanas creadas de la clase.
    int             wxWindows;

  // Member Functions.
  public :
    // Constructor y Destructor.
    WXClass();
    ~WXClass();

    // Información Extra ( Alocación, Acceso y Liberación de los bytes extras de clase ).

    // Retorna el LONG ubicado en el offset especificado dentro de los bytes extras.
    long            WXGetClassLong( int offset );
    // Setea el offset especificado dentro de los bytes extras con el LONG nVal.
    // Retorna el LONG que estaba anteriormente.
    long            WXSetClassLong( int offset, long nVal );
    // Retorna el WORD ubicado en el offset especificado dentro de los bytes extras.
    WORD            WXGetClassWord( int offset );
    // Setea el offset especificado dentro de los bytes extras con el WORD nVal.
    // Retorna el LONG que estaba anteriormente.
    WORD            WXSetClassWord( int offset, WORD nVal );

    // Manejo de Información de Clase

    // Obtiene información sobre la clase.
    // Retorna TRUE o FALSE segun se realice correcta o incorrectamente la operación.
    BOOL            WXGetClassInfo(WNDCLASS *);
    // Obtiene el nombre de la clase.
    // Retorna la longitud en bytes de la clase sin contar el NULL.
    int             WXGetClassName(char * lpcn, int size);
    // Retorna el DC de la clase.
    PWXDC           WXGetClassDC();
```

```
    // Información de Ventanas

    // Retorna la cantidad de ventanas creadas.
    void            WXGetWindows();
    // Retorna la cantidad de bytes extras requeridos por la clase.
    int             WXGetExtraBytes();
    // Retorna el ptr. al procedimiento de ventana.
    WNDPROC *       WXGetWndProc();

  protected :
    // Incrementa la cantidad de ventanas creadas.
    void            WXIncWindows();
    // Decrementa la cantidad de ventanas creadas.
    void            WXDecWindows();
};
```

5. 3. 1. 5. Clase WXWnd

La clase sin duda mas importante de la jerarquia de clases de WExX es la clase **WXWnd**, que es donde se realiza todo el mapeo del sistema de ventanas y mensajes. Esta clase es manipulada principalmente desde las funciones de manejo de ventanas y mensajes de Windows que seran atrapadas y redireccionadas hacia el sistema de ventanas X.

```
class WXWnd
{
  // Data Memebers.
  protected :
    // Cliente al que pertenece la ventana.
    PWXClient     wxClient;
    // Clase a la que pertenece la Ventana.
    PWXClass      wxWndClass;
    // Punteros para navegar en la jerarquia de ventanas y para mantener la jerarquia de
    // ventanas Windows que será muy diferente a la jerarquia de ventanas X.
    PWXWnd        wxParent, wxChildList, wxSiblingList, wxOwner;
    // Ventanas que conforman una ventana Windows.
    Window  xClient, xNoClient;
    PWXWnd        wxVScrollBar, wxHScrollBar, wxMenu;
    // DC cuando la clase tiene el estilo CS_OWNDC o CS_CLASSDC.
    PWXDC         wxOwnDC;
    // Bytes Extras  de la ventana, como los especificados en la clase.
    int           cbClsExtra;
    BYTE *        pExtra;
    // Nombre de la ventana (caption).
    char *        lpszName;
    // Region de la Ventana que necesita ser de actualizada.
    XRegion       xRegion;
    // Procedimiento de ventana.
    // Tiene el valor NULL para las ventanas de las clases default de Windows.
    WNDPROC *     pWndProc;
    // Estilo de la Ventana.
    DWORD         dwStyle, dwExStyle;
    // Propiedades de la Ventana (definidas en Windows).
    HANDLE        hProperties;
    // Mantiene el ultimo error sucedido.
    WXError       wxError;

  // Members Functions.
    // Retorna el identificador de la ventana (utilizado por los controles).
    int           WXGetId(){return(0);}
    // Crea y Destruye las ventanas X relacionadas.
    BOOL          WXCreateXWindows();
    BOOL          WXDestroyXWindows();
    // Agregar y Sacar una ventana de la lista de hijas.
    BOOL          WXAddChild(PWXWnd child);
    BOOL          WXDelChild(PWXWnd child);
    void          ForEachChild(WXActionFunc action, void * params);
    // Retorna y Setea el miembro wxSibling.
    PWXWnd        WXGetSibling(){return(wxSiblingList);}
```

```
    void            WXSetSibling(PWXWnd sibling){wxSiblingList =
                        sibling;}
  public :
    // Setea los miembros de la ventana pero no crea la ventana propiamente dicha.
    WXWnd(PWXClient inst, PWXClass aClass, PWXWnd parent,
      LPCSTR lpszWindowName, DWORD exStyle, DWORD style,
      HANDLE menu, void * lpvCreateParams);
    // Destruye las ventanas asociadas y envia los mensajes de destrucción correspondientes al
    // procedimiento de ventana.
    ~WXWnd();
    // Envia los mensajes de creación al procedimiento de ventana correspondiente y crea las
    // ventanas X necesarias segun la clase especificada.
    HWND            WXCreateWindow(int x, int y, int width, int height);
    // Envia los mensajes de destrucción a la aplicación y destruye las  ventanas X  asociadas.
    BOOL            WXDestroyWindow();
    // Retorna el HWND de la ventana. Este será el XId de la ventana X correspondiente al area
    // cliente de la ventana Windows.
    HWND            WXGetHWnd();
    // Retorna un ountero al padre.
    PWXWnd          WXGetParent();
    // Setea un nuevo padre.
    void            WXSetParent(PWXWnd parent){wxParent = parent;}
    // Retorna la clase a la que pertenece la ventana.
    PWXClass        WXGetClass(){return(wxWndClass);}

    // Información Extra ( Alocación, Acceso y Liberación de los bytes extras de clase ).

    // Retorna el LONG ubicado en el offset especificado dentro de los bytes extras.
    long            WXGetWindowLong( int offset );
    // Setea el offset especificado dentro de los bytes extras con el LONG nVal.
    // Retorna el LONG que estaba anteriormente.
    long            WXSetWindowLong( int offset, long nVal );
    // Retorna el WORD ubicado en el offset especificado dentro de los bytes extras.
    WORD            WXGetWindowWord( int offset );
    // Setea el offset especificado dentro de los bytes extras con el WORD nVal.
    // Retorna el LONG que estaba anteriormente.
    WORD            WXSetWindowWord( int offset, WORD nVal );

    // Procesamiento de Mensajes Windows.
    // Convension para nombres de metodos WXWM NombreMensajeWindows

    // Envia un mensaje al procedimiento de ventana.
    // Si el procedimiento de ventana es NULL (es una ventana de una clase predefinida
    // de Windows) llama al procedimiento de ventana default (WXDefProc).
    LRESULT         WXSendMessage( UINT uMsg, WPARAM wParam,
      LPARAM lParam );
    // Dispatching. Dado un mensaje Windows llama al metodo que lo procesa.
    LRESULT         WXDefProc( UINT msg, WPARAM wParam,
      LPARAM lParam );
    // WMCreate
    // Este mensaje es enviado cuando una aplicación requirio la creación de una ventana. El
    // procedimiento de ventana recibe el mensaje despues de ser creada pero antes de que pase a
    // estar visible. El mensaje es enviado antes de que la función de creación de ventana retorne.
    LRESULT         WXWMCreate(WPARAM wParam, LPARAM lParam);
```

```
    LRESULT         WXWMDestroy(WPARAM wParam, LPARAM lParam);
    LRESULT         WXWMEnable(WPARAM wParam, LPARAM lParam);
    LRESULT         WXWMActivate(WPARAM wParam, LPARAM lParam);
    LRESULT         WXWMPaint(WPARAM wParam, LPARAM lParam);
    LRESULT         WXWMMove(WPARAM wParam, LPARAM lParam);
    LRESULT         WXWMSize(WPARAM wParam, LPARAM lParam);
    LRESULT         WXWMMouseMove(WPARAM wParam, LPARAM lParam);
    LRESULT         WXWMSetCursor(WPARAM wParam, LPARAM lParam);

    // Traslado de Eventos X.
    // Este traslado incluye "Enviar" y/o "Postear" mensajes Windows a las ventanas de la
    // aplicación.
    // Convension para nombres de metodos WXEv<NombreEventoX>

    // Recibe un evento X y lo despacha al metodo que procesa el evento en particular,
    // inspeccionando el campo "type" de la estructura XEvent.
    BOOL            WXTraslateXEvent(XEvent * event);

    // Expose.
    BOOL            WXEvExpose(XEvent * event);
    // FocusIn, FocusOut.
    BOOL            WXEvFocus(XEvent * event);
    // Visibility.
    BOOL            WXEvVisibility(XEvent * event);
    // Map, UnMap.
    BOOL            WXEvMap(XEvent * event);
    // KeyPress y KeyRelease.
    BOOL            WXEvKey(XEvent * event);
    // EnterNoitify
    BOOL            WXEvEnterNotify(XEvent * event);
    // ConfigureNotify.
    BOOL            WXEvConfigureNotify(XEvent * event);
    // MotionNotify.
    BOOL            WXEvMotionNotify(XEvent * event);
    // ButtonPress.
    BOOL            WXEvButtonPress(XEvent * event);
    // ButtonRelease.
    BOOL            WXEvButtonRelease(XEvent * event);
};

// Constructor
// Setea los miembros de la ventana pero no crea la ventana propiamente dicha.
WXWnd::WXWnd(PWXClient inst, PWXClass aClass, PWXWnd parent,
  LPCSTR lpszWindowName, DWORD exStyle, DWORD style, HANDLE menu,
  void * lpvCreateParams)
{
  wxError = 0;
  wxClient = inst;
  wxWndClass = aClass;
  wxOwner = wxParent = wxChildList = wxSiblingList = NULL;
  // Estilo de la ventana.
  dwStyle = style;
  dwExStyle = exStyle;
  if(dwStyle & WS_CHILD)
```

```
      wxParent = parent;
   else
      wxOwner = parent;
   // Tomar procedimiento de ventana desde la clase.
   pWndProc = wxWndClass->WXGetWndProc();
   // Caption.
   lpszName = NULL;
   int len = strlen(lpszWindowName);
   if(len > 0)
      if((lpszName = (char *)malloc(len+1)) != NULL)
         strcpy(lpszName, lpszWindowName);
      else
         wxError |= WXERR_NOTENOUGHMEMORY;
   // Alocar bytes extras segun datos de clase.
   pExtra = NULL;
   if((cbClsExtra = wxWndClass->WXGetExtraBytes()) > 0)
      if((pExtra = (BYTE *)malloc(cbClsExtra)) == NULL)
         wxError |= WXERR_NOTENOUGHMEMORY;
   // Miembros que se inicializan en tiempo de creación de la ventana.
   wxOwnDC = NULL;
   hProperties = NULL;
   xClient = xNoClient = 0;
   wxVScrollBar = wxHScrollBar = wxMenu = NULL;
}

// Crea las ventanas X Window necesarias segun la clase especificada, y envia los mensajes de
// creación al procedimiento de ventana correspondiente (principalmente el mensaje
// WM_CREATE).
HWND WXWnd::WXCreateWindow(int x, int y, int width, int height)
{
   /* Enviar mensaje WM_CREATE al proc. de ventana indicado en la clase a la que
      pertenece la ventana.
   Si la aplicación acepto la creación de la ventana.
      Crear la/s ventanas X correspondientes (WXCreateXWindows).
      Si todo estubo OK
         Agregar la ventana a la jerarquia (WXLink).
         Actualizar la tabla de relaciones handles-ptr.
      Sino
         La creación fue rechazada u ocurrio algun error.
         Enviar mensajes de destrucción(DESTROY).
   Retornar handle a la ventana o NULL si hubo algun error. */
}

// WXDestroyWindow()
// Envia los mensajes de destrucción a la aplicación y destruye las ventanas X asociadas.
// Se envian los mensajes necesarios para liberar el input focus, tambien se destruye el menu de la
// ventana window, se "flushea" la cola de la aplicación y se destruyen timers pendientes.
// Se envian el mensaje WM_DESTROY a la ventana, y si la ventana es padre de ventana/s, se
// destruye/n primero esta/s. Esta función tambien es utilizada para destruir modeless dialog boxes
// que fueron creados con CreateDialog.
BOOL WXWnd::WXDestroyWindow()
{
   WXSendMessage(WM_PARENTNOTIFY, WM_DESTROY, MAKELONG(WXGetHWnd(),
      WXGetId()));
```

```
    WXSendMessage(WM_KILLFOCUS, 0, NULL);
    WXSendMessage(WM_DESTROY, 0, 0);
    if(wxChildList != NULL)
    {
      ForEachChild(destroy, NULL);
      wxChildList = NULL;
    }
    if(wxParent != NULL)
      wxParent->WXDelChild(this);
    WXDestroyXWindows();

    return(TRUE);
}
```



```
// Enviar un mensaje al procedimiento de ventana.
// Si el procedimiento de ventana es NULL (es una ventana de una clase predefinida de Windows)
// llamar al WXDefProc.
LRESULT WXWnd::WXSendMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    if(pWndProc != NULL)
      return(pWndProc(WXGetHWnd(), wParam, lParam));
    else
      return(WXDefProc(uMsg, wParam, lParam));
}

// Retorna el LONG ubicado en el offset especificado dentro de los bytes extras.
long WXWnd::WXGetWindowLong(int offset)
{
    long value = 0;
    if((offset > 0) && ((offset+sizeof(long)) < cbClsExtra))
      memcpy((void *)&value, pExtra+offset, sizeof(long));

    return(value);
}

// Setea el long en el offset especificado dentro de los bytes extras con el parametro LONG nVal.
// Retorna el LONG que estaba anteriormente.
long WXWnd::WXSetWindowLong(int offset, long nVal)
{
    long value = 0;
    if((offset > 0) && ((offset+sizeof(long)) < cbClsExtra))
    {
      memcpy((void *)&value, pExtra+offset, sizeof(long));
      memcpy(pExtra+offset, (void *)&nVal, sizeof(long));
    }

    return(value);
}

// Crea las ventanas X que forman la ventana Windows.
BOOL WXWnd::WXCreateXWindows()
{
    unsigned int border_width;
    int depth;
```

```
  unsigned long fg, bg;
  // Window atributes.
  XSetWindowAttributes xswa;
  // Properties.
  XSizeHints size_hints;
  XWMHints wm_hints;
  XTextProperty windowNamePorperty, iconNamePorperty;
  XClassHint class_hint;

  border_width = 0;
  depth = (int)BlackPixel(dpy, DefaultScreen(dpy));
  bg = WhitePixel(dpy, DefaultScreen(dpy));
  fg = BlackPixel(dpy, DefaultScreen(dpy));
  xswa.background_pixel = bg;
  xswa.colormap = DefaultColormap(dpy, DefaultScreen(dpy));
  xswa.bit_gravity = CenterGravity;
  // Crear la ventana en el XServer.
  xClient = XCreateWindow(
    dpy,                                    /* Conexion con el Xserver. */
    WXGetXParent(),                         /* Ventana padre en X Window. */
    x, y,  (unsigned int)w, (unsigned int)h,        /* Posición y tamaño. */
    border_width, depth, InputOutput, CopyFromParent, /* Atributos de la ventana.
*/
      (CWColormap | CWBitGravity | CWBackPixel), &xswa);
  if(xClient == 0)
    return(FALSE);
  // Seteo de "mandatory propierties" con las funciones de la convención "version 1" adoptada por
  //  el X Consortium a partir del Release 4. Existe la función XmbSetWMProperties para facilitar
  // el 'setting' de estas propiedades pero se utilizara XSetWMProperties para mayor entendimiento
  // de las propiedades.
  // USPosition | USSize : que se respeten la posición y tamano definidos por el usuario.
  size_hints.flags = USPosition | USSize;
  // Pasar nombre de la ventana y texto del icono a una codificación para
  // "internationalized text comunication".
  XStringListToTextProperty(&lpszName, 1, &windowNamePorperty);
  XStringListToTextProperty(&lpszName, 1, &iconNamePorperty);
  // Estado inicial de la ventana.
  wm_hints.initial_state = WXGetInitState();
  // Keyboard input ?. En WExX siempre seran seleccionados eventos desde el teclado.
  wm_hints.input = True;
  // Icon para ser utilizado al minimizar la ventana.
  wm_hints.icon_pixmap = icon_pixmap;
  wm_hints.flags = StateHint | InputHint; //| IconPixmapHint
  XSetWMProperties(dpy, xClient, &windowNamePorperty, &iconNamePorperty,
    NULL, NULL, &size_hints, &wm_hints, NULL);
  // Creación del DC desde el WXGDI.
  wxDC = _WXCreateDC(dpy, xClient);
  // Selección de eventos a recibir.
  XSelectInput(dpy, xClient, clientMask);
  // Mapear la ventana.
  if(dwStyle & WS_VISIBLE)
    XMapWindow(dpy, xClient);
```

```
   return(TRUE);
}

// Expose.
BOOL WXWnd::WXEvExpose(XEvent * event)
{
   XRectangle rect;
   // Agregar la region como area que necesita ser repintada.
   rect.x = event->xexpose.x;
   rect.y = event->xexpose.y;
   rect.width = event->xexpose.width;
   rect.height= event->xexpose.height;
   XUnionRectWithRegion(&rect, xRegion, xRegion);
   // Si no hay mas eventos expose pendientes desde el XServer agregar la ventana a la lista de
   // ventanas que necesitan ser repintadas (solo si ya no pertenece a la lista).
   if(event->xexpose.count == 0)
      wxClient->WXAddWindowNeedingPaint(this);

   return(TRUE);
}

// MapNotify, UnMapNotify.
BOOL WXWnd::WXEvMap(XEvent * event)
{
   WORD fwSizeType;

   fwSizeType = (event->type == MapNotify) ? SIZE_MAXIMIZED :
      SIZE_MINIMIZED;

   return(WXSendMessage(WM_SIZE, fwSizeType, MAKELONG(size.x, size.y)));
}

// VisibilityNotify.
BOOL WXWnd::WXEvVisibility(XEvent * event)
{
   if(event->xvisibility.state == VisibilityPartiallyObscured)
      return(0);
   return(WXSendMessage(
      WM_SHOWWINDOW,(event->xvisibility.state == VisibilityUnobscured),
         0l));
}

// FocusIn, FocusOut.
// Si el foco es ganado por un click del mouse deberia enviarse ademas de los eventos de cambio de
// foco el evento WM_MOUSEACTIVATE pero esta información no puede ser obtenida directamente
// del evento X de cambio de foco.
BOOL WXWnd::WXEvFocus(XEvent * event)
{
   UINT uMsg;
   WORD fActive;

   if(event->type == FocusIn)
   {
      fActive = WA_ACTIVE;
```

```
      uMsg = WM_SETFOCUS;
   }
   else
   {
      fActive = WA_INACTIVE;
      uMsg = WM_KILLFOCUS;
   }
   WXSendMessage(WM_ACTIVATE, fActive, 0l);

   return(WXSendMessage(uMsg, 0, 0l));
}

// ConfigureNotify.
BOOL WXWnd::WXEvConfigureNotify(XEvent * event)
{
   if((pos.x != event->xconfigure.x) || (pos.y != event->xconfigure.y))
   {
      // Notificar que la posición de la ventana ha sido modificada.
      pos.x = event->xconfigure.x;
      pos.y = event->xconfigure.y;
      WXSendMessage(WM_MOVE, SIZE_RESTORED, MAKELONG(pos.x, pos.y));
   }
   if((size.x != event->xconfigure.width) || (size.y != event-
      >xconfigure.height))
   {
      // Notificar que el tamano de la ventana ha sido modificado.
      size.x = event->xconfigure.width;
      size.y = event->xconfigure.height;
      WXSendMessage(WM_SIZE, SIZE_RESTORED, MAKELONG(size.x, size.y));
   }

   return(TRUE);
}

// EnterNoitify
BOOL WXWnd::WXEvEnterNotify(XEvent * event)
{
   // Dar la posibilidad al padre de que maneje el 'setting' del cursor de sus hijas.
   if(wxParent)
      return(wxParent->WXSendMessage(WM_SETCURSOR, WXGetHandle(),
        MAKELONG(HTCLIENT, 0)))

   return(FALSE);
}

// KeyPress, KeyRelease.
BOOL WXWnd::WXEvKey(XEvent * event)
{
   return(WXPostMessage((event->type == KeyPress) ? WM_KEYDOWN :
      WM_KEYUP, event->xkey.keycode, MAKELONG(0, event->xkey.state)));
}

// MotionNotify.
BOOL WXWnd::WXEvMotionNotify(XEvent * event)
```

```
{
  return(WXPostMessage(WM_MOUSEMOVE, mapkeys(event->xmotion.state),
    MAKELONG(event->xmotion.x, event->xmotion.y)));
}

// ButtonPress.
BOOL WXWnd::WXEvButtonPress(XEvent * event)
{
  UINT uMsg;

  switch(event->xbutton.button)
  {
    case 1 : // Left button
      uMsg = WM_LBUTTONDOWN;
      break;
    case 2 : // Midle button
      uMsg = WM_MBUTTONDOWN;
      break;
    case 3 : // Right button.
    default :
      uMsg = WM_RBUTTONDOWN;
      break;
  }
  if((wxWndClass->WXGetClassStyle() & CS_DBLCLKS) &&
    (lastbuttondown.button == event->xbutton.button) &&
    (event->xbutton.time - lastbuttondown.time <= GetDoubleClickTime()))
  {
    uMsg += 2;  // los mensajes de doble click tienen un codigo igual al codigo de
                // BUTTONDOWN + 2;
    memset(&lastbuttondown, 0x0, sizeof(lastbuttondown));
  }
  else
  {
    lastbuttondown.button = event->xbutton.button;
    lastbuttondown.time = event->xbutton.time;
  }

  return(WXPostMessage(uMsg, mapkeys(event->xbutton.state),
    MAKELONG(event->xbutton.x, event->xbutton.y)));
}

// ButtonRelease.
BOOL WXWnd::WXEvButtonRelease(XEvent * event)
{
  UINT uMsg;

  switch(event->xbutton.button)
  {
    case 1 : // Left button
      uMsg = WM_LBUTTONUP;
      break;
    case 2 : // Midle button
      uMsg = WM_MBUTTONUP;
```

```
        break;
      case 3 : // Right button.
        default :
        uMsg = WM_RBUTTONUP;
        break;
    }

    return(WXPostMessage(uMsg, mapkeys(event->xbutton.state),
      MAKELONG(event->xbutton.x, event->xbutton.y)));
}
```

5. 3. 1. 6. Clases Auxiliares

Además de las clases que se relacionan practicamente uno a uno con los componentes de Windows (Modulos, Tareas, Clases de Ventanas, Ventanas), hay algunas clases auxiliares que completan la jerarquia de clases de WExX. Estas clases son **WExXBaseClass** y **WXMessage**.

WExXBaseClass : es la clase base de WExX. Todas las clases mencionadas mas arriba derivan de esta clase base. Esta clase no agrega mucha funciónalidad, solo implementa la obtención de un handle a memoria local en donde se mantendra el puntero si mismo cuando un objeto es creado. Luego ese handle será el que distinga al objeto.

WXMessage : encapsula una estructura MSG de Windows, que es la utilizada para obtener mensajes por medio de GetMessage. WExX mantiene una cola de objetos WXMessage, los cuales tienen una estructura MSG correspondiente al mensaje.

Como son utilizadas las clases de WExX.

Hasta ahora se explicó la jerarquía de clases de WExX, pero... cómo y cuando son creados objetos de estas clases mientras se ejecuta un programa Windows que desea ser controlado remotamente utilizando WExX ?

Cuando un programa Windows invoca un función del API de Windows, como éstas están interceptadas por WExX, el flujo del programa no continua en los módulos de Windows que implementan la función mencionada (GDI, USER, KERNEL), sino que el control es tomado por WExX. En ese momento, dependiendo de la función invocada, pueden ser creados y/o borrados objetos de WExX y/o invocados métodos de éstos.

En general la funciónalidad del API de Windows esta implementada en los métodos de los distintos objetos de WExX, pero en algunos casos existe un procesamiento previo antes de invocar a los métodos correspondientes.

A continuación se da una especificación, y en algunas oportunidades la implementación de WExX de la reimplementación de las funciones del API de Windows mas importantes y mas utilizadas.

```
// CreateWindow : crea una ventana a partir de una clase, y especifica el titulo, posición y tamaño.
// Esta función envia el mensaje WM_CREATE al procedimiento de ventana antes de retornar.
HWND CreateWindow(LPCSTR lpszClassName, LPCSTR lpszWindowName,
   DWORD dwStyle, int x, int y, int nWidth, int nHeight,
   HWND hwndParent, HMENU hmenu, HINSTANCE hinst, void FAR* lpvParam)
{
   // Obtener el objeto WXClient que representa a la tarea que esta creando la ventana, en base al
   // HINSTANCE, obtenido mediante la función del API GetCurrentTask().

   // Obtener el objeto WXClass que representa a la clase en base al nombre de clase especificado
   // por el programa.

   // Crear el objeto WXWnd correspondiente.

   // Llamar al metodo WXCreateWindow del objeto WXWnd creado, para que realice los
   // pasos necesarios para la creación de la ventana.

   // Retornar un handle que identifique univocamente al objeto ventana creado.
}

// GetMessage : este es la función mas importante en el mapeo. Debe obetener eventos desde la cola
// X, mapearlos a mensajes Windows, y retornarlos a la aplicación llamante; pero ademas debe
// permitir que otras aplicaciones continuen ejecutandose en Windows. Su función en Windows es
// obtener un mensaje desde la cola de mensajes de la aplicación. Si no hay mensajes disponibles
// para la aplicación, GetMessage deja el control a otra aplicación, hasta que un mensaje este
// disponible.
BOOL GetMessage(MSG * lpmsg, HWND hwnd, UINT uMsgFilterMin,
   UINT uMsgFilterMax)
{
   // Obtener el ptr. al objeto WXClient que se esta ejecutando, en base al hInstance obtenido
   // mediante GetCurrentTask().

   // Obtener el ptr. al objeto WXWnd en base al hwnd.
getmessage:
   if(client->WXPeekMessage())
   {
      // Si hay mensajes en la cola de la aplicación, obtener uno y retornarselo a la aplicación.
```

```
      ret = client->WXGetMessage(lpmsg, hwnd, uMsgFilterMin,
        uMsgFilterMax);
      if(ret == WM_QUIT)
        ret = 0;
    }
    else
    {
      // Ver si hay alguna ventana que necesita ser repintada chequeando el conjunto de ventanas
      // mantenida por el objeto cliente que necesitan repintarse (miembro wxNeedPaint). Si es
      // asi, verificar que no haya una notificación de un nuevo expose o cambio de configuración
      // de la misma ventana en la cola de eventos de X Window, que pueda producir nuevamente
      // la necesidad de repintar la ventana,que suele ser el procesamiento mas costoso (en
      // tiempo) para el X Server. En el caso de que no exista ninguna notificación de las antes
      // mencionadas es que se retorna un mensaje WM_PAINT a la aplicación indicando que una
      // ventana necesita ser repintada. Esta forma de generación de los mensajes WM_PAINT
      // puede generar un retardo en el refresco de una ventana, pero evita el procesamiento inutil
      // de un repintado, y reduce el tráfico entre el el cliente y el X Server, y la carga de
      // procesamiento del X Server que puede responder a requerimeintos de otros clientes.
      // En caso contrario (no hay una ventana que necesite ser repintada), se chequea la cola de
      // eventos de X Window y se envia a procesar el evento por la ventana correspondiente. Este
      // procesamiento puede incluir enviar mensajes a una ventana y/o poner mensajes en la cola
      // de la aplicación. Luego retornar a chequear la cola de mensajes de la aplicación.
      if(client->WXGetWindowNeedingPaint(lpmsg, hwnd) &&
        !XCheckIfEvent(client->GetXDisplay(), &checkEvent, predicado,
          stuf))
      {
        ret = TRUE;
      }
      else
      {
        client->WXNextEvent();
        goto getmessage;
      }
    }

    // Retornar 0 si el mensaje obtenido es WM_QUIT y distinto de cero en caso contrario.
}

// DispatchMessage : despacha un mensaje a una ventana. Esta función es generalmente utilizada
// luego de obtener un mensaje con GetMessage.
LONG DispatchMessage(MSG * lpmsg)
{
    // Obtener la ventana (puntero al objeto WXWnd) a la que esta asociada el mensaje.

    // Invocar al metodo del objeto WXWnd que despacha un mensaje.
}

// DestroyWindow : destruye la ventana especificada. Envia los mensajes apropiados para
// desactivar y quitar el foco a la ventana.
BOOL DestroyWindow(HWND hwnd)
{
    // Obtener la ventana (puntero al objeto WXWnd) a la que esta asociada el mensaje.
    PWXWnd win = (PWXWnd)WExXClass::GetThis(hwnd);
```

```
   // Invocar al metodo de destrucción que realiza los pasos necesarios de destrucción de una
   // ventana, como por ejemplo generar el requerimiento de destrucción de la/s ventana/s X
   // asociada/s a la ventana que se esta destruyendo y enviar los mensajes de destrucción
   // (WM_DESTROY) a la ventana. Finalmente destruir el objeto WXWnd.
   if(win)
   {
      win->WXDestroyWindow();
      delete(win);
   }

   // Retornar TRUE o FALSE segun se pudo destruir correctamente la ventana o no.
}
```

5. 3. 2. Mapeo de objetos y funciones gráficas (GDI)

5. 3. 2. 1. Objetos Gráficos

El modulo WXGDI mantiene una lista de handles a objetos al igual que el modulo GDI. Al mapear el modulo GDI también tenemos que tener en cuenta estos objetos. Definimos los objetos:

Objeto	ID
WXPenObject	1
WXBrushObject	2
WXFontObject	3
WXPaletteObject	4
WXBitmapObject	5
WXDCObject	6
WXDCObject	7
WXDCObject	8
WXRegionObject	9

Cada objeto tiene información en común, esta información se mantiene en una estructura llamada WXObject.

```
typedef struct
{
  WORD   nextObject;
  WORD   typeObject;
  DWORD  countObject;
} WXObject;
```

5. 3. 2. 1. 1. WXDC

En el WXGC definimos un objeto llamado *WXDC*, el cual se comporta como un DC. Para mapear los atributos del DC usamos el recurso GC de Xlib y para toda la información no existente en el GC, la mantenemos en los atributos del objeto WXDC.

```
typedef struct
{
  WXObject  header;
  HBRUSH    hWXBrush;
  HBITMAP   hWXBitmap;
  HPEN      hWXPen;
  HPALETTE  hWXPalette;
  POINT     PenPosition;
  POINT     BrushPosition;
  WORD      kMode;
  DWORD     TextColor;
  FONT      hWXFont;
  GC        WXGC;
  HANDLE    hWXWin;
  Window    XWin;
  Display   *dpy;
```

```
} WXDC;
```

WXObject: Datos en común de los objetos.
hWXBrush: Handle a un Brush.
hWXBitmap: Handle a un Bitmap.
hWXPen: Handle a un Pen.
hWXPalette: Colormap asociado con la paleta
PenPosition: Posición del Pen
BrushPosition: Posición del Brush.
kMode Modo de dibujo.
TextColor; Color de Texto
hWXFont: Handle a un Font.
WXGC: Graphics Device asociado con el Device Context.
hWXWin: Ventana Windows asociada con el Device Context.
Xwin: Ventana X asociada con el Device Context.
dpy: Display asociado con el Device Context.

5. 3. 2. 1. 2. WXBrush

En el WXGDI se definio el objeto *WXBrush*, el cual contiene información para realizar el mapeo con Xlib. Existen distintos tipos de brush, definimos una estructura para mantener la información en comun los distintos tipos de brush.

```
typedef struct
{
  WXObject  header;
  WORD      typeBrush;
} WXBrush;
```

header: Datos en común de los Objetos
typeBrush: Tipo de Brush.

Tipos de Brush

Tipo	ID	Descripción
BS_DIBPATTERN	1	Especifica un Brush definido por un bitmap.
BS_HATCHED	2	Especifica un brush definido por un hatched.
BS_HOLLOW	3	Especifica un hollow brush.
BS_PATTERN	4	Especifica un brush definido por un bitmap en memoria.
BS_NULL	5	Equivalente a BS_HOLLOW.
BS_SOLID	6	Especifica un brush solid.

Para poder realizar el mapeo de los tipos de brush se usan los atributos *tile, fill_style, background* del GC. Para los tipos *BS _ DIBPATTERN, BS_PATTERN*, y *BS_HATCHED* se usan los atributos tile, fill_style, este ultimo esta modificado con el valor FillTiled y en el atributo tile el bitmap o el hatch.

El hatch es manejado como un bitmap. Con los tipos *BS_HOLLOW*, *BS_SOLID* y *BS_NULL* el atributo fill_style es modificado con el valor FillSolid y el atributo background con un color.

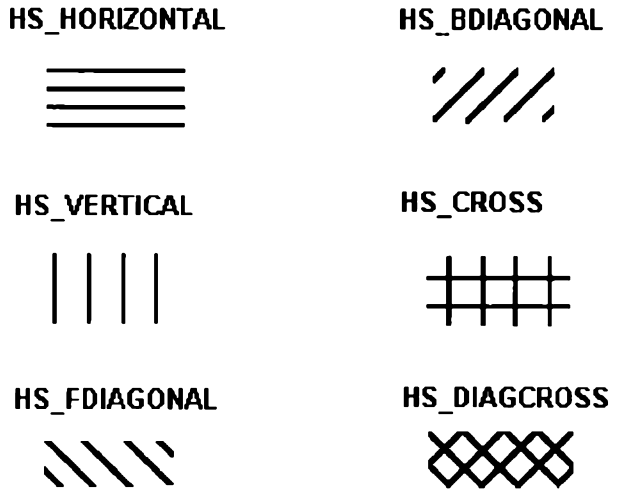
```
typedef struct
{
  WXBrush   headerBrush;
  COLORREF  colorBrush;
} WXSolidBrush;
```

headerBrush: Especifica el Tipo de Brush. (*BS_SOLID*)
colorBrush: Color del Brush.

```
typedef struct
{
  WXBrushheaderBrush;
  COLORREF  colorBrush;
  HBITMAP   hatchBrush;
} WXHatchBrush;
```

headerBrush: Especifica el Tipo de Brush. (*BS_ HATCHED*)
colorBrush: Color del Brush.
hatchBrush: Hatch asociado al Brush.

Los estilos de Hatch son:




```
typedef struct
{
  WXBrush   headerBrush;
  BITMAP    Bitmap;
} WXPatternBrush;
```

headerBrush: Especifica el Tipo de Brush. (*BS_ PATTERN*)
Bitmap: Especifica el Patron de Bits.

5. 3. 2. 1. 3. WXPen

Se definio un objeto llamado *WXPen*, el cual contiene la información para realizar el mapeo con Xlib.

```
typedef struct
{
  WXObject      header;
  int           width;
  int           style;
  unsigned long color;
  unsigned char *dash_list;
  int       dash_list_length;
} WXPen;
```

header: Datos en común de los Objetos
width: Ancho del Pen.
style: Estilo del Pen.
color: Color del Pen
**dash_list*,
dash_list_length: Se usan para los tipos de Pen

Tipos de Pen

Tipo	ID	Descripción
PS_SOLID	1	Especifica un pen solido.
PS_DASH	2	Especifica un pen con dashes.
PS_DOT	3	Especifica un pen con dots
PS_DASHDOT	4	Especifica un pen con dashes y dots.
PS_DASHDOTDOT	5	Especifica un pen with con dashes y dobles dots.
PS_NULL	6	Especifica un pen nulo.
PS_INSIDEFRAME	7	Especifica un pen que dibuja una línea dentro de un frame.

Los atributos del GC para poder emular la funcionalidad de los pens de Windows son: *foreground, line_width*, y *line_style, dashes*,y *dash_offset.*

PS_SOLID ————————
PS_DASH — — — -
PS_DOT ----------
PS_DASHDOT — - — - —
PS_DASHDOTDOT — - - — - - -
PS_NULL
PS_INSIDEFRAME ————————

5. 3. 2. 1. 4. WXFont

Se definio un objeto *WXFont* con los siguientes atributos:

```
typedef struct
{
   WXObject  header;
   LPCSTR    nameFont;
   Font      idFont
} WXFont;
```

5. 3. 2. 1. 5. WXPalette

El objeto *WXPalette* tiene la siguiente estructura:

```
typedef struct
{
   WXObject  header;
   Colormap  colors;
} WXPalette;
```

header: Datos en común de los Objetos
colors: Colormap (*Paleta en Xlib*).

5. 3. 2. 1. 6. WXBitmap

En el *WXGDI* definimos un objeto llamado *WXBitmap*, el cual contiene información del bitmap como el ancho. largo, profundidad y el conjunto de bytes que lo forman.

```
typedef struct
{
   WXObject  header;
   char      *pBitmap;
   int       width;
   int       height;
   UINT      depth;
   Pixmap    idPixmap;
   int       flag;
} WXBitmap;
```

header: Datos en común de los Objetos
**pBitmap*: Bits del Bitmap.
width: Ancho del Bitmap.
height: Largo del Bitmap.
depth: Profundidad del Bitmap.
idPixmap: Identificación del Pixmap en el X Server.
flag: Informa si el Bitmap fue convertido a Pixmap.

5. 3. 2. 1. 7. WXCursor

Definimos un objeto *WXCursor*, el cual contiene información del nombre del cursor de Windows y el cursor asociado a este en Xlib.

```
typedef struct
{
```

```
  WXObject       header;
  LPCSTR         nameCursor;
  unsigned int   shapeCursor;
  Cursor         cursor;
} WXCursor;
```

header: Datos en común de los Objetos
nameCursor: Nombre del Cursor en Windows.
shapeCursor: Identificación del cursor en X Window.
cursor: Cursor en x Window.

5. 3. 2. 1. 8. WXRegion

Definimos el objeto *WXRegion*. En ambos sistemas de ventanas se usan de forma similar, es decir para modificar el area de clipping.

```
typedef struct
{
  WXObject  header;
  HRGN      hRgn;
  Region    idRgn;
} WXRegion;
```

header: Datos en común de los Objetos
hRgn: Identificación de una región en Windows.
idRgn: Identificación de una región en X Window.

5. 3. 2. 2. Mapeo de Funciones

5. 3. 2. 2. 1. Funciones de WXBrush

Para cada función de creación de un brush se creara un objeto *WXBrush* de un determinado estilo en el cual tendrá toda la información necesaria para su uso.

Las funciones a mapear son:

CreateBrushIndirect	Crea un brush con los atributos especificados.
CreateHatchBrush	Crea un Brush con un Hatch.
CreatePatternBrush	Crea un Brush con un bitmap
CreateSolidBrush	Crea un Brush con un color especificado.
GetBrushOrg	Devuelve el origen del corriente brush.
SetBrushOrg	Modifica el origen del corriente brush.

5. 3. 2. 2. 1. 1. WXCreateBrushIndirect

La función *WXCreateIndirect* crea un crea un Brush con un estilo, color y patrón de bits especificado en la estructura llamada LOGBRUSH.

```
typedef struct
{
  UINT      lbStyle;
  COLORREF  lbColor;
  int       lbHatch;
} LOGBRUSH;


HBRUSH WXCreateBrushIndirect(LOGBRUSH FAR* lplb)
{
  switch (lplb->lbStyle)
  {
    case BS_HATCHED:
      // creo un brush de tipo hatch
      return(WXCreateHatchBrush(lplb->lbHatch, lplb->lbColor));
    case BS_PATTERN:
      // creo un brush de tipo Pattern
      return(WXCreatePatternBrush(lplb->lbHatch));
    default:
      // creo un brush de tipo Solid
      return(WXCreateSolidBrush(lplb->lbColor));
   }
}
```

5. 3. 2. 2. 1. 2. WXCreateSolidBrush

La función *WXCreateSolidBrush* crea un Brush con el color especificado. El parámetro clrref es de tipo COLORREF. El valor de retorno es el handle del Brush si la función es valida. En caso contrario es NULL. Esta información se mantendrá en el objeto llamado WXSolidBrush.

```
HBRUSH WXCreateSolidBrush(COLORREF clrref)
{
  WXSolidBrush     *pBrush;
  HBRUSH           hbrush;

  hbrush=LocalAlloc(LHND, sizeof(WXSolidBrush));
  if (hbrush!=NULL)
  {
    pBrush=(WXSolidBrush *)LocalLock(hbrush);
    if (pBrush!=NULL)
    {
      // guardo en el objeto brush el ripo de objeto, tipo de brush, color del brush
      pBrush->headerBrush.header.typeObject = WXBrushObject;
      pBrush->headerBrush.typeBrush=BS_SOLID;
      pBrush->colorBrush=clrref;
      LocalUnlock(hbrush);
    }
    return(hbrush);
  }
  return(NULL);
}
```

5. 3. 2. 2. 1. 3. WXCreateHatchBrush

La función *WXCreateHatchBrush* crea un Brush con un hatched y color. Recibe como parámetros el estilo del hatch para el brush y el parámetro color del foreground del brush. El valor de retorno es el handle del brush si la función es valida, en caso contrario es NULL. La información será guardada en el objeto *WXHatchBrush*. En Xlib no existen los hatch, por ese motivo el brush Hatched es similar al Brush de tipo BS_PATTERN, pero en vez de leer un bitmap y realizar la conversion a pixmap, directamente crearemos un pixmap. El atributo flag se usa para saber si el pixmap asociado con el hatch fue creado o no.

Pseudocodigo

```
// creo un objeto brush de tipo Hatched
// guardo las caracteristicas del brush como el estilo, color y un flag
brush->headerBrush.header.typeObject = WXBrushObject;
brush->headerBrush.typeBrush=BS_HATCHED;
brush->colorBrush=clrref;
brush->fnStyle=fnStyle;
brush->flag=FALSE;
return(brush)
```

5. 3. 2. 2. 1. 4. WXCreatePatternBrush

La función *WXCreatePatternBrush* crea un brush con el patrón de bits, el cual es especificado por un bitmap. El valor de retorno es el handle del brush si la función es valida. En caso contrario es NULL. Los bitmaps usados como patrones deben ser 8 pixels por 8 pixels. Si el bitmap es mas grande, se usaran los bits correspondientes a las primeras 8 filas y 8 columnas de pixels empezando desde la esquina superior izquierda del bitmap.

Pseudocodigo

```
// guardo el tipo de objeto, tipo de brush y el bitmap asociado a este
brush->headerBrush.header.typeObject = WXBrushObject;
brush->headerBrush.typeBrush=BS_DIBPATTERN;
brush->Bitmap = hbmp;
return(brush);
```

Al usarlo se creara el brush hatched, eso sucede porque igual que el brush con pattern necesitamos información que en el momento de creación no la tenemos.

En los dos ultimos casos al usar el brush por medio de las funciones *WXSelectObject* o *WXFillRect* se crea el pixmap y es guardado en el objeto *WXHatchBrush* o *WXPaternBrush* de acuerdo al tipo de brush usado.

5. 3. 2. 2. 1. 5. WXSetBrushOrg

La función *WXSetBrushOrg* modifica el origen del actual brush seleccionado en el WXDC. En Xlib las coordenadas de origen del brush estan indicadas en los atributos *ts_x_origin* y *ts_y_origin* en el objeto WXGC del WXDC. La función *XSetTSOrigin* modifica los atributos *ts_x_origin* y *ts_y_origin* del GC, estos atributos especifican la coordenada inicial del tile.

El valor default de las coordenadas de origen de un brush es el punto (0,0). Esta función retorna las coordenadas del origen anterior, si la función fue exitosa. El valor formado por la parte mas baja del WORD es la coordenada X y la mas alta forma la coordenada Y.

Pseudocodigo

```
// genero un long con la coordenada de origen de brush para devolverlo
values = MAKELONG(pDC->BrushPosition.x, pDC->BrushPosition.y);
// guardo la nueva coordenada X en el WXDC
DC->BrushPosition.x=nXOrg;
// guardo la nueva coordenada Y en el WXDC
DC->BrushPosition.y=nYOrg;
// guardo la coordenada de origen el el GC del WXDC
XSetTSOrigin(DC->dpy, DC->WXGC, nXOrg, nYOrg);
// devuelvo la coordenada anterior en la variable values
return(values);
```

5. 3. 2. 2. 1. 6. WXGetBrushOrg

La función *WXGetBrushOrg* devuelve las coordenadas de origen del brush del WXDC. Un parámetro es un handle a una estructura de tipo HWDC. Esta función retorna en el byte mas bajo del WORD la coordenada X y en el mas alto la Y.

Pseudocodigo

```
// Leo la coordenada de origen del WXDC.
values = MAKELONG(DC->BrushPosition.x, DC->BrushPosition.y);
return(values);
```

5. 3. 2. 2. 2. Funciones de WXDC

Las funciones esta seleccionadas en dos grupos:

- Manipulación de los miembros del WXDC. (modificación, consulta de los miembros)
- Manipulación del objeto WXDC. (creación, etc)

5. 3. 2. 2. 2. 1. Manipulación de los miembros del WXDC.

5. 3. 2. 2. 2. 1. 1. WXSelectObject

La función *SelectObject* es una de las funciones mas importantes del GDI porque selecciona objetos gráficos en el Device Graphics para usarlos. El objeto a seleccionar reemplaza al objeto actual en el DC. Definimos la función *WXSelectObject* que mapea a la función SelectObject, tiene como parámetro el HANDLE al WXDC y el HANDLE a un Objeto del WXGDI. En Xlib no existe ninguna función similar a esta., en cambio en el código de la función *WXSelectObject* se realizan acciones diferentes de acuerdo al objeto a seleccionar

Los objectos del WXGDI pueden ser: WXBitmap, WXBrush, WXFont, WXPen. La función retorna el handle al objeto viejo, en otro caso es NULL.

```
HGDIOBJ WXSelectObject(HDC hdc, HGDIOBJ hgdiobj)
{
  WXDC      *pDC;
  WXObject  *pObject;
  HPEN      oldObject=NULL;
  Pixmap    idPixmap;

  pDC=(WXDC *)LocalLock(hdc);
  pObject=(WXObject *)LocalLock(hgdiobj);
  if ((pDC!=NULL) && (pObject!=NULL))
  {
    switch (pObject->typeObject)
    {
      case WXPenObject:
      {
        WXPen *pPen;

        pPen = (WXPen *)pObject;

        // se guarda el handle del pen anterior para devolverlo
        oldObject = pDC->hWXPen;
        // modifica el ancho y estilo del graphics context con los datos del WXPen
        XSetLineAttributes(pDC->dpy,pDC->WXGC,pPen->width,
          pPen->style,CapNotLast, JoinMiter);
        // modifica el color de la linea
        XSetForeground(pDC->dpy, pDC->WXGC, (pPen)->color);
        // modifica los dashed
        XSetDashes(pDC->dpy, pDC->WXGC, 0, pPen->dash_list,
          pPen->dash_list_length);
        break;
      }
      case WXBrushObject:
      {
        WXBrush *pBrush;
```

```
// se guarda el handle del brush anterior para devolverlo
oldObject = pDC->hWXBrush;
switch (pBrush->typeBrush)
{
    case BS_HATCHED:
   {
     WXHatchBrush *pHatchBrush;

     pHatchBrush= (WXHatchBrush *)pBrush;
     //modifica el modo de llenado con FillTiled
        XSetFillStyle(pDC->dpy, pDC->WXGC, FillTiled);
     //modifica el origin de llenado con el origen del brush
     XSetTSOrigin(pDC->dpy, pDC->WXGC, 0,0);
     if (!pHatchBrush->flag)
     {
        // crea un pixmap con el hatched asociado
        idPixmap = XCreatePixmapFromBitmapData(
          pDC->dpy,
          (Drawable)pDC->XWin,
          hatch_bits[pHatchBrush->fnStyle],
          hatch_width,
          hatch_height,
          pHatchBrush->colorBrush,
          RGB(255, 255, 255),
          DefaultDepth(pDC->dpy, DefaultScreen(pDC->dpy)));
        if (idPixmap > 0)
        {
          pHatchBrush->idPixmap=idPixmap;
          pHatchBrush->flag=TRUE;
        }
     }
     // modifica el tile del graphics context con el pixmap creado
     XSetTile(pDC->dpy, pDC->WXGC, pHatchBrush->idPixmap);
     break;
   }
   case BS_SOLID:
   {
     // modifica el color de foreground con el color del brush
     XSetForeground(pDC->dpy, pDC->WXGC,
        ((WXSolidBrush *)pBrush)->colorBrush);
     // modifica el modo de llenado con FillSolid
     XSetFillStyle(pDC->dpy, pDC->WXGC, FillSolid);
     break;
    }
   case BS_DIBPATTERN:
   {
     WXBitmap      *pBitmapBrush;

     pBitmapBrush=(WXBitmap *)
       LocalLock(((WXPatternBrush *)pBrush)->Bitmap);
     // modifica el modo de llenado con FillTiled
     XSetFillStyle(pDC->dpy, pDC->WXGC, FillTiled);
     // modifica el origin de llenado con el origen del brush
     XSetTSOrigin(pDC->dpy, pDC->WXGC, 0,0);
```

```
        if (!pBitmapBrush->flag)
        {
          // crea un pixmap con el pattern asociado
          idPixmap = XCreatePixmapFromBitmapData(
            pDC->dpy,
            pDC->XWin,
            pBitmapBrush->pBitmap,
            pBitmapBrush->width,
            pBitmapBrush->height,
            RGB(0, 0, 0),
            RGB(255, 255, 255),
            DefaultDepth(pDC->dpy, DefaultScreen(pDC->dpy)));
          if (idPixmap > 0)
          {
            pBitmapBrush->idPixmap=idPixmap;
            pBitmapBrush->flag=TRUE;
          }
        }
        // modifica el tile del graphics context con el pixmap creado
        XSetTile(pDC->dpy, pDC->WXGC, pBitmapBrush->idPixmap);
        LocalUnlock(((WXPatternBrush *)pBrush)->Bitmap);
        break;
      }
    }
   break;
}
case WXBitmapObject:
  {
   WXBitmap *pBitmap;

   pBitmap = (WXBitmap*)pObject;
   // se guarda el handle del bitmap anterior para devolverlo
   oldObject = pDC->hWXBitmap;
   if (!pBitmap->flag)
      {
      // crea el pixmap asociado con el bitmap a seleccionar
      idPixmap = XCreatePixmapFromBitmapData(
        pDC->dpy,
        (Drawable)pDC->XWin,
        pBitmap->pBitmap,
        pBitmap->width,
        pBitmap->height,
        RGB(0, 0, 0),
        RGB(255, 255, 255),
        DefaultDepth(pDC->dpy, DefaultScreen(pDC->dpy)));
      if (idPixmap > 0)
      {
        pBitmap->idPixmap=idPixmap;
        pBitmap->flag=TRUE;
        }
   }
   pDC->hWXBitmap=hgdiobj;
   break;
  }
```

```
        case WXDCObject:
          break;
        }
      LocalUnlock(hdc);
      LocalUnlock(hgdiobj);
      return(oldObject);
    }
    return(0);
}
```

5. 3. 2. 2. 2. 1. 2. WXDeleteObject

La función *WXDeleteObject* libera toda la memoria asociada con el objeto.(pens, brushes, fonts, bitmaps, y palettes.). Esta función usa las funciones de *Xlib XFreeCursor, XFreePixmap, XFreeFont, XFreeColormap, XDestroyRegion* para liberar cualquiera de los recursos asociados al objeto.

La función retorna un valor distinto a 0 si fue valida, en caso contrario cero. Luego de la liberación el handle asignado al objeto no es valido

```
BOOL WDDeleteObject(HGDIOBJ hgdiobj)
{
  WXObject  *pObject;

  pObject=(WXObject *)LocalLock(hgdiobj);
  if (pObject!=NULL)
  {
    switch (pObject->typeObject)
    {
      case WXCursorObject:
      {
        WXCursor *pCursor;

        pCursor = (WXCursor*)pObject;
        if (pCursor->cursor>0)
          // Libero  la memoria asociada con el cursor
          XFreeCursor( pCursor->dpy, pCursor->cursor);
        break;
      }
      case WXBrushObject:
      {
        WXBrush *pBrush;

        pBrush = (WXBrush*)pObject;
        switch (pBrush->typeBrush)
        {
            case BS_HATCHED:
          {
            WXHatchBrush *pHatchBrush;

            pHatchBrush= (WXHatchBrush *)pBrush;
            if (pHatchBrush->idPixmap>0)
              // Libero la memoria asociada con el pixmap
              XFreePixmap( pHatchBrush->dpy, pHatchBrush->idPixmap);
```

```
            break;
          }
        }
        break;
        }
      case WXPaletteObject:
       {
        WXPalette *pPalette;

        pPalette = (WXPalette *)pObject;
         if (pPalette->colors>0)
        {
          XFreeColormap(pPalette->dpy, pPalette->colors);
          break;
         }
      case WXBitmapObject:
      {
        WXBitmap *pBitmap;

        pBitmap = (WXBitmap*)pObject;
        free(pBitmap->pBitmap);
        if (pBitmap->idPixmap>0)
          // Libero la memoria asociada con el pixmap
          XFreePixmap( pBitmap->dpy, pBitmap->idPixmap);
        break;
      }
      case WXRegionObj:
      {
        WXRegion *pRegion;

        pRegion = (WXRegion*)pObject;
        if (pRegion->idRgn > 0)
          // Libero la memoria asociada con la region
          XDestroyRegion(pRegion->idRgn);
         break;
      }
    }
     LocalUnlock(hgdiobj);
    LocalFree(hgdiobj);
     return(1);
  }
  return(0);
}
```

5. 3. 2. 2. 2. 1. 3. WXSetBkColor

El WXDC tiene un atributo el cual indica el color del background. La función *WXSetBkColor* modifica el mismo por el color especificado. Para realizar esto en Xlib se usa la función *XSetBackground* en cual modifica el atributo background en el WXGC. El valor por default es blanco. El valor de retorno es un valor RGB del color previo del background, si la función es valida. En caso contrario retorna el valor 0x80000000.

Pseudocodigo

// modifica el background en el graphics context
`XSetBackground`

5. 3. 2. 2. 2. 1. 4. WXGetBkColor

La función *WXGetBkColor* retorna el color del background del GC asociado al WXDC usando la función de Xlib *XGetGCValues*.

Pseudocodigo

// devuelve el background con la mascara GCBackground
`XGetGCValues`

5. 3. 2. 2. 2. 1. 5. WXSetROP2

La función *WXSetROP2* modifica el modo de dibujo. El modo de dibujo es la forma de como los colores del pen y en el llenado de los figuras (rectángulo, etc) son combinados con el color de la superficie de la pantalla.

En Xlib existe el atributo *function* en el recurso GC que es modificado por la función *XSetFunction* y controla el modo de dibujo en X. El mapeo del modo de dibujo entre ambos sistemas es:

Windows	X Window	Operación
R2_BLACK	GXclear	0
R2_NOTMERGEPEN	GXnor	~(P|D)
R2_MASKNOTPEN	GXandInverted	~P&D
R2_NOTCOPYPEN	GXcopyInverted	~P
R2_MASKPENNOT	GXandReverse	P&~D
R2_NOT	GXinvert	~D
R2_XORPEN	GXxor	P^D
R2_NOTMASKPEN	GXnand	~(P&D)
R2_MASKPEN	GXand	P&D
R2_NOTXORPEN	GXequiv	~(P^D)
R2_NOP	GXnoop	D
R2_MERGENOTPEN	GXorInverted	~P|D
R2_COPYPEN	GXcopy	P
R2_MERGEPENNOT	GXorReverse	P|~D
R2_MERGEPEN	GXor	P|D
R2_WHITE	GXset	1

Pseudocodigo

// realiza el matching usando la tabla anterior
// se guarda la función anterior para devolverla
`XSetFunction`
// modifica la función logica en el graphics context

5. 3. 2. 2. 2. 1. 6. WXGetROP2

La función *WXGetROP2* retorna el modo de dibujo usando la función *XGetGCValues* de Xlib.

Pseudocodigo

```
// devuelve el función logica de X con la mascara GCFunction
// realiza el matching usando la tabla anterior
XGetGCValues
return( La función en Windows);
```

5. 3. 2. 2. 2. 1. 7. WXSetPolyFillMode

La función *WXSetPolyFillMode* modifica el modo de llenado de los polígonos. En Xlib usando el atributo *fill_rule* del GC logramos el mismo efecto. Para modificar este atributo se usa la función *XSetFillRule*. Mapeamos el modo de llenado usando el atributo *fill_rule* del WXGC del WXDC. Los tipos de filling son:

Windows	X Window	Acción
ALTERNATE (default)	EvenOddRule	No pinta la intercepción.
WINDING	WindingRule	Pinta todo.

Pseudocodigo

```
// realiza el matching usando la tabla anterior
// se guarda el fillrule función anterior para devolverla
// modifica el fillrule con el modo de fill calculado en el matching
XSetFillRule
```

5. 3. 2. 2. 2. 1. 8. WXGetPolyFillMode

La función *WXGetPolyFillMode* retorna el modo de llenado de polígonos usando la función *XGetGCValues* sobre el WXGC del WXDC.

Pseudocodigo

```
// devuelve el fillrule con la mascara GCFillRule
XGetGCValues
```

5. 3. 2. 2. 2. 2. Manipulación del objeto WXDC.

5. 3. 2. 2. 2. 2. 1. WXCreateCompatibleDC

La función *WXCreateCompatibleDC* crea un WXDC en memoria compatible con el WXDC ingresado como parámetro. Creamos un nuevo objeto WXDC.

Pseudocodigo

```
// guardo en el nuevo DC el tipo de objeto, el identificador de la conexion entre el cliente y el server, el
identificador de la ventana X, el GC y modifica la posición del pen en el origen.
pNewDC->header.typeObject = WXDCObject;
pNewDC->dpy= pDC->dpy;
```

```
pNewDC->XWin=pDC->XWin;
pNewDC->WXGC=pDC->WXGC;
pNewDC->PenPosition.x=0;
pNewDC->PenPosition.y=0;
// retorno el handle al nuevo DC.
return(hNewDC);
```

5. 3. 2. 2. 3. Funciones de WXPen

Por default las coordenadas de origen de un pen son (0,0). Como ya explicamos un pen tiene atributos como el ancho, estilo y color. En Windows las coordenadas de origen del pen se mantienen en el DC mientras que en X Windows no posee dicho punto, por esa razón en el objeto WXDC mantendrá esas coordenadas para poderlas usar en funciones de X como XDrawLine, la cual necesita dos puntos para poder dibujar una línea, la de origen y la de destino.

5. 3. 2. 2. 3. 1. WXGetCurrentPosition

La función *WXGetCurrentPosition* devuelve la posición actual del pen. Esta posición puede ser modificada usando la función *WXMoveTo*. La posición actual esta especificada por el atributo *PenPosition* del objeto WXDC. Este atributo es usado al realizar requerimiento gráficos usando el pen seleccionado en el WXDC.

Esta función retorna un WORD, en la parte baja la coordenada X y en la parte alta la coordenada Y de la posición del pen.

Pseudocodigo

```
// genero un long con la posición del pen
values = MAKELONG(pDC->PenPosition.x, pDC->PenPosition.y);
// retorno la posición del pen actual en la variable values
return(values);
```

5. 3. 2. 2. 3. 2. WXMoveTo

La función *WXMoveTo* modifica las coordenadas de origen de un objeto WXPen asociado con el WXDC.

Pseudocodigo

```
// genero un long con la posición del pen
values = MAKELONG(pDC->PenPosition.x, pDC->PenPosition.y);
// modifico la posición del pen con la nueva posición
pDC->PenPosition.x=x;
pDC->PenPosition.y=y;
return(values);
```

5. 3. 2. 2. 3. 3. WXLineTo

La función *WXLineTo* dibuja una línea en la ventana asociada al WXDC. Las coordenadas de destino son pasadas como parámetros y las de origen son tomadas del WXDC. Se usa la función de Xlib *XDrawLine*.

Pseudocodigo

```
// dibuja una linea
XDrawLine
```

5. 3. 2. 2. 3. 4. WXCreatePen

La función *WXCreatePen* crea un objeto WXPen con un estilo, ancho y color especificados. En Xlib existe la función *XSetLineAttributes*, la cual permite crear todos los estilos de pen. Esta función se llamara en el código de la función *WXSelectObject* debido a que necesitamos información como el identificador de la conexión entre el cliente y el server y el GC para llamar a la función *XSetLineAttributes* que están en el WXDC no conocida en esta función.

La función *WXCreatePen* crea un objeto *WXPen* modificando los atributos width, style y color pasados como parámetros como así también dash_list y dash_list_length usados en forma interna para dibujar los distintos tipos de líneas.

En Xlib también se pueden generar dash de distintos tamaños y la distancia entre ellos usando la función *XSetDashes* la cual modifica el atributo dashes del GC. La función *WXCreatePen* retorna el handle al pen creado, en caso contrario devuelve NULL. Los tipos de pen son:

```
PS_SOLID
PS_DASH
PS_DOT
PS_DASHDOT
PS_DASHDOTDOT
PS_NULL
PS_INSIDEFRAME
```

Pseudocodigo

```
si fnPenStyle = case PS_NULL:
{
  case PS_INSIDEFRAME:
  case PS_SOLID:
    pPen->dash_list = dash_list[0];
    pPen->dash_list_length = dash_list_length[0];
    style = LineSolid;
            break;
  case PS_DASH:
    pPen->dash_list = dash_list[1];
    pPen->dash_list_length = dash_list_length[1];
    style = LineOnOffDash;
    break;
  case PS_DOT:
    style = LineOnOffDash;
    break;
  case PS_DASHDOT:
    pPen->dash_list = dash_list[3];
    pPen->dash_list_length = dash_list_length[3];
    style = LineOnOffDash;
    break;
  case PS_DASHDOTDOT:
    pPen->dash_list = dash_list[4];
    pPen->dash_list_length = dash_list_length[4];
```

```
        style = LineOnOffDash;
        break;
}
pPen->header.typeObject = WXPenObject;
pPen->width = nWidth;
pPen->style = style;
pPen->color = clrref;
```

En Xlib existen tres estilos de líneas, las cuales definen como dibujar las secciones de una línea.

Line Solid Especifica que la toda línea será dibujada con un mismo color del pen.
LineOnOffDash Especifica que partes de la línea será dibujada con el color del pen
LineDoubleDash Especifica que partes de la línea será dibujada con el color del pen y otras partes con el color de background.

Definimos la cantidad de estilos de líneas como así también el tamaño de la lista de dash asociado a cada estilo.

```
NUMLINES          5
SOLID_LENGTH      2
DASH_LENGTH       2
DOT_LENGTH        2
DASHDOT_LENGTH    4
DASHDOTDOT        5
```

Lista de Tamaños de cada lista de dash.

```
static int dash_list_length[]=
{
   SOLID_LENGTH,
   DASH_LENGTH,
   DOT_LENGTH,
   DASHDOT_LENGTH,
   DASHDOTDOT,
};
```

Inicializo con valores las lista de dash de cada estilo.

```
static unsigned char solid[SOLID_LENGTH] = {0,0};
static unsigned char dash[DASH_LENGTH] = {4,2};
static unsigned char dot[DOT_LENGTH] = {2,2};
static unsigned char dashDot[DASHDOT_LENGTH] = {4,2,2,2};
static unsigned char dashDotDot[DASHDOTDOT_LENGTH] = {4,2,2,2,2};
```

Por Ejemplo la inicialización odd_dashed[DASHDOTDOT_LENGTH] = {1, 2, 3} quiere decir el primer dash con el color del pen es de tamaño 1, el segundo dash de color del background o la distancia al otro dash es de 2 y el ultimo dash es de tamaño 3. La inicialización dotted[SOLID_LENGTH] = {0,0} no se usa porque directamente se modifica el estilo LineSolid, para los otros estilos de Window se usa el estilo LineOnOffDash de X Window.

```
static unsigned char *dash_list[] =
{
```

```
  solid,
  dash,
  dot,
  dashDot,
  dashDotDot
};
```

Windows	X Window	Lista de Dash
PS_SOLID	LineSolid	
PS_DASH	LineOnOffDash	{4, 2}
PS_DOT	LineOnOffDash	{2, 2}
PS_DASHDOT	LineOnOffDash	{4, 2, 2, 2}
PS_DASHDOTDOT	LineOnOffDash	{4, 2 , 2, 2, 2}
PS_NULL	LineSolid	
PS_INSIDEFRAME	LineSolid	

Estos valores se sestean en los atributos **dash_list, dash_list_length* del objeto *WXPen* para luego al seleccionarlo con la función *WXSelectObject* se manden como parámetros de entradas en el llamado a *XSetDashes*

5. 3. 2. 2. 3. 5. WXCreatePenIndirect

La función *WXCreatePenIndirect* crea un crea un Pen con un estilo, color y patrón de bits especificado en la estructura llamada LOGPEN.

```
typedef struct
  UINT        lopnStyle;
  POINT     lopnWidth;
  COLORREF  lopnColor;
} LOGPEN;
```

El parámetro lplgpn de la función WXCreatePenIndirect apunta a una estructura LOGPEN. El valor de retorno es un handle a un pen si la función es valida, en caso contrario es NULL. El pseudocodig es similar al de WXCreatePen.

5. 3. 2. 2. 3. 6. WXRectangle

La función *WXRectangle* dibuja un rectángulo usando el pen seleccionado en el WXDC.Para dibujar un rectángulo esta función tiene cinco parámetros:

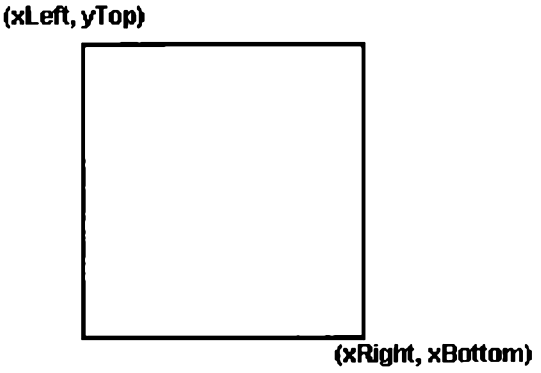
hdc El Device Context el cual contiene el pen.
nLeftRect
nTopRect Las coordenadas de la esquina superior izquierda.
nRightRect
nBottomRect Las coordenadas de la esquina inferior derecha.

Esta función retorna un valor diferente a cero si fue valida, en caso contrario cero. En X existe una función similar llamada *XDrawRectangle* que tiene como parámetros el el identificador a la conexion entre el cliente y el server, la ventana X, la posición superior izquierda y en lo que cambia

con respecto de Windows es que tiene como parámetros el ancho y el largo del rectángulo y no la posición inferior derecha. estos valores se calculan con 2 simples operaciones. (width = nRightRect - nLeftRect y height = nBottomRect - nTopRect)

Pseudocodigo

```
// calcular ancho y largo
// dibuja un rectangulo
XDrawRectangle
```



5. 3. 2. 2. 3. 7. WXArc

La función *WXArc* dibuja un arco, la función tiene los siguientes parámetros:

hdc	Especifica el DC.
nLeftRect, nTopRect	Las coordenadas de la esquina superior izquierda.
nRightRect, nBottomRect	Las coordenadas de la esquina inferior derecha.
nXStartArc, nYStartArc	Las coordenadas x e y del principio del arco.
nXEndArc, nYEndArc	Las coordenadas x e y del final del arco.

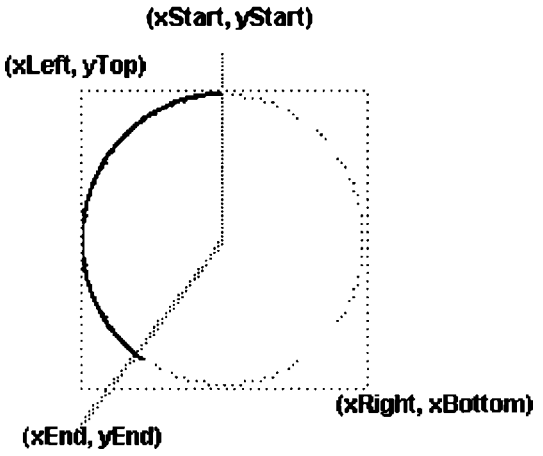
Esta función dibuja un Arco elíptico. La función WXArc devuelve un valor distinto a cero si la función fue valida en caso contrario cero. En X existe una función llamada *XDrawArc* la cual difiere de la Arc de Windows que en vez de usar como parámetros las coordenadas de principio y fin del arco necesita el ángulo de principio y fin.

La función *Angle*(int x1, int y1, int x2, int y2) calcula un ángulo dado 2 puntos, con esta función mapeamos las coordenadas de principio y fin a los ángulos de principio y fin. Para calcular el ángulo la función primero calcula la pendiente de la recta formada por esos 2 puntos y luego el arcotangente. Hay que tener en cuenta en que cuadrante están los puntos para este calculo.

Pseudocodigo

```
// calcular ancho, largo, angulo de principio y fin
// dibuja un arco
```

`XDrawArc`

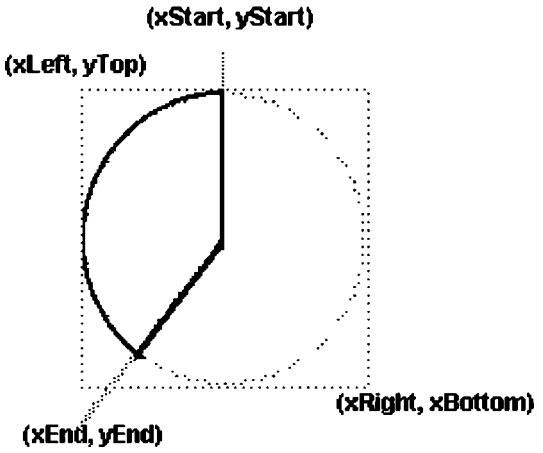


5. 3. 2. 2. 3. 8. WXPie

La función *WXPie* dibuja un pie. Se usan las funciones *XDrawArc* y *XDrawLine* de Xlib para dibujar un arco con dos lineas. Una linea desde el origen a un extremo del arco y la otra del origen al otro extremo.

Pseudocodigo

// calcular ancho, largo, angulo de principio y fin
// dibuja un arco
`XDrawArc`
// dibuja una linea
`XDrawLine`
// dibuja una linea
`XDrawLine`

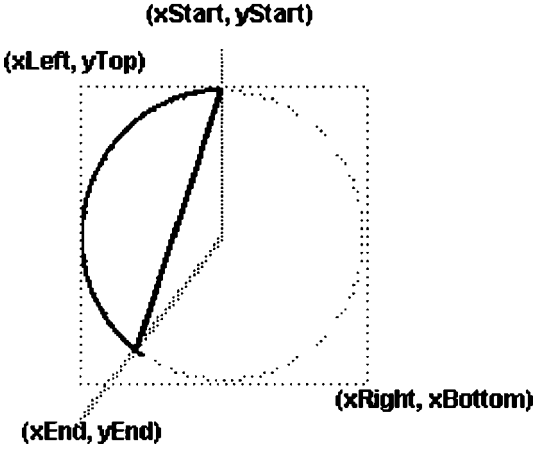


5. 3. 2. 2. 3. 9. WXChord

La función *WXChord* dibuja un chord. Se usan las funciones *XDrawArc* y *XDrawLine* de Xlib para dibujar un arco con una línea que unen los dos extremos del arco.

Pseudocodigo

```
// calcular ancho, largo, angulo de principio y fin
// dibuja un arco
XDrawArc
// dibuja una linea
XDrawLine
```

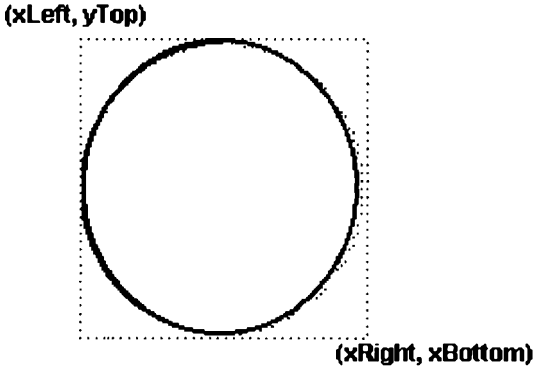


5. 3. 2. 2. 3. 10. WXEllipse

La función *WXEllipse* dibuja un elipse usando la función *XDrawArc* de Xlib con el ángulo de comienzo igual a 0 grados y el de final con 360 grados.

Pseudocodigo

```
// calcular ancho, largo, angulo de principio y fin
// dibuja un arco
XDrawArc completo es decir de 0 a 360 grados.
```



5. 3. 2. 2. 3. 11. WXFillRect

La función *WXFillRect* llena un rectángulo dado usando el brush especificado. Esta función usa la función *XSetFillStyle* para indicar el modo de llenado de la figura gráfica que puede ser *FillTiled* o *FillSolid,* también si es necesario crear un pixmap usa la función *XCreatePixmapFromBitmapData.* Por ultimo selecciona el pixmap creado en el GC usando la función *XSetTile*. Estas funciones se usan dependiendo del tipo de brush a utilizar. La función *WXFillRect* tiene como parametros:

hdc Especifica el device Context.
lprc Especifica el rectángulo a llenar.
hbr Especifica el Brush.

El brush debe ser creado usando una de las funciones brush *WXCreateHatchBrush*, *WXCreatePatternBrush*, o *WXCreateSolidBrush*.

Pseudocodigo

```
// guardo los valores anteriores
XGetGCValues
si pObject->typeObject es un  WXBrushObject:
   // se guarda el handle del brush anterior para devolverlo
   si pBrush->typeBrush = BS_HATCHED:
```

```
    // modifica el modo de llenado con FillTiled
    XSetFillStyle
    // modifica el origin de llenado con el origen del brush
    XSetTSOrigin
    // crea un pixmap con el hatched asociado
    XCreatePixmapFromBitmapData
    // modifica el tile del graphics context con el pixmap creado
    XSetTile
    // llena un rectangulo
    XFillRectangle
  si pBrush->typeBrush = case BS_SOLID:
    // setea el color de foreground con el color del brush
    XSetForeground
    // setea el modo de llenado con FillSolid
    XSetFillStyle
    // llena un rectangulo
    XFillRectangle
  si pBrush->typeBrush = BS_DIBPATTERN:
    // idem BS_HATCHED
// modifico con los valores anteriores
XSetFillStyle
XSetForeground
```

5. 3. 2. 2. 3. 12. WXSetPixel

La función *WXSetPixel* dibuja un pixel en un coordenada especifica con un color dado. Para mapear esta función usamos la función *XSetForeground*, para indicar el color del pixel y posteriormente usando la función *XDrawPoint* se dibuja el pixel. Los parametros de la función WXSetPixel son:

hdc	Especifica el Device context.
nXPos	
nYPos	Especifica la coordenada del pixel.
clrref	Especifica el color del pixel.

Pseudocodigo

```
// guardo el color del foreground
XGetGCValues
// modifica el color del foreground con el color del pixel
XSetForeground
// dibujo un pixel
XDrawPoint
// modifica el color del foreground con el color anterior
XSetForeground
```

5. 3. 2. 2. 4. Funciones de WXFont

El GDI de Windows ofrece una variedad de stock fonts. Para poder crear un font se usa la función *WXGetStockObject*. WXGetStockObject crea a handle a un font. luego la aplicación selecciona este font en el device context. En X es similar, la función se llama *XLoadFont*. Cuando la aplicación llame a la función *WXDeleteObject*, esta llamara a la función *XFreeFont*. La función

WXGetStockObjet recibe como parámetro el tipo de fonts, en X la función XLoadFont recibe como parámetro el

5. 3. 2. 2. 4. 1. WXTextOut

La función *WXTextOut* escribe un string en una posición especificada con el font seleccionado. En Xlib usamos la función *XDrawString*. Tiene como parametros:

hdc	Especifica el Device Context.
nXStart, nYStart	Especifica las coordenadas en donde se escribe el string.
lpszString	Especifica el Texto a escribir.
cbString	Especifica la cantidad de caracteres del texto.

Pseudocodigo

```
// dibuja un string
XDrawString
```

5. 3. 2. 2. 5. Funciones de WXPalette

Windows para usar colores maneja una paleta del sistema (*hardware*), la cual tiene los colores visualizados en el screen. Cuando una aplicación no crea ninguna paleta lógica, usa la paleta default, es decir usan los colores predefinidos en la paleta del sistema. Cuando la aplicación usa sus propios colores primero debe crear una paleta lógica con los colores a usar y luego usando las funciones de GDI. La aplicación temporariamente reemplaza la paleta del sistema por la paleta lógica, la cual permite que la aplicación use los colores de la paleta lógica. Existen tres funciones que Windows usa para lograr este objetivo:

CreatePalette	Crea una paletta logica.
SelectPalette	Selecciona una logica paleta para un DC.
RealizePalette.	Mapea todos los colores de la paleta logica del DC a la paleta del sistema.

En X Window existe el concepto de "colormap". El colormap contiene "colorcells" las cuales especifican el valor RGB, para acceder a cada entrada del colormap se usa como índice el valor del pixel. El numero de bits por pixel determina la cantidad de entradas disponibles en un colormap y el numero de colores que se pueden mostrar simultáneamente. X maneja los múltiples colormaps guardándolos en memoria. El proceso de instalación (installing) es el movimiento de un colormap virtual a un colormap por hardware. Cuando una aplicación crea un colormap virtual, este colormap debe estar indicado en el atributo de la ventana top-level para que el window manager pueda buscar el colormap a instalar. Para manejar los colormaps se usan las siguientes funciones:

XCreateColormap	Crea un colormap virtual.
XFreeColormap	Desinstala un especificado colormap virtual y libera los recursos asociados con el colormap.
XSetWindowColormap	Setea atributo colormap de la ventana.

5. 3. 2. 2. 5. 1. WXCreatePalette

La función *WXCreatePalette* crea una paleta lógica, el parámetro es un puntero a una estructura LOGPALETTE que contiene información de los colores de la paleta. Retorna un handle a una paleta lógica si la función es valida y en caso contrario NULL.

Usaremos la función de X llamada *XCreateColormap* para crear un colormap asociado a la paleta. Esta función tiene 4 parámetros: display, w, visual, y alloc. Los visual que se usaran de X serán: *GrayScale* (Monocromo / Escala de Grises), *PseudoColor* (RGB con un solo índice). El parámetro alloc es *AllocAll* que indica que se aloquen todas las celdas para ser compartidas. Una vez alocadas las celdas para poder modificarlas se usa la función *XStoreColors* la cual cambia múltiples entradas read/write del colormap en un solo llamado. Los parámetros color y ncolors de la función XStoreColors indican los colores a guardar y el numero de colores respectivamente.

Pseudocodigo

```
// crea un colormap
// convertir los colores de Windows a X Window
XCreateColormap
// modifica las colorcell del colormap
XStoreColors
palette->colors = colormap creado anteriormente
return(palette)
```

5. 3. 2. 2. 5. 2. WXSelectPalette

La función *WXSelectPalette* guarda el objeto *WXPalette* creado usando la función *WXCreatePalette* en el atributo palette del device context. Esta función reemplaza el WXPalette por uno nuevo. Esta función tiene como parámetros el handle de un objeto WXDC, el handle al objeto WXPalette y un parámetro que especifica el comportamiento de windows cuando mapea los valores RGB de la paleta a la paleta del sistema.

Esta función retorna un handle al objeto WXPalette anterior en el device context, si la función es valida, en caso contrario retorna NULL. Una aplicación puede seleccionar una Paleta lógica en mas de un device context, los cambios de la paleta afectan a todos los device context asociados con ella.

Pseudocodigo

```
// guardo el paleta anterior anterior
DC-> hWXPalette = palette->colors
Return(paleta anterior)
```

5. 3. 2. 2. 5. 3. WXRealizePalette

La función *WXRealizePalette* copia los colores de una paleta lógica en la paleta del sistema. El parámetro especifica en que objeto WXDC esta la paleta a mapear. Esta función retorna el numero de entradas en la paleta del sistema que fueron mapeadas. Como se explico anteriormente, X Windows usa colormaps en vez de paletas para usar los colores y el colormap por hardware, es decir, el colormap tiene los colores actualmente mostrados es el que esta asociado con la ventana que tiene el foco, por ese motivo la idea es que cuando se haga un llamado a la función *WXRealizePalette*, en su codigo se llama a la función *XSetWindowColormap* que asocia el colormap a la ventana.

Pseudocodigo

```
// modifica el colormap en el graphics context
XSetWindowColormap
```

5. 3. 2. 2. 6. Funciones de WXBitmap

5. 3. 2. 2. 6. 1. WXCreateBitmap

La función *WXCreateBitmap* crea un bitmap. Esta función guarda en memoria los bits del bitmap hasta que se pueda crear el pixmap en el server. Esto sucede porque al crear el pixmap necesitamos el identificador de la conexión entre el cliente y el server el cual es desconocido en esta función.

Para crear un bitmap usando la función *WXCreateBitmap* definimos un método de conversión que al recibir el conjunto de bits que forman el bitmap lo convierte al formato pixmap de Xlib, luego con los bits convertidos del pixmap se usa la función *XCreatePixmapFromBitmapData* de Xlib para crear el pixmap en el server. La función *WXCreateBitmap* tiene como parámetros:

nWidth Especifica el ancho del bitmap.
nHeight Especifica el largo del bitmap.
cbPlanes Especifica el numero de planos del bitmap. El numero de bits por plano es el producto de ancho, largo y bits por pixels. (nWidth*nHeight*cbBits).
cbBits Especifica el numero de bits por pixels.
lpvBits Buffer con los bits que forman el bitmap.

Retorna un handle del bitmap si la función es valida. En caso contrario este handle NULL.

Pseudocodigo

```
// guardo el ancho, largo y la profundidad de un bitmap en el objeto WXBitmap
pBtimap->header.typeObject = WXBitmapObject;
pBtimap->width=nWidth;
pBtimap->height=nHeight;
pBtimap->depth=cbPlanes;
// creo un conjunto de bytes
pBtimap->pBitmap= conjunto de bytes
```

5. 3. 2. 2. 6. 2. WXLoadBitmap

La función *WXLoadBitmap* carga a memoria un WXBitmap. Para lograr esto primero usando la función *LoadBitmap* de Windows cargamos un bitmap a memoria, con la función *GetObject* obtenemos las dimensiones del bitmap (plano, ancho, largo) y con la función *GetBitmapBits* obtenemos los bits que forman el bitmap, por ultimo se llama a la función *WXCreateBitmap* del WXGDI con los datos obtenidos para crear el WXBitmap. La función *WXLoadBitmap* tiene dos parámetros:

hinst Especifica el modulo el cual contiene el bitmap.
lpszBitmap Especifica el nombre del Bitmap en el recurso.

Esta función retorna el handle al bitmap si la función fue valida o NULL en caso contrario.

Pseudocodigo

```
// leo un bitmap
LoadBitmap
// leo información del bitmap (plano, ancho, largo)
GetObject
// leo los bytes de datos del bitmap
// convierto un bitmap a un pixmap
GetBitmapBits
// creo un pixmap (función de WXGDI)
WXCreateBitmap
```

5. 3. 2. 2. 6. 3. WXBitBlt

Una aplicación usa la función *WXBitBlt* para copiar pixels de una ventana a otra o desde un rectángulo fuente a uno destino. Para lograr esto primero creamos el pixmap usando la función *XCreateBitmapFromData* y luego usando la función *XCopyArea* copiamos el pixmap de un area a otra area. Si la función fue valida retorna un valor distinto a 0 en caso contrario retorna 0.

Pseudocodigo

```
// creo un pixmap
XCreateBitmapFromData
// copio un pixmap de una area a otra area
XCopyArea
```

5. 3. 2. 2. 6. 4. WXGetBitmapBits

La función *WXGetBitmapBits* copia los bits del bitmap en un buffer.

hbm Especifica en bitmap.
cbBuffer Especifica el numero de bytes a copiar.
lpvBits Especifica el puntero al buffer.

Esta función retorna los valores en el atributo pBitmap del objeto WXBtimap.

5. 3. 2. 2. 7. Funciones de WXCursor

5. 3. 2. 2. 7. 1. WXLoadCursor

La función *WXLoadCursor* carga un especifico cursor desde un modulo ejecutable. Usamos la función *XCreateFontCursor* para crear un cursor en Xlib. Mapeamos los cursores default de Windows con los cursores default de Xlib. Los parametros de la función *WXLoadCursor* son:

hinst Especifica el modulo el cual contiene el cursor.
pszCursor Especifica el nombre del Cursor en el recurso.

Esta función retorna el handle al cursor si la función fue valida o NULL en caso contrario.

Pseudocodigo

```
// guardo en elobjeto cursor el nombre del cursor, el tipo de objeto, el identificador del cursor
// en Xlib y el cursor predefinido en X.
pCursor->header.typeObject = WXCursorObject;
pCursor->nameCursor=pszCursor;
pCursor->shapeCursor=mapCursor(pszCursor);
// creo un cursor en Xlib
pCursor->cursor=XCreateFontCursor(display,pCursor-  >shapeCursor);
```

Para los cursores default de Windows se mapean con los cursores predefinos en Xlib. La función *mapCursor* busca el cursor en Xlib asociado con el cursor en Windows.

Windows	X Window
IDC_ARROW	XC_arrow
IDC_IBEAM	XC_xterm
IDC_WAIT	XC_watch
IDC_CROSS	XC_cross
IDC_UPARROW	XC_sb_up_arrow
IDC_SIZE	XC_fleur
IDC_ICON	XC_icon
IDC_SIZENWSE	XC_top_left_corner
IDC_SIZENESW	XC_top_right_corner
IDC_SIZEWE	XC_sb_h_double_arrow
IDC_SIZENS	XC_sb_v_double_arrow

5. 3. 2. 2. 8. Funciones de WXRegion

5. 3. 2. 2. 8. 1. WXCreateRectRgn

La función *WXCreateRectRgn* crea una región, la función *WXDeleteObject* destruye una región. En Xlib usamos la función *XCreateRegion* para crear una región vacia, luego usando *XUnionRectWithRegion* se modifica la región vacía con la nueva región.

Pseudocodigo

```
// inicializar un rectangulo con x, y, ancho y largo
XCreateRegion();
// crea una region vacia
XUnionRectWithRegion(&rectangle, region, regionDest);
// modifica la region con un rectangulo
pRegion->header.typeObject = WXRegionObject;
pRegion->idRgn = regionDest;
```

5. 3. 2. 2. 8. 2. WXSelectClipRgn

La función WX*SelectClipRgn* selecciona una región en el área de clipping de un WXDC. Esta región esta indicada en el objeto WXGC asociado al WXDC, para modificarlo se usa la función de Xlib llamada *XSetRegion*.

Pseudocodigo

```
// modifica el area de clipping
XSetRegion
```

Apéndice A - Personalidades.

A. 1. Soluciones del mercado para ejecutar aplicaciones Windows en otras plataformas.

En un sistema operativo (S.O.) una personalidad es un módulo de software, el cual puede o no ser parte integral del S.O., que permite ejecutar aplicaciones de otro S.O..

Aclaremos que WExX y una personalidad Windows sobre un S.O. con una UI basada en X intentan solucionar problemas conceptualmente diferentes. La analogía que existe es que WExX da soluciones a algunos de los problemas que una personalidad Windows debe resolver.

Nuestro objetivo es separar la GUI de aplicaciones Windows de 16 bits haciendo uso del X Window System.

Veremos que nuestra solución resuelve parte de los problemas que se presentan cuando se quieren *correr aplicaciones no nativas*, específicamente cuando se quieren correr applicaciones Windows sobre otras plataformas cuyo sistema de ventanas está basado en X.

Esta es una de las características de la nueva generación de sistemas operativos (S.O.), la posibilidad de correr software no nativo. La elección del S.O. ya no limitará la elección de las aplicaciones. Tener aplicaciones Macintosh, Windows y Unix compartiendo el mismo desktop está comenzando a ser y será algo standard.

Lo que sin embargo no es standard es la manera en que los S.O. implementan la característica de correr aplicaciones no nativas. OS/2, Windows NT, Unix, Workplace OS y Mac, todos ofrecen aproximaciones diferentes.

Para que una computadora ejecute software destinado a otra computadora (ej.: Mac corriendo aplicaciones Windows), la computadora debería ejecutar instrucciones que no son nativas para ella. El µP 680x0 no entiende código del µP 80x86, por lo tanto la personalidad debe emular instrucciones del 80x86 con código 680x0 externo (no microcódigo interno). El punto más crítico de la emulación es la performance.

Las aplicaciones de hoy en día corren bajo UIMS como Windows, Mac, Motif u Open Look bajo Unix, Next, etc.. Estas aplicaciones estan contínuamente invocando funciones del API del sistema de ventanas. Esta particularidad del software más el nivel de abstracción provisto por los APIs de GUI son características que explotan las personalidades.

Una personalidad implementa librerías que simulan las librerías del sistema de ventanas no nativo con funciones del sistema de ventanas nativo que están escritas con código nativo. Esta aproximación se llama translation (traducción). Por ejemplo, cuando se hace una llamada para abrir una ventana, la personalidad invoca a la rutina de abrir ventana provista por las librerías nativas del S.O.. Por lo tanto, en aquellas secciones de código en las que se invoca el ABI (Application Binary Interfase) del sistema de ventanas, la aplicación puede alcanzar e incluso exceder su performance en su procesador nativo ya que se evita la emulación de instrucciones.

Apple dice que una aplicación Mac emplea entre el 60% y el 80% de su tiempo de procesamiento ejecutando código del Toolbox (servicios de UI y QuickDraw -motor gráfico de las Mac, análogo al GDI de Windows-). SunSelect dice que una aplicación Windows emplea un porcentaje similar de su tiempo ejecutando código de los módulos de Windows. En consecuencia, **existe una pérdida de performance mucho menor en la emulación de aplicaciones basadas en una GUI**.

Existen fuertes presiones del mercado que obligan a las empresas de software a considerar que el soporte de múltiples personalidades sea un punto crucial de cualquier S.O. exitoso. Afortunadamente, la modularidad de la nueva generación de S.O. simplifica el soporte de múltiples personalidades. Es

más fácil incorporar al S.O. módulos de software que implementen emulación de procesadores y traducción de librerías de UIMS.

A. 2. Sistemas Operativos y sus personalidades soportadas

Entre los S.O. avanzados que incorporan múltiples personalidades están OS/2 2.x y Workplace OS de IBM (sucesor de OS/2 basado en el microkernel Match 3.0), Microsoft Windows NT, PowerOpen de PowerOpen Association y versiones de Unix de Sun Microsystems (Solaris), IBM (AIX) y Hewlett-Packard (HP-UX). Además existen proveedores de terceras partes que ofrecen personalidas en forma separada (ver gráfico *Sistemas Operativos que ofrecen múltiples personalidades*).

Windows NT ofrece 5 personalidades : DOS, Windows 16 bits, Win32 (32 bits), OS/2 1.x modo caracter y la especificación Posix (Portable Operating System Interface based on Unix) IEEE 1003.1 y parte de la 1003.2.

A modo ilustrativo se describe como Windows NT corre aplicaciones Windows de 16 bits, ver 'Windows on Windows'.

Windows on Windows.

Window NT corre aplicaciones Win16 mediante el módulo WOW (Windows 16 on Win32) que es escencialmente una VDM (Virtual DOS Machine) multithreaded, donde cada thread ejecuta una aplicación Windows de 16 bits. Una VDM es un proceso NT que provee una sesión DOS. El módulo WOW invoca funciones del subsistema Win32 (convirtiendo - thunking - las llamadas de 16 bits a 32 bits) concernientes a la administración de ventanas y primitivas gráficas para cada una de las aplicaciones Win16. El subsistema Win32 implementa el sistema de ventanas y provee un nivel de abstracción a los servicios de sistema de NT. El módulo WOW, desde el punto de vista de la entrada de usuario, se comporta como una aplicacion Windows de 32 bits.

Sistemas Operativos que ofrecen múltiples personalidades.

	OS/2 2.x	Workplace OS	Windows NT	PowerOpen (con MAS)	Unix (con WABI)
Vendedor	IBM	IBM	Microsoft	PowerOpen Association	SunSoft (Solaris), IBM (AIX), Hewlett-Packard (HP-UX), Novell (SVR4.2)
Personalidades disponibles	DOS, Windows 3.1	DOS, Windows 3.1, OS/2, AIX	DOS, Windows 3.1, Win32, OS/2 1.x, Posix	Macintosh, AIX	Windows 3.1
Look and feel	Desktop Windows completo en un ventana Workplace Shell	OS/2 Workplace Shell o Unix CDE	Windows	Desktop Mac completo en una ventana X.	Interfase externa Motif
Aplicaciones soportadas	Aplicaciones. Windows 3.1.	*desconocidas (el S.O. está en desarrollo)*	Aplicaciones. Windows que no acceden al hardware, OS/2 modo carácter y POSIX recompil. para NT.	System 7	13 aplicaciones. Windows

A. 3. Dos alternativas para correr aplicaciones Windows en plataformas Unix: WABI de SunSelect vs. SoftWindows de Insignia Solutions.

WABI.

WABI (Windows Application Binary Interfase) se basa en los servicios de sistema de Unix (manejo de archivos, administración de memoria, comunicación entre procesos etc.), y en el X Window System para todo lo relacionado con la GUI.

La ejecución de una aplicación Windows por parte de WABI consiste en decodificar y emular cada una de las instrucciones 80x86 hasta que se encuentra una invocación a DOS o Windows, en cuyo caso el emulador pasa a modo nativo realizando la función DOS o Windows haciendo las invocaciones necesarias a X, Unix u otras facilidades. Para esto WABI tiene que implementar el soporte de linking dinámico de Windows.

WABI preserva el look de la interfase interna de una aplicación Windows; no ocurre lo mismo con la interfase externa de las aplicaciones que bien sabemos es política del window manager[1]. Los desktops de las versiones de Unix para los cuales está implementado WABI están basados en Motif, en cuyo caso el window manager es mwm (motif window manager)[2].

En lugar de correr el desktop completo de Windows dentro de una ventana, como lo hace SoftWindows de Insignia Solutions, WABI abre una nueva ventana en el desktop X para cada aplicación Windows.

En lo que respecta al GUI, WABI provee librerías de rutinas que reescriben el API de Windows de 16 bits sobre un S.O. de 32 bits a nivel del sistema de ventanas base de X, es decir haciendo uso de la funcionalidad provista por Xlib. Como sabemos, Xlib no impone ninguna política de UI, por lo tanto WABI preserva el look de las aplicaciones Windows.

WABI no soporta muchas de las características de Windows incluyendo las extensiones de multimedia, ODBC (Open Database Connectivity), MAPI (Messaging API) y el networking, que está limitado al acceso de sistemas de archivos remotos e impresoras.

La primera versión de WABI soporta el API de Windows 3.1, con los servicios de DDE y OLE 1.0 soportados vía emulación 80x86. El manejo de fonts TrueType fue adquirido a otra empresa especializada en el desarrollo de tecnologías de fonts.

Son sólo 13 las aplicaciones Windows "WABI - certified" que se garantizan que corren bajo WABI: Lotus 1-2-3 y Ami Pro; WordPerfect; Microsoft Word, Excel, PowerPoint y Project; Borland Paradox y Quattro Pro; Aldus PageMaker; Harvard Graphics; Corel Draw y Procomm Plus.

Sun distribuye WABI con el S.O. Solaris y vendió la licencia del producto a IBM, HP y Novell para que lo incluyan en sus versiones de Unix (Windows estaría entonces corriendo en el 70 % de las workstations basadas en Unix).

SoftWindows.

La solución provista por Insignia Solutions, llamada SoftWindows, corre aplicaciones. Windows en workstations Unix, pero aquí termina la semejanza con WABI.

SoftWindows emula, por software, Windows 3.1 y DOS sobre distintas plataformas Unix. A pesar de disponer del código fuente de Windows, no es posible pensar en una solución basada en la

[1] Es posible tener un window manager con el look de Windows.

[2] Open Look tiene su propio window manager, olwm (open look window manager), el cual sigue los lineamientos del Toolkit Open Look.

recompilación ya que gran parte del código de Windows y la mayor parte del de DOS no son portables, están escritos en Assembler.

SoftWindows corre el desktop Windows completo en una ventana de la UI nativa y, debido a que usa el código fuente de Windows, SoftWindows corre un mayor número de aplicaciones que WABI.

La ejecución de cada línea de código Windows crea un gran overhead, especialmente porque Windows es una aplicación de 16 bits que corre arriba de DOS. Por el contrario, Unix es un S.O. de 32 bits cuyos servicios están implementados más eficientemente. Por esto último SunSelect argumenta que al mapear las funciones de Windows a Unix, las aplicaciones Windows sobre WABI pueden ser más eficientes que una aplicación Windows 80x86, especialmente aquellas que hagan un alto uso de funciones gráficas.

Para fines del '94 se había planificado que SoftWindows correría en workstations Unix de Sun, HP, IBM, DEC, Next, Silicon Graphics y Power Macs.

Conceptualmente, la aproximación de SoftWindows es similar, en cuanto a la emulación de instrucciones, a la que utiliza NT para correr aplicaciones Windows en plataformas no-80x86.

Actualmente la solución de SoftWindows se ha extendido al API Win32s, también sobre plataformas basadas en el S.O. Unix.

A. 4. Otras personalidades

Apple ha apostado a la emulación, sus workstations basadas en el procesador PowerPC emulan por hardware al procesador 680x0 para poder correr su S.O. System 7 y así poder ejecutar aplicaciones destinadas al System 7.

Apple está también desarrollando una personalidad Mac que corre en sistemas Unix. La primera versión, MAS (Machintosh Application Services) corre en workstations basadas en el procesador Power-PC corriendo la versión PowerOpen de Unix y permite ejecutar software Mac 680x0 o software Mac basado en el PowerPC.

MAS pone el desktop completo de Mac en una ventana del desktop X. MAS mapea las funciones del Toolbox directamente al sistema de ventanas base de X y no hacen uso de los Toolkits basados en X como Motif y Open Look, logrando de esta manera preservar el look and feel de Mac.

Apple también está desarrollando MAE (Machintosh Application Environment) que es un emulador de Mac sobre Unix que corre una sesión del System 7 (S.O. de las Mac) en una ventana del desktop X en workstations Sun (Solaris) y HP (HP-UX). MAE preserva el look and feel Mac.

SoftWindows, MAS y MAE son **aproximaciones de desktop completo**.

La empresa Quorum Software Systems desarrolló el producto Equal, una personalidad Mac para workstations Sun y Silicon Graphics basadas en Unix. La tecnología utilizada está basada en la emulación 680x0 (para código no-Toolbox) y en el mapping de las funciones gráficas y de administración de ventanas de Mac a funciones del Toolkit Motif y de la librería Xlib. El producto pone cada aplicación Mac en una ventana del desktop Unix, pero como mapea a funciones de un Toolkit, el estilo del GUI es el del Toolkit.

Tanto WABI, Equal como nuestra solución son **aproximaciones basadas en aplicaciones**.

Personalidades de 3ras partes.

	Macintosh Application Services	**Macintosh Application Environment**	**Liken**	**Equal Application Adapter ***	**SoftPC**	**SoftWindows**	**WABI ***
Proveedor	Apple	Apple	Andataco	Quorum Software Systems	Insignia Solutions	Insignia Solutions	SunSelect
Personalidad esoportadas	MAC	MAC	MAC	MAC	DOS, Windows 3.1	Windows 3.1	Windows 3.1
Tecnología	Fue portado parte del Toolbox + emulación.	Fue portado parte del Toolbox + emulación.	Reescribir el código en ROM de las MACs (incluye parte del System 6 (incluye)) + emulación 680x0 y hardware MAC.	Se reescribieron las rutinas del Toolbox a Xlib, Motif / OpenLook + emulación 680x0 p/ código no-Toolbox	Emulación del 80x86 y el hardware de PC.	Principalmente emulación, se portó muy poco código.	Reescribir el API Win16 haciendo uso de Xlib.
Look & feel	MAC System 7 en una ventana X (se preserva)	MAC System 7 en una ventana X (se preserva, excepto lo referente a ICCCM)	MAC System 6 en una ventana X (se preserva)	X/Motif o X/OpenLook , cada app. MAC en una ventana X. (se pierde)	Aplicaciones . DOS en ventanas X y Desktop Windows Completo en una ventana X (se preserva)	Desktop Windows completo en una ventana X (se preserva)	Cada app. Windows en una ventana X (se preserva, excepto lo referente a ICCCM)
S. Ops. soportados	Unix (las versiones PowerOpen)	Unix: Solaris y HP-UX	Unix: Solaris y HP-UX	Unix: Solaris (Sun) y Silicon Graphics	MAC y Unix (muchas versiones)	Unix: Solaris, HP-UX, AIX, Silicon Graphics, DEC OSF/1	Unix: Solaris, AIX, HP-UX y versiones basadas en SVR4

* La tecnología utilizada se conoce como 'reversed-engineered' (a partir de la especificación reescribir parte del System 7 en caso de Liken y el API Win16 en caso de WABI).

A. 5. Wind/U

Todos estos productos (personalidades) permiten portar aplicaciones a nivel binario, es decir, las aplicaciones no se modifican.

Otra manera de permitir la portabilidad es a nivel del código fuente, para lo cual es necesario disponer de :

- código fuente para poder recompilarlo en la plataforma destino
- implementación del API que usan las aplicaciones en la plataforma origen haciendo uso del API sobre la plataforma destino.

Wind/U permite este tipo de portabilidad para aplicaciones Windows. Wind/U es un conjunto de librerías que implementan el API Win16 completo (ya existe la versión Win32s) en plataformas Unix.

Wind/U permite que una aplicación Windows, escrita en C o en C++ utilizando MFC (Microsoft Foundation Classes), corra como una aplicación Unix/Motif nativa, soportando características importantes de Windows tales como DDE, MDI y Common Dialogs. Las aplicaciones Windows solo deben ser recompiladas en Unix y enlazadas con las librerías de Wind/U. Las aplicaciones Windows recompiladas con Wind/U adquieren el look de Motif.

Wind/U implementa la característica MDI, pero no con Motif, ya que ningún toolkit X soporta este concepto. Esto es así ya que MDI viola claramente el concepto de separación de interfases de una aplicación basada en X.

Apéndice B - X Window System, Version 11, Release 6[1] (X11R6)

B. 1. Nuevas características y mejoras introducidas en X11R6.

Todas las librerías, protocolos y servers son compatibles hacia arriba con R5, es decir, los clientes basados en X11R5 siguen funcionando con las librerías y servers de R6.

A continuación se describen algunas de las nuevas características y mejoras que pudieran estar relacionadas con nuestro objetivo:

Soporte de Windows NT

Aunque nuestra solución en principio no está pensada para ser portada a Windows NT, es importante saber que X11R6 puede compilarse para Microsoft Windows NT.

Las librerías Xt, Xaw y Xmu no se construyen como DLLs sino como librerías estáticas.

Como novedad, **Xlib soporta acceso multithreaded sobre una única conexión con el diplay**. Las funciones de Xlib bloquean la estructura display, causando que otros threads que invocan a funciones de Xlib sean suspendidos hasta que el primer thread la libere, pero los threads dentro de Xlib que están esperando leer o escribir desde el X Server no mantienen el display bloqueado, posibilitando, por ejemplo, que un thread esperando en XNextEvent no impida a otros threads que envíen requerimientos al server.

ICEP (Inter-Client Exchange Protocol)

ICE define un framework sobre el cual pueden construirse otros protocolos de comunicación.

Soporta multiplexar múltiples protocolos sobre un único transporte. ICElib es el API a todos los mecanismos provistos por ICE. En particular la versión de ICCCM de X11R6 es un ejemplo de un protocolo construído utilizando ICE.

ICEP podría ofrecer una alternativa para implementar comunicación de aplicaciones Windows (por ejemplo por DDE) que no estubieran ejecutándose en el mismo nodo.

LBX (Low Bandwidth X).

LBX es un nuevo standard incorporado a la distribución del X Window System para correr aplicaciones X sobre lineas series, WANs u otros links de baja velocidad. El propósito de este standard es mejorar la respuesta interactiva de aplicaciones X que corren sobre transportes de bajo ancho de banda.

LBX logra esto interponiendo un pseudo-server entre los clientes X y el X server. **El pseudo-server intercepta los paquetes X que viajan por la conexión X, los junta y los comprime**. El X Server en el otro extremo descomprime el stream de datos separando los requerimientos (esto hace que los X Servers actuales no sean compatibles con LBX).

El objetivo es lograr que se puedan usar transparentemente aplicaciones X sobre modems a 9600 bps.

Las caraterísticas básicas de LBX son:

- Compresión del stream de datos binario.

[1] La primer distribución del X Window System, Version 11, **Release 6** fue provista el 16 de **Marzo de 1994**.

- Compresión Delta de los paquetes X (representando los paquetes como diferencia de paquetes previamente enviados).
- Sustitución de algunos requerimientos gráficos por otros en forma equivalente (puntos, líneas, rectángulos, arcos).
- Caching de ciertos datos en el pseudo-server para evitar que sean transmitidos múltiples veces.
- Restringir el flujo de eventos MotionNotify.

Apéndice C - Entorno de Desarrollo

C. 1. Configuración de la red TCP/IP utilizada

Para las pruebas de desarrollo hemos implementado un 'mini' laboratorio compuesto por 4 PCs con placas de red Ethernet. Utilizamos **Windows for Workgroup 3.11** (WFW), el cual incluye los protocolos NetBEUI e IPX/SPX como transportes 'built-in', para la compartición de recursos (archivos e impresoras).

Como el transporte utilizado por el protocolo X (protocolo base del X Window System) es principalmente TCP/IP, necesitamos también de implementaciones de estos protocolos para Windows (ver *TCP/IP Software* mas adelante) que no son incluídos por WFW.

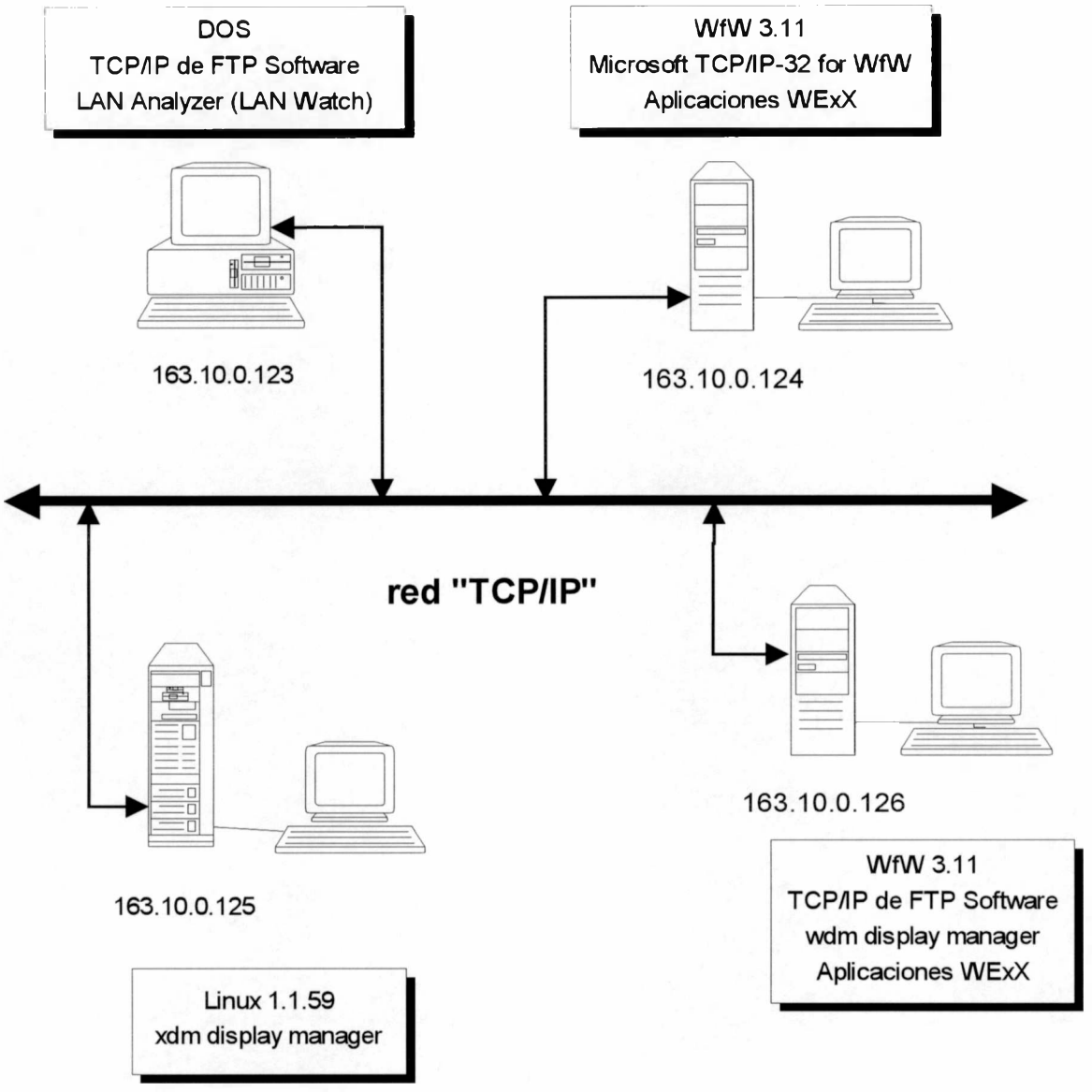
Como parte de la instalación de una implementación de TCP/IP, es requisito indispensable la definición de las direcciones IP. En este caso utilizamos las direcciones IP de clase B asignada a la red de la U.N.L.P. (163.10), aunque debido al tamaño reducido de nuestro laboratorio (nuestra red está compuesta por un pequeño número de hosts) hubiera sido mas que suficiente utilizar direcciones IP de clase C (ver en la siguiente pagina gráfico *Entorno de trabajo*).

C. 2. TCP/IP Software

Existen muchas implementaciones de la familia de protocolos TCP/IP para Windows, todos con la interfase standard Windows Sockets. Opcionalmente estos productos incluyen algunas implementaciones de otros protocolos de nivel de aplicación de esta familia como FTP (File Transfer Protocol), NFS (Network File System), Telnet, etc.

Uno de las implementaciones mas conocidas de TCP/IP para Windows es **PC/TCP de FTP Software Inc.**, y que es una de las que utilizamos en nuestro desarrollo. La otra implementación de WinSockets utilizada fue la de Microsoft (**Microsoft TCP/IP**), en sus dos sabores: 16 bits (implementación basada en TSRs) y 32 bits (implementación basada en VxDs)

Entorno de trabajo.



C. 3. Recompilación del código fuente del X Window System

Una obstáculo que debimos superar y que hasta ahora no hemos detallado es la **recompilación del código fuente de la distribución del X Window System**. Los fuentes del X Window System son de dominio público y pueden ser obtenidos desde Internet via FTP desde diferentes maquinas entre las que se encuentra **expo.lcs.mit.edu**.

Si bien los fuentes del X Window System están pensados para ser portados sobre diferentes plataformas (además de adherirse al standard ANSI C, tienen consideraciones especiales para funcionar sobre distintas arquitecturas), principalmente éstas comprenden sistemas operativos UNIX y sobre arquitecturas que no son Intel (DEC, HP, IBM, Sun, etc.). Debido a esto, las principales diferencias que debieron ser tenidas en cuenta para la recompilación de los fuentes del X Window System en Windows 3.1 (obviamente sobre la arquitectura Intel) fueron:

- **Tamaño de los enteros** : Windows es un entorno operativo de 16 bits, por lo que un entero desde cualquier entorno de programación tiene un tamaño de 16 bits, mientras que en los sistemas operativos UNIX los enteros son de 32 bits.

- **Interface de programación al protocolo de transporte** : la interface de programación mas difundida existente para acceder a los servicios de TCP (el principal protocolo de transporte utilizado por el X Window System y el utilizado por WExX para comunicarse con los X servers) es **sockets.** , y en particular la implementación realizada en la Universidad de California en Berkley (BSD Sockets). La especificación de ésta interface en Windows es llamada **Windows Sockets**[1] . Windows Sockets extiende la funcionalidad a la provista por los sockets standards, principalmente para adherirse al modelo de programación de Windows. Existe una diferencia fundamental entre los sockets standard y los sockets de Windows; un socket en la implementación standard es un descriptor de archivos, por lo que es posible aplicar a un socket cualquier función de manipulación de archivos, mientras que en Windows no ocurre lo mismo.

Para solucionar éstas diferencias fue necesario modificar los archivos fuentes del X Window System y prepararlos para funcionar sobre una arquitectura Intel y en un entorno de 16 bits como es Windows 3.x.

C. 4. X Servers para Windows

Un X Server para Windows hace el 'mapping' inverso al realizado por WExX, es decir, mapea requerimientos X en llamados a funciones del API de Windows y mensajes Windows en eventos X.

Existen muchas empresas que proporcionan implementaciones de X Servers para Windows, que difieren principalmente en performance. Los X servers con los cuales probamos nuestro desarrollo son :

- **X Vision** de Visionware Ltd.
- **Multiview/X** de JSB Corporation.
- **Exceed/W** de Hummingbird Communications Ltd.

Todos estos X Servers implementan X11R5. Sólo Multiview/X no implementa el protocolo XDMCP. Exceed/W incluye un tracer de PDUs del protocolo X y de XDMCP, utilitario fundamental para el debugging de aplicaciones X.

Todos estos X Servers pueden operar en dos modos, en modo 'single window' y en modo 'multiple window'. En el modo 'single window' el X Server 'pone' el desktop completo del entorno gráfico en una ventana window; en el modo 'multiple window' cada cliente X tendrá una o más ventanas windows, es decir que Windows provee la funcionalidad del window manager a diferencia del modo anterior donde se utiliza el window manager que corre en el desktop X y por lo tanto es preservado el look & feel de la interfase externa a la aplicación.

Existe un modo alternativo de operación no implementado por ninguno de estos tres X Servers, en donde las aplicaciones Windows y los clientes X están todos en el desktop Windows, pero el X Server preserva el look & feel de la interfase externa a la aplicación de los clientes X. Para lograr esto el X Server redibuja el área cliente de las ventanas Windows con los requerimientos enviados por

[1] La especificación Windows Sockets fue desarrollada por las empresas JSB Corporation, Microdyne Corporation, Sun Microsystems Inc. y Microsoft Corporation.

el window manager de X (recordemos que en Windows es posible esto ya que no existe la diferencia entre interfase externa e interna a la aplicación). Así, con este modo de operación, se tiene un desktop Windows con ventanas con dos tipos de 'frames', el de Windows y el implementado por el window manager del desktop X.

C. 5. Plataformas X utilizadas

Como una de las características importantes de hacer el 'mapping' del API de Windows sobre el sistema de ventanas X Window System, y no sobre un sistema de ventanas propietario, fue el hecho de poder interactuar con implementaciones del X Window System existentes en diferentes plataformas. Es por esto que fue necesario probar nuestro desarrollo interactuando con X servers ejecutandose sobre otros sistemas operativos distintos a DOS/Windows.

Los entornos gráficos probados fueron el de **SCO UNIX** y el de **LINUX 1.1.59.**

Bibliografía

1) Charles Petzold. *Programming Windows*. The Microsoft Guide to writing applications for Windows 3.0. Microsoft Press. 1990.

2) Matt Pietrek. *Windows Internals. The Implementation of the Windows Operating Environment*. Addison-Wesley. 1993.

3) Andrew Schulman, David Maxey y Matt Pietrek. *Undocumented Windows*. A Programmer's Guide to Reserved Microsoft Windows API functions. Addison-Wesley. 1992.

4) Al Williams. *Dos and Windows Protected Mode*. Addison-Wesley. 1993.

5) Robert W. Scheifler y James Gettys. *ACM Transaction on Graphics*, Vol. 5, No. 2, Abril de 1986. *The X Window System*.

6) Robert W. Scheifler y James Gettys. *X Window System. The Complete Reference to Xlib, X Protocol, ICCCM, XLFD. X Version 11, Release 5*. Digital Press. 1992.

7) Adrian Nye. *The Definitive Guides to the X Window System. Xlib Programming manual for Version 11*. Volume One. O' Reilly & Associates, Inc. 1990.

8) Adrian Nye y Tim O'Reilly. *X Toolkit Intrinsics Programming Manual for Version 11*. Volume Four. O' Reilly & Associates, Inc. 1994.

9) Douglas A. Young. *X Window System. Programing and Applications with Xt*. Prentice Hall, Inc. 1989.

10)Douglas E. Comer. *Internetworking with TCP/IP volume I. Principles, Protocols, and Architecture*. Second Edition. Prentice-Hall. 1991.

11)Douglas E. Comer, David L. Stevens. *Internetworking with TCP/IP volume III. Client-Server Programming and Applications - BSD Socket version*. Prentice-Hall. 1993.

12)FTP Software, Inc. *PC/TCP Development Kit for DOS & Windows. Socket Reference for Windows*. Version 2.2. 1992.

13)*The complete X Window System, Version 11, Release 5 distribution from MIT*. (Código fuente y documentación del X Window System).

14)The Santa Cruz Operation Inc. *SCO Open Desktop / SCO Open Server Graphical Environment Administrator's Guide*.