

Proceso de booteo en Linux

Linux es un kernel monolítico. Eso significa que todo corre en el anillo 1, el anillo privilegiado. Normalmente la CPU tiene varios anillos, como mínimo 2: el anillo donde corre el sistema operativo, que puede hacer todo y el segundo anillo, el modo protegido, donde corren las aplicaciones, los procesos de usuario. En el modo protegido hay muchas cosas prohibidas, como el acceso a E/S, es decir que ningún proceso de usuario interactúa con el hardware si no es a través del sistema operativo. Si el sistema operativo fuera microkernel, se busca que sea lo menos posible lo que corre en el anillo 1. Microkernel es una tecnología más moderna, lo cual no significa que sea mejor. El kernel monolítico sigue siendo más rápido porque comparte todo el mapa de memoria pero es un kernel inmenso, desde controladores, administradores, etc. Eso quiere decir que hay que ser muy cuidadoso en la programación. Los viejos kernel de Linux tenían un montón de programas vinculados estáticamente. Son módulos que tienen objetos (espacio de memoria reservado a variables y rutinas) vinculadas estáticamente en tiempo de compilación para generar un único binario. Cómo se instalaba un Linux monolítico hace 20 años atrás? Te traías el código fuente, corrías un utilitario o editabas un archivo con `defines` que decían que ibas a incluir y que no. Ahí decías mi máquina tiene una disquetera marca tal, controladora de tal tipo, una tarjeta de video tal. Entonces, recorrías esa larga lista e ibas habilitando las cosas que ibas a usar. Una vez que tenías eso, compilabas. La compilación estaba automatizada con la herramienta *make*. Una cosa importante: como estaba vinculado estáticamente no tenía acceso a librerías. El compilador vinculaba todos los objetos y armaba el binario. Al binario del kernel se le agregaba un pequeño código delante de compresión para pasar las limitaciones de memoria de los BIOS primitivos.

Qué hace el sistema cuando arranca (bootstrap)? Después de ser descomprimido en la memoria, lo primero que hace es salir del modo de 16 bits (compatible 8086, lanzado en 1978) y preparar la CPU para trabajar en modo multitarea, 32 bits o 64 bits, depende qué CPU sea. Una vez que prepara la CPU y el ambiente, levanta. El primer programa toma el anillo 1, empieza a correr en modo privilegiado. Por eso es fácil atacar una máquina en el booteo: si algo bootea antes que el kernel le puede robar el anillo 1 al kernel. Una vez que el kernel está listo, empieza a correr las rutinas de prueba de todos los controladores que tiene para ver si encuentra los dispositivos que le dijimos que tenía que tener. En la época dura de montaje “a pelo”, los buses no estaban tan evolucionados como ahora, los buses ahora tienen un sistema

de base de datos, un sistema de reportes, los dispositivos arrancan, dicen qué es lo que usan, el controlador organiza el bus de control, asigna interrupciones, DMA, puertos de E/S, direccionamiento de memoria. Todo eso no existía antes, al principio había que agarrar la tarjeta y, mediante jumpers, hacer toda esa configuración y más vale que no pusieras dos en el mismo puerto porque iba a haber problemas. En esa época las rutinas de prueba del sistema operativo eran bastante salvajes: intentaban conectar y veían si respondía algo o no. Cuando el sistema operativo terminaba de probar todos los dispositivos, iniciaba el planificador de tareas, luego monta el sistema raíz. UNIX tiene un árbol de directorios único y un sistema de archivos virtual que propone un sólo árbol de directorios. Las diferentes particiones con sus sistemas de archivos reales se van colgando en diferentes puntos de ese árbol de directorios. Entonces, el código durante el arranque, hace el montaje del sistema raíz. Una vez que montó el sistema raíz, dispara la primer tarea automáticamente, clásicamente el *init*. El *init* es un proceso guardián que no se puede morir. Si el *init* se muere todo el kernel se detiene. Entonces, el proceso *init* interpreta una serie de scripts para configurar el sistema. A partir de scripts se levanta todo el resto del sistema de archivos, se configuran las placas de red, y se levantan todos los servicios. En un UNIX clásico, lo que se levantaba además de los servicios residentes, los demonios que están sirviendo por los puertos de red, eran los manejadores o servidores de terminal. El clásico es el *getty* (abreviatura de "get tty") y sus variantes. Eso viene de la época en que los UNIX funcionaban en mainframes. Los mainframes no tenían consola, tenían terminales en la línea serial. Las terminales solamente tienen teclado, pantalla y un emisor serie, podía recibir caracteres en su codificación y los proyectaba por la pantalla, lo mismos con los caracteres por teclado. Si tenía 5 puertos serie, inicializaba 5 *getty*'s, cada uno se conectaba al respectivo puerto serie. Cuando detectaba que tenía una terminal conectada, ejecutaba un programa que se llama *login* que transmite hacia la terminal y que se queda esperando que el usuario ingrese su usuario y contraseña. Si había 3 intentos de login fallidos, se caía el login. Eso lo capturaba el *init* como proceso guardián. Estaban 10-15 segundos caídos y se volvían a levantar. Cuando lograbas loguearte exitosamente, el login clásico leía los datos de un archivo de texto que se llama *etc/passwd*, que contiene los atributos de cada usuario o cuenta, incluído la contraseña cifrada con un algoritmo sencillo de 40 bits y el id de usuario. En *etc/passwd* también dice cual es la *shell* o intérprete de comandos que vos usas, Entonces se levanta ese shell para que puedas trabajar.

Niveles de ejecución del *init*

Ahora, las distribuciones más nuevas reemplazaron al *init* por *systemd* pero les voy a contar cómo funcionaba el *init* porque *systemd* es parecido. La ventaja de *systemd* es que tiene más flexibilidad, pero es más complejo. *Systemd* permite un booteo en paralelo y por eso es más rápido.

Clásicamente, *init* es lo primero que se ejecuta en modo usuario y tiene varios niveles de ejecución. Los niveles de corrida son estados internos del *init*. Eran 7 niveles:

- El 0 es parada de la máquina (halt).
- El 1 es el nivel monousuario. Normalmente apaga todas las terminales y los servicios que conectan por red. Dejan la máquina como para que el administrador, desde la consola pueda hacer el mantenimiento que requiere que los usuarios salgan.
- Los niveles de 2 a 5 son configurables y cada distribución los usa como quiere.
- El 6 es reinicio de la máquina (reboot)

Mediante un comando le puedo cambiar el estado al init y eso sigue funcionando. Por ejemplo, para apagar la máquina se puede hacer:

```
init 0
0
telinit 0
```

Por compatibilidad, systemd acepta esas mismas órdenes. Un comando como *shutdown* o *reboot*, en definitiva invocan a init 0 y a init 6 respectivamente.

En init comienza ejecutando un script que se llama rcS y eso se hace una única vez durante el arranque. El init luego ejecuta el script de shell rc -rc significa run commands- y que permite controlar el arranque de la máquina. rc se vuelve a ejecutar cada vez que se cambia de estado o nivel de ejecución. rc toma como parámetros el estado del que venía y el estado al que se va, entonces, ya puede saber qué cambiar. Finalmente, init ejecuta una serie de procesos en un modo que se llama RESPAWN. Eso quiere decir, que si se llegan a caer, los vuelve a arrancar. init se queda para siempre esperando que alguien lo cambie de estado o si alguno de los procesos que se están ejecutando se cae, lo vuelve a levantar.

Al cambiar de estado, se vuelve a ejecutar rc, lo cual permite parar o iniciar procesos y además, los procesos en modo RESPAWN pueden cambiar en función del estado. Este sencillo principio de funcionamiento permite armar de manera inteligente, un sistema de arranque de servicios. Con lo cual se puede configurar qué servicios quiero arrancar y en qué estados. Para implementar eso se utiliza una serie de scripts que son los scripts de arranque BSD. El arranque BSD tiene un script rcS que ejecuta todos los scripts que se encuentren en el directorio etc/rc.S, con lo cual, cada paquete que necesite arrancar algo solamente durante el booteo de la máquina, lo único que tenía que hacer era poner un script en este directorio. Similarmente, el script rc ejecuta todos los scripts que se encuentran en el directorio etc/rc.N, donde N es el nivel de corrida. Así son 7 directorios.

Cualquier paquete puede arrancar o parar un servicio, lo único que tiene que hacer es instalar un script en el directorio etc/init.d. El script recibe el parámetro “start” o “stop” para arrancar o parar el servicio respectivamente. Hay un enlace simbólico (archivo que contiene el camino a otro archivo) en etc/rc.N que apunta al script en etc/init.d. Como varios enlaces simbólicos apuntan al mismo script, si lo cambio, cambia para todos a la vez. La estructura del enlace simbólico tiene la forma: una letra S o K en el primer lugar, seguido de un número entre 00 y 99, seguido del nombre del script. Por ejemplo etc/rc.3/S50apache apunta a etc/init.d/apache.

Eso quiere decir que en el nivel 3 arranque con prioridad 50 el script apache después de haber arrancado todos los que tienen números más chicos. Entonces, simplemente con pedir un directorio en orden sale solo por la forma en que está armado.

Si en algún lado pongo start, el script rc es lo suficientemente inteligente como para ejecutar stop de cualquier servicio que aparezca como arrancado en el estado anterior y como parado en el estado nuevo y similarmente ejecuta start para cualquier servicio parado en el estado anterior y arrancado en el estado nuevo. rc no hace nada con servicios que no aparece en alguno de los dos o en los dos del mismo modo (arrancado o apagado).

Las versiones más modernas de rc -antes de systemd- ejecutaban en paralelo los scripts con el mismo número de prioridad.

Incorporación de initrd

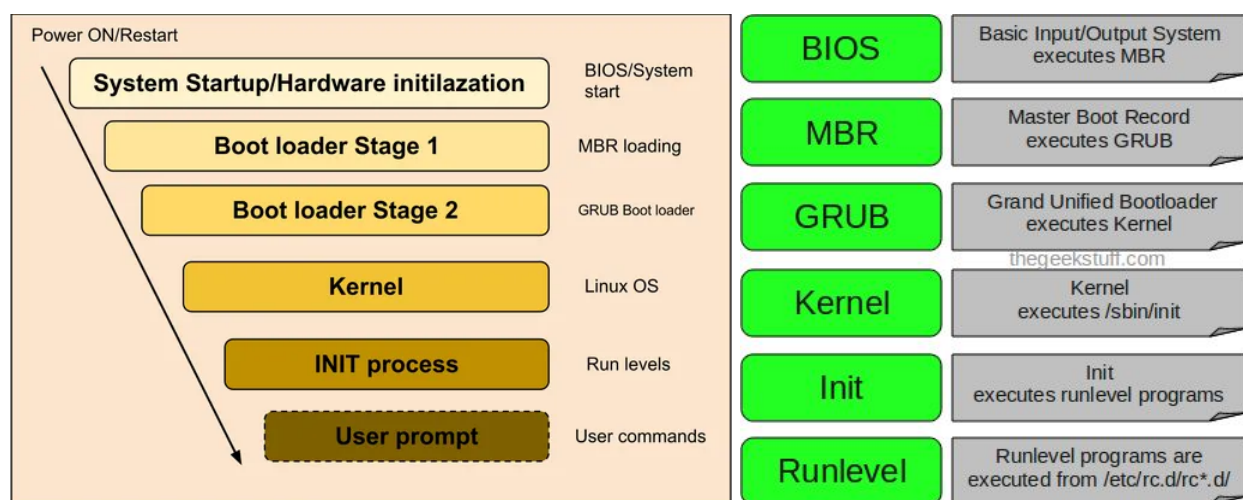
Qué se le fue incorporando al arranque con el tiempo? Desde el punto de vista del kernel, el problema más grande era compilar el kernel para cada máquina. Los instaladores de las distribuciones, cómo sabían en qué máquina iban a arrancar? Los instaladores venían con una lógica de incluir todo lo esperable que una máquina vaya a necesitar. Entonces eran kernels inmensos. Eso fue bien hasta que se llenó el BIOS. Luego, empezaron a comprimir. Cuando comprimidos también tenían problemas, apareció la posibilidad de vincular dinámicamente a partir del kernel versión 2 (1997-98). Es el concepto de módulo. La vinculación dinámica, que no es la misma que se usa para aplicaciones de usuario, es un mecanismo que permite cargar dinámicamente un objeto y vincularlo al espacio de memoria del kernel. Podía compilar el kernel con una cantidad mínima de cosas indispensables y el resto compilarlo como módulo que fuera a parar al sistema de archivos. Después, mediante un script de inicio podía decir qué módulos levantar. Para evitar las instalaciones “monstruo” y agilizar el proceso de instalación crearon un mecanismo que se llama *initrd* (initial ramdisk). Es un pequeño sistema de archivos de solo-lectura sencillo y fácil de acceder. Lo que necesita el kernel ahora, es el controlador del sistema de archivos temporal de *initrd* y nada más. Así queda pequeño de nuevo. Con eso se arma una imagen que también se comprime y se le pide al dispositivo de arranque que cargue dos archivos y no uno, el kernel y el *initrd*, en diferentes partes de la memoria. El kernel bootea, prepara el CPU, prueba los dispositivos, que son muchos menos porque ahora el kernel tiene los mínimos indispensables para funcionar. Ya no es un kernel grande, sí tiene el driver de ROMFS (ROM filesystem) para descomprimir en memoria el *initrd* y montarlo. En vez de montar el sistema raíz, monta *initrd*. El *initrd* tiene un script que se ejecuta automáticamente, que prueba y levanta los módulos. Una vez que están cargados los módulos y montado el sistema raíz se pivota este sistema de archivos pequeño por el sistema de archivos definitivo. Una vez que están los módulos para acceder al disco, al sistema de archivos, se levanta el sistema de archivos definitivo y continúa el proceso de arranque como lo habíamos explicado antes.

No es imprescindible un instalador para armar un Linux. Si tenés un sistema de archivos básico, le pones un kernel y le instalas un gestor de arranque, podés hacer que una máquina bootee sin instalador. Con Debian es relativamente fácil de hacer.

Booteo en PC

En el esquema clásico de arranque de una PC, los procesadores de Intel arrancan binarios de 16 bits y la primera rutina de arranque es la BIOS, la cual está almacenada en una memoria de solo-lectura en el motherboard. BIOS testea el procesador y la memoria principal. Tiene incorporados algunos controladores sencillos implementados en 16 bits para los dispositivos más importantes: disco rígido, tarjeta de red, USB, unidad de CD, por si se quiere bootear por alguno de ellos. Al ser en 16 bits tiene cantidad de limitaciones hardcodeadas en cuanto a tamaños de acceso. El principio de funcionamiento es simple: a partir de la configuración del BIOS (Apretando la tecla DEL) los usuarios dicen en qué orden quieren que arranquen los dispositivos. Si eligen el disco rígido, BIOS accede al sector cero del disco, lo carga en memoria, y le pasa el control (jump). Este esquema, que es sencillo, trajo problemas. Si se rompe el sector cero de un disco, hay que tirar el disco.

En el arranque se habla de cadena de arranque porque vas cargando una cosa más pequeña que le pasa el control a otra, y así hasta que llegas a un kernel. Se puede arrancar un Linux si el kernel se copia directamente, sin sistema de archivos, bloque a bloque, sobre un dispositivo, y arranca, pero es una solución muy primitiva y muy rígida porque no puedo pasarle parámetros, particularmente el sistema de archivos raíz. Se necesita un programa de arranque.



Para más detalles, las figuras son de [Debugging problems in the boot process](#) y [6 Stages of Linux Boot Process \(Startup Sequence\)](#)

Los gestores de arranque son programas en binario de 16 bits que administran el arranque. Por ejemplo, se le puede decir cuál es el sistema raíz. Entre los clásicos está el LILO (Linux

Loader). Después apareció el GNU Grub (Grand Unified Bootloader), que es más completo y está pensado para que ande en todas las máquinas, soporta muchas arquitecturas. Cuando se configura GNU Grub para un booteo clásico en PC compila e instala un código pequeño en el sector cero, en el Master Boot Record del disco, e inmediatamente atrás hay un salto porque todos los discos tienen un pequeño espacio libre después del sector cero. La primera partición arranca 64 bloques más adelante. Ese espacio libre, clásicamente fue aprovechado por virus, malware, o para esconder algo. Grub aprovecha ese espacio para su primer nivel de arranque (Stage 1) donde tiene un controlador para el sistema de archivos. Se puede abrir el sistema de archivos y buscar el segundo nivel que ya es más largo. Grub está partido en dos niveles. Su segundo bloque (Stage 2) es el que muestra el menú que te permite elegir qué arrancar. Cuando elegís qué arrancar, automáticamente Grub va a buscar el kernel que elegiste, lo sube a la memoria, sube el initrd, y le pasa el control al kernel y así continúa el booteo. Si ustedes tenían un Windows en la máquina podían hacer un chainload.

Así funcionaba el esquema clásico. Después apareció UEFI (Unified Extensible Firmware Interface). UEFI reemplaza la antigua interfaz del BIOS después de 30 años. Dos cosas vinieron con el UEFI. La primera es un nuevo esquema de particionado distinto del original de IBM. El esquema original de IBM (MBR, Master Boot Record), introducido en 1983 aceptaba cuatro particiones. Después se le hizo un parche para que pudiera aceptar más. Por otra parte GPT² (GUID Partition Table) tiene una zona de metadatos, y tiene dos copias, una al principio y otra al final del disco, con lo cual es un poco más seguro. Si cambias el tamaño de un disco virtual, o copias bloque a bloque de un disco chico a uno grande, se da cuenta que cambió el tamaño porque no encuentra su superbloque en el fondo. En la tabla de particiones, ya no hay código binario ejecutable ni espacios libres. Si uso GPT y quiero hacer un arranque clásico de IBM, lo puedo hacer porque en el sector cero de GPT hay un arranque de compatibilidad. Hay que declarar una partición donde el Grub pueda guardar su nivel 1. En MBR eso no hacía falta, el GPT es obligatorio porque si no, no se ve.

El salto más importante de UEFI, es que el sistema arranca en 32 o 64 bits. El manejador de arranque se instala en un sólo código y está guardado en un sistema FAT32, una partición pequeña de unos 100 MB. En esa partición se guardan todos los gestores de arranque. UEFI nativamente soporta que tengas varios guardados. Las implementaciones más completas de UEFI te permiten navegar la partición para buscar los gestores y ordenarlos. Las implementaciones más básicas no permiten eso y van por default. Otra cosa que incorpora UEFI es una supuesta característica de seguridad que permite firmar digitalmente el manejador de arranque para que no bootee nada que no esté firmado. Es una manera de parar rápidamente virus pero obtener la firma digital del gestor de arranque era bastante difícil. Ahora ya se puede usar más fácil. Esa característica, Secure Boot, siempre se pudo deshabilitar. Grub tiene soporte para UEFI y es más robusto y más rápido -cuando está bien configurado. UEFI es una solución más moderna y es lo que se recomienda usar hoy. Es importante entender que pueden bootear UEFI con una máquina antigua con particiones MBR o en una más moderna con particiones GPT.

Recordemos que BIOS (en este contexto, también llamado Legacy) era la primera rutina de arranque y evolucionó hacia el UEFI. Por otro lado tenemos los gestores de arranque: LILO evolucionó hacia GRUB. E independientemente, tenemos los esquemas de particionado: MBR que evolucionó hacia GPT. El sistema de arranque (BIOS/UEFI) es independiente del sistema de particionado (MBR/GPT) y las 4 combinaciones son posibles en una PC.

Fallas en el booteo

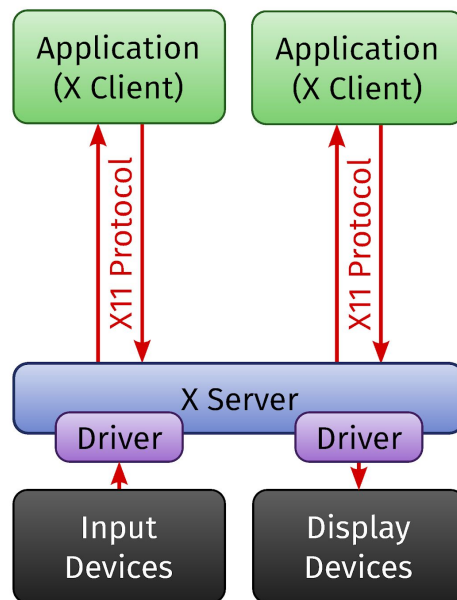
Si una máquina con Linux no arranca, los puntos de falla más probables son, que haya un conflicto con el hardware como querer bootear un kernel de 64 bits en una CPU de 32 bits. Si pasó ese punto, el primer punto de falla importante es que no pueda encontrar el sistema de archivos raíz (*kernel panic*), por diferentes razones, por ejemplo si le indicamos una partición que no era por error, o nos falta un controlador y no pudo encontrar el sistema de archivos. Otro punto es en el sector de arranque, que no haya podido encontrar la imagen del kernel o el initrd. Otro es en el initrd porque no encontró los módulos que tenía que encontrar para continuar el booteo. Los kernel modernos, si se caen ahí, te dejan con un shell integrado que se llama busybox^{*1} y te permite a veces continuar el proceso de booteo manualmente. Finalmente, otro punto de falla es que levantó un sistema de archivos en una partición equivocada y ese no era el sistema raíz, con lo cual no encontró el binario para arrancar.

El Sistema de Ventanas X^{*3}

Si pensamos en la evolución de Windows, por ejemplo, primero fue DOS, que es un sistema operativo antiguo, monousuario. Cuando apareció el procesador 386 Windows empezó a desarrollar, parcialmente, el soporte para multitarea real con Windows 3.1. Booteaba sobre DOS y arrancaba el Windows aparte y como estaban obligados a mantenerse compatibles, estaba la emulación de DOS arriba y cualquiera le podía robar el control: te colgabas de una interrupción de DOS y le robabas el modo kernel a Windows fácilmente. Por eso proliferaron tanto los virus. Nunca fue un criterio de diseño importante la seguridad en ese sistema. Vino Windows 98, y luego Windows 2000 no fue continuación del 98, sino de Windows NT. NT era un sistema operativo de verdad, levantaba en modo 32 bits desde el origen y tenía un sistema de archivos con protección de escritura. Pero los programadores de DOS podían abrir archivos en modo lectura, escritura, o lecto-escritura con lo cual no andaba nada de DOS en Windows 2000, y no estaban los códigos fuente con lo cual no se puede recompilar: un desastre. Entonces, Microsoft deshabilitó las protecciones y publicaron el Windows 2000 que no es un mal sistema pero las políticas de Microsoft son terribles y deshabilitaron toda la seguridad para que ande. Por eso fue, y va a seguir siendo, peligroso. No por la calidad técnica que tenga, sino por las políticas que toman a la hora de comercializarlo. Salió Windows Milenium a partir del 98 pero no duró.

En los UNIX, la aparición de terminales con capacidades gráficas tuvo otra evolución. En Windows, si lo miramos, se centraron en la parte gráfica. El mismo nombre te lo dice. Si la placa de video falla, todo el Windows se cae. Eso afloja un poco con la inclusión de la tecnología microkernel de NT y sus sucesores. UNIX sigue manteniendo hasta el día de hoy la política de login a través de terminales a pesar de que las máquinas hace años que no tienen terminales. Cómo se corre hoy si no hay terminales? Hay un controlador de software que es un emulador de terminales. El emulador de terminal utiliza la consola, o sea el teclado y la pantalla de la máquina, como terminal. La placa de video, clásicamente estaba en modo VGA, en modo texto. Finalmente, este controlador se conecta con el `getty`. El emulador de terminal es configurable: tiene muchas terminales. Por eso ustedes, con el teclado^{*4}, o con el comando `chvt` (necesita privilegios de root) pueden cambiar de terminal (TTY1, TTY2, etc.). En resumen, el login de texto se hace contra un emulador de terminal. Qué pasa con la terminal gráfica? Se enfocó con el mismo criterio. La terminal gráfica también va a estar emulada. Entonces, se armó un servidor, no se hizo monolítico, al cual se le dio acceso a la placa de video para poder levantarla y ponerla en modo gráfico. Funcionaba dentro del emulador de terminal. Una terminal que podía tener placa de video. Con el tiempo apareció una estandarización que se llama frame buffer y permite manejar la placa directamente en estado gráfico a alto nivel. Se dejó atrás el modo VGA. El login de texto se hace a través de un frame buffer (por eso las letras se ven pequeñas). O podemos instalar un servidor que permite manejar la placa gráfica hasta sus niveles más elevados, que es el servidor X.

El kernel le da un salvoconducto al servidor X para manejar la placa de video. El servidor X levanta la placa de video hasta el modo gráfico que pueda o hasta que ustedes le digan. El servidor X cierra un lazo local en el que maneja el teclado y el mouse, y emula una terminal gráfica. Ustedes mueven el mouse y la flechita la dibuja el servidor X y a la pantalla de fondo la dibuja el servidor X. Por defecto, X no dibuja nada, una pantalla "en blanco". El servidor X se conecta a sockets. Acepta sockets de UNIX clásicos y acepta sockets TCP, con lo cual uno se puede conectar a través de la red. X se queda esperando que algún cliente se conecte con él. Cualquier programa puede estar programado con la librería de X. La librería de X es el cliente. Para qué sirve el protocolo X? Este protocolo vectorizado es muy eficiente para presentar gráficos; le permite al cliente renderizar rápidamente los vectores en el servidor, y así dibujar las ventanas, marcos, dibujos, letras. Uno escribe un programa en C y, utilizando las primitivas de esta librería de X, uno puede conectarse a través del socket de UNIX o a través de la red con el servidor X y dibujar una ventana. Las librerías de X se caracterizan por la simplicidad; es muy duro el manejo de recursos en el X clásico. Hay un cliente X privilegiado, que se llama manejador de ventanas o Windows Manager. Es un cliente al que X le delega la construcción de los marcos de las ventanas, esto es el borde, título, los botones para manejar las ventanas. Como X propaga eventos, porque está orientado a eventos, si se mueven por la zona del marco, se activa el manejador de ventanas y atiende él. Si se mueven por el interior de la ventana, al evento lo atiende la aplicación que tenga esa zona de la ventana. Si se mueven por afuera, atiende el que esté conectado a la ventana raíz o fondo de pantalla. Aún no existía el concepto de escritorio, lo máximo que había era un manejador de ventanas.



Compartida desde Wikipedia, con licencia CC-BY-2.0.

Es muy interesante entender que, mucho antes que hubiera VNC, o Terminal Server, o Remote Desktop, esto ya estaba funcionando en la década de 1990 y, con mucha facilidad permitía que un cliente de X se conectara a un servidor X a través de la red. Hoy parece muy fácil, pero Remote Desktop, VNC, y otros, transmiten mapas de bits, videos de las pantallas, renderizan localmente y transmiten el video. X es vectorizado, el cliente manda los vectores por la red, y X dibuja la pantalla. Puede ser que ejecute un cliente por ssh en una máquina remota, y le pida a ese cliente que mande la ventana a la máquina donde estoy yo. En otras palabras, el servidor X normalmente corre en una computadora con una persona enfrente, mientras que las aplicaciones clientes X corren en esa o en cualquier máquina en la red y se comunican con la computadora del usuario para renderizar el contenido gráfico y recibir los eventos de los dispositivos de entrada, o sea teclado y mouse. El término “server” para software en la máquina local puede confundir, pero hay que pensar que un server administra un recurso. En este caso, la terminal gráfica es el recurso que queda disponible para programas cliente locales y remotos que necesitan compartir la pantalla y dispositivos de entrada del usuario por medio del servidor X.

El protocolo X11^{*5} no es un protocolo fuerte como para conectarse en Internet, además, los vectores son pesaditos y si tengo que dibujar una ventana a través de una conexión lenta, me va a costar. Por eso VNC, Remote Desktop andan bastante mejor a través de Internet porque le transmiten un video de la pantalla. Es más rápido aunque es de menos calidad. En X la calidad es tal que no detectas la diferencia entre una ventana que está ejecutando en la máquina local de una ventana que está ejecutando una aplicación en otra máquina. Si están en una red local tampoco se nota la velocidad (latencia). Un protocolo como ssh que permite hacer

sockets de túnel, permite tunelear fácilmente X, con lo cual yo me puedo conectar por ssh a través de Internet, con compresión, y ejecutar una aplicación remota X. Si configuré el túnel, el ssh tunelea el protocolo X y yo puedo ver la ventana localmente. Los vectores no necesariamente son más eficientes. En un video, si tenés una conexión mala, le bajas la calidad. Con X, si enlace es malo, vos movés el mouse pero el refresco de la ventana demora. Secure Shell con túnel y compresión, ayuda. X es un protocolo viejo que no está pensado para ser transmitido a través de Internet (considerando seguridad y latencia). Es importante entender que X te da el concepto de terminal (terminal server), VNC y Remote Desktop duplican una pantalla que ya está dibujada en otro lado.

Esta conexión, se puede generalizar a través de un login gráfico. Me había logueado usando ssh, que en definitiva se conecta con el login original de la máquina remota y así obtuve los privilegios. Alternativamente, le puedo decir al servidor X que se conecte con un login gráfico. En la máquina remota hay un servidor equivalente al getty, pero gráfico, que se llama XDM y sus sucesores GDM, KDM, MDM, LightDM, etc. XDM utiliza un protocolo que se llama XDMCP y es viejo ya. Cuando arrancas el servidor X, le podés decir que actúe en este caso como cliente por protocolo XDMCP. XDM funciona como el getty. Una vez que recibe una conexión del servidor X y la acepta, automáticamente genera una conexión para el otro lado y le manda los clientes de sesión. O sea, permite iniciar una sesión. Por ejemplo, te levanta el login gráfico, maneja automáticamente la seguridad. En la máquina local ves un login gráfico. Si te acepta, XDM inicia la sesión y levanta todos los clientes de X en la máquina remota y X te dibuja la pantalla localmente. En resumen, los componentes más importantes que tenemos son:

- Servidor X: es un servidor de pantalla, recibe de los comandos, vectores, de aplicaciones cliente a través de un socket para dibujar ventanas.
- Servidor XDM: es equivalente a getty. Es un manejador de terminales gráficas. Una terminal gráfica se inicia y se puede pedir que se conecte como cliente a un servidor XDM.

Este concepto evolucionó a lo que hoy son los ambientes de escritorio. Básicamente son muchas aplicaciones X, todas con una interfaz integrada, con un framework de desarrollo común y con una aplicación que maneja el fondo de la pantalla y es el escritorio. Algunos escritorios usan características de aceleración de video, pero eso consume recursos que normalmente están destinados a las aplicaciones. Además, he visto fallar a más de una aplicación por no poder “compartir” correctamente los recursos de la aceleración gráfica con el manejador de ventanas.

Notas sobre X

Cuando se creó X en 1984, Bob Scheifler and Jim Gettys establecieron una serie de principios de diseño. Uno de los más contundentes y que puede tener su gracia es *"The only thing worse than generalizing from one example is generalizing from no examples at all."*

X solo define primitivas gráficas y deliberadamente no define un diseño de GUI. Por eso, hay varios ambientes de escritorio populares (GNOME 2, KDE, Xfce, etc.). Los ambientes de escritorio, aparte del manejador de ventanas, viene con un conjunto de aplicaciones que tienen una GUI consistente. Por otro lado, esta independencia entre X y las GUI generó problemas de interoperabilidad.

A partir de la implementación original de X, aparecieron una cantidad de variantes, incluso algunas propietarias. X.org Server (a partir de 2005) es la implementación que usan las distribuciones de Linux, la desarrolla la X.org Foundation y tiene el dominio X.org.

Hay un tutorial sobre X de Jasper St. Pierre y otros en <https://magcius.github.io/xplain/article/>.

A veces, por contexto se sobreentiende, pero los componentes diferentes de los que mencionamos son: el Sistema de Ventanas X, el protocolo X11, el servidor X, y los clientes X.

Referencias

^{*1} Busybox viene en un solo ejecutable, y corre en Linux, Android, entre otros. Se creó para sistemas embebidos con muy pocos recursos pero incluye las funciones de más de 300 comandos comunes. Más información en www.busybox.net

^{*2} GPT no tiene límite, más allá que los que establezcan los propios sistemas operativos, tanto en tamaño como en número de particiones. Por ejemplo, Windows tiene un límite de 128 particiones.

^{*3} X también se conoce como X11 porque ha estado en la versión 11 desde 1987. Lo desarrolla la X.org Foundation y se distribuye con licencia MIT.

^{*4} Para cambiar de terminal se usa la combinación de teclas Ctrl + Alt + FunKey(1-6).

^{*5} Las aplicaciones no usan el protocolo X11 directamente, sino a través de librerías. Igualmente la documentación completa del protocolo X11 está en:

Robert W. Scheifler, "X Window System Protocol" 2004.

(<https://www.x.org/releases/current/doc/xproto/x11protocol.html>)