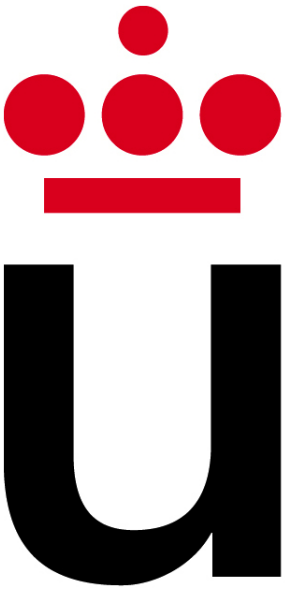
Práctica Obligatoria

Diseño y Arquitectura del Software

GII+GIS: *Jesús Alcalde Alcázar*
Germán Alonso Azcutia
Adrián Gutiérrez Jiménez
GIS+MAT: *José Ignacio Escribano Pablos*
GIS: *Carlos Vázquez Sánchez*



Móstoles, 7 de enero de 2014

Índice general

Prefacio

Durante la realización esta práctica, se han seguido una serie de pautas o condiciones; por ello nos gustaría relatar brevemente cuáles han sido dichas pautas y cómo han afectado al desarrollo de la práctica.

Objetivos y motivaciones

En ocasiones, para los distintos y diversos ámbitos que recoge la informática, puede ser necesario un cambio de sistema operativo con el que trabajar. Esto puede estar ocasionado por diversos motivos: soportar cierto tipo de características, cuestiones de seguridad, mayor rendimiento o simplemente por cuestión de preferencias; este tipo de cambios implican, generalmente, grandes costes y si hablamos de entidades públicas como pudiera ser una universidad, realizar un cambio en de sistema operativo en todos sus ordenadores supondría un gran cantidad de costes.

Por ello, sería interesante buscar y conocer una arquitectura de software que permita a un computador escoger entre más de un sistema operativo que trabaje a un nivel óptimo. Esta arquitectura se conoce como *arquitectura de virtualización*.

En otro ámbito, pero centrándonos más en las llamadas arquitecturas software; se suele hablar de éstas arquitecturas en cuanto al diseño de productos software se refiere, en asignaturas destinadas a tal efecto; pero, *¿Cómo implemento una aplicación con una cierta arquitectura?*, *¿Es posible implementar una arquitectura de una forma genérica?* y *¿Se podría crear una interfaz con una arquitectura software y utilizarla en varios productos software?*.

Por tanto, los objetivos de esta práctica pasan por satisfacer estas dudas e inquietudes, es decir:

- Definir la arquitectura de virtualización y describir un caso en concreto.
- Describir la arquitectura y diseño de una plataforma que implemente una arquitectura que pueda ser utilizada por otros programas.

Estructura de la memoria

En cuanto a la distribución de las hojas en este documento, debido a que en la práctica se encuentran dos partes claramente diferenciadas, la memoria ha sido dividida en dos secciones:

- **Parte de observación:** En la que tras una pequeña introducción se detallará la arquitectura del sistema de virtualización de la universidad.
- **Parte de diseño:** Correspondiente con el diseño de la plataforma, se empezará describiendo qué es una arquitectura de pizarra, qué componentes la forman, etc para posteriormente diseñar una plataforma que implemente una arquitectura de este tipo, finalmente,se dará un manual con las operaciones básicas de nuestra aplicación.

Es necesario destacar, que estas partes han sido estrictamente obtenidas del enunciado de la práctica.

Metodología de trabajo

Para la realización de esta práctica, uno de los pilares fundamentales ha sido la manera en la que nos hemos organizado.

Básicamente y en gran parte debido a las fechas en las que había que realizar la práctica, decidimos probar algo nuevo para nosotros pero que ya conocíamos de oídas: realizar la práctica de una manera distribuida. Por ello uno de los primeros pasos fue buscar alguna herramienta que nos permitiera este tipo de trabajo, entre las opciones se encontraban: dropbox, google drive, google code o GitHub.

La elección dependía de cómo fuéramos a afrontar la práctica, es decir, si decidíamos que cada uno se encargará de una cosa, seguramente *Dropbox* hubiera sido la mejor opción, pero al ser gran cantidad de contenido decidimos que lo mejor sería ir mirando poco a poco que era necesario hacer.

Por este motivo también, un simple archivo **.doc* no nos valía, ya que *¿Qué sucede si escribimos los dos a la vez?*.

Por todos ello el resultado de una tarde de organización fue realizar la memoria de la práctica en lenguaje LaTeX en un repositorio en GitHub[13].

¿Por qué ?

Esta plataforma junto con el control de versiones git[11] nos ha permitido trabajar sobre un repositorio al que todos los miembros del grupo teníamos acceso y mediante el cuál podíamos compartir nuestras dudas y sugerencias y trabajar cada uno a su aire, sin estar limitado a una repartición de trabajo; si nos atascábamos en una cosa, podíamos continuar con otra sin problemas porque todos teníamos el control sobre el estado actual de la práctica.

Hemos elegido usar GitHub por los siguientes motivos:

- Es un conocido programa para desarrolladores de software, por lo uqe tiene una extensa documentación que nos podía solucionar casi cualquier tipo de dudas.
- Al ser un alto número los integrantes del grupo, nos ha sido más fácil y sencillo trabajar mediante repositorios.

- Pero principalmente debido a que dicho programa sigue una arquitectura de pizarra lo que nos acercaba a un entorno práctico de aspectos a tratar en la práctica: la arquitectura de pizarra.

Todos los archivos de esta práctica se encontrarán disponibles en la dirección `https://github.com/KekoAlk/Practica_DAS`, correspondiente con el repositorio de trabajo.

¿Por qué LaTeX?

LaTeX es un sistema de composición de textos profesional, es la herramienta más potente de creación de documentos que conocemos, que permite separar de forma fácil el contenido de un único archivo en varios, ya que en el fondo se trata de un lenguaje de programación, de forma que trabajar de forma distribuida sea bastante cómoda.

Además, los archivos son texto plano por lo que, junto a GitHub, hace que se pueda ver qué se ha añadido o borrado en un documento de forma sencilla. También es un sistema que separa el contenido de la presentación, por lo que facilita centrarse en el contenido y no tanto en su presentación.

Sin duda, haber aclarado este tipo de cosas antes del comienzo de la práctica nos ha permitido centrarnos mucho más en el trabajo y dejar a un lado detalles como pudiera ser los aspectos visuales de la memoria.

Parte I

Observación

Capítulo 1

Introducción

En esta primera parte de la memoria nos vamos a encargar de hacer una descripción a nivel arquitectónico de los sistemas de virtualización de las aulas de la Universidad Rey Juan Carlos.

Para llevar a cabo esta tarea hemos decidido que la mejor opción pasa por describir o analizar qué es un sistema de virtualización, para después poder centrarnos en el caso concreto de las aulas de la URJC, que se encuentran virtualizadas con VMware[9] lo que significa que en el caso concreto nos centraremos en VMware, eso sí, haciendo continuas analogías a lo que se ve en las aulas.

Pero para entrar un poco en materia vamos a intentar explicar brevemente qué es esto de la *virtualización*.

1.1. ¿Qué es la virtualización?

Con la intención de hacernos una idea precisa del término *virtualización* hemos buscado diversas definiciones, de las cuales nos vamos a quedar con:

> *«Virtualización es la creación -a través de software- de una versión virtual de algún recurso tecnológico, como puede ser una plataforma de hardware, un sistema operativo, un dispositivo de almacenamiento u otros recursos de red.»*
>
> –Wikipedia[17]

Es otras palabras, se trata de la creación mediante software de elementos hardware virtuales. Un buen ejemplo de esto es cuando particionamos un disco duro; físicamente tenemos un único *HDD* pero a nivel de software existen dos, pues el disco duro esta *virtualizado* en dos particiones, el sistema operativo se encarga de abstraer las propiedades físicas del disco.

Dando una vuelta de tuerca más, podemos definir *virtualización* como el proceso por el cual una capa de software *Virtual Machine Monitor (VMM)* abstrae los recursos de la computadora física al Sistema Operativo o *Máquina virtual*.

Cabe destacar el uso de los términos *Máquina virtual* y VMM, pues son el eje central de los sistemas de virtualización y conviene introducirlos. Y es por esto que nuestro siguiente apartado está dedicado a ellas.

1.2. ¿Qué es una máquina virtual?

Al igual que antes, encontramos diversas acepciones de *Máquina virtual*, pero vamos a partir de la definición de *Wikipedia* pues es la que consideramos más completa.

> *«Una máquina virtual es un software que simula a una computadora y puede ejecutar programas como si fuese una computadora real. Este software en un principio fue definido como "un duplicado eficiente y aislado de una máquina física".»*
>
> –Wikipedia[18]

O lo que es lo mismo, un programa o aplicación software que simula de manera total o parcial una computadora de forma que el usuario final no note la diferencia.

Podemos encontrar dos tipos de éstas dependiendo de cómo se ejecuten en el sistema:

- **Máquinas virtuales de proceso** o Máquina virtual de aplicación, se ejecuta como un proceso normal dentro de un sistema operativo y soporta un solo proceso. Su objetivo es el de proporcionar un entorno de ejecución independiente del sistema operativo y del hardware. Una Máquina virtual de proceso muy popular es la de Java o *Java Virtual Machine (JVM)*.
- **Máquinas virtuales de sistema** o máquinas virtuales de hardware, que permiten a la máquina física multiplicarse entre varias máquinas virtuales, cada una con su propio sistema operativo. A la capa de software que se permite la virtualización se le llama *monitor de máquina virtual* o VMM, anteriormente mencionado.

Como es obvio nosotros nos vamos a centrar en el último tipo, puesto que vamos a analizar la virtualización en las computadoras de la universidad, y éstas se ejecutan como un sistema operativo propio y no como una aplicación de éste.

1.3. Condiciones para la virtualización

A la hora de crear un sistema virtualizado existen propiedades que hacen que se pueda evaluar, es decir *¿Cuán bueno es el sistema?*, estas propiedades se refieren al el monitor de la máquina virtual o *VMM*. Listémoslas pues:

- **Equivalencia o fidelidad:** Un programa corriendo bajo el VMM debe exhibir un comportamiento esencialmente idéntico a aquel demostrado cuando se ejecuta directamente en una máquina equivalente.

- **Control de recursos y seguridad:** El VMM debe estar en completo control de los recursos virtualizados.
- **Eficiencia:** La mayor parte de las instrucciones de máquina debe ser ejecutada sin la intervención del VMM.

Para llevar a cabo una virtualización del sistema que cumpla estas propiedades, Popek y Goldberg escribieron en un artículo en 1974 en la revista *Communications of the ACM*[15], qué condiciones se han de dar para una virtualización eficiente, en dicho artículo realizan un análisis del repertorio de instrucciones del computador o *Instruction Set Architecture (ISA).*

Dividiendo las instrucciones en:

- **Instrucciones privilegiadas:** Las instrucciones que sólo funcionan en modo kernel del procesador y no lo hacen en modo usuario.
- **Instrucciones sensibles de control:** Las instrucciones que al ejecutarse cambian la configuración del sistema de alguna manera.
- **Instrucciones sensibles de comportamiento:** Aquellas que dependen del estado de los recursos del sistema en el momento que se ejecutan.

Y como resultado de este análisis formularon los siguientes teoremas.

Teorema 1. *Para cualquier computadora convencional de tercera generación, se puede construir un VMM efectivo si el conjunto de instrucciones sensibles es un subconjunto de las instrucciones privilegiadas.*

Este teorema implica que para construir un VMM basta con que todas las instrucciones que pueden afectar a su correcto funcionamiento, las instrucciones sensibles, pasen siempre el control al VMM. Esto garantiza la propiedad de control de recursos y seguridad. En cambio, las instrucciones no privilegiadas se deberán ejecutarse de manera nativa, para lograr eficiencia.

Teorema 2. *Una máquina convencional de tercera generación es recursivamente virtualizable si es virtualizable y se puede construir para ella un VMM sin ninguna dependencia de sincronización.*

Este teorema es fundamental, en lo que a nosotros se refiere, pues este último teorema no se cumple para arquitecturas de computadores x86, que casualmente son las que utilizan los computadores de la universidad.

Ahora centrémonos en la arquitectura de los sistemas de virtualización.

Capítulo 2

Descripción de la arquitectura de virtualización

Una vez introducido el vocabulario básico y tras haber indagado un poco más en la materia vamos a centrarnos en la arquitectura como tal.

2.1. Tipos de Virtualización

En el apartado anterior ya se pudo vislumbrar que existen diferentes formas de virtualización, y por ello, vamos a hacer un pequeño análisis de cada uno, pero antes nos interesa conocer el término *Hypervisor*, ya que es el elemento central de un sistema de máquinas virtuales.

2.1.1. ¿Qué es un hipervisor?

El *Hypervisor* o *Virtual Machine Monitor (VMM)* se trata de una plataforma que permite aplicar diversas técnicas de control para utilizar, al mismo tiempo, diferentes sistemas operativos en una misma computadora.

Se trata de un elemento software que dependiendo de cómo se sitúe en relación con el Hardware da lugar a dos maneras diferentes de virtualizar, dos tipos de *Hypervisor*[16]:

2.1.2. Hipervisor de Tipo 1 o *Nativo*

El software del hipervisor se ubica directamente entre el hardware y las distintas máquinas virtuales, para ofrecer la funcionalidad descrita, siguiendo la siguiente estructuración:

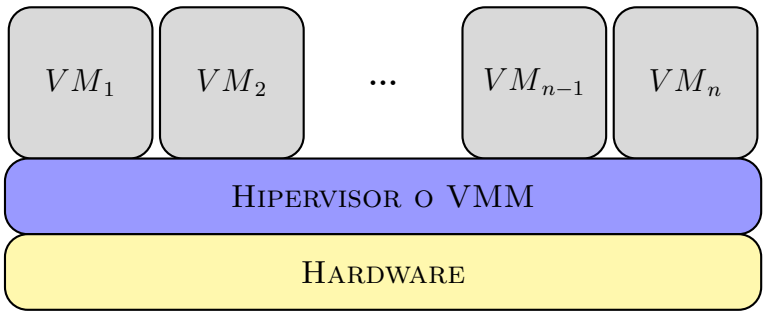


Figura 1: Esquema de un hipervisor de primer nivel

Este tipo de *hipervisor* también es conocido como *unhosted* o *bare metal*, que en inglés hacen referencia a que no es huésped o que se ejecuta a bajo nivel, respectivamente.

Dentro de este tipo se encuentran VMware ESXi, VMware ESX y Microsoft Hyper-V Server, pero nos gustaría prestar una atención especial a Xen por ser un hipervisor de código abierto desarrollado por la Universidad de Cambridge[12][6].

2.1.3. Hipervisor de Tipo 2 o *Huésped*

Es una arquitectura alternativa para la máquina virtual insertando una capa de virtualización encima del sistema operativo *host* o huésped, siendo éste responsable de administrar el hardware. Los sistemas operativos invitados se instalarán encima del nivel de virtualización, en máquinas virtuales. Tiene la siguiente estructura:

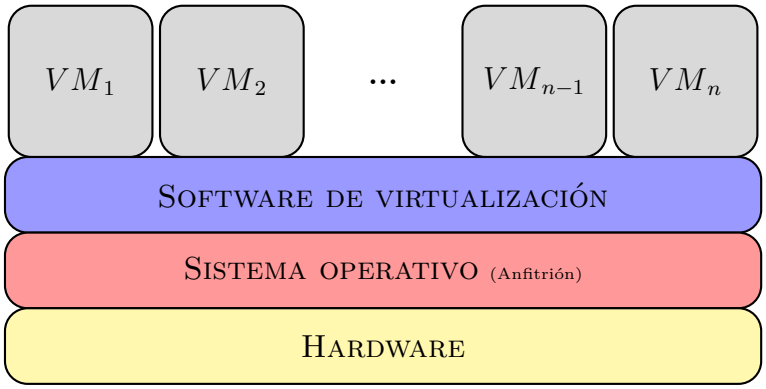


Figura 2: Esquema de un hipervisor de segundo nivel

Este tipo de hipervisor tiene una ventaja muy destacada, el usuario puede instalar esta arquitectura de máquina virtual sin modificar el sistema operativo host, pudiendo descansar en el sistema operativo host para entregar los controladores de dispositivos y otros servicios de bajo nivel (se simplifica el diseño de la máquina virtual y facilita la implementación).

Otra característica suya es que al ejecutarse sobre un sistema operativo huésped, éste puede ejecutar sus propias aplicaciones, por lo que en realidad el esquema completo sería:

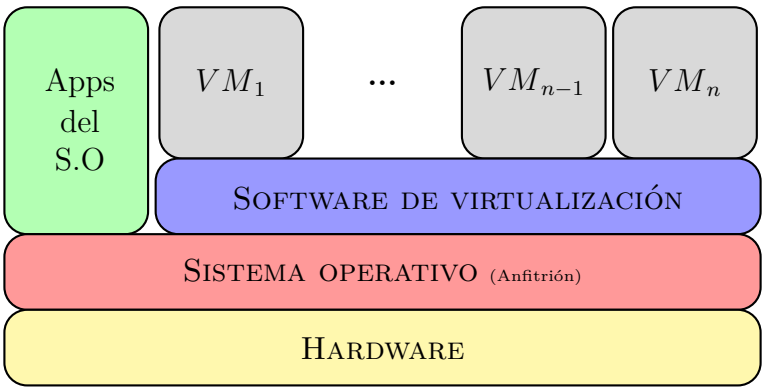


Figura 3: Esquema de un hipervisor de segundo nivel real

Algunos de los hipervisores tipo 2 más utilizados son: Oracle: VirtualBox, VirtualBox OSE, VMware: Workstation; siendo éste último en el que más nos vamos a centrar.

2.2. Componentes

En las figuras mostradas anteriormente, se han hecho referencias a *hardware*, *Hypervisor* o *Máquina virtual (VM)* en las que no explicamos exactamente que significa y por ello, ahora, vamos a analizar cada uno de estas partes descomponiendo sus componentes; es decir, qué elementos lo forman, para después poder juntarlos todos y obtener una idea real de la arquitectura.

Siguiendo un orden de menor a mayor nivel, comenzaremos con el *hardware* para terminar con las distintas *VMs*.

2.2.1. Hardware o Hardware real

Quizás lo primero que tengamos que hacer es explicar el encabezado del apartado, ¿Por qué *"hardware real"*?.
Para contestar a esto es necesario conocer los componentes de una máquina virtual (*Véase apartado 2.2.4*), pues ésta tiene también elementos hardware y el calificativo *"real"* es lo que los diferencia.

Este elemento hardware se refiere, como es obvio, al computador físicamente hablando, es decir, al aparato electrónico generalmente conectado a una toma de luz y sus componentes físicos: la memoria RAM, el HDD y su CPU. Por tanto, a partir de ahora vamos a representar el hardware como:

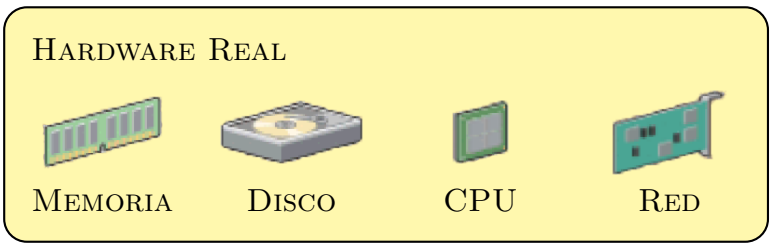


Figura 4: Esquema del hardware real

2.2.2. Sistema Operativo huésped

En este caso no se trata nada más que de un simple sistema operativo y lo que esto conlleva, su funcionalidad y sus restricciones.

Cualquier sistema operativo se puede descomponer en los archivos del sistema o *Kernel* y de las aplicaciones que se ejecutan sobre este sistema, es decir que el esquema es el siguiente:

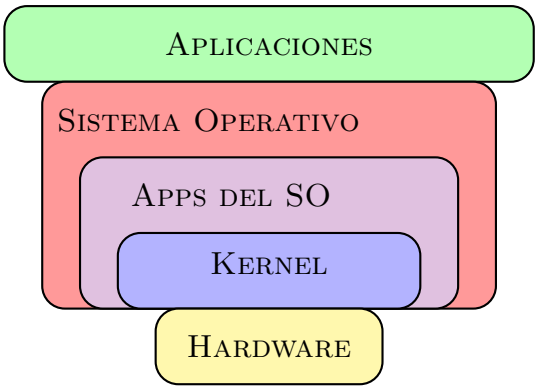


Figura 5: Esquema de componentes de un Sistema Operativo *host* o huésped

Cabe destacar que el hardware se comunica con el sistema operativo a través de su *kernel*. Pero esta última representación profundiza en términos que en estos momentos no nos son de utilidad, por lo que nos vamos a quedar con esta representación:

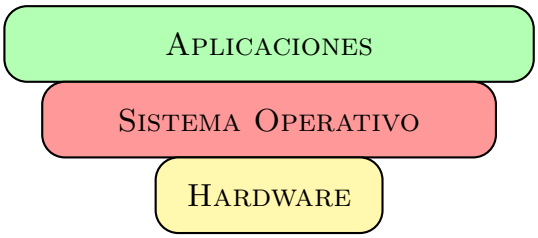


Figura 6: Esquema de componentes de un Sistema Operativo *host* o huésped de manera simplificada

2.2.3. Hipervisor

Ahora nos centramos en el meollo de asunto, pero,en realidad, ¿Nos es relevante el contenido del Hypervisor o Virtual Machine Monitor (VMM)?

Para lo que nosotros vamos a profundizar y a lo que queremos llegar, nos es completamente irrelevante, pero como nunca está de más conocer cosas nuevas simplemente vamos a decir los componentes de un hipervisor varían según el *"fabricante"* y el tipo.

Aun así, existen una serie de elementos comunes para todos ellos, son los referentes a la organización del sistema virtual, por lo que podemos concluir con la siguiente figura:

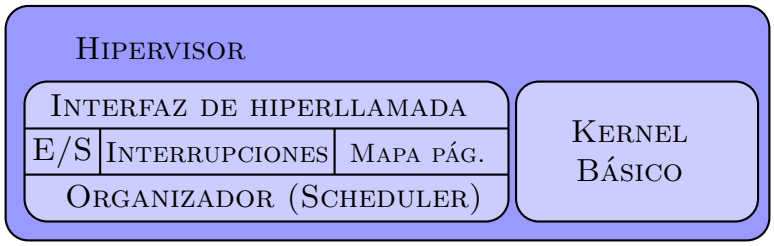


Figura 7: Esquema de componentes del hipervisor

Aquí, al igual que en el caso del sistema operativo (*Véase apartado 2.2.2*) utilizaremos el esquema que ya teníamos en lugar de éste.

2.2.4. Máquina Virtual

Sin necesidad de pensar mucho, y con la comparativa de que una máquina virtual simula el comportamiento de un computador real, es fácil pensar que ésta se compone de manera similar a un computador y acertaríamos, este es su esquema:

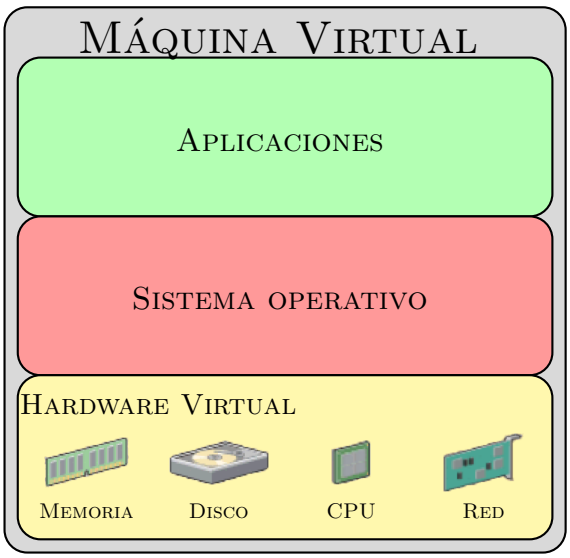


Figura 8: Esquema de componentes de una máquina virtual

Si seguimos la lógica de capas y abstracción, el encapsulamiento que se produce entre el *harware virtual* y el sistema operativo, hacen que éste último no diferencie sobre qué se esta ejecutando. Esto es un punto importante en los sistemas de virtualización, ya que gracias a esto no es necesario adaptar el software a poder ser ejecutado en máquinas virtuales, tan sólo necesitamos instalar un Hypervisor.

2.3. Esquema general

Puestos a hacer un resumen, vamos a agrupar todos los elementos que participan en los sistemas de virtualización con sus correspondientes componentes en una única representación.

Como esta representación varía dependiendo del tipo de virtualización que escojamos, vamos a realizar una representación para cada uno.

2.3.1. Virtualización *nativa*

Para todos los sistemas de virtualización que utilizan un hipervisor de tipo 1 como elemento central, esta es su arquitectura:

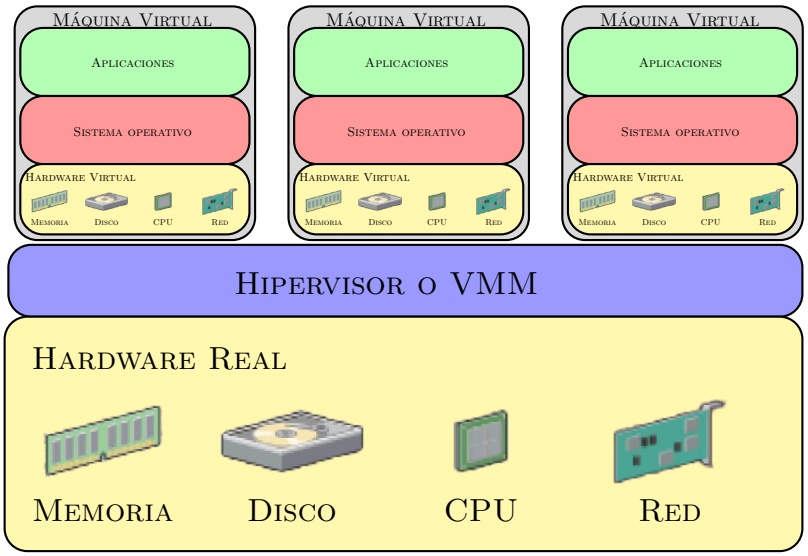


Figura 9: Esquema general de un sistema de virtualización de primer nivel

2.3.2. Virtualización *host*

Para todos los sistemas de virtualización que utilizan un hipervisor de tipo 2 como elemento virtualizador, esta es su arquitectura:

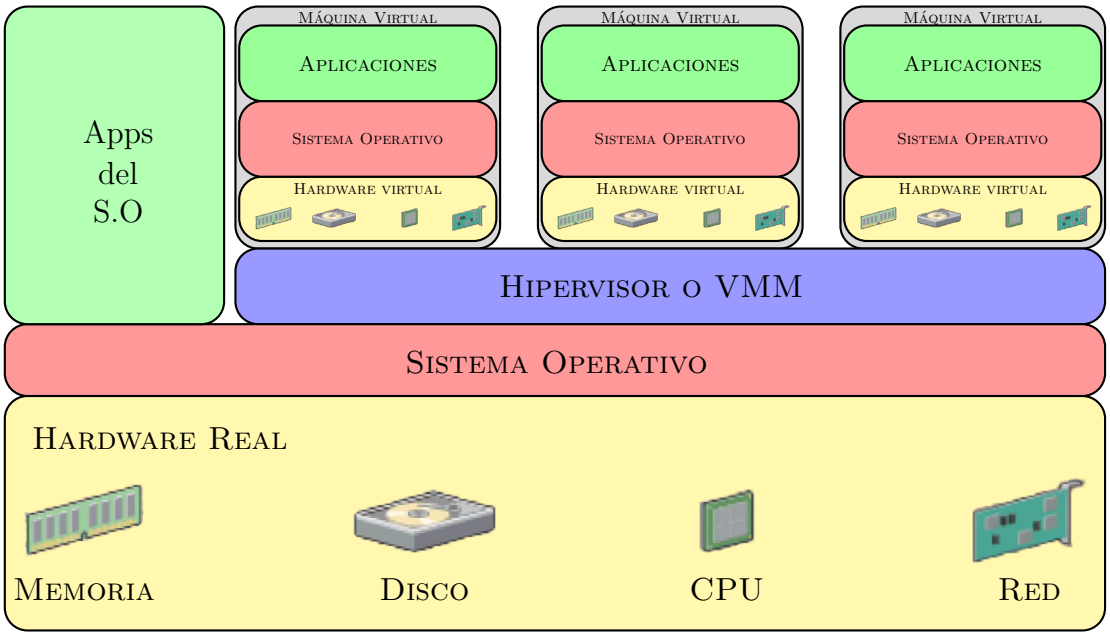


Figura 10: Esquema general de un sistema de virtualización de segundo nivel

Capítulo 3

VMware: Workstation

En este capítulo nos vamos a centrar en la plataforma de VMware, VMware: Workstation[10], los pasos a seguir serán: analizar y ver qué tipo de virtualización utiliza, describir sus principales características y dar ejemplos sobre su uso e instalación.

Pero, antes de nada, es necesario explicar por qué escogemos esta plataforma.

3.1. ¿Por qué VMware: Workstation?

Aunque la respuesta es obvia era necesario dejar una constancia explícita de ello: hemos escogido esta plataforma pues es la que se encuentran normalmente instalada en los ordenadores de las aulas y laboratorios de la Universidad Rey Juan Carlos.

Para llegar a esta conclusión basta con tener acceso de uno de estos ordenadores de sobremesa y los encendamos para su uso, al arrancar descubriremos que sale una ventana con el logo de VMware y con título *VMware: Workstation vX.Y build ZZZZ* siendo X, Y y Z los valores de la versión y compilación utilizada en ese ordenador.

3.2. Arquitectura VMware: Workstation

Vamos pues, a analizar detalladamente la arquitectura de esta plataforma en concreto, para ello vamos a fijarnos en sus especificaciones, en el tipo de virtualización y consultaremos la pagina oficial[10][7].

3.2.1. Tipo de virtualización

Para esta parte, al igual que para la mayoría, podemos saberlo simplemente mediante la observación y con ayuda de los conocimientos previos arriba mencionados (*Véase apartados 2.1, 2.1.2 y 2.1.3*).

Fijándonos en el proceso de arranque de los ordenadores de la URJC es fácil deducir que nos encontramos ante un tipo 2 de hipervisor, ya que para que arranque la máquina virtual

y VMware, primero es necesario que arranque el ordenador con *Windows® XP*.

Si buscamos información en la web[4], y especialmente en la página oficial[1][14], son muchas las referencias que se hacen a un *SO host*, elemento que sólo se incluye en el tipo 2 de virtualización, siendo este su elemento característico.

Por lo tanto concluimos que VMware: Workstation (y por lo tanto los ordenadores de la URJC también) necesitan de un Sistema Operativo base que cumpla los requisitos mínimos. Se trata por tanto de una virtualización de segundo nivel.

3.2.2. Especificaciones técnicas

Ahora vamos a comprobar las distintas especificaciones que tiene esta plataforma en particular, qué la difiere de un sistema genérico de virtualización de segundo nivel.

Entre las que hemos encontrado nos gustaría destacar:

- Es la herramienta con mayor soporte en cuanto a empresas se refiere del mercado.
- Especialmente diseñada para dar funcionalidad a múltiples computadoras.
- Soporta redes virtuales, lo que permite un mayor control sobre las conexiones que se realizan.
- Soporta la virtualización de sistemas x64 en ordenadores x86.

Pero sin duda si queremos explicar por qué la universidad escogió este sistema, se debe a su potencia para un gran número de ordenadores; tal y como se da el caso de la universidad, donde prácticamente todos los computadores se encuentran virtualizados.

3.2.3. Esquema de arquitectura

Por tanto, siguiendo la línea del capítulo anterior, este sería el esquema de la arquitectura instalada en los ordenadores de las aulas de la universidad:

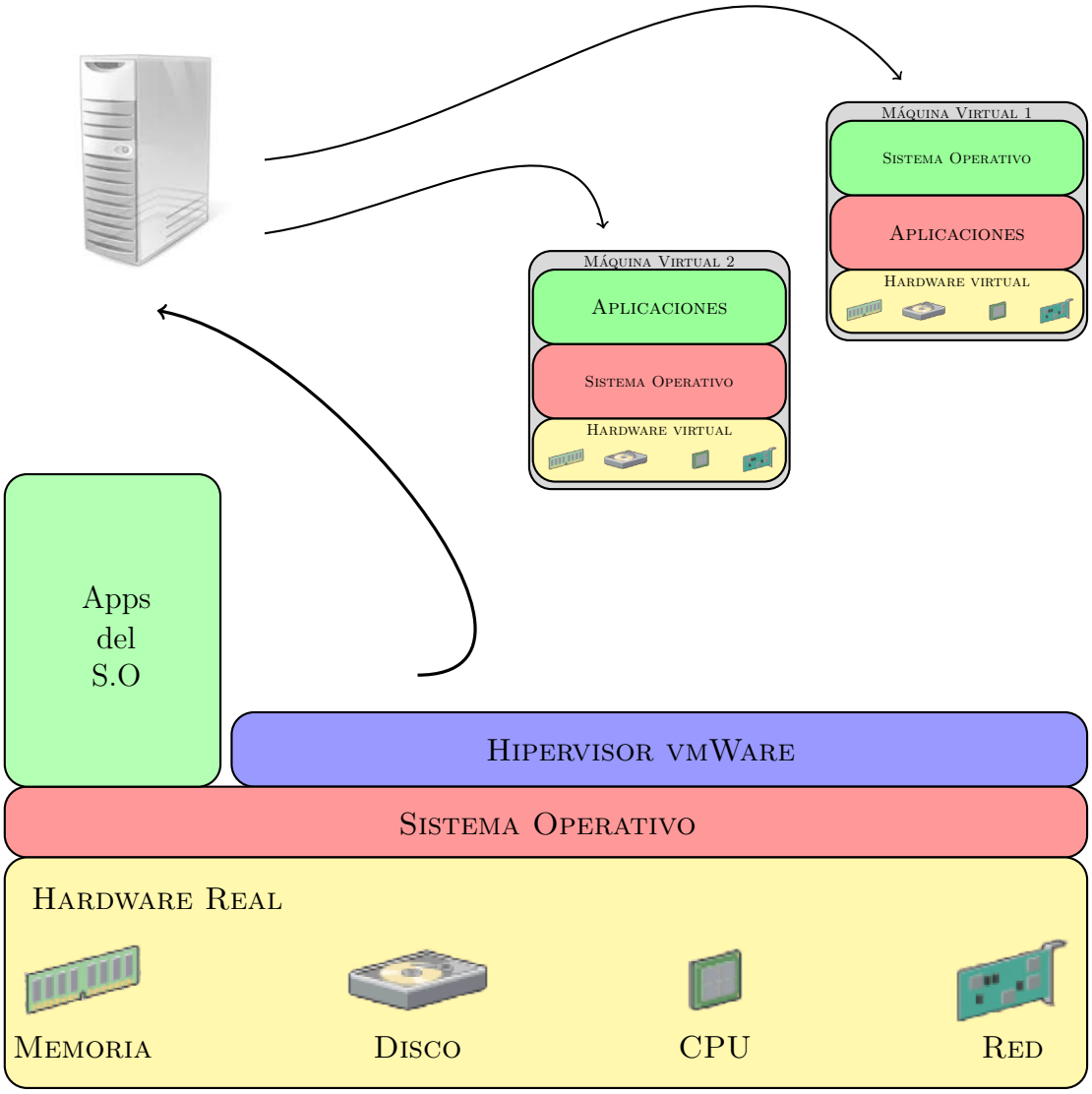


Figura 1: Esquema de la arquitectura de los sistemas virtualizados implementada en la universidad. VMware: Workstation

3.3. Instalación y uso

En esta sección nos gustaría explicar brevemente cómo creemos que se instaló dicho software en los ordenadores de la universidad.

Por comenzar por algún lado, empecemos analizando el sistema operativo huésped[2][3].

3.3.1. Instalación *fresh* de Windows®

Indagando por la web, hemos encontrado en diversos *post*, *guías*, *tutoriales* y *blogs* una continúa mención a una *instalación fresh Windows® XP*, bastante relacionada con lo que a nosotros nos incumbe, los ordenadores de la URJC.
Esta *instalación* no es otra cosa que un sistema optimizado para ser el huésped de un sistema de virtualización de segundo nivel.

Esto es exactamente lo que encontramos en los ordenadores de la universidad; cuando arrancamos los ordenadores, en el arranque, nos da a escoger entre varias opciones y generalmente, la que escogemos para darles su uso normal es *"windows xp vmware worstation"*, donde, al escoger dicha opción, nos arranca un entorno windows minimalista (no posee ninguna aplicación externa) a excepción de la aplicación de VMWare.

3.3.2. Funcionamiento básico

El funcionamiento de VMware es simple, vamos a detallar el proceso que sigue un computador de la universidad desde que se pulsa el botón de arranque hasta que se inicia la máquina virtual:

1. Se arranca el ordenador.
2. En la selección de arranque, escogemos iniciar con VMware.
3. Se inicializa el sistema operativo huésped.
4. Se inicia automáticamente la aplicación de VMware.
5. En la aplicación se introducen los datos de la máquina virtual (para saber cuál escoger entre todas las disponibles en el servidor).
6. El servidor le devuelve la información a la aplicación.
7. Se inicia la maquina virtual.
8. Se pide el usuario y contraseña, cómo un ordenador normal, pero en este caso serán los valores que se usan para todas las cuentas de la URJC.
9. El ordenador esta listo para su uso.

Aunque parece un proceso complicado, en realidad es de lo más simple y conciso. Esto permite distribuir los permisos en los ordenadores y evitar que se instalen programas innecesarios, al tratarse de una máquina virtual, si se requiere la instalación de una aplicación, con instalarse una vez bastaría y ésta se encontraría en todos los ordenadores de la universidad.

Capítulo 4

Conclusiones

Como no podía ser de otra manera, y a modo de aportar nuestra opinión, añadimos en este capítulo las conclusiones obtenidas durante el desarrollo de esta primera parte de la práctica.

En primer lugar nos gustaría recalcar que aunque los sistemas de la universidad *"aparentemente"* poseen grandes ventajas que se han ido comentando a lo largo de este documento, también poseen desventajas que hacen que ninguno de nosotros prefiera un ordenador de la universidad a uno propio.

Uno de estos impedimentos a los que nos referimos es a la capacidad de los ordenadores de almacenar datos. Éstos solo perduran durante la sesión actual, esto significa, para muchos, auténticos quebraderos de cabeza pues si no se guardan los archivos que se hayan modificado en un lugar seguro (*en: tu cuenta de dropbox, mega, correo, pendrive*) y apagas el ordenador, cierras sesión o se va la luz; éstos desaparecen, cosa que hace que trabajar con dichas máquinas sea una tarea difícil.

El término *ordenador* tan sólo es usado en España entre los países hispanohablantes, proviene de *ordinateur*, palabra francesa, es por tanto un galicismo que significa que *ordena* u organiza datos, pero resulta que éstos no pueden ser guardados, cosa que resulta un tanto inútil. Esto es debido a que las máquinas virtuales están virtualizadas a través de un servidor y este tiene un espacio finito, bastante inferior al de todos los ordenadores de la universidad.

Por otro lado, nos ha resultado de lo más interesante averiguar cómo están funcionando actualmente los ordenadores de la universidad. Este proceso de ingeniería inversa nos ha propuesto grandes retos, que creemos que hemos sido capaces de afrontar con alguna dificultad.

En resumen, nos ha gustado mucho el proceso que hemos seguido para entender los sistemas de virtualización, con los que hemos aprendido bastante. pero no nos ha agradado saber que la universidad puede utilizar otros métodos de virtualización que aportarían lo mismo que el sistema actual, con el añadido que harían que trabajar con ellos fuera de una manera más amena, empezando por pasar a un sistema de virtualización de primer nivel y con un sistema de revisiones periódico para las modificaciones de las máquinas virtuales. Un sencillo

ejemplo: Que cada día a las 22:00 (tras cerrar la universidad) el servidor donde se almacenan los datos de las VM busque actualizaciones y demás, al iniciar cada ordenador compruebe si posee la ultima versión de los datos de la VM, y si no es así que actualice.

Por lo tanto, un sistema de virtualización adecuado puede hacer obtener un mayor control sobre un sistema a la par que se obtiene un mayor rendimiento.

Parte II

Diseño

Capítulo 1

Introducción

En la segunda parte de la práctica se nos pide el diseño de una plataforma que implemente una arquitectura de pizarra, para ello vamos a seguir un modelo de desarrollo en cascada[1].

Por tanto, hemos dividido esta parte de la memoria en cada una de las fases de éste modelo de desarrollo, pero antes es necesario centrarnos en cómo debe diseñarse esta plataforma y que implica que se diseñe de esta manera.

1.1. Arquitectura de pizarra

Cómo bien se nos explica en el enunciado de la práctica, tenemos que crear una plataforma que siga, en su diseño, una arquitectura de pizarra; pero, ¿Qué es una arquitectura de pizarra?

Para definir una arquitectura de pizarra o de tuplas debemos entender que se trata de una arquitectura software, es decir un modelo básico en el que basarse para la creación de software similar y , por lo tanto, debemos describir qué es exactamente una arquitectura de pizarra, para seguidamente poder analizar en detalle la plataforma a crear.

Para entender en concepto básico de esta arquitectura basta con fijarse en su nombre: *"pizarra"* y es que, en realidad se trata de eso, de una gran pizarra en la que cualquiera que tenga los conocimientos necesarios puede aportar sus: cálculos, procesos, metodologías, etc. para la resolución de problemas de cualquier tipo.

Siguiendo con este ejemplo, en una resolución de problemas cualquiera encontraríamos a varias personas trabajando sobre una misma pizarra, a estos elementos en la arquitectura de pizarra se les conoce cómo *agentes* y *pizarra*, respectivamente.

Siendo un poco más técnicos podemos definir estos dos componentes como:

[1] **Modelo en cascada:** Divide el desarrollo de software en etapas rigurosamente definidas, de forma que para el inicio de una etapa debe esperarse previamente que finalice la anterior. Las etapas son: análisis de requerimientos, diseño, implementación, pruebas y mantenimiento.

- **Pizarra:** Instrumento de control o estructura de datos que representa el estado actual de la plataforma.
- **Agente:** Elementos funcionales independientes entre sí que operan sobre la pizarra.

La siguiente figura muestra como interactúan estos *agentes* con la *pizarra*:

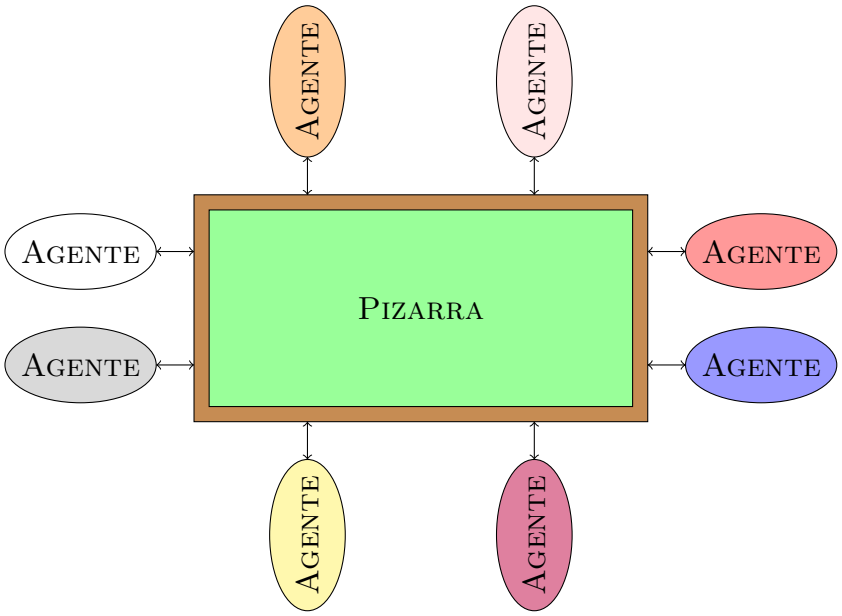


Figura 1: Interacción entre los agentes y la pizarra

La función de los agentes es, basándose en el contenido de la pizarra, realizar la tarea que se les asigna y escribir sobre ella sus resultados.

La función principal de la pizarra es coordinar la interacción de los agentes con la pizarra y la comunicación entre ellos. La pizarra irá cambiando de estado según van escribiendo en ella los distintos agentes. Éstos podrán leer lo que fue escrito por otros agentes anteriores a él. De esta forma el contenido de la pizarra será cada vez más completo, hasta alcanzar una solución final.

Una arquitectura de pizarra suele ser utilizada en los siguientes casos:

- Problemas en los que no existe una solución analítica.
- Problemas en los que, aunque exista solución analítica, es inviable por la dimensión del espacio de búsqueda.
- Problemas extremadamente complejos en términos cognitivos.

Las arquitecturas en pizarra presentan una serie de inconvenientes importantes, estos radican principalmente en la complejidad de los problemas que se resuelven con este tipo de arquitecturas. Son los siguientes:

- Una vez obtenida una solución, es difícil ver de forma ordenada los pasos que llevaron a esa solución.
- No garantiza obtener una solución.
- No se puede saber a priori el tiempo necesario para realizar el problema.

1.2. ¿Qué implica la arquitectura de pizarra?

El utilizar una arquitectura de pizarra para el diseño de la plataforma supone restringir ciertos aspectos.

Implica que no existe interrelación entre los agentes, sólo a través de la pizarra de manera que los agentes colaboran entre ellos para obtener una solución común al problema que se plantea. Por ejemplo, en una plataforma como GitHub, donde distintas personas, de distintos países del mundo, colaboran entre ellos, en este caso, en conseguir hacer un producto software, generalmente.

Por otro lado, se nos dice en el enunciado que esta pizarra tendrá la característica de que se implementara siguiendo un modelo cliente-servidor, es decir: Debe estar preparada para poder crear la pizarra en un computador y cada agente en otro, conectados a través de la red, esto no quiere decir que en un mismo ordenador no se puedan ejecutar una aplicación cliente y otra servidor, al contrario, en caso de implementar algo de código, el modo de probarlo sería de esta manera.

1.3. ¿Cómo va a ser la plataforma?

Nuestra plataforma irá destinada a la interacción de distintos agentes que podrán subir y modificar archivos a través de la pizarra, con tres operaciones básicas: Escribir(**in**), Leer(**out**) y Escribir sin eliminar(**rd**).

La plataforma constará de dos partes:

- **Pizarra:** Será la estructura que contenga la información. Permite a los agentes interactuar sobre ella, escribiendo y leyendo su contenido.
- **Librería:** Dará las funcionalidades añadidas necesarias para que un programador pueda diseñar un agente que interactúe con la pizarra.

La implementación de la plataforma se realizará con el lenguaje de programación C++, ya que al ser un lenguaje orientado a objetos nos permite una mayor libertad y comodidad.

Capítulo 2

Documentación

Una vez introducida la arquitectura de pizarra y la estructura básica que tendrá la plataforma daremos paso a una documentación detallada de la misma.

Para abordar la documentación de la plataforma ha sido necesario obtener previamente una lista de requisitos, para después, una vez desarrollados y clasificados convenientemente, dar lugar a la realización de los diagramas UML correspondientes que definen el diseño de la plataforma.

Ahora centrémonos en cómo hemos obtenido esos requisitos y a que conclusiones hemos llegado.

2.1. Obtención de los requisitos

Para abordar la obtención de los requisitos, y debido a que obtener los requisitos de la plataforma de la qué no teníamos una idea muy clara y concisa de su funcionalidad, hemos partido de una posible aplicación que pudiera ser implementada con nuestra librería para así poder buscar su funcionalidad, es decir, hacernos una idea clara de qué debería llevar y a partir de ahí, obtener tanto los requisitos de esta posible aplicación como separarlos y extrapolarlos a la librería.

La aplicación que tomamos como referencia para la extracción de requisitos permite a un profesor, así como a sus alumnos subir y modificar archivos en una pizarra externa, permitiendo así la cómoda realización de trabajos en grupo por parte de los alumnos y un seguimiento por parte del profesor.

Pensar de un modo más aplicado nos ha permitido extraer requisitos con más facilidad, pensando en qué cosas serían necesarias para el buen funcionamiento de la aplicación.

2.1.1. Clasificación de requisitos

Una vez extraídos los requisitos es necesario clarificarlos, ya que no todos hacen referencia a una misma parte de la funcionalidad, por ello se han clasificado según los siguientes criterios:

- **Funcionales:** Son declaraciones de los servicios que debe proporcionar el sistema. Especifica la manera en que éste debe reaccionar a determinadas entradas. Especifica cómo debe comportarse el sistema en situaciones particulares.
- **No funcionales:** Restricciones de los servicios o funciones ofrecidas por el sistema (fiabilidad, tiempo de respuestas, capacidad de almacenamiento, etc.). Generalmente se aplican al sistema en su totalidad. Surgen de las necesidades del usuario debido a restricciones de presupuesto, políticas de la organización, necesidad de interoperatividad, etc. Estos a su vez se pueden dividir en:
  - **Del producto:** Especifican el comportamiento del producto, por ejemplo fiabilidad o rapidez.
  - **Organizacionales:** Derivan de políticas y procedimientos existentes en la organización del cliente y del desarrollador, por ejemplo la utilización de un lenguaje de programación en concreto.
  - **Requisitos externos:** Se derivan de factores externos al sistema y a su proceso de desarrollo, por ejemplo requisitos de interoperatividad, la capacidad de intercambiar información de un sistema, o éticos, como el cumplimiento de normativas.

2.1.2. Pensamos en la posible aplicación

En cuanto a la hora de pensar en qué tipo de aplicación podría ser implementada se nos ocurrieron varias ideas[5], pero optamos por quedarnos con una aplicación del ámbito educativo. Concretamente, esta aplicación permitiría almacenar archivos en la pizarra o servidor y dependiendo del nivel del usuario (alumno o profesor), éste tendría habilitada unas u otras opciones, por ejemplo el profesor podría comparar los resultados de un grupo de prácticas de alumnos con los de otros para buscar copias o un alumno podría preguntar a otros grupos.

Tras esta pequeña descripción de la aplicación, vamos a redactar una lista con sus requisitos para poder trabajar con ellos:

2.1.3. Requisitos de la aplicación

Lista de requisitos de una posible aplicación:

- **Requisitos funcionales**
  - La pizarra y los agentes siguen un modelo cliente-servidor.
  - Los agentes pueden leer, escribir y modificar la pizarra.
  - La pizarra debe permitir gestionar la BD de los distintos usuarios.
  - Los agentes no deben poder interrelacionarse entre ellos si no es a través de la pizarra.

- Debe haber distintos tipos de usuarios: profesores y alumnos.
- La pizarra debe permitir la creación de apartados para los ejercicios, por ejemplo Tema 1.
- La pizarra debe guardar el progreso de cada usuario, mediante gráficos, estadísticas, etc.
- La pizarra debe distinguir los distintos tipos de ejercicios de los alumnos.
- La pizarra debe reconocer los tipos de archivo de los ejercicios.

- **Requisitos no funcionales**
  - **De producto**
    - La pizarra no debe permitir a los alumnos modificar los apuntes subidos por un profesor.
    - La pizarra debe bloquear a los usuarios para que no puedan acceder a ejercicios de clases en las que no están inscritos.
    - La pizarra debe almacenar una contraseña y un identificador único para cada usuario y no permitir que otros accedan.
    - La pizarra debe permitir al profesor bloquear contenido a los alumnos.
  - **Organizacionales**
  - **Externos**

Pero, como se puede apreciar, estos requisitos hacen referencia a ciertos elementos que no pertenecen para nada a la plataforma a implementar, como pudieran ser *alumno* o *profesor*.

2.1.4. Requisitos de la plataforma

Para solucionar esto, se han extraído de los requisitos anteriores, la funcionalidad que va a tener nuestra plataforma y se han vuelto a redactar sus requisitos.

- **Requisitos funcionales**
  - Los agentes pueden leer, escribir y modificar la pizarra.
  - La pizarra debe permitir gestionar los distintos tipos de agentes.
  - Los agentes no deben poder interrelacionarse entre ellos si no es a través de la pizarra.
  - Debe haber distintos tipos de agentes, organizados por una jerarquía de nivel (p.e. directores, profesores, alumnos).
  - La pizarra debe permitir la creación de carpetas donde guardar los elementos para organizarlos.
  - La pizarra debe permitir crear nuevos usuarios.
  - La pizarra permitirá la creación de nuevos usuarios.
  - La pizarra debe guardar las estadísticas de colaboración de cada usuario, mediante gráficos, estadísticas, etc.

- La pizarra debe distinguir los distintos tipos de elementos de los agentes.
- La pizarra debe reconocer los tipos de archivo de los elementos.
- La pizarra debe ser capaz de comparar dos archivos con el objetivo de buscar diferencias.
- La pizarra debe permitir realizar una búsqueda de un archivo por su nombre.
- La pizarra debe permitir ser configurada como "pública" o "privada".
- El agente pedirá los credenciales del usuario al comenzar.
- El agente se encargará de revisar la veracidad de los credenciales del usuario.
- El agente adquirirá permisos y funciones dependiendo del usuario que lo ese usando.

- **Requisitos no funcionales**

  - **De producto**
    - La pizarra debe gestionar los permisos, y poder configurarlos para decidir que puede hacer cada agente.
    - La pizarra debe bloquear a los usuarios para que no puedan acceder a problemas que no se les han asignado.
    - La pizarra debe almacenar una contraseña y un identificador único para cada usuario y no permitir que otros accedan.
    - La pizarra debe permitir a los agentes de mayor nivel, bloquear contenido.
    - Por seguridad para los usuarios deben proteger su cuenta con una contraseña de más de 6 caracteres.
    - La pizarra no revelará información personal acerca de los agentes a parte del nombre.
    - Cada usuario accederá a la pizarra a través de un agente.
  - **Organizacionales**
    - La plataforma se realizará con el lenguaje de programación C++.
    - La plataforma estará implementada siguiendo una arquitectura de pizarra.
    - La pizarra y los agentes siguen un modelo cliente-servidor.
    - La comunicación entre la pizarra y el agente (modelo cliente-servidor) seguirá una encriptación SSL.
  - **Externos**
    - La comunicación entre la pizarra y el agente (modelo cliente-servidor) será a través del protocolo HTTP.
    - La pizarra no revelará información acerca de los agentes, excepto su nombre y número de referencia.

Ahora, ya tenemos unos requisitos reales de nuestra plataforma con los que poder afrontar el diseño.

2.1.5. ¿Son válidos estos requisitos?

Como se trata una librería, nos surgía la duda de que si los requisitos obtenidos nos eran válidos para el diseño.

Pronto descubrimos que sí eran válidos, pero no eran eficientes, al estar todos mezclados era difícil saber por donde había que comenzar el diseño, por lo que optamos por organizar los requisitos de otra manera.

2.2. Requisitos destinados al diseño

Para que el diseño sea más conciso hemos optado por separar los requisitos en dos apartados adicionales: requisitos de la librería, que serán los requisitos de las funcionalidades que se le añaden a la pizarra y requisitos de la pizarra en sí, donde se definirá y diseñará la arquitectura propiamente dicha.

Requisitos de la pizarra

Lista con todos los requisitos de pizarra.

- **Requisitos funcionales**

  1. Los agentes pueden leer, escribir y modificar la pizarra.
  2. La pizarra debe permitir gestionar los distintos tipos de agentes.
  3. Los agentes no deben poder interrelacionarse entre ellos si no es a través de la pizarra.
  4. Debe haber distintos tipos de agentes, organizados por una jerarquía de nivel (p.e. directores, profesores, alumnos).
  5. La pizarra debe permitir la creación de carpetas donde guardar los elementos para organizarlos.
  6. La pizarra debe distinguir los distintos tipos de elementos de los agentes.
  7. La pizarra debe reconocer los tipos de archivo de los elementos.
  8. La pizarra debe ser capaz de comparar dos archivos con el objetivo de buscar diferencias.
  9. La pizarra debe permitir realizar una búsqueda de un archivo por su nombre.
  10. La pizarra debe permitir ser configurada como “pública” o “privada”.

- **Requisitos no funcionales**

  - **De producto**

    1. La pizarra debe gestionar los permisos, y poder configurarlos para decidir que puede hacer cada agente.

    1. La plataforma se realizará con el lenguaje de programación C++.
    2. La pizarra y los agentes siguen un modelo cliente-servidor.

3. La comunicación entre la pizarra y el agente (modelo cliente-servidor) seguirá una encriptación SSL.

- **Externos**
  1. La comunicación entre la pizarra y el agente (modelo cliente-servidor) será a través del protocolo HTTP.
  2. La pizarra no revelará información acerca de los agentes, excepto su nombre y número de referencia.

Requisitos de la librería

Lista con todos los requisitos de librería.

- **Requisitos funcionales**
  - La plataforma se gestiona mediante usuarios.
  - La plataforma permitirá la creación de nuevos usuarios.
  - La plataforma debe guardar las estadísticas de colaboración de cada usuario, mediante gráficos, porcentajes, etc.
  - La plataforma pedirá las credenciales del usuario al comenzar.
  - La plataforma se encargará de revisar la veracidad de los credenciales del usuario.
  - El agente adquirirá permisos y funciones dependiendo del usuario que lo ese usando.
- **Requisitos no funcionales**
  - **De producto**
    - La plataforma debe bloquear a los usuarios para que no puedan acceder a problemas que no se les han asignado.
    - La plataforma debe almacenar una contraseña y un identificador único para cada usuario y no permitir que otros accedan.
    - La plataforma debe permitir a los usuarios de mayor nivel bloquear contenido.
    - Por seguridad los usuarios deben proteger su cuenta con una contraseña de más de 6 caracteres.
    - La plataforma no revelará información personal acerca de los usuarios a parte del nombre.
    - Cada usuario accederá a la pizarra a través de un agente.
  - **Organizacionales**
    1. La plataforma se realizará con el lenguaje de programación C++.
    2. La plataforma estará implementada siguiendo una arquitectura de pizarra.
  - **Externos**

Ahora sí, con estos requisitos estamos plenamente preparados para comenzar el diseño de la plataforma, eso sí, comenzando primero por la arquitectura de pizarra.

Capítulo 3

Arquitectura y Diseño

En este apartado nos centramos plenamente en lo que al diseño se refiere.

Los pasos a seguir son: primero describiremos la arquitectura de pizarra mediante un esquema y un diagrama UML de casos de uso; después, analizaremos el diseño en función de los requisitos obtenidos anteriormente y finalmente, diseñaremos la plataforma.

3.1. Arquitectura

Ahora vamos a centrarnos en definir gráficamente la arquitectura, en todo momento tenemos en cuenta los requisitos obtenidos en el capítulo anterior (*Véase apartado 2.2*).

3.1.1. Estructura de nuestra arquitectura de pizarra

Para conocer un poco más nuestra arquitectura, es necesario realizar ciertos esquemas o diagramas. Este es uno de ellos que hacen que se entienda bastante bien la arquitectura:

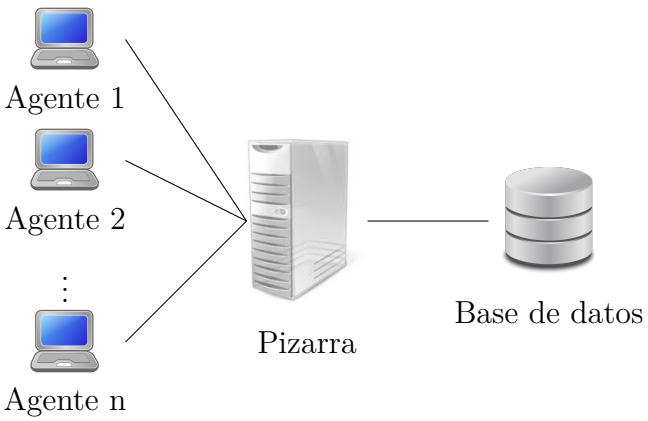


Figura 1: Esquema donde se define la estructura básica de nuestra arquitectura de pizarra

Con esto queda claro como se organizan los agente con la pizarra, pero a grandes rasgos.

3.1.2. Análisis de la estructura

Para dar un poco más de detalle y ver de que manera se relacionan agente y pizarra, hemos construido un diagrama de casos de uso (*Véase apartado 3.2.1*) a partir de los requisitos obtenidos previamente (*Véase apartado 2.2*), este ha sido el resultado:

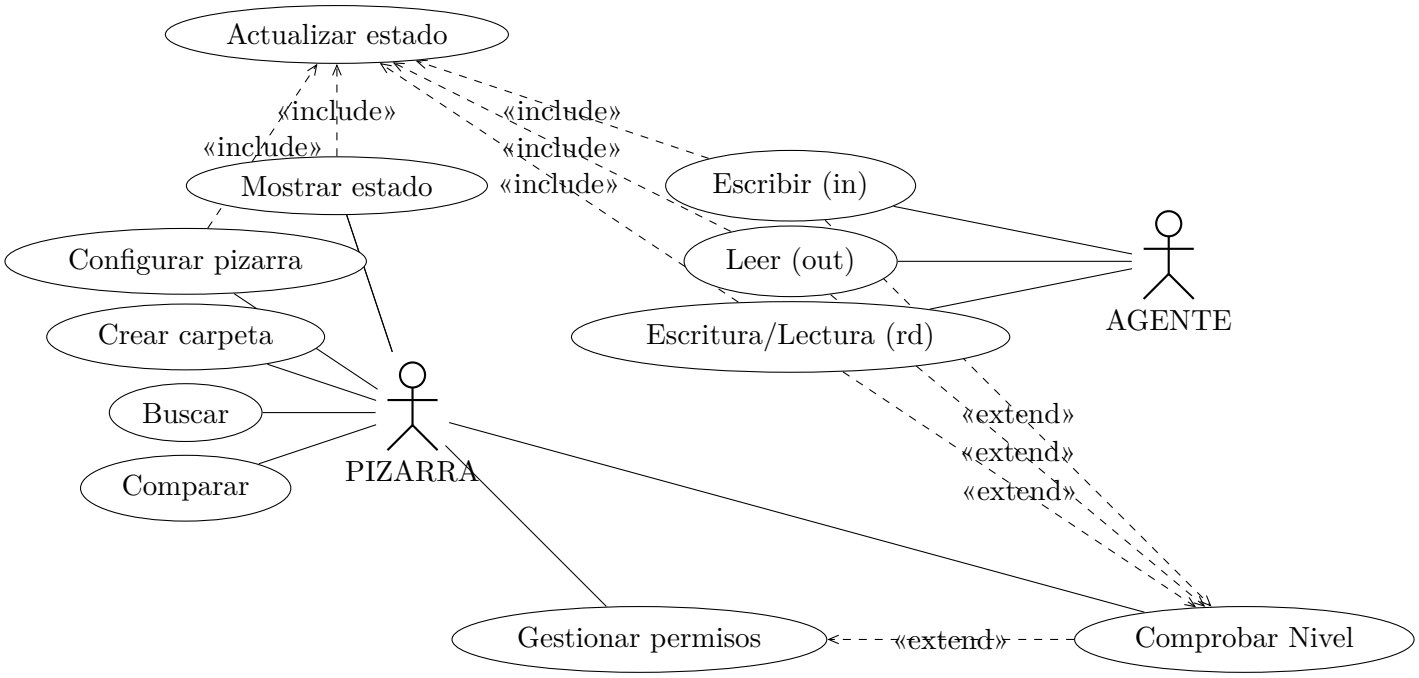


Figura 2: Diagrama de casos de uso de la arquitectura

Como se puede apreciar en la *figura 2* los agentes se comunican a la pizarra mediante *Comprobar nivel* y *Actualizar estado*, esto significa que en esas funcionalidades habrá que incluir la conexión entre ambos.

Por otro lado, se trata de un diagrama sencillo que no tiene gran contenido, como era de esperar.

3.2. Diseño inicial o análisis

En este apartado el principal objetivo es realizar un análisis de los requisitos obtenidos y un diseño previo, similar al realizado en el apartado anterior, sólo que en este caso de una manera más completa.

En primer lugar diseñaremos un diagrama de casos de uso que permite ver de una forma más detallada los requisitos extraídos anteriormente así como su relación con los distintos actores, para después analizar de forma detallada mediante diagramas de actividad cada uno de los casos de uso resultantes. Para finalizar, incluiremos un diagrama de clases que permita una visión general de las clases que componen la plataforma y su relación.

3.2.1. Diagrama de casos de uso

Un diagrama de casos de uso permite la visualización de las actividades que se permite realizar la plataforma. A estas actividades se las denomina *casos de uso*. El diagrama de casos de uso define también los actores o roles que interactúan con la aplicación y las relaciones que existen entre los distintos casos de uso. Las relaciones pueden ser de dos tipos:

- **Inclusión:** Un caso de uso depende de la salida que el caso de uso con el que se relaciona produce.
- **Extensión:** Un caso de uso que extiende su comportamiento a otro caso de uso, que realiza la misma función.

Diagrama de casos de uso de la paltaforma

En nuestro caso el diagrama se encuentra el la *figura 3*, en ella se aprecia que nuestra plataforma consta de tres actores:

- **Usuario:** Cualquier persona con una cuenta a la que se le permita la interacción con el servidor.
- **Administrador:** Es un usuario normal, pero con funcionalidad añadida; sólo existe uno en el sistema.
- **Pizarra:** Es el actor principal de nuestra aplicación; es el responsable de interrelacionar a los usuario y al administrador.

También se ha de apreciar que los casos de uso son muy similares a los que encontramos en la *figura 2*, correspondiente con el diseño de la arquitectura, en realidad el único cambio que se produce es el cambio entre agente y usuario.

Ahora, en su lugar, encontramos un actor *Usuario* que solo tiene acceso a *Iniciar Sesión*, y es que, en resumidas cuantas, podemos descomponer un *Agente* como un usuario que inicia sesión desde un ordenador.

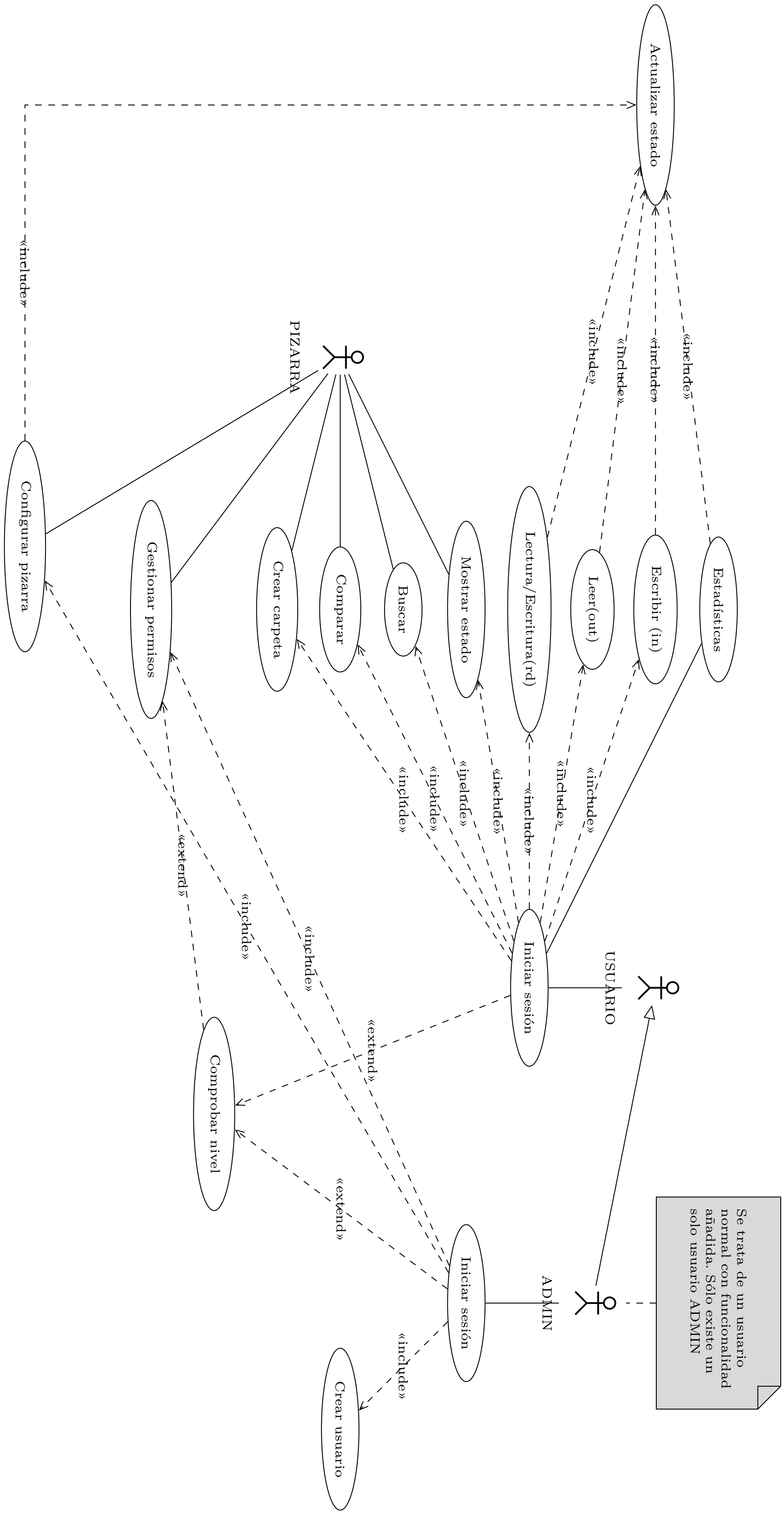


Figura 3: Diagrama de casos de uso

3.2.2. Diagramas de actividad

Un diagrama de actividad es una representación de un proceso de forma gráfica. Consta de una serie de símbolos que representan los distintos pasos a seguir y flechas que indican el flujo de ejecución que se sigue un caso de uso. Hemos realizado uno para cada caso de uso:

Comprobar nivel: (*Figura 4*) este caso de uso permite comprobar el nivel en el que se encuentra el usuario.

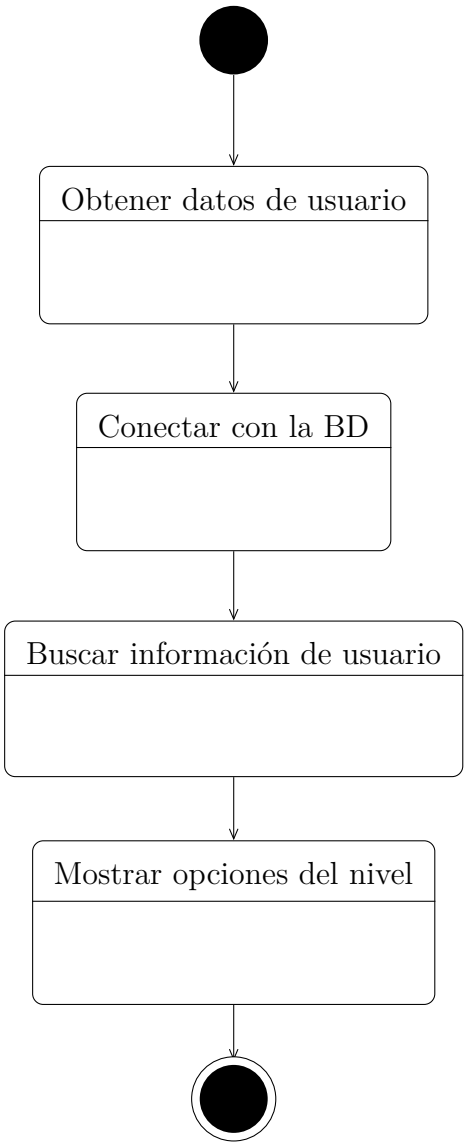


Figura 4: Diagrama de actividad de Comprobar nivel

Iniciar sesión: (*Figura 5*) se introduce el id de usuario y la contraseña para poder tener acceso a la plataforma.

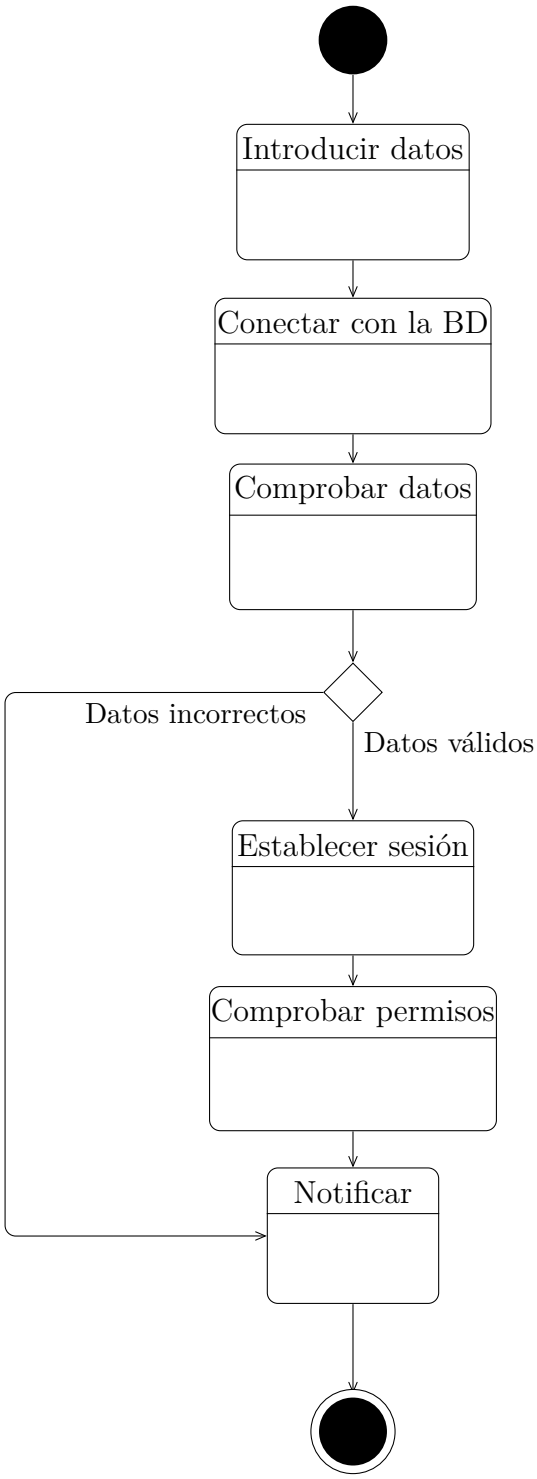


Figura 5: Diagrama de actividad de Iniciar sesión

Actualizar Pizarra: (*Figura 6*) éste es uno de los estados básicos de la pizarra, ya que cada vez se escribe, se borra o se modifica algún archivo hay que hacer uso de éste.

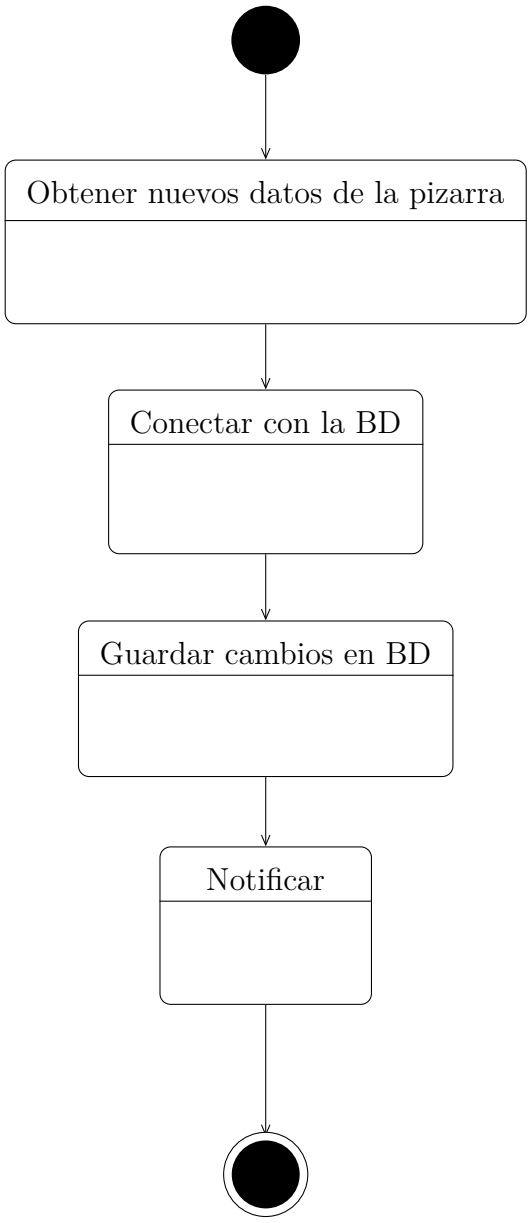


Figura 6: Diagrama de actividad de Actualizar pizarra

Estadísticas: (*Figura 7*) permite ver las estadísticas de un determinado usuario.

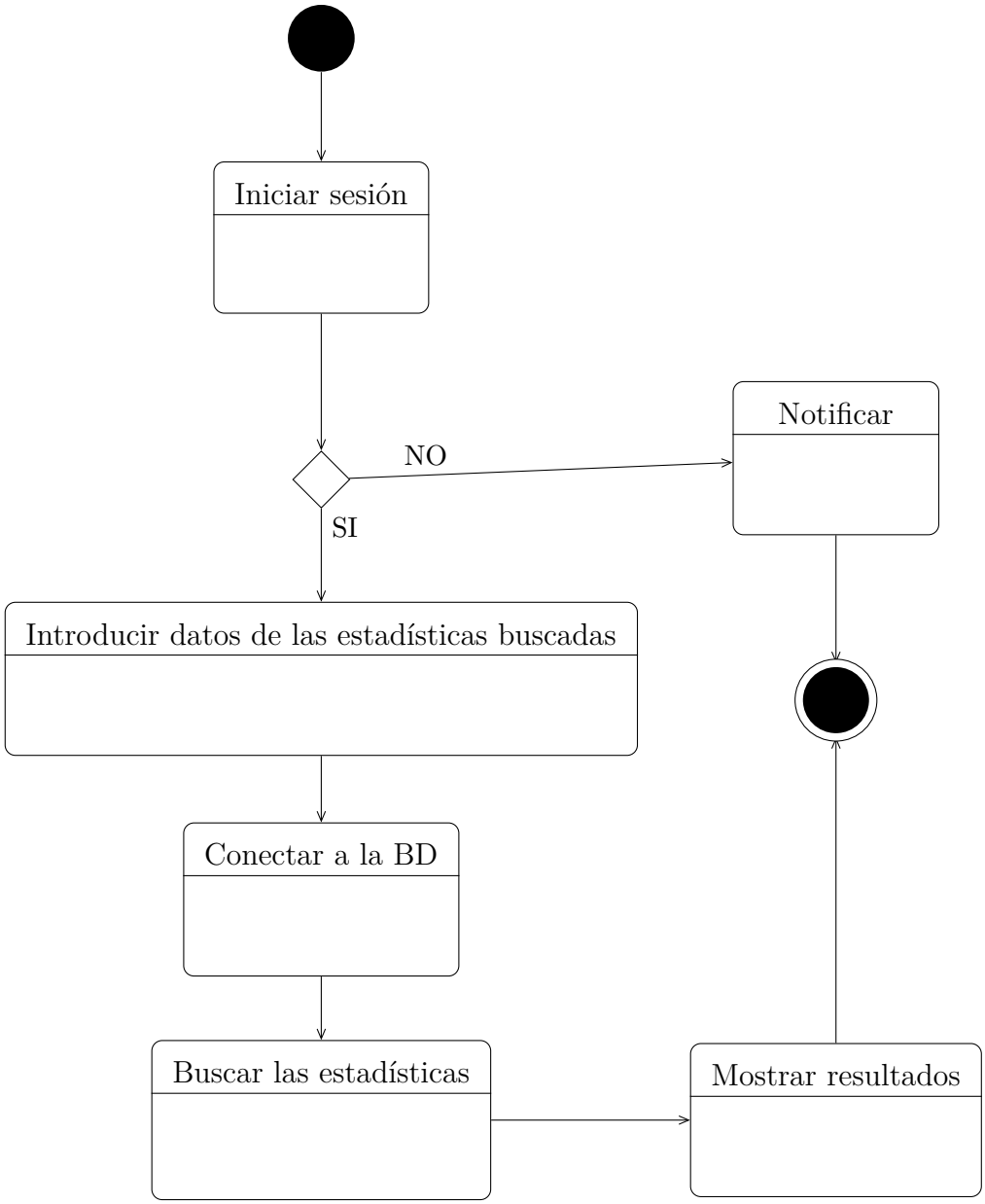


Figura 7: Diagrama de actividad de Estadísticas

Escribir (in): (*Figura 8*) éste es uno de las operaciones básicas de la pizarra. Escribe un nuevo dato en la pizarra.

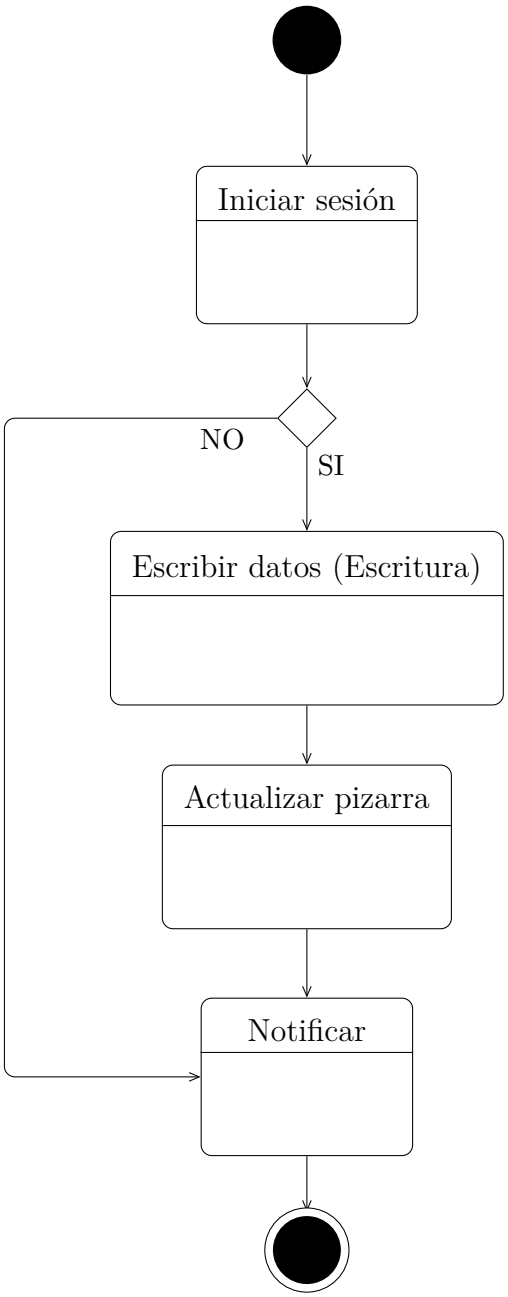


Figura 8: Diagrama de actividad de Escribir

Leer (out): (*Figura 9*) leer datos de la pizarra.

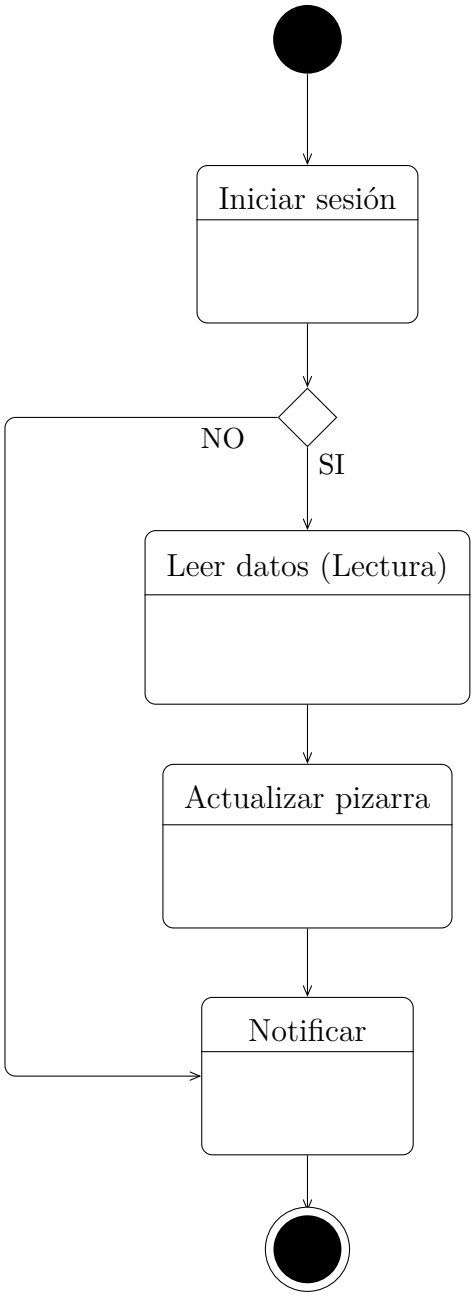


Figura 9: Diagrama de actividad de Leer

Lectura/Escritura (rd): (*Figura 10*) este caso de uso permite la escritura en la pizarra pero sin eliminar los datos anteriores, es decir que permite tanto leer como escribir en la pizarra.

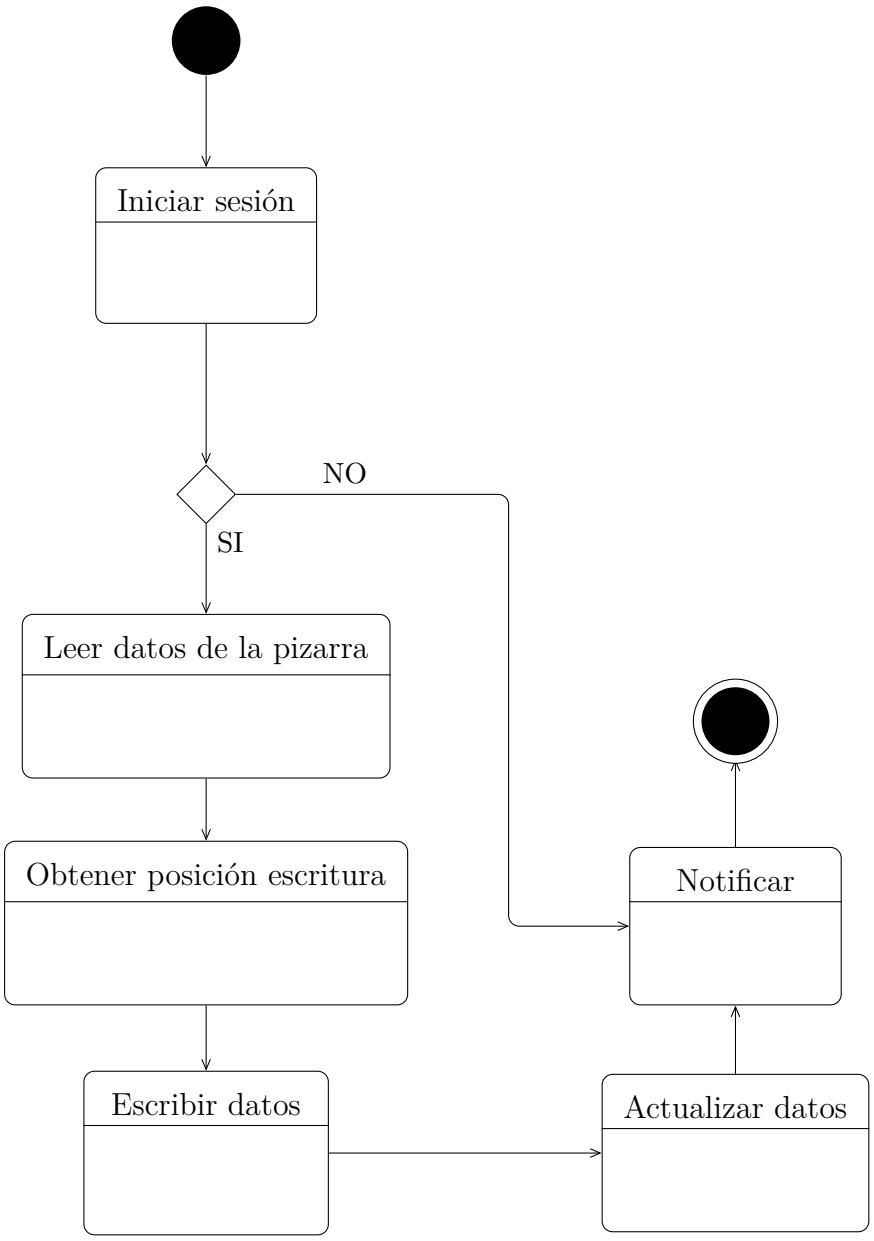


Figura 10: Diagrama de actividad de Lectura/Escritura

Mostrar estado: (*Figura 11*) muestra el estado actual de la pizarra.

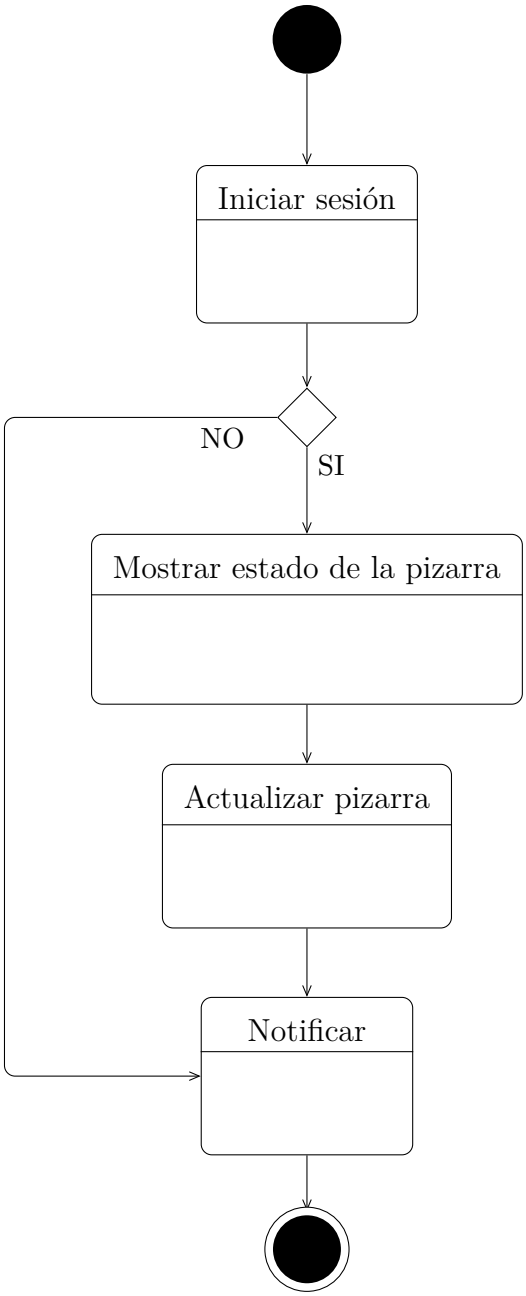


Figura 11: Diagrama de actividad de Mostrar estado

Buscar: (*Figura 12*) este caso de uso busca entre los archivos de la pizarra y devuelve los datos de los archivos en caso de que se hayan encontrado de acuerdo a los datos de la búsqueda.

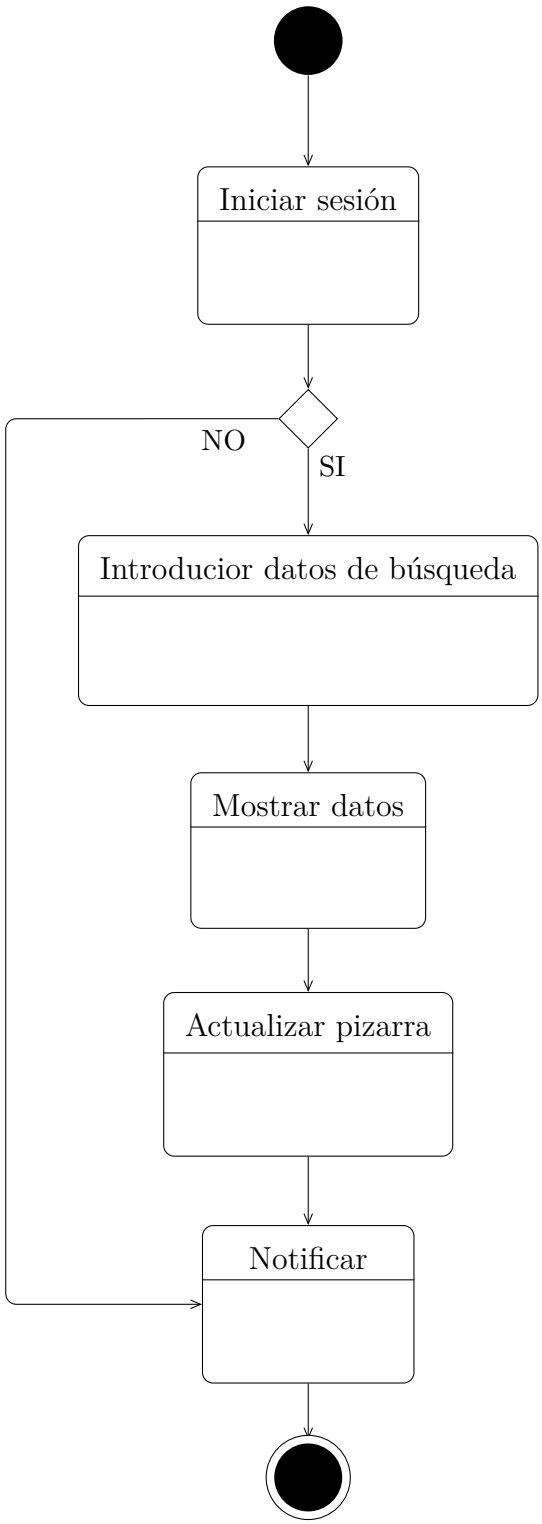


Figura 12: Diagrama de actividad de Buscar

Comparar: (*Figura 13*) este caso de uso compara dos o más archivos y devuelve si se ha modificado algo y qué es lo que se ha modificado.

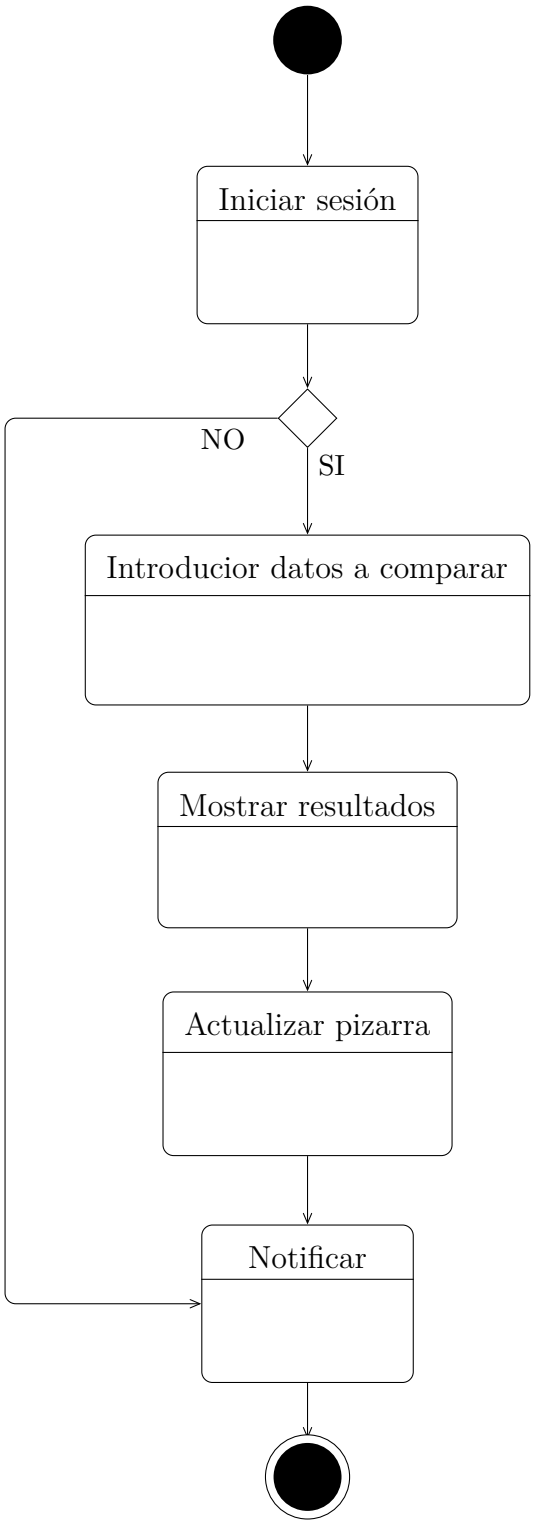


Figura 13: Diagrama de actividad de Comparar

Crear carpeta: (*Figura 14*) este caso de uso permite crear nuevas carpetas a los usuarios.

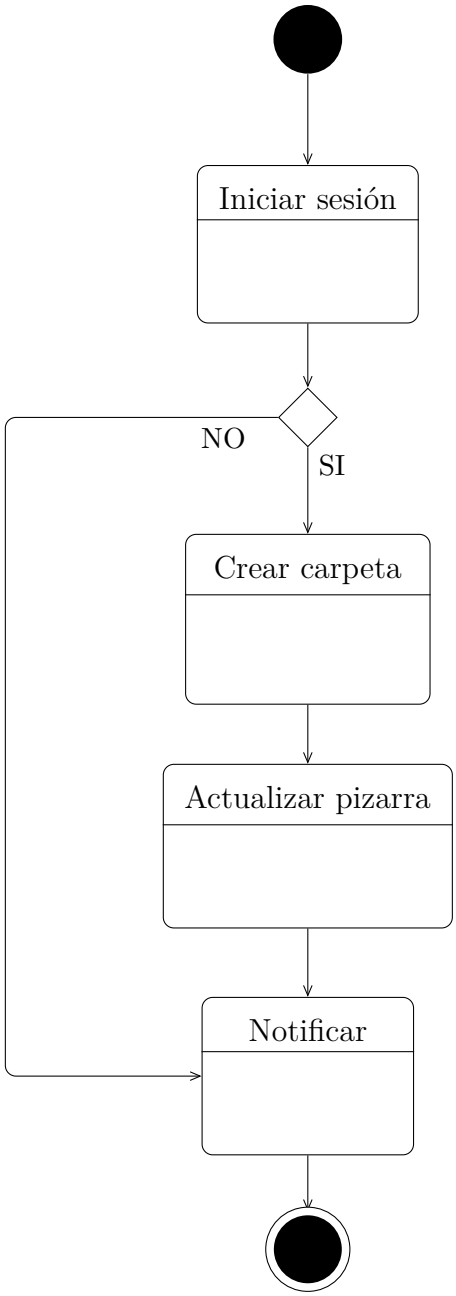


Figura 14: Diagrama de actividad de Crear carpeta

Gestionar permisos: (*Figura 15*) permite cambiar los permisos tanto de lectura como de escritura de un determinado usuario.

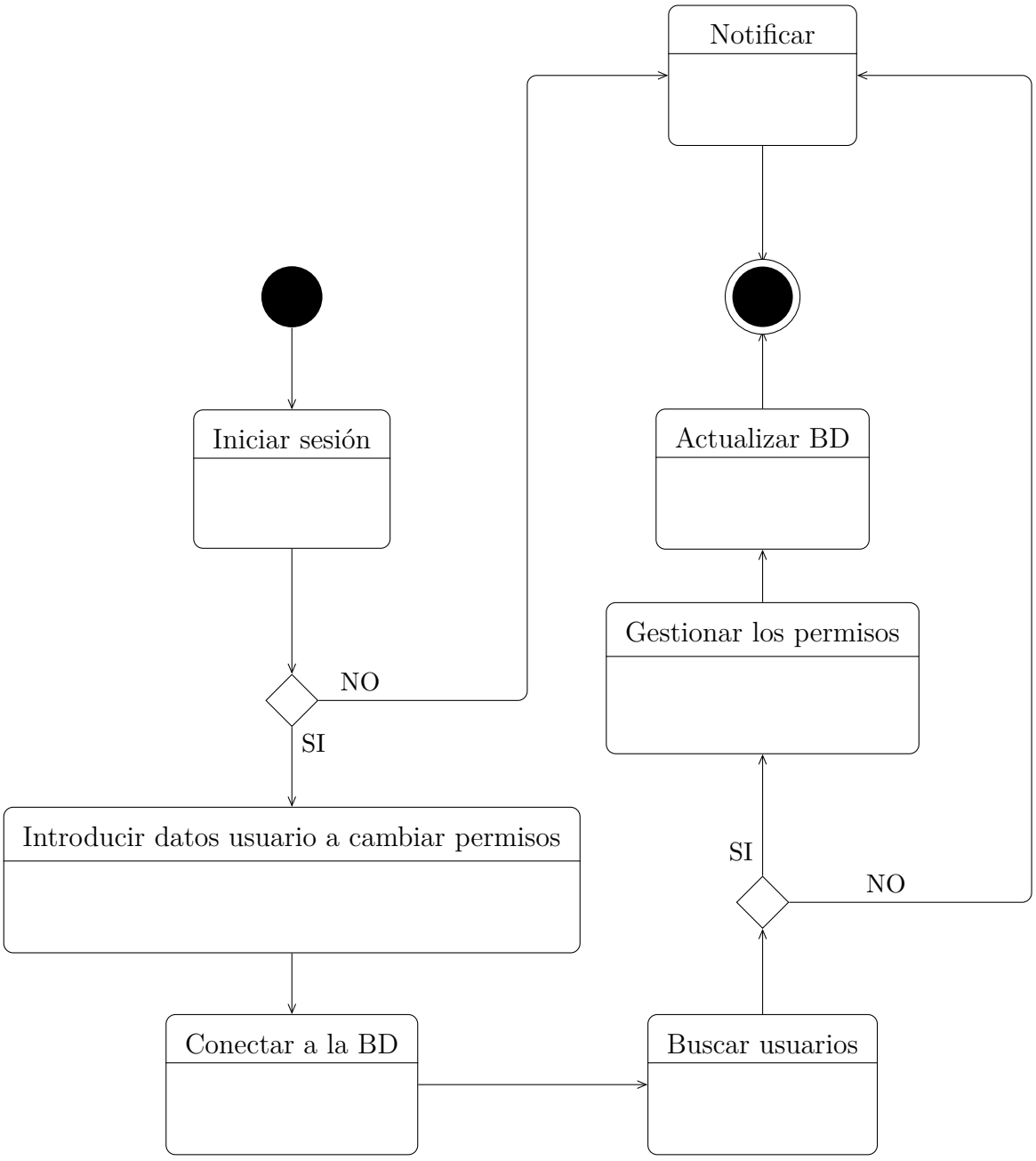


Figura 15: Diagrama de actividad de Gestionar permisos

Configurar pizarra: (*Figura 16*) este caso de uso sirve para configurar la pizarra, cambiando los distintos parámetros.

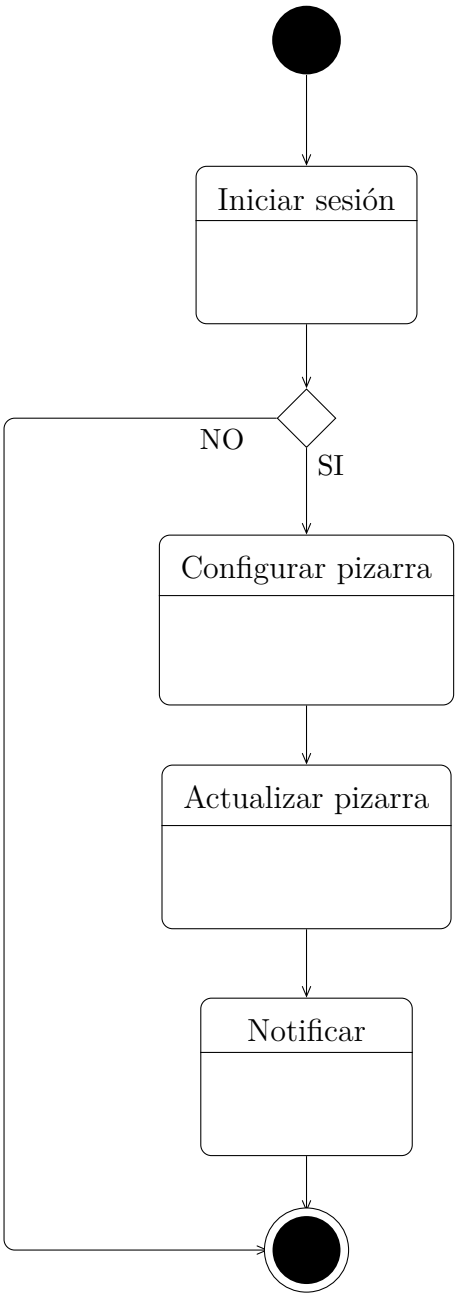


Figura 16: Diagrama de actividad de Configurar pizarra

Crear usuario: (*Figura 17*) este caso de uso está restringido sólo al administrador. Crea un nuevo usuario en la pizarra.

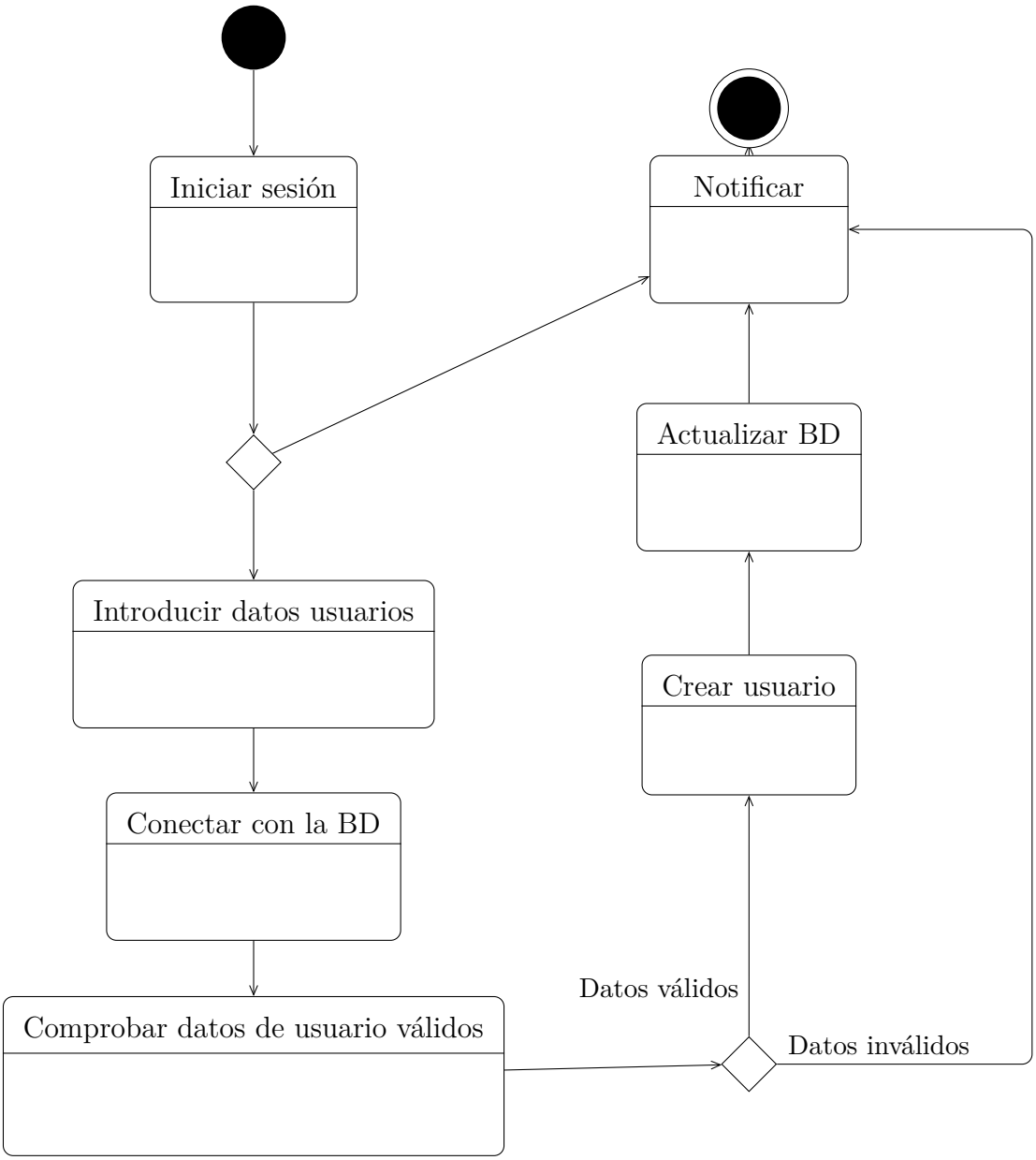


Figura 17: Diagrama de actividad de Crear usuario

3.2.3. Diagrama de clases

Un diagrama de clases es un diagrama estático destinado a la programación orientada a objetos que permite describir las clases de un sistema, así como sus propiedades, operaciones, relaciones entre ellas y herencia.

Se suelen hacer varias versiones de estos diagramas, generalmente dos, una para la parte de análisis que abordaremos ahora y otra para la parte de diseño que describiremos más adelante.

Clases de la plataforma

A continuación detallamos cada una de las clases que aparecen en el diagrama (*figura 18*) centrándose en sus propiedades y las relaciones entre ellas.

Pizarra: El diagrama de clases se centra en esta clase. Almacena los datos relacionados con el estado de la pizarra, así como la lista de usuarios, las estadísticas, los permisos y la configuración.

Agente: Permite interactuar con la pizarra y hace las veces de interfaz al usuario para usar la pizarra. Contiene el nombre de usuario y la contraseña con la que se interactúa con la pizarra.

Estado: Contiene la lista de elementos.

Elemento: Puede ser de dos tipos, representado como herencia. Un archivo o una carpeta. Una carpeta contendrá a su vez un listado de elementos.

Nivel: Proporciona las operaciones necesarias para comprobar y editar los permisos de la pizarra.

Configuración: Permite la visualización y modificación de las configuraciones de la pizarra.

Estadísticas: Permite visualizar las estadísticas.

Usuario: Contiene el nombre, el id, la contraseña y los permisos del usuario.

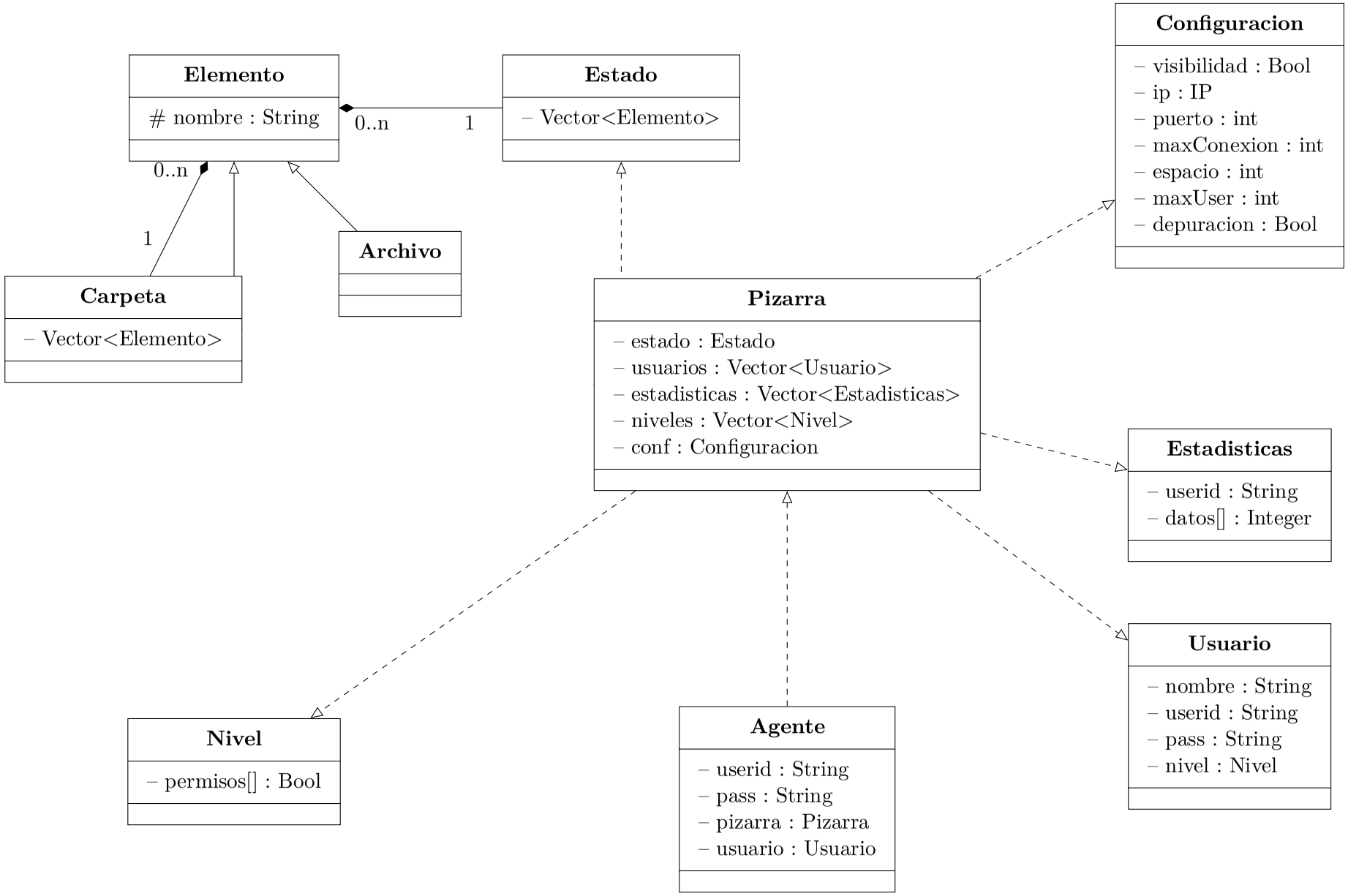


Figura 18: Diagrama de clases

3.3. Diseño final

En este apartado nos centramos en un diseño más profundo, destinado a ser la documentación que se proporcionaría a los encargados de la implementación de la plataforma.

Para comenzar, vamos a completar el diagrama de clases añadiéndole los métodos necesarios para su correcto funcionamiento y después realizaremos una serie de diagramas de secuencia, sólo correspondientes a la parte de *Agente* debido a que es lo que se nos pide en esta práctica.

3.3.1. Diagrama de clases

Completamos el diagrama anterior (*Figura 18*) añadiendole los métodos correspondientes, obteniendo como resultado la *figura 19*.

Ahora vamos a analizar la funcionalidad de cada clase:
Pizarra: Contiene las operaciones necesarias para que los agentes puedan interactuar con ella.

Agente: Permite interactuar con la pizarra, dando opciones para leer o escribir en la misma, así como modificar su configuración o permisos.

Estado: Permite actualizar su contenido agregando, modificando o eliminando archivos.

Nivel: Proporciona las operaciones necesarias para comprobar y editar los permisos de la pizarra.

Configuración: Permite la visualización y modificación de las configuraciones de la pizarra.

Estadísticas: Permite visualizar las estadísticas.

Usuario: Permite la modificación y visualización de los datos del usuario.

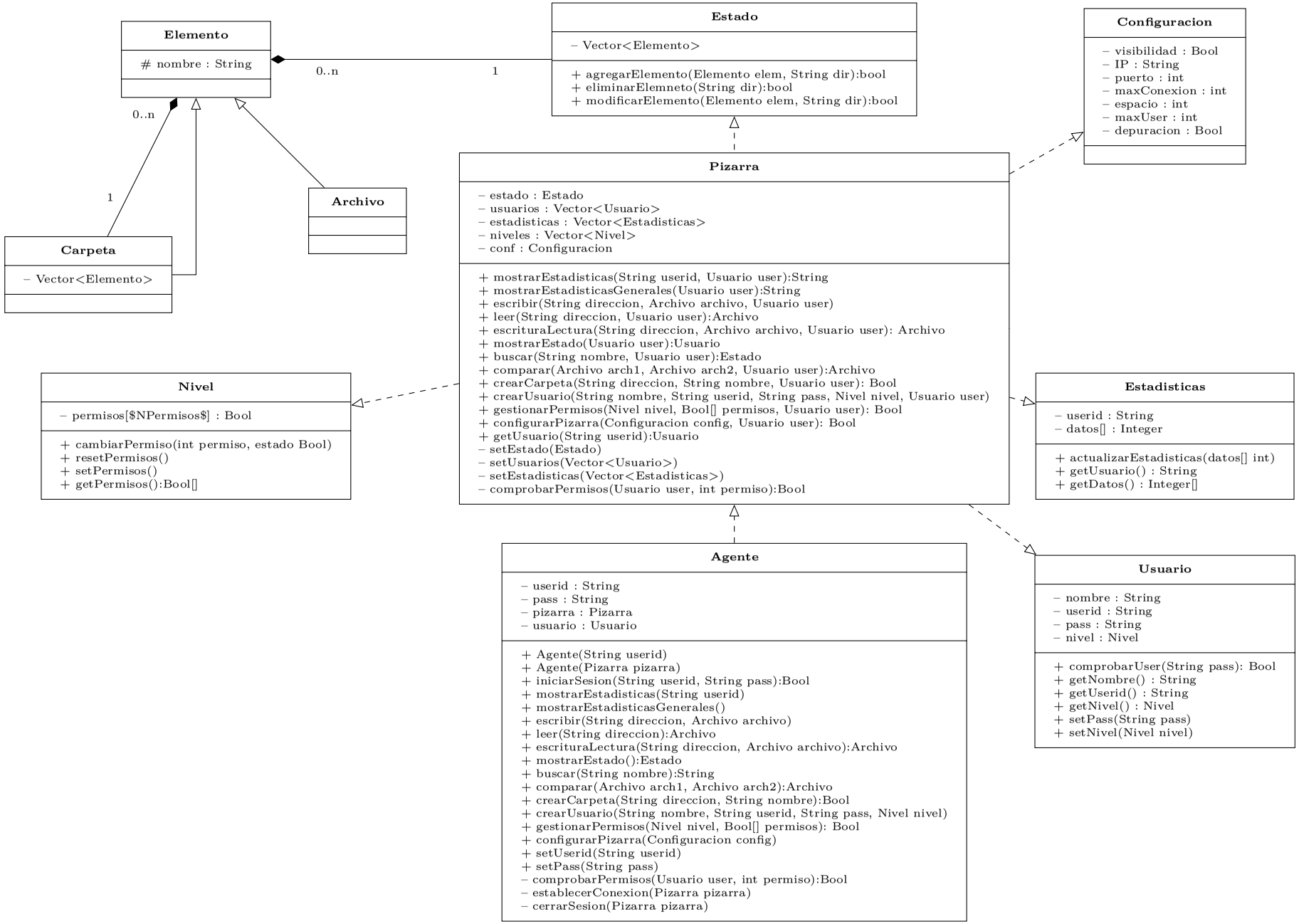


Figura 19: Diagrama de clases

3.3.2. Diagramas de secuencia

El diagrama de secuencia es un diagrama UML utilizado para modelar la interacción entre objetos. Este diagrama se modela para cada caso de uso. Consta de dos tipos de mensajes:

1. **Síncrono:** Corresponden con llamadas a métodos. Se representan con flechas rellenas en negro.
2. **Asíncrono:** Terminan inmediatamente y crean un nuevo hilo de ejecución dentro de la secuencia. Se representan con flechas sin rellenar.

Los diagramas de secuencia correspondientes a los casos de uso de nuestra plataforma se ponen a continuación.

Actualizar estado: actualiza el estado actual de la pizarra. (*Figura 20*)

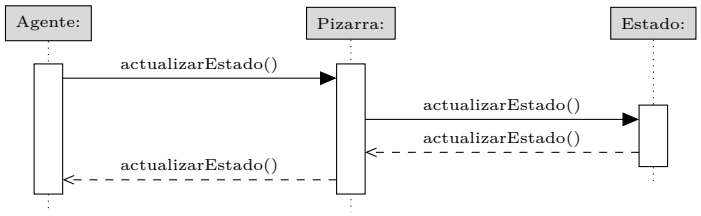


Figura 20: Diagrama de secuencia de Actualizar estado

Comprobar Nivel: Comprueba los permisos del usuario. (*Figura 21*)

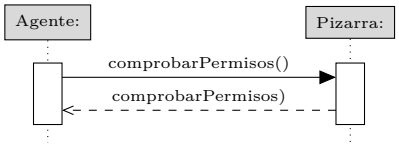


Figura 21: Diagrama de secuencia de comprobar nivel

Crear Usuario: Crea un nuevo usuario en la BD de usuarios. (*Figura 22*)

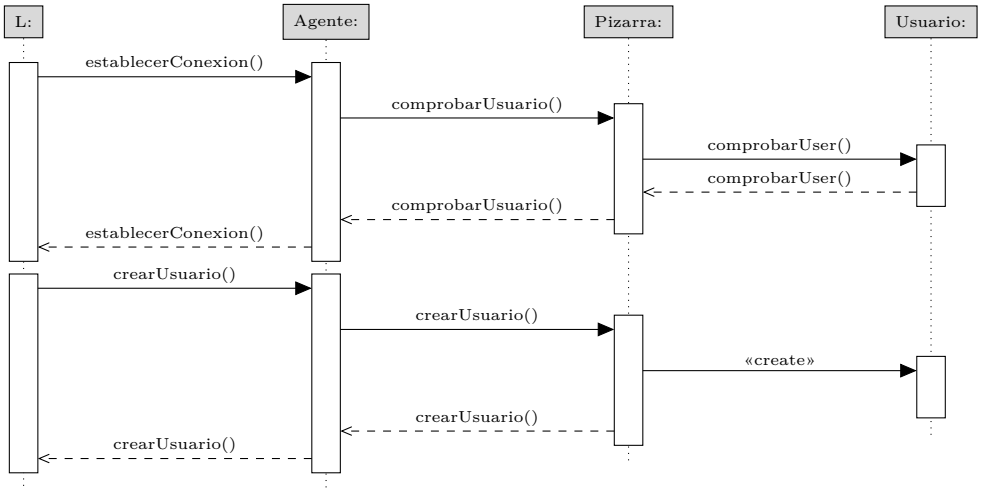


Figura 22: Diagrama de secuencia de crear usuario

Iniciar sesión: Comprueba el usuario en la BD comprobando los permisos de dicho usuario. (*Figura 23*)

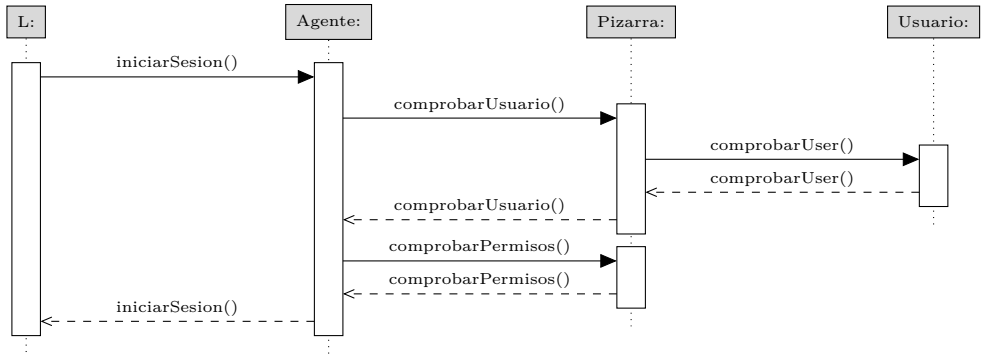


Figura 23: Diagrama de secuencia de iniciar sesión

Mostrar estadísticas: Muestra las estadísticas de la actividad del usuario en la pizarra. (*Figura 24*)

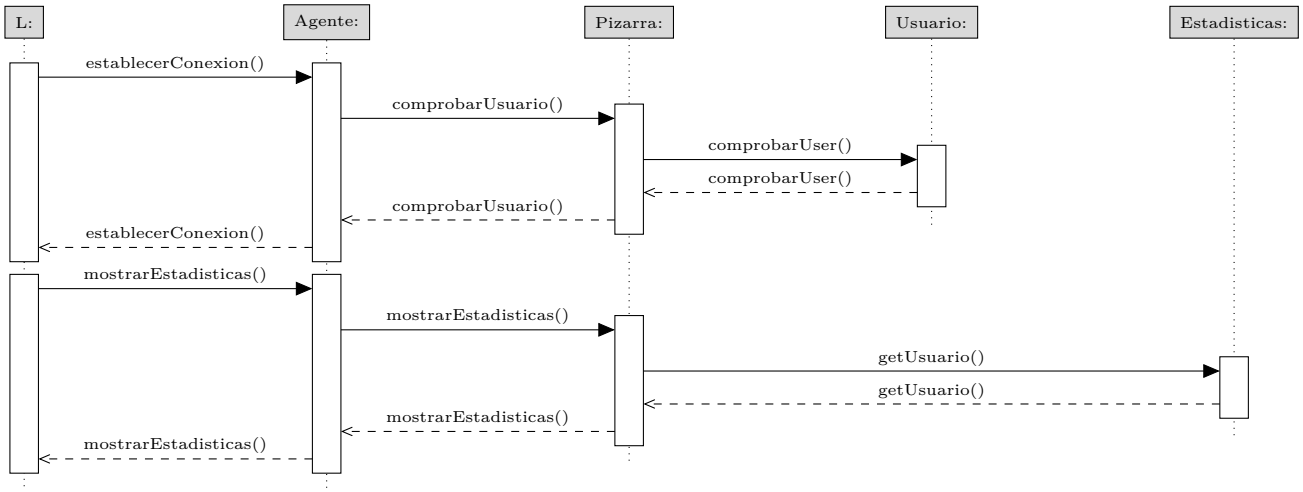


Figura 24: Diagrama de secuencia de mostrar estadísticas

Mostrar estadísticas generales: Muestra las estadísticas generales de la actividad de todos usuarios en la pizarra(*Figura 25*)

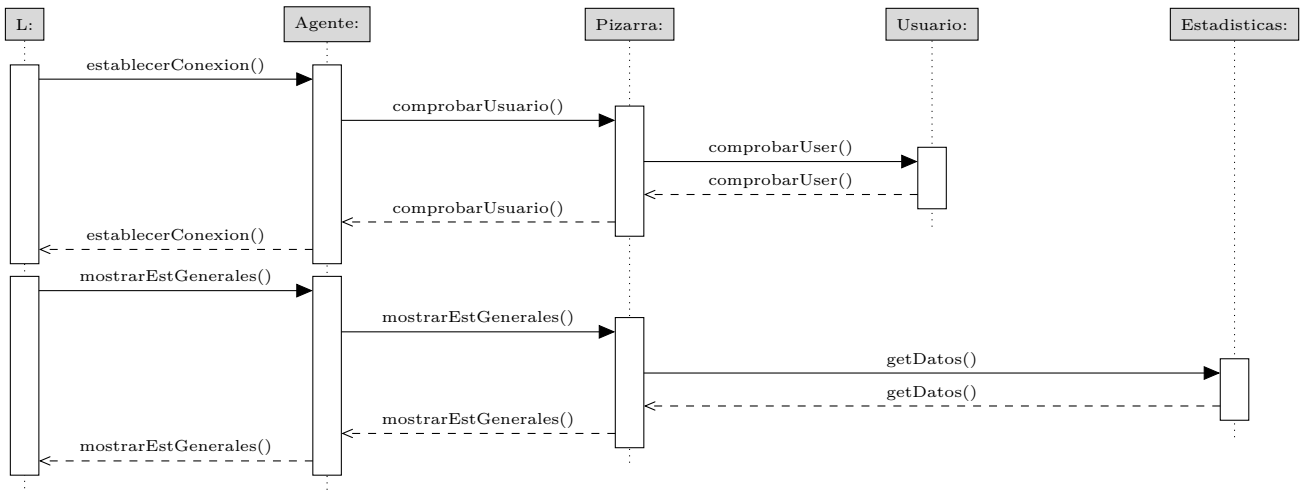


Figura 25: Diagrama de secuencia de mostrar estadísticas generales

Mostrar estado: Muestra el estado actual de la pizarra.(*Figura 26*)

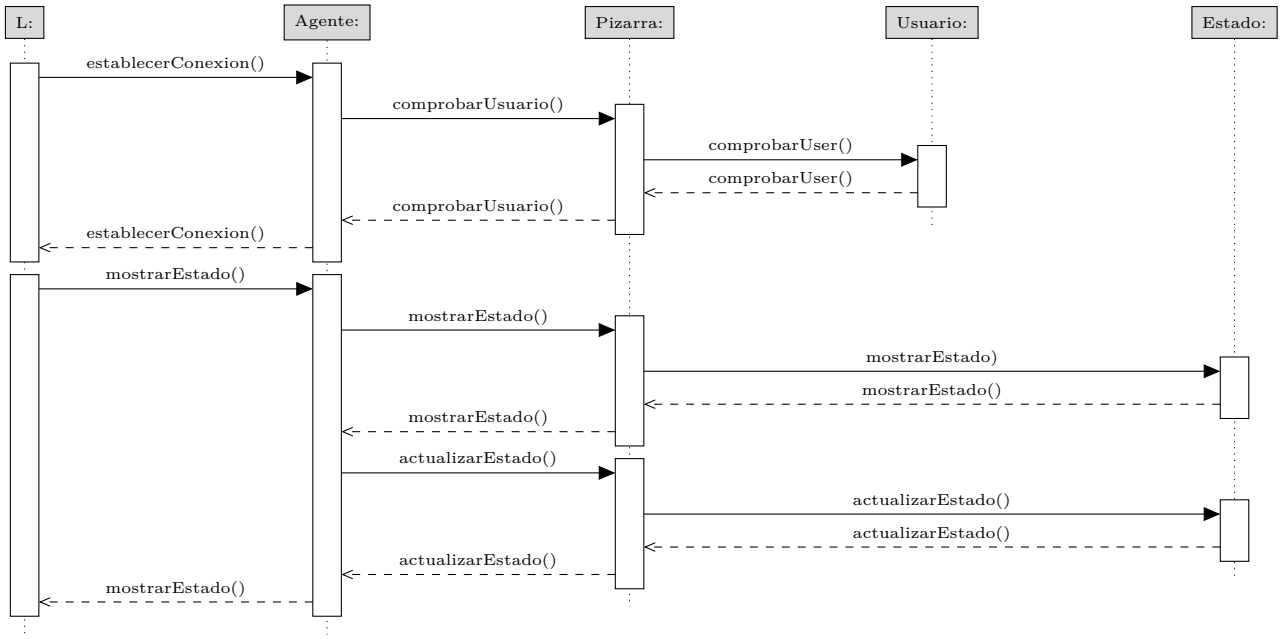


Figura 26: Diagrama de secuencia de mostrar estado

Buscar: Busca un archivo en el estado de la pizarra.(*Figura 27*)
Comparar: Compara dos archivos que se encuentran en el estado de la pizarra.(*Figura 28*)
Configurar pizarra: Cambia la configuración de la pizarra. (*Figura 29*)
Crear Carpeta: Crea una nueva carpeta en el estado de la pizarra. (*Figura 30*)
Escribir: Sobreescribe el estado de la pizarra. (*Figura 31*)
Lectura/Escritura: Escribe en el estado de la pizarra sin sobreescribir. (*Figura 32*)
Leer: Lee el estado de la pizarra(*Figura 33*)
Gestionar permisos: Modifica los permisos de un usuario.(*Figura 34*)

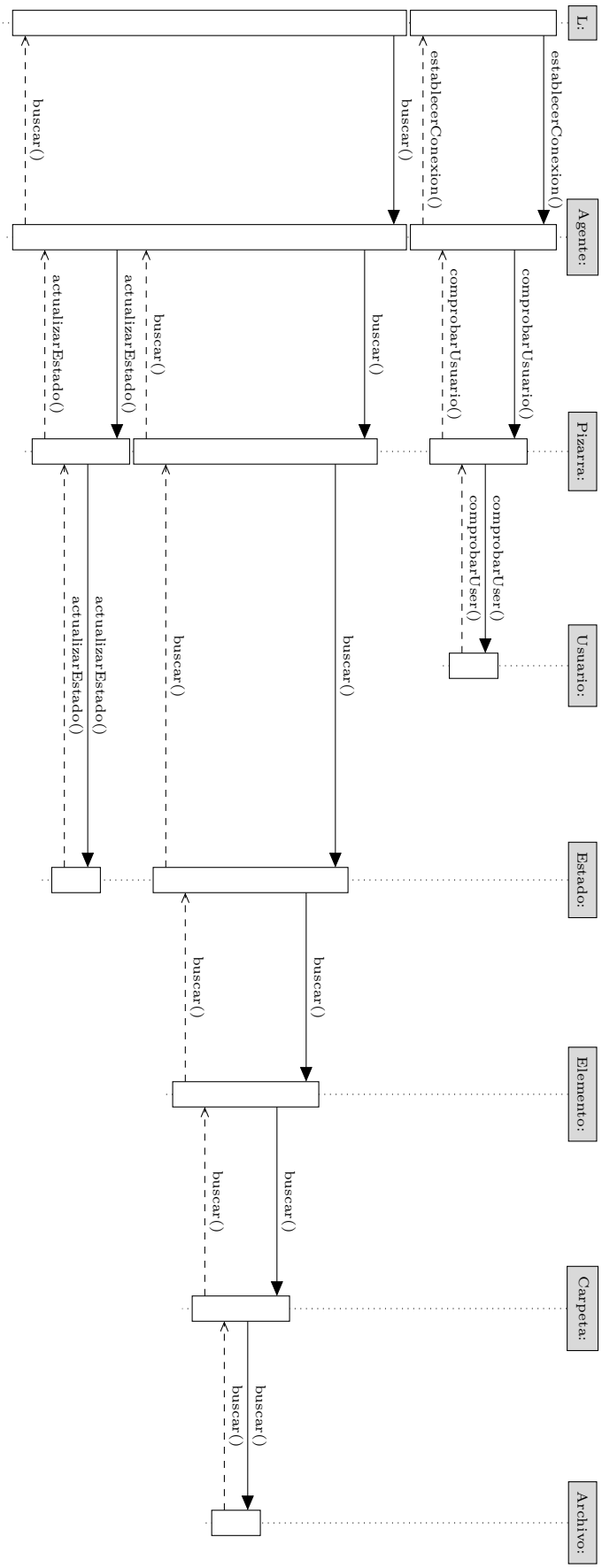


Figura 27: Diagrama de secuencia de buscar

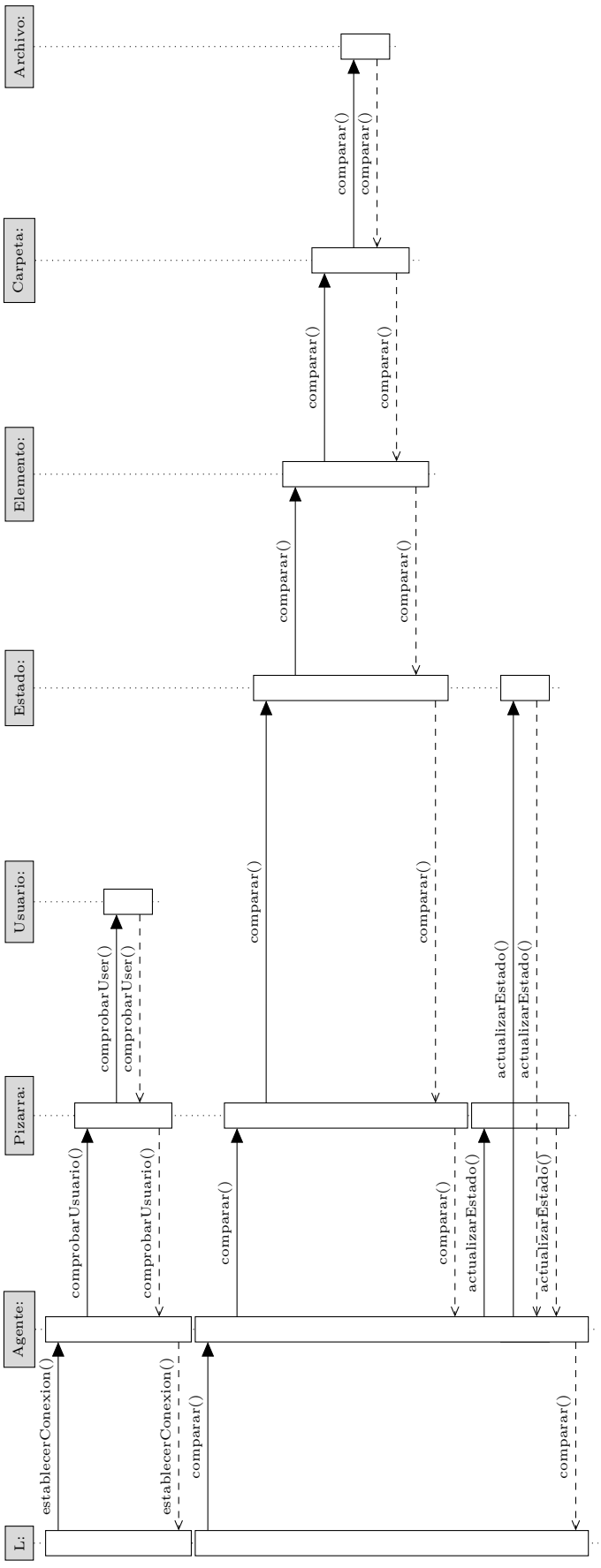


Figura 28: Diagrama de secuencia de comparar

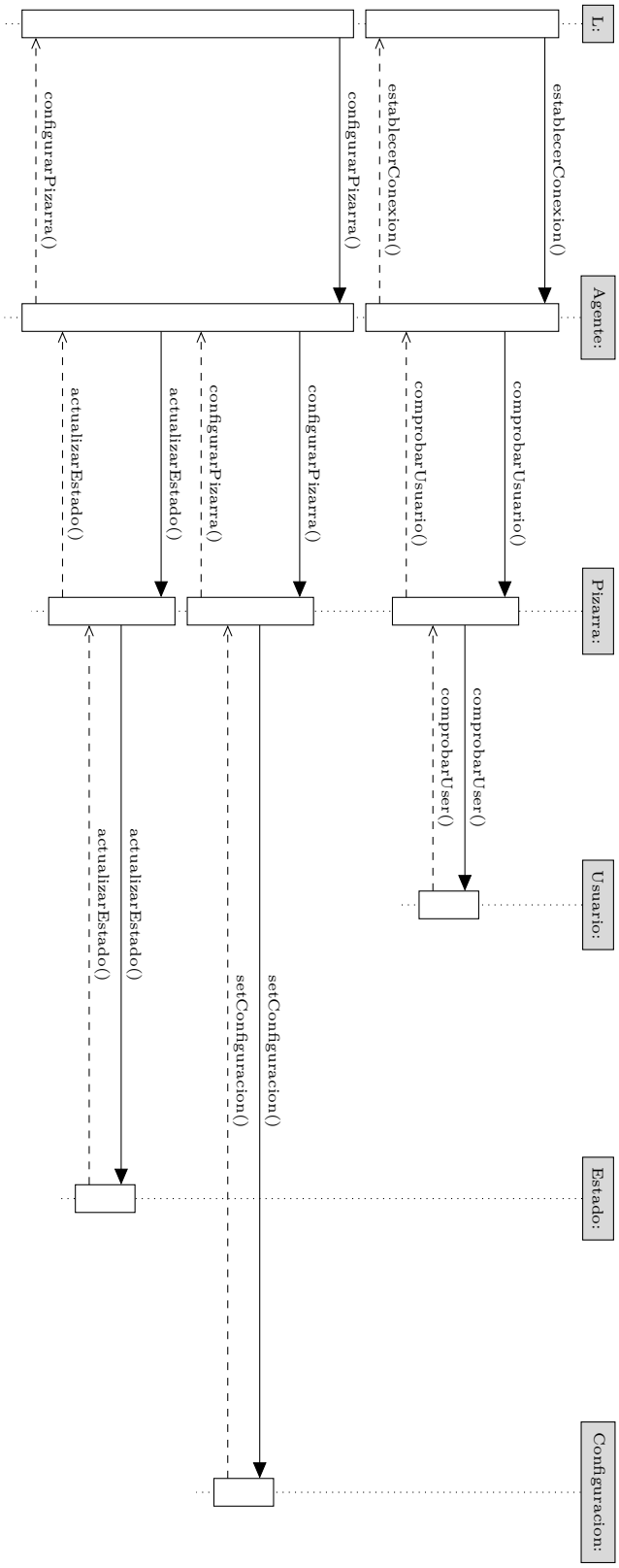


Figura 29: Diagrama de secuencia de configurar pizarra

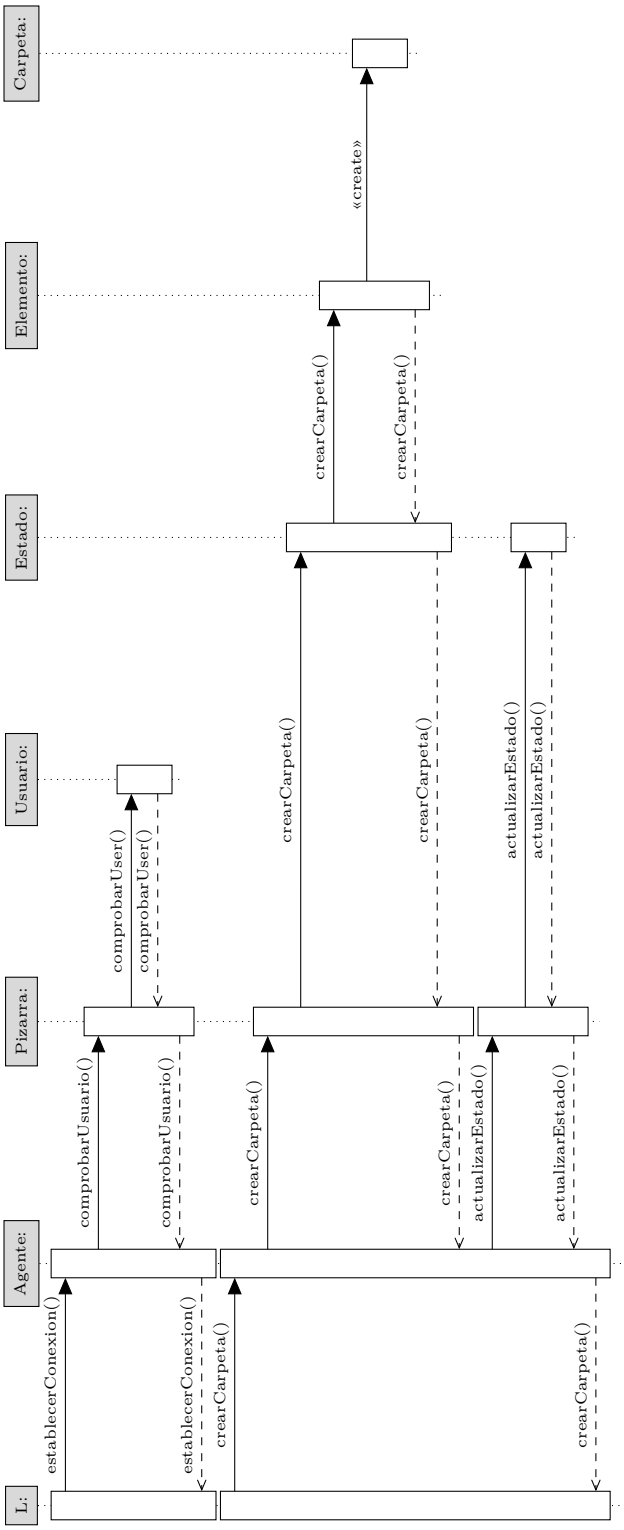


Figura 30: Diagrama de secuencia de crear carpeta

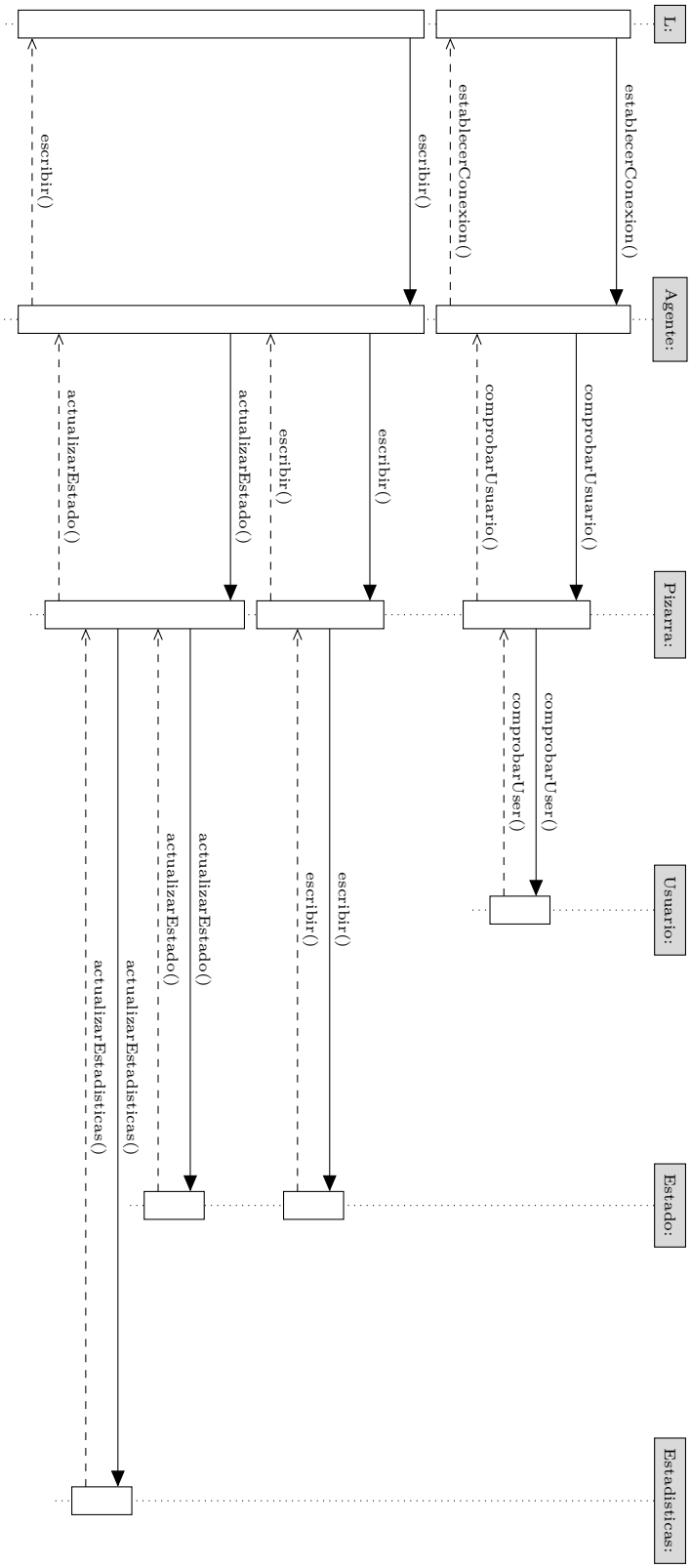


Figura 31: Diagrama de secuencia de escribir

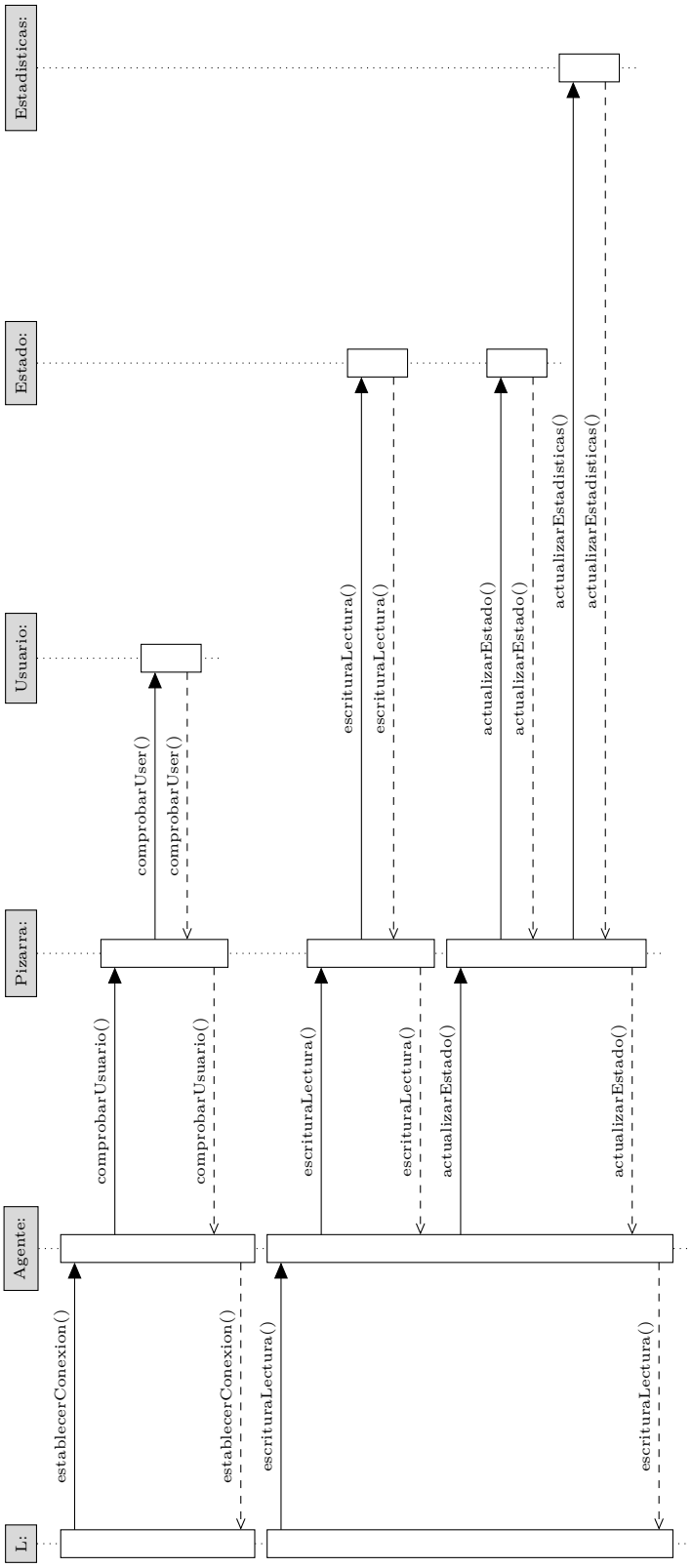


Figura 32: Diagrama de secuencia de lectura/escritura

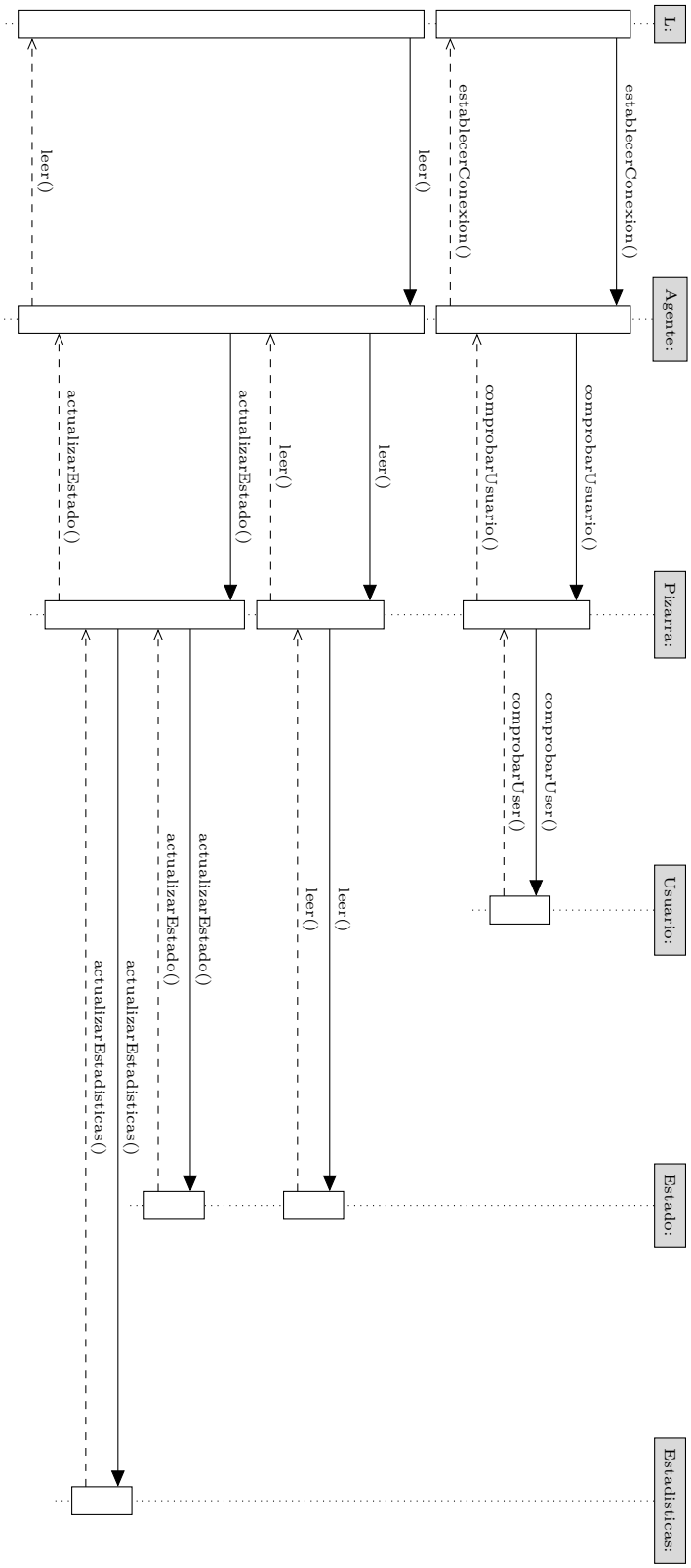


Figura 33: Diagrama de secuencia de leer

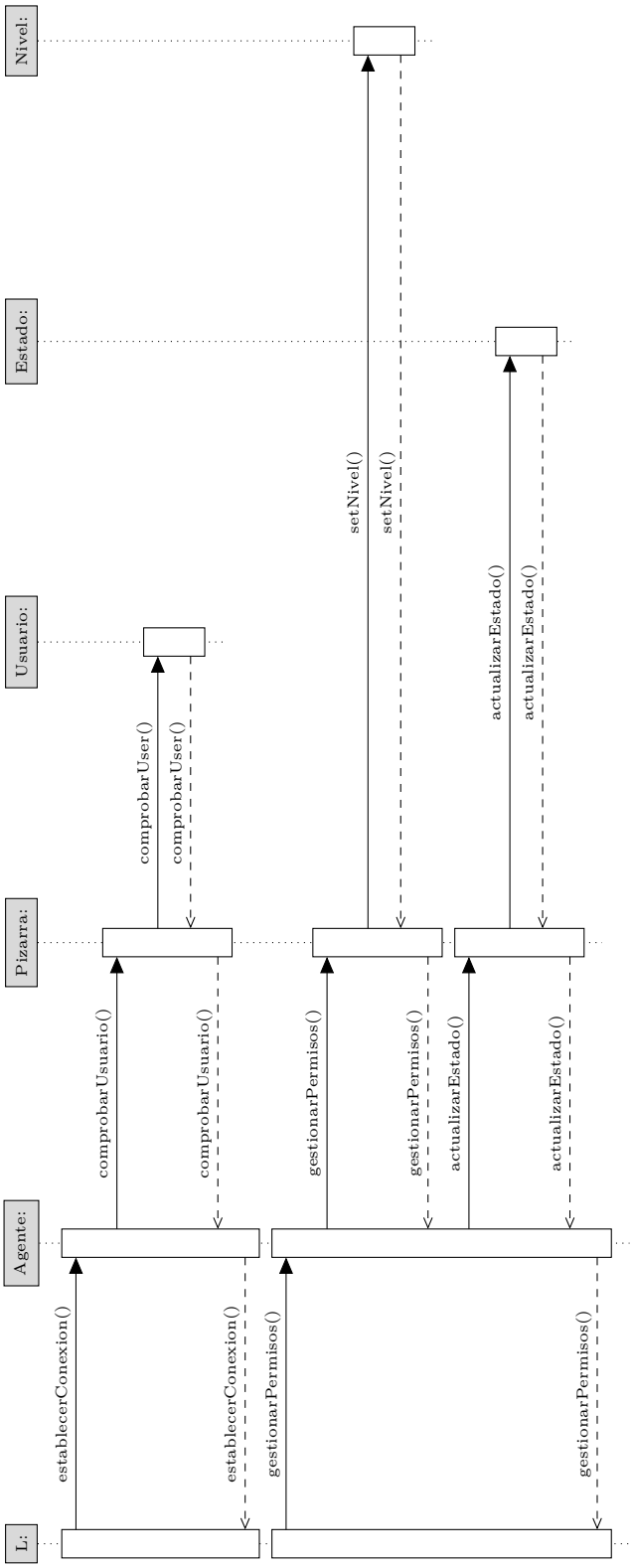


Figura 34: Diagrama de secuencia de gestionar permisos

Capítulo 4

Manual de uso de la librería

Una vez especificados los requisitos de nuestra pizarra, además de los UML necesarios que describen su funcionamiento, incorporamos un breve manual de usuario con el que un programador ajeno podría construir fácilmente una aplicación que implemente un agente de nuestra plataforma.

4.1. Consideraciones previas

Se considerará que nuestro PC cumple con todos los requisitos necesarios para que tanto nuestra librería como nuestra aplicación puedan funcionar sin problemas. Ello conlleva que se disponga de conexión a la red y tenga instalado algún compilador de C++, preferiblemente algún entorno de desarrollo como Qt[8]. En definitiva, que se pueda compilar una aplicación que incluya la plataforma.

4.2. Primeros pasos

En primer lugar se detallarán los primeros pasos típicos que realizará nuestra aplicación.

Primeramente será necesario conectarnos. Se considerará que el registro (donde se especifican tanto el nombre de usuario como la contraseña) se ha realizado anterior y correctamente, por ejemplo en nuestra aplicación).

4.2.1. Tipos de usuarios

Antes de comenzar es necesario recalcar que existen dos tipos de usuarios: administradores y normales. Los primeros cuentan con más opciones a la hora de organizar la pizarra, tal y como se explicará más adelante, tanto para administradores como para usuarios corrientes será iniciar sesión.

4.2.2. Conectarse a la pizarra, Iniciar sesión

Ahora se proporciona un ejemplo básico para crear un objeto agente que se conecte a la pizarra:

```
#include <pizarra.h>
#include <iostream.h>

using namespace std;

Agente app = new Agente(loginuser, passuser, new Pizarra(IP, puerto, nombre));
//Creamos un agente con los parametros que hemos introducido por teclado

Usuario user = app.iniciarSesion(app.getUser(), pass);
//Llama al metodo que nos permitira iniciar sesion

if(user != null){
// Si el usuario ha introducido sus datos correctamente
// y estos se encuentran en la base de datos

}else{
// No ha encontrado el usuario o la contrasena incorrecta, y
// se mostrara el pertinente mensaje

    Pizarra.existe(user) ? cout << "Invalid Password":
    cout << "Invalid user name";
}
```

Nota: En el ejemplo anterior se incluyen las credenciales del usuario en el código por simplificar, deben ser tratados con cuidado estos parámetros, la plataforma no se hará cargo si se pierden o son robados por culpa de una mala implementación.
A continuación se hará una breve descripción de las funcionalidades de las que dispone un agente:

4.3. Funcionalidades y ejemplos

Ahora vamos a analizar las distintas funcionalidades que se proporciona con el agente de la plataforma.

4.3.1. Funcionalidades básica

Aquí encontraras una lista con ejemplos de la funcionalidad más básica de la plataforma.

Iniciar sesión

Como ya se ha explicado en el apartado de primeros pasos (Véase apartado 4.2) , este método será invocado constantemente por parte de los usuarios de nuestra aplicación, y su

funcionamiento es el siguiente:

```
Usuario user = app.iniciarSesion(app.getUser(), pass);
```

En primer lugar el usuario deberá introducir su nombre de usuario y contraseña formados por sendos string y si los datos con correctos, el método devolverá el usuario con sus correspondientes permisos, lo que permitirá al agente comprobar que puede y que no puede hacer.

Mostrar estadísticas

Esta funcionalidad permite saber las estadísticas a nivel de usuario, es decir, el número de archivos editados, datos borrados etc.
Para ello será necesario que el agente introduzca su nombre de usuario, que mediante:

```
Estadisticas stats = mostrarEstadisticas(String userid);
```

Se podrán consultar los cambios realizados por cada usuario, con los modos de la clase estadísticas.

Mostrar estadísticas generales

Funciona de un modo muy parecido a mostrar estadísticas, con la diferencia de que ahora podremos consultar las estadísticas a nivel global, de todos los usuarios como un solo grupo de trabajo. Esto permitirá consultar el número de archivos subidos al repositorio, archivos borrados, etc pero todo desde un punto de vista grupal.

```
Estadisticas stats = mostrarEstadisticasGenerales();
```

Escribir

Es una operación básica de la pizarra, junto a leer y lectura/escritura. Esta operación permite tanto añadir, no permite sobrescribir archivos de la pizarra. Esta operación está limitada por los permisos del usuario que quiera escribir (salvo el usuario Admin).

```
app.escribir(". \problemas\tema1", Archivo ejemplo);)
```

Para poder escribir será necesario que el agente especifique el archivo que se quiere escrbir (*".\problemas\tema1\ejemplo.getNombre()" en nuestro caso*).

Debido a la importancia de esta funcionalidad se adjunta un pequeño código de ejemplo (consideramos que ya ha iniciado sesión) :

```
. . .

if(this.user.getPermiso() == 0){
```

```
// El usuario dispone de los permisos necesarios

app.escribir(''.\problemas\tema1'', Archivo ejemplo)


}else{
// No tiene los permisos y salta un mensaje de advertencia

    Pizarra.existe(user) ? cout << "No dispone de los permisos necesarios":
}

. . .
```

Lectura/Escritura

Lectura/Escritura funciona de manera análoga a escribir, salvo que en esta operación si soporta la sobrescritura.

```
app.lecturaEscritura(".\problemas\tema1", Archivo ejemplo);)
```

Por lo tanto no escribirá si el archivo ya existe.

Leer

Esta operación es análoga a escribir, salvo que en lugar de escribir se leen archivos.

```
Archivo ejemplo = app.lectura(".\problema \tema1");
```

Es altamente recomendable utilizar conjuntamente los métodos *leer* y *lecturaEscritura* para poder editar archivos, cómo por ejemplo así:

```
. . .

if(this.user.getPermiso() == 1 && this.user.getPermiso() == 2){
// El usuario dispone de los permisos necesarios

Archivo ejemplo = app.leer(''.\problemas\tema1\ej1.c'')

\\Se edita el archivo ejemplo
    . . .
\\Se edita el archivo ejemplo

app.lecturaEscritura(''.\problemas\tema1'', ejemplo);

}else{
// No tiene los permisos y salta un mensaje de advertencia
```

```
    Pizarra.existe(user) ? cout << "No dispone de los permisos necesarios
}

. . .
```

Mostrar estado

Permite mostrar el estado actual de la pizarra, es decir, el número de usuarios totales, el número de archivos en la pizarra, el historial de todas las modificaciones, etc.

```
app.mostrarEstado();
```

Buscar

Permite buscar archivos en la pizarra. Sólo se podrá especificar el nombre del archivo o carpeta a buscar. El agente debe introducir un String con el nombre del archivo deseado y se facilitará una lista con los archivos encontrados, en caso de que los hubiera.

```
app.buscar("ejercicio1");
```

Comparar

Permite comparar dos archivos, pudiendo por tanto apreciar las diferencias existentes en aspectos como tamaño, formato o líneas modificadas.

Para ello será necesario que el agente introduzca el archivo en comparar(Archivo archivo), y al igual que en otras funcionalidades se guardará el usuario que ha realizado dicha comparación. Finalmente se devuelve un archivo donde se indican las diferencias.

```
Archivo diff = app.comparar(ejercicio1, solucion1);
```

Crear Carpeta

Permite crear una carpeta dentro de la pizarra. Para ello será necesario introducir la ruta y nombre de la carpeta que deseamos crear (debe tener un nombre único).

```
app.crearCarpeta(".\ejercicios\tema1\soluciones");
```

Este método, al igual que todos requiere una comprobación de los permisos.

Crear Usuario

Permite crear nuevos agentes en la pizarra. Esta función solo puede ser usada por el Administrador.

```
app.crearUsuario(new Usuario(nombre, userid, pass, niveluser));
```

Para ello se deberán especificar, como bien se muestra arriba, el nuevo nombre de usuario, contraseña, y nivel que dispondrá el nuevo usuario. Finalmente se comprobará que el nombre de usuario deseado no esté ya en uso, en cuyo caso se deberá cambiar.

Configurar pizarra

Mediante esta función podemos configurar algunos aspectos de nuestra pizarra, como podrían ser asuntos de conexión, máximo de conexiones simultáneas, número máximo de usuarios permitidos y algunas opciones de depuración.

```
app.configurarPizarra(new Configuracion(datos[]));
```

4.3.2. Gestionar permisos

Una explicación más detallada precisan los permisos, que pueden tener los distintos usuarios. En nuestra aplicación los usuarios podrán tener los 7 permisos básicos típicos, tal y como se explica en la siguiente tabla:

Número	Permiso	Descripción
0	Leer	El usuario puede obtener información de la pizarra
1	Escribir	El usuario puede añadir datos a la pizarra
2	Lectura Escritura	El usuario puede sobrescribir datos en la pizarra
3	Buscar	El usuario puede buscar datos en la pizarra
4	Comparar	El usuario puede comparar archivos de la pizarra
5	Buscar	El usuario puede buscar archivos en la pizarra
6	Crear Carpeta	El usuario puede crear carpetas en la pizarra
7	Admin	El usuario que posee este permiso se comporta como admin.

[1]

Dichos permisos sólo pueden ser modificados por el usuario Administrador mediante:

```
app.gestionarPermisos(usuario);
```

[1] Sólo un usuario puede poseer los permisos de admin

Capítulo 5

Conclusiones

Durante todo el proceso, tanto de documentación de los requisitos como en el de análisis y el de diseño, hemos experimentado una extraña sensación, cada vez conocíamos mejor la plataforma, cada vez la entendíamos mucho mejor, hasta tal punto que prácticamente hemos sido capaces de escribir pequeñas lineas de código destinadas a la utilización de la misma.

Esto es debido a la gran cantidad de tiempo que le hemos dedicado y en lo que se lo hemos dedicado; centrándonos la mayor parte del tiempo en el diseño, en la creación de los distintos diagramas que sin duda alguna nos ha permitido conocer este producto.

Todos coincidimos en que por las fechas tan poco adecuadas, en Navidad, el desarrollo de la memoria y de la práctica se ha visto truncado. Si hubiéramos tenido los mismos días para hacerla, pero en enero, seguramente nos hubiera dado tiempo a implementar completamente la plataforma e incluso a probar alguna aplicación que la implementase.

En cualquiera de los casos, se ha tratado de un proceso interesante cuanto menos para nosotros, hemos descubierto las propiedades del uso de arquitecturas de una manera más práctica y hemos realizado un pseudo[1] modelo de cascada por el cuál hemos sido partícipes por primera vez de un proyecto en el que se utilicen este tipo de metodologías.

En conclusión, podemos resumir esta práctica como única y extensa, con respecto a esto último no estamos en desacuerdo, nos parece correcto hacer trabajos de este tipo de envergadura, pero no nos agrada que haya tenido que ser durante el período de Navidad, en el que pasamos gran parte de tiempo con la familia, que nos ha impedido dedicarle tiempo a la práctica.

[1]En realidad no hemos trabajado mediante un modelo de cascada real, más bien lo hemos simulado, puesto que algunos diagramas se hacían mientras se obtenían todavía requisitos

Glosario

A

Association for Computing Machinery (ACM) Primera sociedad científica y educativa acerca de la Computación, p. 5.

C

C++ Lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup., p. 30.

Central Unit Process (CPU) Unidad de procesos de un computador, es el elemento encargado de realizar las operaciones aritmeticológicas necesarias para el correcto funcionamiento del mismo, p. 9.

H

Hard Disk Drive (HDD) Disco Duro, p. 3.

Hypervisor Plataforma que permite aplicar diversas técnicas de control de virtualización, p. 7.

I

Instruction Set Architecture (ISA) Especificación que detalla las instrucciones que una CPU de un ordenador puede entender y ejecutar, su repertorio de instrucciones, p. 5.

J

Java Virtual Machine (JVM) Máquina virtual de la plataforma Java, p. 4.

K

Kernel Traducción al español: Núcleo; En informática se refiere al elemento central de un programa, generalmente un Sistema Operativo, p. 10.

L

LaTeX sistema de composición de textos, orientado especialmente a la creación de libros, documentos científicos y técnicos que contengan fórmulas matemáticas., p. ix.

M

Máquina virtual (VM) Software que simula a una computadora y puede ejecutar programas como si fuese una computadora real., p. 4.

O

Organizador (Scheduler) Tan bien conocido como planificador, es el elemneto encargado de gestionar los recursos reales que posee para proporcionarselos a la máquina virtual, p. 11.

R

RAM Memoria de acceso aleatorio, se utiliza como memoria de trabajo para el sistema operativo, los programas y la mayoría del software., p. 9.

U

UML Lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad., p. 27.

V

Virtual Machine Monitor (VMM) Monitor de la Máquina Virtual, p. 3.

X

x64 Conjunto de instrucciones utilizada en la microarquitectura de CPU de 64 bits., p. 16.

x86 Conjunto de instrucciones utilizada en la microarquitectura de CPU de 32 bits., p. 16.

Xen Hipervisor de código abierto desarrollado por la Universidad de Cambridge, p. 8.

Índice de figuras

Bibliografía

[1] Guía de instalación de un sistema operativo huésped para VMware: Workstation. partnerweb.vmware.com/GOSIG/home.html.

[2] Guía de instalación de VMware Workstation 5.5. https://www.vmware.com/support/ws55/doc/ws_newguest_setup_simple_steps.html.

[3] Guía de instalación de windows xp en unaMáquina virtual VMware. https://www.vmware.com/support/ws5/doc/new_guest_example_ws.html.

[4] Guía de Instalación y Configuración VMware Workstation. http://cadascu.wordpress.com/2012/10/16/2272/.

[5] Issue de discusión del repositorio de la práctica sobre las posibles aplicaciones y requisitos, howpublished=https://github.com/kekoalk/practica_das/issues/14,.

[6] Página de la Universidad de Cambridge del proyectoXenoserver. http://www.cl.cam.ac.uk/research/srg/netos/xeno/.

[7] Página de soporte oficial de VMware sobre VMware: Workstation. http://www.vmware.com/es/support/workstation.html.

[8] Página oficial de qt. http://qt.digia.com.

[9] Página oficial de VMware. http://www.vmware.com/es/.

[10] Página oficial de VMware sobre VMware: Workstation. http://www.vmware.com/es/products/workstation/.

[11] Página oficial del control de versiones git. http://git-scm.com/.

[12] Página oficial del proyecto Xen. http://www.xenproject.org/about/history.html.

[13] Repositorio de trabajo para el desarrollo de la memoria. https://github.com/KekoAlk/Practica_DAS.

[14] Requisitos de instalación de VMware Workstation v.5. https://www.vmware.com/support/ws5/doc/intro_hostreq_ws.html.

[15] Popek Gerald J and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412 – 421, 1974.

[16] M. Tim Jones. *«La anatomía de un hipervisor Linux»*. http://www.ibm.com/developerworks/ssa/library/l-hypervisor/index.html.

[17] Wikipedia. Página sobre el concepto de virtualización. http://es.wikipedia.org/wiki/Virtualizacion.

[18] Wikipedia. Página sobre las máquinas virtuales. http://es.wikipedia.org/wiki/Maquina_virtual.