Reingeniería del Sistema de Gestión de Menús y Permisos de la Plataforma Comunidad Académica (COMA)

Nestor Javier Clavijo Hernández, Jesus David Ramirez Celis

Trabajo de Grado para Optar al Título de Ingeniero de Sistemas

Director

Luis Ignacio Gonzalez Ramírez

Magíster en Informática

Universidad Industrial de Santander

Facultad de Ingenierías FisicoMecánicas

Escuela de Sistemas e Informática

Ingeniería de Sistemas

Bucaramanga

2026

Dedicatoria

Primero que todo, quiero agradecerle a Dios por darme la fortaleza y la sabiduría necesarias para recorrer este camino.

*A mis padres, **Medardo Clavijo y Margarita Hernández**, pilares fundamentales en mi desarrollo personal y en mi ingreso a la universidad. Sin ellos, este logro no habría sido posible, pues fueron el apoyo emocional que nunca me desamparó y que me sostuvo tanto en los momentos altos como en los más difíciles.*

*A mis hermanos, **José Clavijo y Mónica Clavijo**, que han sido como unos segundos padres para mí. Me brindaron una ayuda que solo ellos podían darme, un respaldo no solo económico sino también emocional. Muchas veces me entendieron más que nadie y, con su sabiduría y experiencia, me ayudaron a sobrellevar cada situación en todos sus aspectos.*

*Un agradecimiento muy especial a **Juan David Lipez Guevara**, quien no solo fue un amigo, sino que actuó como mentor, guía, maestro y compañero. Asumió muchas responsabilidades que, aunque no le correspondían, realizó con toda la disposición de ayudarnos y acompañarnos para que este proyecto y este proceso fueran más llevaderos. Estuvo presente de principio a fin, y sin él este trabajo no sería una realidad.*

*A todo el grupo **CALUMET**, que es como una familia para mí. Allí creé vínculos e hice amigos que nunca imaginé, y viví momentos y experiencias muy valiosas que no olvidaré.*

*Por último, quiero agradecer a mi director de proyecto, el profesor **Luis Ignacio Gonzalez**, por recibirme en el grupo, orientarme y permitirme ser parte de la familia CALUMET.*

A todos ustedes, gracias por esta oportunidad, por las experiencias que me regalaron y por los momentos tan especiales que hicieron de este proceso una etapa inolvidable en mi vida.

Nestor Javier Clavijo Hernández

Agradezco primeramente a Dios, por darme la fortaleza, la sabiduría y la perseverancia necesarias para culminar esta etapa tan importante de mi vida.

*A mi familia, especialmente a **Katerin Celis y Marco Ramírez**, mis padres, por brindarme todo su apoyo a lo largo de este proceso, por creer y confiar en mí. Gracias por motivarme a seguir adelante y por ser una base fundamental en mi formación personal y profesional.*

*A mi novia, **Lucia Franco**, que ha sido mi apoyo desde el principio y testigo de todo mi esfuerzo. Siempre supo lo que me costaba cada paso de este camino, y nunca me dejó sentir que lo recorría solo. Gracias por tu amor, por tu paciencia y por ser esa compañía constante que me dio fuerzas para seguir. También agradezco a su familia, **Edwin Fernando y Bleidy Castillo**, mis suegros, por su cariño y respaldo.*

A los compañeros y amigos que conocí en la universidad, por compartir conmigo aprendizajes, experiencias y momentos que hicieron de esta etapa algo especial e inolvidable.

*A mis compañeros de CALUMET, y en especial a **Juan David Lipez**, quien desde el inicio nos apoyó y nos transmitió sus conocimientos. Gracias por ser, desde el primer día, una persona honesta y amable, y por aportar a mi crecimiento académico y personal.*

*Finalmente, agradezco a mi director de proyecto, el profesor **Luis Ignacio**, por su orientación, paciencia y acompañamiento durante el desarrollo de este trabajo.*

A todos ustedes, gracias por hacer parte de este logro y por acompañarme en este camino.

Jesus David Ramirez Celis

Contenido

Pág.

Lista de Figuras

Pág.

Lista de Tablas

Pág.

Lista de Apéndices

Los apéndices estana djuntos y puede visualizarlos en la base de datos de la biblioteca UIS.

Apéndice A Plantillas Plan de Pruebas Estructurado**¡Error! Marcador no definido.**.

Resumen

Título: Reingeniería del sistema de gestión de menús y permisos de la plataforma Comunidad Académica (COMA)*.

Autor: Nestor Javier Clavijo Hernández, Jesus David Ramírez Celis**

Palabras Clave: Reingeniería de software, Gestión de menús, Control de acceso, Middleware, Caché a nivel de aplicación, Optimización de memoria.

Descripción: Este trabajo de grado presenta la reingeniería del sistema de gestión de menús y permisos de la plataforma Comunidad Académica (COMA) de la Universidad Industrial de Santander, con el propósito de reducir el consumo de memoria, mejorar la mantenibilidad del código y fortalecer el control de acceso. El diagnóstico permitió identificar tres deficiencias con una causa común: el árbol de menús, construido sobre la tabla TB_Menu, se reconstruía en cada inicio de sesión y se replicaba íntegramente en la sesión de cada usuario, lo que elevaba la memoria y la latencia; además, la lógica de autorización estaba dispersa en más de mil archivos mediante fragmentos copiados manualmente; finalmente, el modelo de seguridad acoplaba permisos y navegación, dejando rutas a servicios protegidos fuera de una política uniforme.

A partir de este diagnóstico, la solución se organizó en seis fases: análisis, rediseño del inicio de sesión mediante un middleware de autenticación, construcción de un gestor central de menús que opera como caché a nivel de aplicación por tipo de usuario, identificación del inventario de servicios mediante análisis estático, reestructuración de la base de datos y evaluación posterior. De este modo, se creó una única instancia de menú por tipo de usuario, de la que cada sesión guarda solo una referencia, mientras el middleware concentra la verificación de identidad y permisos sobre el recurso realmente solicitado.

Como resultado, bajo 200 usuarios concurrentes, la latencia de autenticación se redujo un 96,6 % (de 840,04 ms a 28,58 ms) y la memoria un 95,2 % (de 1,40 GB a 69 MB), con un 47 % más de capacidad y un 78,7 % menos de recolección de basura. Asimismo, los 1.789 servicios alcanzables quedaron con permiso explícito y denegación por defecto, cerrando el vector de acceso indebido detectado, sin regresiones en pruebas de extremo a extremo.

* Trabajo de Grado

** Facultad de Ingenierías Fisicomecánicas. Escuela de Ingeniería de Sistemas e Informática. Ingeniería de Sistemas. Director: Luis Ignacio Gonzáles Ramírez

Abstract

Title: Reengineering of the menu and permission management system of the Academic Community (COMA) platform*

Authors: Nestor Javier Clavijo Hernández, Jesus David Ramírez Celis**

Keywords: Software reengineering, Menu management, Access control, Middleware, Application-level cache, Memory optimization.

Description: This undergraduate thesis presents the reengineering of the menu and permission management system of the Academic Community (COMA) platform at the Industrial University of Santander, with the purpose of reducing memory consumption, improving code maintainability, and strengthening access control. First, the diagnosis made it possible to identify three deficiencies linked by a common cause: the menu tree, built on the TB_Menu table, was rebuilt at each login and fully replicated in each user's session, which increased memory consumption and latency; in addition, the authorization logic was scattered across more than a thousand files through manually copied fragments; finally, the security model coupled permissions with navigation, leaving routes to protected services outside a uniform policy.

Based on this diagnosis, the solution was organized into six phases: analysis, redesign of the login routine through an authentication middleware, construction of a central menu manager that operates as an application-level cache by user type, identification of the service inventory through static code analysis, database restructuring, and post-implementation evaluation. In this way, the manager creates a single menu instance per user type, of which each session stores only a reference, while the middleware concentrates identity and permission verification on the resource actually requested.

As a result, under 200 concurrent users, authentication latency dropped by 96.6% (from 840.04 ms to 28.58 ms) and memory consumption by 95.2% (from 1.40 GB to 69 MB), with 47% more capacity and 78.7% less garbage collection. Likewise, the 1,789 reachable services were assigned explicit permissions with default denial, thereby closing the improper-access vector that had been detected, with no regressions in the end-to-end tests.

* Undergraduate Thesis

** Faculty of Physical-Mechanical Engineering. School of Systems Engineering and Informatics. Systems Engineering. Director: Luis Ignacio Gonzáles Ramírez.

Introducción

La plataforma Comunidad Académica (COMA), desarrollada y administrada por el grupo Calumet de la Universidad Industrial de Santander, complementa los espacios de interacción de las diferentes escuelas y apoya el cumplimiento de la misión universitaria en docencia, investigación y extensión. A través de ella, la comunidad puede consultar y actualizar información personal, publicar noticias, organizar eventos y cargar documentos, entre otras funciones. En este contexto, debido a su importancia operativa, su arquitectura debe garantizar eficiencia en el uso de recursos, facilidad de mantenimiento y un modelo de seguridad robusto.

Con el paso del tiempo, sin embargo, el sistema de gestión de menús y permisos de la plataforma acumuló deficiencias que comprometían esas tres cualidades. Los menús constituyen la estructura de navegación en forma de árbol que enlaza los servicios de COMA y, en el diseño vigente, cada inicio de sesión reconstruía ese árbol y almacenaba una copia completa de la matriz de servicios habilitados dentro de la sesión del usuario. En consecuencia, la misma estructura se replicaba tantas veces como sesiones activas existieran, lo que incrementaba de forma sostenida la memoria ocupada, generaba consultas redundantes a la base de datos y añadía latencia a la construcción del menú. Además, este costo aumentaba tanto con la concurrencia de usuarios como con la incorporación de nuevos servicios.

A esta situación se sumaban dos problemas estrechamente relacionados. Por un lado, la lógica de autorización se encontraba dispersa, ya que el control de acceso se aplicaba mediante la inclusión manual de un fragmento de verificación en cada página protegida. Este patrón, repetido en más de mil archivos, elevaba el costo de cambio y aumentaba el riesgo de omisiones. Por otro lado, el modelo de seguridad acoplaba permisos y menús, de manera que proteger un servicio

exigía incorporarlo a un menú y, al mismo tiempo, existían rutas a servicios no visibles en la navegación. Por ello, el sistema dependía de controles frágiles, sin una definición uniforme de qué se permite y a quién.

A partir de este diagnóstico, se planteó una reingeniería del sistema de gestión de menús y permisos orientada a abordar la raíz común de las tres deficiencias. Para ello, la propuesta introdujo un gestor central de menús que actúa como caché a nivel de aplicación, organizado por tipo de usuario, de modo que se construye una única instancia de cada menú y cada sesión conserva solo una referencia a ella, en lugar de una copia completa. De forma complementaria, se centralizó la autenticación y la autorización en un middleware que valida cada solicitud en un único punto del flujo. Asimismo, se separaron la navegación y el control de acceso en la base de datos, de manera que los permisos se validan por servicio sobre el recurso realmente solicitado, con independencia de la organización del menú.

El desarrollo se llevó a cabo sobre la base tecnológica de la propia plataforma, una aplicación web Java EE desplegada sobre Tomcat. Con este propósito, se recurrió a técnicas de ingeniería inversa, análisis estático de código para inventariar los servicios existentes, filtros de servlet para la capa intermedia y pruebas automatizadas de rendimiento y de extremo a extremo. Así, se verificó el comportamiento antes y después de la intervención. En este marco, la reingeniería se organizó en seis fases: análisis del estado actual, rediseño de la rutina de inicio de sesión mediante middleware, desarrollo del gestor de menús, servicios y permisos, identificación de los servicios existentes, reestructuración de la base de datos y evaluación posterior a la implementación.

En este sentido, el presente trabajo de grado constituye un aporte orientado a la optimización y el aseguramiento de una plataforma institucional en operación. En particular, se

resolvieron problemas concretos de consumo de memoria, mantenibilidad y control de acceso. Como resultado, se fortaleció una herramienta clave para la comunicación y la gestión de información dentro de la Universidad Industrial de Santander.

1. Planteamiento y Justificación del Problema

La plataforma Comunidad Académica (COMA), desarrollada y administrada por el grupo Calumet de la Universidad Industrial de Santander, complementa los espacios de interacción de las diferentes escuelas y apoya el cumplimiento de la misión universitaria en docencia, investigación y extensión. A través de ella, la comunidad puede actualizar información personal, publicar noticias, organizar eventos, gestionar agendas, intercambiar documentos y, en general, comunicarse de manera más eficiente. Dada su criticidad operativa, su arquitectura debe garantizar eficiencia, mantenibilidad y un modelo de seguridad robusto.

En primer lugar, el problema central es el consumo de memoria derivado del manejo de menús. En COMA, los menús son la estructura de navegación que se organiza en forma de árbol, estos contienen enlaces a los servicios de la plataforma; cada servicio corresponde a una funcionalidad o a un punto de acceso a otras funciones. En el diseño vigente, cuando un usuario accede, la plataforma construye una matriz extensa con todos los servicios habilitados para ese tipo de usuario y la almacena íntegramente en su sesión dentro del servidor para utilizarla al generar el menú. Esta práctica duplica la misma estructura tantas veces como sesiones activas haya, lo que incrementa de manera sostenida la memoria ocupada y tensiona los recursos del servidor. Como efecto secundario, se generan consultas redundantes a la base de datos y se añade latencia en la construcción y uso del menú, lo que agrava la carga del servidor. Además, la incorporación continua de nuevos servicios incrementa el tamaño de la matriz, de modo que cada copia residente en sesión ocupa más memoria; en consecuencia, el consumo total crece tanto por la concurrencia de usuarios como por la expansión funcional de la plataforma con el paso del tiempo.

La propuesta es introducir un gestor central de menús que actúe como caché a nivel de aplicación y que se organice por tipos de usuario (estudiante, egresado, planta, etc.), porque en la práctica los usuarios que comparten un mismo tipo acceden a los mismos menús. De este modo, el gestor crea una única instancia para cada tipo de usuario, por ejemplo, una para el menú de Estudiante, otra para el de Docente y así sucesivamente; cada sesión no almacena la estructura completa, sino únicamente un puntero a la instancia que le corresponda. Por puntero se entiende un identificador que señala dónde está guardada la instancia en el gestor sin copiarla. Con ello se mantiene el uso de la sesión como referencia, se evita la multiplicación de estructuras idénticas entre miles de usuarios y se preserva un acceso rápido a los menús, al tiempo que disminuye de forma significativa el consumo de memoria y la necesidad de consultas redundantes a la base de datos.

En segundo lugar, el sistema presenta un problema de mantenibilidad a nivel de código por la dispersión de la lógica de autorización. Actualmente, el acceso se controla incluyendo manualmente un fragmento de verificación en cada página protegida. Este patrón, repetido en más de mil archivos, eleva el costo de cambio y aumenta el riesgo de omisiones. La propuesta es centralizar la autenticación y la autorización mediante un middleware, entendido como una capa intermedia que recibe cada solicitud, válida la identidad y los permisos del usuario y solo entonces permite que la petición continúe hacia el servicio solicitado. Al ubicar la verificación en un único punto del flujo de solicitudes, se elimina la repetición de código, se reducen las inconsistencias y se facilita la evolución de la plataforma.

No obstante, implementar esta solución conlleva un reto principalmente operativo. Para consolidar el control de acceso en el middleware, será necesario modificar los más de mil archivos que hoy contienen la lógica incrustada. En muchos casos, esa verificación no se incluyó como

componente reutilizable, sino que se copió y pegó dentro de cada servicio, lo que exige una depuración cuidadosa, caso por caso, para retirar duplicaciones sin alterar el comportamiento funcional. Una vez completada esta limpieza, las páginas dejarán de contener validaciones propias y el control de acceso quedará concentrado en la capa intermedia, con beneficios directos en coherencia y mantenimiento.

En tercer lugar, el modelo de seguridad actual acopla permisos y menús, lo que introduce vulnerabilidades, ya que proteger un servicio exige incorporar lo a un menú, de modo que existen rutas a servicios no visibles en la navegación. Esa dependencia de la estructura de menús obliga a controles contextuales frágiles que no se apoyan en una definición uniforme de qué se permite y a quién. Entonces, la solución propuesta combina dos acciones coordinadas: separar navegación y autorización en la base de datos, de manera que los menús se conservan sólo como mecanismo de visibilidad, y crear un sistema a nivel de aplicación que, apoyado en esa nueva estructura, valide los permisos por servicio de forma eficiente y consistente. Este sistema consultará una única fuente sobre servicios y tipos de usuarios, pudiendo reutilizar decisiones en caché y aplicando una política de acceso uniforme con independencia de la organización del menú. Con esta separación, cualquier solicitud a una ruta no registrada o sin permiso es detenida por el middleware, incluso si la URL y los parámetros necesarios se conocen.

Para hacer viable esta propuesta, uno de los desafíos más relevantes es construir un inventario completo y confiable de los servicios para registrarlo en la base de datos. Esto implica localizar todas las rutas operativas, incluidas variantes con parámetros, alias y puntos de entrada indirectos; decidir la granularidad con la que se definirá cada servicio; y asignar sus reglas por tipo de usuario. Para lograrlo, se requiere combinar análisis del código con revisión de registros de acceso, pruebas exploratorias y validación con responsables funcionales. Durante la transición se

busca aplicar una política de denegación por defecto de servicios, la creación controlada de excepciones temporales y un monitoreo cercano de errores, hasta completar el registro y las pruebas de regresión.

En conjunto, esta reingeniería para la integración de las medidas no solo suma beneficios, sino que establece una dependencia positiva entre ellas. La separación entre menús y permisos crea un modelo único de autorización por servicio que el middleware consulta de forma consistente, sin depender de cómo esté organizado el menú. El gestor central de menús, organizado por tipos de usuario, provee a esa verificación un acceso rápido a la información necesaria y evita reconstrucciones repetidas, porque cada sesión guarda solo un puntero a la instancia de menú que le corresponde. Así, el middleware aplica las reglas sobre una misma fuente, el gestor reduce la carga de datos que esa aplicación requiere y la independencia de los menús garantiza que ninguna ruta quede accesible por fuera de la política central. El resultado es coherencia en las decisiones de acceso, menor latencia bajo carga y una plataforma más sencilla de auditar y mantener.

2. Objetivos

2.1. Objetivo General

Evaluar y rediseñar el sistema de gestión de menús y permisos de la plataforma Comunidad Académica (COMA) de la Universidad Industrial de Santander, centralizando el almacenamiento de los menús en el servidor para optimizar el uso de memoria en las sesiones de los usuarios, al tiempo que se separa la lógica de menús y permisos según los diferentes roles, con el fin de mejorar la gestión de memoria, tiempos de carga y reforzar la seguridad en el control de accesos.

2.2. Objetivos Específicos

Evaluar el estado actual del sistema de gestión de menús y permisos de la plataforma COMA, con el fin de identificar falencias, problemas de seguridad y deficiencias en el uso de memoria.

Diseñar una arquitectura centralizada para la gestión de menús y permisos que elimine la duplicación de código, facilite el mantenimiento y permita un desarrollo más organizado y escalable.

Rediseñar el sistema de gestión de menús a partir de un mecanismo centralizado de almacenamiento en memoria, que permita cargar los menús una única vez y optimice el uso de recursos

Separar la lógica de control de permisos en el código y en la base de datos, de manera que los permisos no dependan directamente de los menús y se fortalezca la seguridad en el control de accesos.

Eliminar procesos redundantes y operaciones innecesarias presentes en la creación y consulta de menús, con el fin de reducir la carga sobre la base de datos y mejorar el rendimiento del sistema.

3. Marco Teórico

La reingeniería de la gestión de menús y permisos de COMA se apoya en conceptos de varios campos de la ingeniería de software: el método que guía la intervención sobre un sistema en producción, la arquitectura web en Java sobre la que la plataforma opera, y las nociones que explican la memoria, el rendimiento y el control de acceso, las tres dimensiones que el trabajo mejora. Este capítulo reúne esas bases, revisa antes los trabajos que le anteceden y fija al final los términos que el libro emplea.

3.1. Antecedentes

La reingeniería de software es una práctica establecida, con una base que va desde las taxonomías que ordenan sus técnicas (Chikofsky y Cross, 1990) hasta los catálogos de patrones que guían su aplicación sobre sistemas orientados a objetos (Demeyer, Ducasse y Nierstrasz, 2002). En un contexto cercano al de este trabajo, García López, Viñas Álvarez y Dávalos Hernández (2024) propusieron un modelo de reingeniería en fases para modernizar los sistemas de una institución de educación superior, con el fin de mejorar su productividad y reutilizar sus componentes. Estos trabajos comparten el punto de partida de la reingeniería de COMA, la intervención de un sistema en producción para renovar su estructura sin perder su función.

El control de acceso roto es, por su parte, un riesgo documentado y frecuente en las aplicaciones web, y encabeza la clasificación de riesgos más críticos que mantiene OWASP (2025a). La defensa recomendada consiste en decidir el acceso en el servidor, sobre el recurso que realmente se solicita, y en denegarlo por defecto salvo permiso explícito, en vez de confiar en comprobaciones dispersas o en datos que el cliente controla. Sobre ese principio se rediseña el control de acceso de la plataforma.

En el plano local, la plataforma COMA ya ha motivado trabajos de optimización. Acuña Vargas (2026) desarrolló para ella un sistema de caché de consultas a la base de datos, con el que alivió la carga que las consultas repetidas imponían a sus módulos. La presente reingeniería se suma a esa línea de mejora de la plataforma y la amplía, al abordar en un mismo trabajo el consumo de memoria, el rendimiento y la seguridad de la gestión de menús y permisos, y al verificar sus efectos con mediciones comparables entre el estado inicial y el final.

3.2. Reingeniería de Software

La reingeniería de software es el examen y la alteración de un sistema existente para reconstituirlo en una forma nueva, y abarca tanto ese rediseño como su posterior implementación (Chikofsky y Cross, 1990). Se distingue del desarrollo desde cero en que parte de un sistema en funcionamiento y conserva lo que este hace mientras mejora cómo lo hace, un enfoque preferible cuando el sistema presta un servicio que debe seguir operando (Sommerville, 2011). Sus técnicas van desde el análisis previo del sistema hasta la transformación de su código y de sus datos, y se describen enseguida.

3.2.1 Ingeniería Inversa

La ingeniería inversa es el proceso de analizar un sistema para identificar sus componentes y las relaciones entre ellos, y producir una representación del sistema en otra forma o en un nivel de abstracción más alto (Chikofsky y Cross, 1990). Se limita a comprender el sistema, sin modificarlo. Precede a cualquier reingeniería, porque recuperar el diseño de un sistema poco documentado es la condición para transformarlo sin romperlo.

3.2.2 Reestructuración del Código y Refactorización

La reestructuración transforma la organización interna de un sistema sin cambiar su comportamiento externo ni la funcionalidad que ofrece (Chikofsky y Cross, 1990). En la programación orientada a objetos, su forma disciplinada es la refactorización, una sucesión de transformaciones pequeñas que preservan el comportamiento y mejoran la estructura del código; cada una es demasiado menor para justificarse por sí sola, pero su acumulación reordena el diseño (Fowler, 2018).

3.2.3 Reestructuración del Código y Refactorización

La optimización de código mejora el rendimiento o el consumo de recursos de un programa sin alterar el resultado que produce. Actúa sobre los algoritmos y las estructuras de datos que concentran ese costo, y su ganancia se establece por medición, comparando el estado anterior con el posterior bajo las mismas condiciones. Conviene por ello guiarla con datos y no con suposiciones, pues el esfuerzo dedicado a acelerar lo que no domina el costo rinde poco (Knuth, 1974).

3.2.4 Reestructuración de Datos

La reestructuración de datos reorganiza el modelo de datos de un sistema, sus tablas y las relaciones entre ellas, para ajustarlo a un diseño nuevo (Chikofsky y Cross, 1990). Es una de las intervenciones más sensibles de una reingeniería, porque los datos persisten más allá de cada ejecución y numerosos componentes dependen de su forma.

3.3 La Plataforma: Arquitectura Web en Java

3.3.1 Arquitectura Cliente/Servidor

COMA es una aplicación web, y como tal sigue el modelo cliente/servidor. Un cliente, el navegador, envía peticiones a un servidor, que las procesa y devuelve una respuesta, sobre el protocolo HTTP (Sommerville, 2011). HTTP no conserva estado entre peticiones, pues atiende cada una de forma independiente, de modo que mantener la continuidad de la sesión de un usuario exige un mecanismo aparte, que se describe más adelante.

3.3.2 Java, Servlets y JavaServer Pages

Java es un lenguaje de programación orientado a objetos que se compila a un código intermedio, el bytecode, ejecutado por la máquina virtual de Java, lo que le da independencia de la plataforma (Oracle, s.f.). Sobre Java, los servlets son las clases que atienden las peticiones y generan las respuestas en el lado del servidor, y son la base de las aplicaciones web en esta tecnología. Las JavaServer Pages (JSP) son plantillas que intercalan contenido dinámico en una página y que el servidor traduce a servlets para ejecutarlas, con lo que facilitan la generación de las vistas que ve el usuario (Eclipse Foundation, s.f.).

3.3.3 Apache Tomcat

Una aplicación web en Java se ejecuta dentro de un contenedor que implementa las especificaciones de servlets y JSP y administra su ciclo de vida. Apache Tomcat es el contenedor de código abierto que COMA emplea para ello (The Apache Software Foundation, s.f.).

3.3.4 MySQL

MySQL es un sistema de gestión de bases de datos relacionales, en el que la información se organiza en tablas relacionadas entre sí y se consulta con el lenguaje SQL (Oracle, s.f.). COMA guarda en MySQL sus datos, entre ellos las tablas de las que se derivaba el menú de cada usuario.

3.3.5 Middleware y Filtros de Servlet

El middleware es el software que se interpone entre los componentes de una aplicación para atender asuntos transversales, comunes a muchas peticiones. En el modelo de servlets, esa idea se materializa en el filtro, una pieza que intercepta cada petición antes de que alcance el recurso solicitado, y su respuesta antes de que llegue al cliente (Eclipse Foundation, s.f.). Esa posición lo convierte en el lugar natural para la lógica transversal, como el control de acceso.

3.3.6 La Sesión HTTP

Como HTTP no conserva estado, la continuidad de la interacción de un usuario se sostiene con una sesión, un espacio de datos que el contenedor asocia a cada usuario y mantiene entre sus peticiones, identificado por una credencial que viaja en cada una (Eclipse Foundation, s.f.). En ese espacio, una aplicación puede guardar información propia de cada usuario mientras dura su visita, y de esa posibilidad depende buena parte del consumo de memoria que este trabajo estudia.

3.4 El Almacenamiento en Memoria

3.4.1 Caché a Nivel de Aplicación

Una caché guarda en memoria datos costosos de calcular o de recuperar para servirlos de inmediato y evitar rehacer el trabajo (Fowler, 2002). A nivel de aplicación, la caché conserva resultados que muchas peticiones comparten, de modo que se calculan una vez y se reutilizan. Su beneficio depende de que lo almacenado se lea muchas más veces de las que cambia.

3.4.2 Referencia Frente a Copia

Una referencia señala un objeto sin duplicarlo, de manera que varias referencias pueden apuntar a un mismo objeto en memoria. Compartir un objeto por referencia, en lugar de copiarlo para cada usuario, ahorra la memoria que ocuparían las copias. El patrón Flyweight recoge esta idea, pues cuando muchos contextos necesitan objetos iguales mantiene una sola instancia compartida que cada contexto referencia, y evita replicar lo que puede ser único (Gamma et al., 1994). Sobre este principio se apoya el rediseño del menú de la plataforma.

3.4.3 Concurrencia y Sincronizzación

Cuando una sola estructura se comparte entre muchas sesiones que se ejecutan a la vez, sus accesos deben coordinarse para que las lecturas y las escrituras no se interfieran ni devuelvan datos inconsistentes. La concurrencia estudia esa ejecución simultánea, y la sincronización reúne los mecanismos que ordenan el acceso a los datos compartidos (Goetz et al., 2006). Una estructura que muchos leen y que rara vez cambia puede compartirse con seguridad si sus escrituras, poco frecuentes, se controlan.

3.5 La Memoria de la Máquina Virtual de Java

3.5.1 El Heap y la Recolección de Basura

La máquina virtual de Java guarda los objetos que un programa crea en una región de memoria llamada heap. La administra de forma automática, con un recolector de basura que libera los objetos que ya no son alcanzables desde el programa y releva al desarrollador de liberar la memoria a mano (Jones et al., 2011). El tamaño ocupado del heap y la actividad del recolector son las dos magnitudes que revelan cómo un programa usa la memoria.

3.5.2 Retención y Fugas de Memoria

Un objeto permanece en el heap mientras algo alcanzable lo referencie, y a esa permanencia se le llama retención. Cuando objetos que ya no se necesitan siguen referenciados, el recolector no puede liberarlos y el heap crece sin freno, en lo que se conoce como fuga de memoria (Jones et al., 2011). No toda retención alta es una fuga. El estado que una aplicación mantiene a propósito mientras hace falta ocupa memoria de forma legítima y se libera cuando deja de necesitarse. Distinguir una cosa de la otra, y averiguar qué referencias retienen más memoria, es la tarea del análisis del heap.

3.6 Rendimiento y Concurrencia

3.6.1 Latencia, Capacidad y Carga

El rendimiento de un sistema se mide con dos magnitudes: la latencia, el tiempo que tarda en responder una petición, y la capacidad, el número de peticiones que atiende por unidad de

tiempo, también conocida como throughput (Jain, 1991). Ambas dependen de la carga, la cantidad de trabajo simultáneo que el sistema enfrenta. A medida que la carga crece, un sistema bien dimensionado sostiene su latencia, mientras que uno que llega a su límite la ve dispararse.

3.6.2 Pruebas de Rendimiento y Estrés

Una prueba de rendimiento mide el comportamiento de un sistema bajo una carga definida, y una prueba de estrés eleva esa carga hasta que el sistema se degrada o falla, para hallar su límite (Molyneaux, 2014). Automatizarlas por completo hace que las dos corridas que se comparan sean idénticas, y elimina al operador como fuente de variación.

3.6.3 El Pool de Conexiones

Abrir una conexión a la base de datos tiene un costo, y bajo concurrencia las conexiones abiertas pueden agotar los recursos del servidor. Un pool de conexiones mantiene un conjunto de conexiones reutilizables y las presta a medida que se necesitan, con lo que acota su número y ahorra el costo de crear una por cada petición (Fowler, 2002). Sin ese control, un servidor puede caer por agotamiento antes de que se pueda medir su rendimiento real.

3.6.4 Contenedorización

Un contenedor empaqueta una aplicación con sus dependencias para que se ejecute igual en cualquier entorno, aislada del anfitrión y de los demás contenedores (Merkel, 2014). Desplegar dos versiones de un sistema en contenedores idénticos permite atribuir la diferencia entre ellas al código, y no al entorno de ejecución.

3.7. Seguridad: el Control de Acceso

3.7.1 Autenticación y Autorización

La autenticación establece quién es un usuario, y la autorización decide qué le está permitido hacer. Son distintas, pues un sistema puede saber quién es el usuario y aun así negarle una acción. El control de acceso es la aplicación efectiva de la autorización. Un modelo extendido es el control de acceso basado en roles (RBAC), que concede los permisos a roles y asigna roles a los usuarios, en vez de conceder los permisos a cada usuario uno por uno (Sandhu et al., 1996).

3.7.2 OWASP Top Ten y el Control de Acceso Roto

OWASP mantiene el Top Ten, una lista que se actualiza de forma periódica con los riesgos de seguridad más críticos de las aplicaciones web, con el fin de concienciar sobre ellos (OWASP, 2025a). El primero de esos riesgos, el control de acceso roto (A01), reúne las fallas en las que un usuario actúa más allá de los permisos que le corresponden. Es el riesgo que este trabajo diagnostica y corrige.

3.7.3 OWASP ASVS Como Estándar de Verificación

El estándar de verificación de seguridad de aplicaciones de OWASP (ASVS) es un catálogo de requisitos de seguridad, organizado por dominios, contra el que una aplicación puede verificarse (OWASP, 2025b). Mientras el Top Ten difunde los riesgos, el ASVS ofrece un criterio medible con el que comprobar el cumplimiento de cada requisito. Su capítulo de autorización reúne los requisitos del control de acceso, y es el que este trabajo emplea para verificar el diseño reingenierizado.

3.8 Análisis y Prueba

3.8.1 Análisis Estático y Grafos Dirigidos

El análisis estático examina el código de un programa sin ejecutarlo, para deducir propiedades de su estructura. Una de esas propiedades aparece al representar el sistema como un grafo dirigido, cuyos nodos son sus componentes y cuyas aristas son las dependencias entre ellos; sobre ese grafo se razona qué componentes son alcanzables desde uno dado. Un recorrido en amplitud visita el grafo por niveles a partir de un nodo de origen, y descubre así todo lo que cuelga de él (Cormen et al., 2009). Este análisis es la base del inventario de servicios que el trabajo levanta en la Fase 4.

3.8.2 Pruebas Automatizadas de Extremo a Extremo

Una prueba de extremo a extremo ejercita el sistema a través de su interfaz, como lo haría un usuario real, y comprueba que un flujo completo funciona a lo largo de todas sus partes (Fewster y Graham, 1999). Automatizarla hace que cada corrida sea reproducible y quita la variabilidad del operador. Sobre estas pruebas se apoya la verificación de que el nuevo control de acceso no rompió el acceso legítimo de los usuarios.

3.8.3 Cifrado Asimétrico RSA

La criptografía asimétrica usa un par de claves, una pública y una privada, de modo que lo que una cifra solo la otra puede descifrarlo. RSA es uno de esos algoritmos, y toma su nombre de las iniciales de sus autores (Rivest, Shamir y Adleman, 1978). COMA genera un par de claves

RSA por sesión para proteger el inicio de sesión, y ese estado por sesión es el principal retenedor de memoria que subsiste tras la reingeniería.

3.9 Marco Conceptual

Los términos que el libro emplea con un sentido preciso se recogen aquí para su consulta.

- Reingeniería de software. Examen y alteración de un sistema existente para reconstituirlo en una forma nueva, preservando su función.
- Servlet. Clase de Java que atiende una petición web y genera su respuesta en el servidor.
- Filtro. Componente que intercepta las peticiones antes de que alcancen el recurso solicitado, empleado aquí para el control de acceso.
- Sesión. Espacio de datos que el servidor mantiene para cada usuario entre sus peticiones.
- Caché. Almacenamiento en memoria de datos costosos de obtener, para reutilizarlos sin rehacer el trabajo.
- Heap. Región de memoria de la máquina virtual de Java donde residen los objetos que un programa crea.
- Recolección de basura. Liberación automática de los objetos que ya no son alcanzables desde el programa.
- Retención. Permanencia de un objeto en el heap mientras algo lo referencia.
- Latencia. Tiempo que un sistema tarda en responder una petición.
- Capacidad. Número de peticiones que un sistema atiende por unidad de tiempo.

- Control de acceso. Aplicación efectiva de la autorización, que decide si una petición puede alcanzar un recurso.
- Denegación por defecto. Criterio según el cual se niega el acceso salvo que un permiso explícito lo conceda.
- Grafo dirigido. Representación de un sistema como nodos, sus componentes, y aristas, las dependencias entre ellos.

4. Metodologia

La metodología adoptada para el desarrollo de este proyecto se estructuró en seis fases consecutivas, cada una orientada a dar solución progresiva al problema identificado en la gestión de autenticación, servicios y permisos. A continuación, se describe cada una de ellas:

4.1. Fase 1: Análisis del Estado Actual de la Aplicación

En esta primera etapa se realiza un diagnóstico de la plataforma con el fin de identificar su comportamiento en condiciones reales de uso. Para ello se definen y ejecutan actividades de evaluación que cubren desempeño y seguridad. Con el fin de garantizar repetibilidad y trazabilidad en futuros ciclos de desarrollo o mantenimiento, se prioriza la definición de un plan de pruebas reutilizables.

4.1.1. Actividades

- Actividad 1: Se elaborará un plan de pruebas formal y reutilizable que incluya: propósito y alcance, criterios de entrada y salida, especificación de casos y datos de prueba,

guiones de ejecución, criterios de automatización, plantillas de reporte y gestión de incidencias. Este plan servirá como base operativa para las actividades de medición de latencia y consumo de memoria.

- Actividad 2: Se evaluará el rendimiento de la aplicación bajo distintos niveles de usuarios concurrentes, siguiendo el plan de pruebas definido en la Actividad 1. Se registran métricas como tiempo medio, así como el comportamiento bajo aumentos graduales de carga.
- Actividad 3: Con base en los casos y guiones del plan de pruebas, se medirá el uso de memoria asociado a sesiones de usuario. Se analizará el impacto de la carga sobre los recursos del sistema y la estabilidad durante períodos sostenidos de uso.
- Actividad 4: Se revisará la correcta gestión de permisos, roles y controles de acceso, así como la interacción de los usuarios con los servicios disponibles.

4.2. Fase 2: Rediseño de la Rutina de Inicio de Sesión Mediante Middleware

En esta segunda etapa se busca eliminar la redundancia y la complejidad que actualmente existen en los procesos de autenticación. La estrategia consiste en centralizar la validación de usuarios a través de un middleware unificado, capaz de gestionar el acceso a los diferentes servicios de forma eficiente.

4.2.1. Actividades

- Actividad 5: Se eliminarán las rutinas de inicio de sesión independientes, identificando y retirando los métodos de autenticación propios de cada servicio para centralizar la validación de usuarios en el nuevo componente de autenticación.

- Actividad 6: Se unificarán las rutinas en el middleware de autenticación, desarrollando un filtro encargado de centralizar la validación de credenciales, gestionar el mapeo de rutas de acceso y asegurar un flujo homogéneo de inicio de sesión en toda la plataforma.

4.3. Fase 3: Desarrollo del Gestor de Menús, Servicios y Permisos

En esta fase se implementa un módulo que proveerá al middleware la información de menús, servicios y permisos mediante un mecanismo de almacenamiento local, con el fin de optimizar la validación de accesos y reducir los tiempos de respuesta.

4.3.1. Actividades

- Actividad 7: Se diseñará e implementará el módulo de gestión de menús, servicios y permisos.

4.4. Fase 4: Identificación y Registro de los Servicios Existentes

Para garantizar que el middleware tenga cobertura sobre todos los recursos de la aplicación, en esta fase se realiza un barrido exhaustivo de los archivos de la plataforma, con el fin de localizar y documentar cada uno de los servicios implementados o invocados.

4.4.1. Actividades

- Actividad 8: Se establecerán los factores que determinarán cuándo un servicio está aislado o no debidamente integrado, definiendo criterios que servirán como referencia para su detección automática en análisis posteriores.

- Actividad 9: Se implementará un script, basado en los criterios definidos, que recorrerá los archivos de la plataforma para identificar servicios aislados, estableciendo un mecanismo sistemático y repetible para su detección.
- Actividad 10: Se depurará manualmente la lista de servicios generada por el script, corrigiendo errores y recuperando posibles registros que no hayan sido detectados automáticamente.

4.5. Fase 5: Reestructuración de la Base de Datos

En esta fase se redefine la estructura de la base de datos con el propósito de separar la relación entre menús, servicios y permisos. El objetivo es que cada uno de estos elementos pueda administrarse de forma independiente, evitando dependencias rígidas y permitiendo que el middleware gestione los accesos con mayor flexibilidad y escalabilidad.

4.5.1. Actividades

- Actividad 11: Se separarán los menús y los servicios en tablas independientes para evitar que compartan la misma estructura y permitir su gestión de forma autónoma.
- Actividad 12: Se implementarán en el backend las clases necesarias para administrar las nuevas tablas de menús y servicios, garantizando la correcta interacción de la aplicación con la base de datos reorganizada.
- Actividad 13: Se estructurará la lógica de la aplicación para que cada componente (menús, servicios y permisos) sea gestionado de manera diferenciada y coherente con la nueva arquitectura de la base de datos.

4.6. Fase 6: Evaluación Posterior a la Implementación

En esta etapa se realiza un nuevo análisis del estado de la aplicación, replicando las mismas métricas de la fase inicial: latencia de inicio de sesión, consumo de memoria por usuario y validación de la seguridad en los accesos. Estos resultados serán comparados con los obtenidos en la Fase 1, con el fin de establecer las mejoras alcanzadas, cuantificar los beneficios del rediseño e identificar oportunidades de optimización futura.

4.6.1. Actividades

- Actividad 14: Se medirá la latencia de inicio de sesión aplicando los casos, datos y guiones definidos en el Plan de Pruebas Estructurado, con escenarios de concurrencia tras los cambios realizados.
- Actividad 15: Se evaluará el consumo de memoria por usuario usando los mismos escenarios y criterios del Plan de Pruebas Estructurado, garantizando repetibilidad y comparabilidad tras los cambios
- Actividad 16: Se revisará la correcta gestión de permisos, roles y controles de acceso, así como la interacción de los usuarios con los servicios disponibles tras los cambios implementados.
- Actividad 17: Consolidar métricas y analizar resultados conforme al Plan de Pruebas Estructurado, comparándolos con las ejecuciones anteriores del plan para identificar variaciones y documentar conclusiones en las plantillas establecidas.

5. Fase 1: Análisis del Estado Actual de la Aplicación

La Fase 1 estableció el diagnóstico de COMA previo a la reingeniería en tres dimensiones: comportamiento funcional, rendimiento bajo concurrencia y modelo de control de acceso. Para cubrirlas se ejecutaron cuatro actividades: la elaboración del plan de pruebas estructurado (Actividad 1), la medición del rendimiento bajo distintos niveles de concurrencia (Actividad 2), el análisis del consumo de memoria por sesión (Actividad 3) y la revisión del control de acceso (Actividad 4).

Los resultados de esta fase cumplieron dos funciones. Sirvieron como línea base para la evaluación posterior a la implementación (Fase 6) y orientaron las decisiones de diseño de las fases intermedias.

5.1. Actividad 1: Plan de Pruebas Estructurado

La Actividad 1 elaboró el plan de pruebas formal y reutilizable, que rige las mediciones de las Fases 1 y 6. Se estructuró conforme a la norma de pruebas de software ISO/IEC/IEEE 29119-3:2021, organizada según el desglose de secciones del IEEE 829 que la serie 29119 reemplazó, y define el propósito y el alcance, los criterios de entrada y de salida, los casos y datos, los guiones de ejecución y automatización, las plantillas de reporte y el registro de incidencias; se recoge completo en el Anexo A. Esta sección detalla su componente funcional, las pruebas de extremo a extremo (E2E) de la suite `coma-tests`, desarrollada desde cero para este trabajo de grado, que verifican que la plataforma conserve su comportamiento antes y después de la reingeniería. Las mediciones de rendimiento y de consumo de memoria de las Actividades 2 y 3 aplican el mismo plan y se describen en las secciones 5.2 y 5.3.

En esta fase, la suite se ejecutó sobre la rama `develop`, la rama principal de desarrollo de COMA que recoge el estado de la plataforma antes de la reingeniería, para fijar la línea base funcional. En la Fase 6 se ejecutará sobre la rama de la reingeniería, y cualquier diferencia entre ambas ejecuciones señalará una regresión. Por eso se optó por la automatización total en lugar de pruebas manuales, que hace idénticas las dos ejecuciones y elimina la variabilidad del operador como factor de confusión.

5.1.1. La Suite Coma-Tests

La suite se construyó con *pytest* y Playwright (Krekel et al., 2024; Microsoft, 2024). Playwright controla un navegador real, con el que cada prueba reproduce la interacción de un usuario sobre la interfaz, sin simular el protocolo.

La suite se organizó en cuatro capas con responsabilidades separadas (Tabla 1) y conectadas en cascada (Figura 1). Esta separación es la decisión de diseño central,porque agregar un caso de prueba consiste en recombinar **steps** existentes en lugar de duplicar código. Por ejemplo, *step_login_admin* se reutiliza en las pruebas de humo y en el flujo de noticias, y *step_logout_to_home* se invoca diecisiete veces dentro del flujo de trabajo de grado. Un cambio en el inicio o el cierre de sesión se ajusta en una sola definición y se propaga a todos los flujos que lo usan.

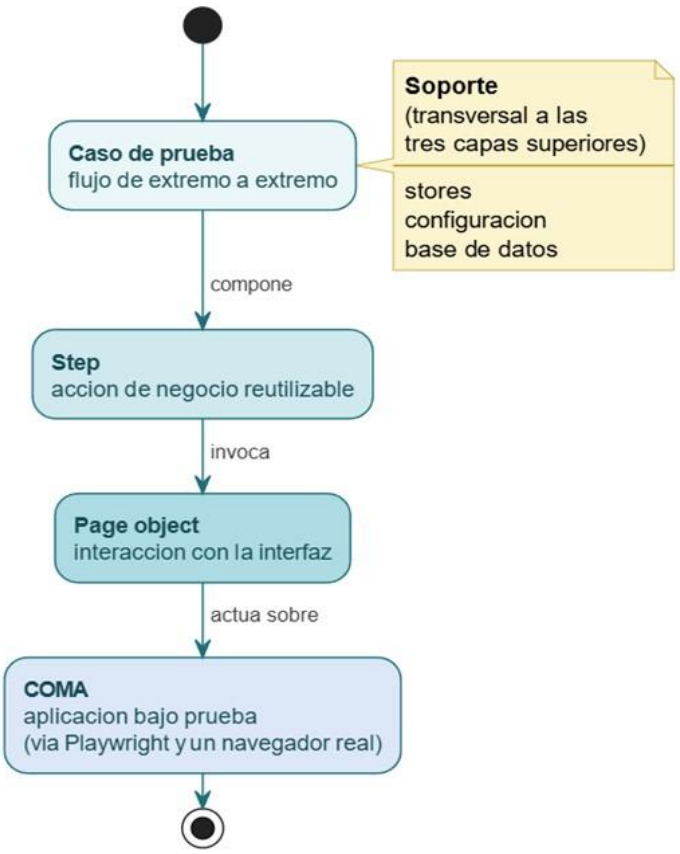
Tabla 1

Capas de la suite de coma-tests

Capa	Responsabilidad
Caso de prueba	Define un flujo como una secuencia ordenada de **steps** y sus verificaciones
Step	Encapsula una acción de negocio reutilizable: iniciar sesión, crear una noticia, aprobar un tema
Page	Encapsula la interacción con una pantalla concreta: localizadores y acciones de interfaz
Soporte	Estado compartido entre **steps** (**stores**), configuración por variables de entorno y consultas a la base de datos para obtener datos reales

Figura 1

Arquitectura en capas de la suite coma-tests



5.1.2. Selección de Módulos

Se seleccionaron dos módulos por su representatividad funcional. El módulo de Noticias concentra el flujo de gestión de contenido (creación, edición y persistencia entre sesiones) y se opera según el tipo de usuario (usuario general o administrador), lo que permite verificar de forma transversal la autenticación y el control de acceso. El módulo de Trabajos de Grado involucra a varios roles con permisos diferenciados y reproduce un proceso administrativo extenso, lo que lo convierte en el caso más exigente de la plataforma.

5.1.3. Casos de Prueba Implementados

Se implementaron cuatro casos: dos pruebas de humo y dos flujos de extremo a extremo.

Pruebas de humo. Verificaron las condiciones mínimas de operación antes de ejecutar los flujos. *test_inicio_de_sesion_usuario_default* comprueba que un usuario sin privilegios administrativos inicia sesión correctamente. *test_inicio_de_cambio_de_rol_administrador* comprueba que una cuenta administradora inicia sesión, mantiene la sesión activa y cambia su tipo de usuario a Administrador.

Flujo de gestión de noticias. El caso *test_crear_y_editar_noticia* recorre el ciclo de creación y edición de una noticia a lo largo de dos sesiones. La misma cuenta administradora crea la noticia operando como usuario general, cierra la sesión y vuelve a ingresar, cambia al tipo de usuario de Administrador y edita la noticia creada.

El caso valida dos comportamientos. El primero es la persistencia de la noticia entre sesiones, ya que la creada en la primera sesión se recupera y se edita en la segunda. El segundo es el cambio de tipo de usuario, pues la edición ocurre después de activar el tipo de usuario

Administrador. La noticia usa un título único derivado de una marca de tiempo, lo que evita colisiones entre ejecuciones.

Flujo de trámite de trabajo de grado. El caso *test_thesis_submission_flow* reproduce el trámite completo de un trabajo de grado, desde el registro del tema hasta la calificación final. Representa el camino favorable, en el que todas las aprobaciones proceden. El caso se descompone en once etapas autocontenidas, orquestadas por una función principal.

Cada etapa agrupa la intervención de uno o más roles. Para cada rol, la prueba inicia sesión, ejecuta sus acciones, verifica en la vista los datos esperados cuando corresponde y cierra la sesión. Dos etapas comienzan, además, con un paso de preparación de datos en la base de datos, como limpiar actas previas o matricular a los autores, antes de cualquier autenticación. La Tabla 2 resume las once etapas, los roles que actúan en cada una y su verificación o resultado principal.

Tabla 2

Etapas del flujo de trámite de trabajo de grado

Etapa	Roles que actuan	Verificacion o resultado
1.registro del tema	Autor 1 y Autor 2	El Autor 2 visualiza los campos del proyecto registrados por el Autor 1 y acepta la coautoría
2. Envió al director	Autor 1	Acción de transición: el tema se envía al director
3. Remisión al comité	Director	El director visualiza los datos del tema antes de remitirlo al comité
4. Aprobación del tema	Miembro del comité	El acta del comité registra el tema como aprobado

Etapa	Roles que actuan	Verificacion o resultado
5. Solicitud de evaluador del plan	Autor 1 y Director	El director visualiza los datos del proyecto en la vista de solicitud de evaluador
6. Asignación de evaluador	Miembro del comité	El comité visualiza los datos del proyecto en la vista de asignación
7. Evaluación del plan	Evaluador	El evaluador visualiza los datos del proyecto en la vista de evaluación
8. Aprobación del plan	Miembro del comité	El acta registra el evaluador asignado y el resultado aprobado
9. Solicitud de jurados	Autor 1 y Director	Acción de transición: se carga el documento final y se solicita la asignación de jurados
10. Asignación de jurados	Miembro del comité	El acta registra los dos jurados calificadores asignados
11. Calificación	Jurado 1, Jurado 2 y autores	Los autores visualizan la nota final, calculada como el promedio de las dos calificaciones

La ejecución en una sola corrida fue posible gracias al **fixture** *thesis_schedule*. Un **fixture** es una función de pytest que prepara el estado necesario antes de ejecutar una prueba; en este caso, *thesis_schedule* preparó en la base de datos un cronograma con las ventanas de recepción de tema y de plan vigentes, y con las fechas de grado y de entrega del documento ajustadas para habilitar la calificación. Así se validaron en una única ejecución etapas que, en operación real, ocurren en fechas distintas. Los usuarios que intervienen (autores, director, miembros del comité, evaluador

y jurados) se obtuvieron de la propia base de datos de COMA mediante consultas a usuarios elegibles, de y las pruebas operaron sobre datos reales de la plataforma.

La arquitectura de **steps** habilita la derivación de flujos alternativos del trámite sin reescribir el caso, mediante la sustitución del **step** de decisión correspondiente; por ejemplo, un rechazo en lugar de la aprobación del tema daría lugar al flujo de tema no aprobado, y unas notas reprobatorias, al de proyecto no aprobado. Estos flujos no se implementaron en la suite, que cubre el camino favorable; quedan como extensión prevista por el diseño.

5.1.4. Criterios de Aceptación

Cada caso se consideró exitoso cuando se cumplió una condición observable en la interfaz al finalizar el flujo. La Tabla 3 reúne los criterios de los cuatro casos.

Tabla 3

Criterios de aceptación de los casos de prueba

Caso de prueba	Criterio de aceptación (Observable en la interfaz)
test_inicio_de_sesion_usuario_default	El enlace de cierre de sesión es visible tras la autenticación.
test_inicio_de_cambio_de_rol_administrador	El control de tipo de usuario muestra el texto "Administrador" tras el cambio.
test_crear_y_editar_noticia (creación)	Aparece el mensaje "La noticia se ha creado exitosamente" y la tarjeta con el título único es visible en el listado.
test_crear_y_editar_noticia (edición)	Aparece el mensaje "La noticia se ha modificado exitosamente" y la tarjeta con el título editado es visible en el listado.

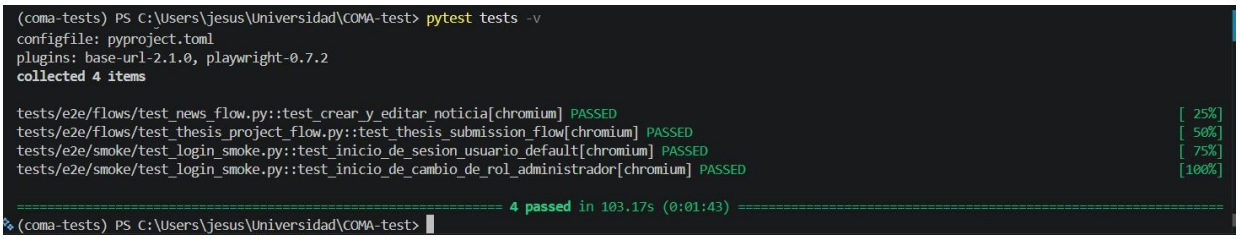
Caso de prueba	Criterio de aceptación (Observable en la interfaz)
test_thesis_submission_flow	En cada etapa, los datos del proyecto se muestran correctamente en la vista del rol correspondiente, y los autores ven la nota final como promedio de las dos calificaciones (ver Tabla 5.2).

5.1.5. Ejecución Sobre la Rama Develop

Una vez verificados los criterios, la suite se ejecutó completa sobre la rama *develop*. Los cuatro casos (dos de humo y dos de flujo) finalizaron de forma satisfactoria, con lo que quedó establecida la línea base funcional. La Figura 2 muestra la ejecución.

Figura 2

Ejecución exitosa de la suite coma-tests sobre la rama develop



5.2. Actividad 2: Evaluación del Rendimiento bajo Concurrencia

La Actividad 2 midió el rendimiento de COMA bajo carga concurrente para establecer la línea base de tiempos de respuesta. La medición se realizó con SISIFO, un entorno de pruebas de estrés construido para este trabajo de grado. En esta fase, SISIFO midió la rama `develop`, que constituye la línea base con la cual se comparará la rama de la reingeniería en la Fase 6.

5.2.1. Entorno de Pruebas: SISIFO

SISIFO despliega COMA y su base de datos MySQL en contenedores Docker aislados, una pareja por rama del repositorio. La rama, fijada como argumento de construcción (*BRANCH*), se compila en el contenedor con Ant sobre Tomcat 9 y JDK 17. Esta separación permite comparar dos ramas bajo la misma carga y atribuir las diferencias observadas a los cambios de código, no al entorno de ejecución.

La carga se generó con Locust (Locust, 2024), que simula usuarios concurrentes a nivel de protocolo HTTP, sin navegador. Se descartó un navegador real porque, por encima de veinte usuarios, los navegadores **headless** compiten por los recursos de la máquina y contaminan la medición. Los tiempos medidos corresponden, por lo tanto, al procesamiento del lado del servidor.

Cada usuario simulado repite, durante los 60 segundos del nivel, un ciclo que reproduce el costo de iniciar sesión en la plataforma. En cada vuelta limpia sus **cookies**, se autentica (solicita la página de login, obtiene las claves públicas RSA y envía las credenciales cifradas) y luego carga la página de inicio autenticada, que renderiza su menú de sesión. Entre una vuelta y la siguiente hace una pausa de entre uno y tres segundos. El diseño concentra la carga en el inicio de sesión, que es el camino donde se construye el árbol de menús, la estructura jerárquica del menú que la rutina de inicio de sesión arma para el usuario y guarda en la sesión. El nivel de concurrencia es el número de estos usuarios que operan en paralelo, cada uno con su propia sesión (Figura 3).

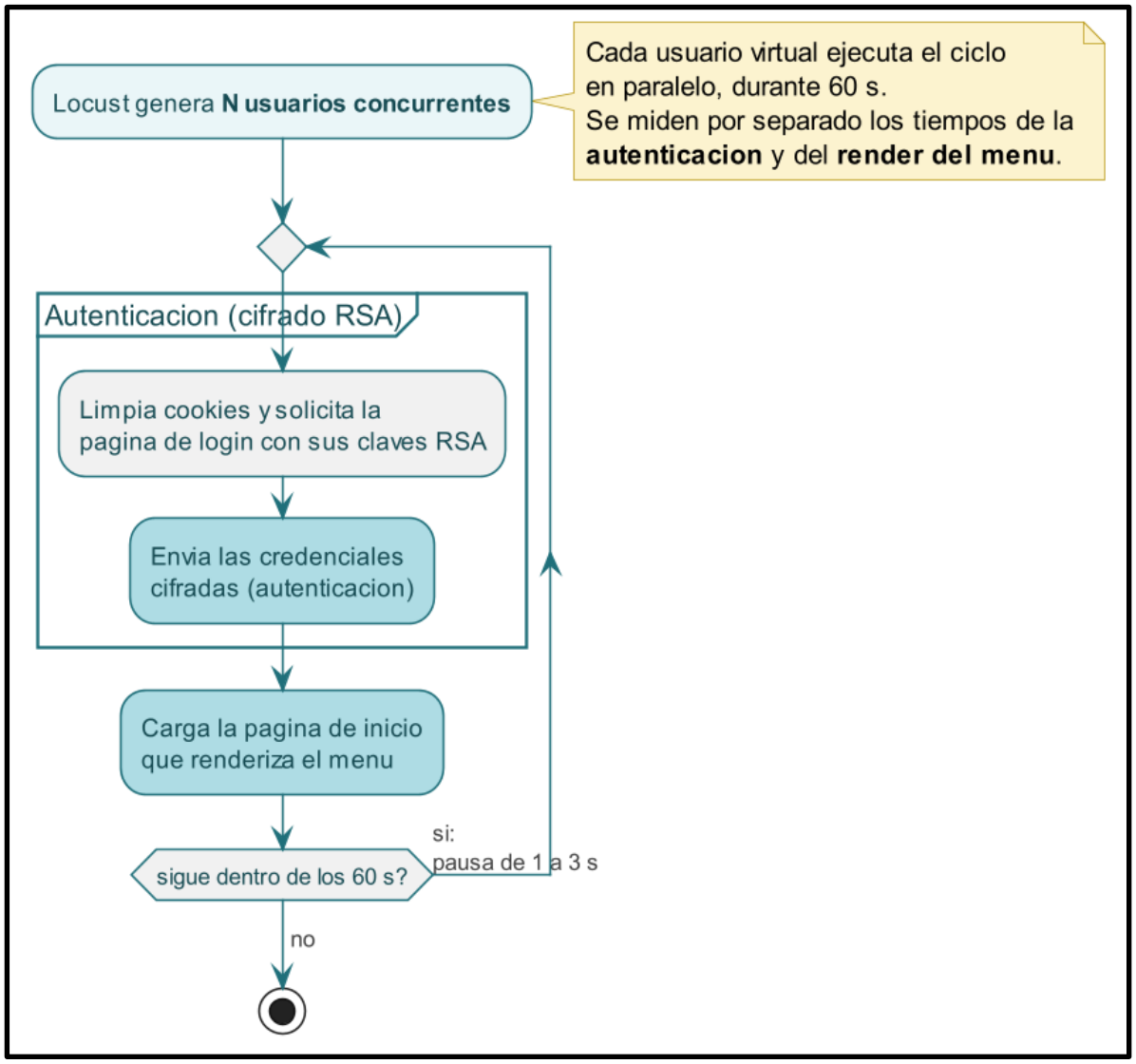
La prueba mide por separado dos pasos del ciclo: la **autenticación**, que construye el árbol de menús, y el **render del menú**, es decir, la carga de la página de inicio que lo muestra. Comparar ambos permite distinguir el costo de construir el menú del de mostrarlo.

SISIFO evalúa una serie configurable de niveles de concurrencia. Para este estudio se usaron, mediante la opción *--concurrency*, los niveles 1, 5, 10, 20, 50, 100 y 200 usuarios

simultáneos, con una duración de 60 segundos por nivel. Para cada paso del ciclo y cada nivel se registraron el número de solicitudes, la tasa de fallos, el tiempo de respuesta promedio, la mediana (p50), el percentil 95 (p95) y el percentil 99 (p99).

Figura 3

Esquema de la prueba de carga de SISIFO



5.2.2. Resultados de Rendimiento de un Obstáculo Previo: Agotamiento de Conexiones

Las primeras ejecuciones de las pruebas de estrés revelaron un obstáculo que impedía medir el rendimiento real de la plataforma, con caídas del servidor bajo carga moderada. La causa fue que COMA no cerraba de forma fiable las conexiones a la base de datos, y cada inicio de sesión abre varias. Bajo la carga sostenida de reautenticación, las conexiones y los puertos efímeros se

agotaban, y el servidor de base de datos respondía con el error "Too many connections" (Figura 4).

La Tabla 4 muestra el efecto sobre la rama *develop* sin la corrección. Desde 10 usuarios concurrentes, la página de login empezó a fallar, con una tasa que subió del 57 % al 92 % entre 10 y 200 usuarios, mientras la latencia de la autenticación trepaba por encima de los ocho segundos. En estas condiciones no era posible observar el comportamiento real de la gestión de menús, porque el agotamiento de conexiones dominaba la medición.

Tabla 4

Rendimiento de la rama develop sin el pool de conexiones

Concurrencia	Autenticación: promedio (ms)	Autenticación: fallos	Página de login: fallos
1 usuario	863	0%	0%
5 usuarios	1584	0%	0%
10 usuarios	1918	2,2%	56,9%
20 usuarios	2821	8,7%	76,7%
50 usuarios	8608	12,3%	88,5%
100 usuarios	8821	58,1%	90,8%
200 usuarios	6180	77,7%	91,8%

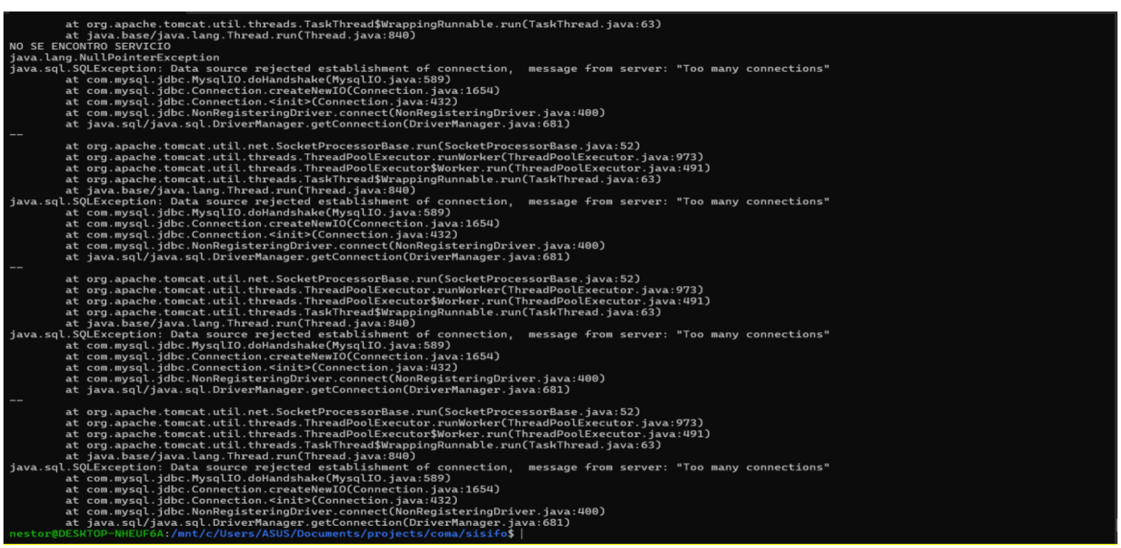
*Nota***.** Medición sobre la rama *develop* sin el pool de conexiones. El promedio de autenticación con 200 usuarios baja respecto al de 100 porque la mayoría de las solicitudes falla de inmediato y solo se promedian las que sobreviven.

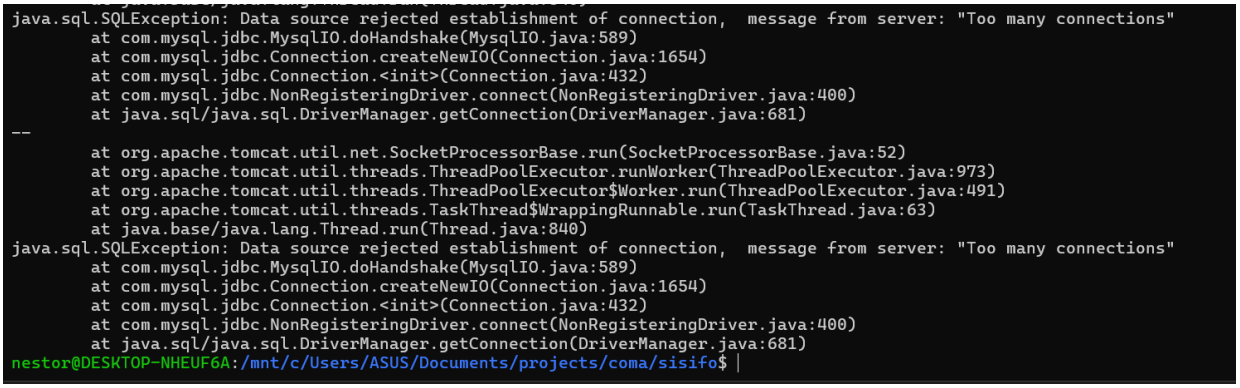
Para poder medir el rendimiento real, se resolvió primero este problema. Se introdujo un **pool** de conexiones, es decir, un conjunto de conexiones a la base de datos que se reutilizan en lugar de abrir una nueva en cada consulta. El pool envuelve las clases de acceso a datos de COMA y emplea *CachedRowSet*, un contenedor que conserva el resultado de la consulta en memoria de forma desconectada. Así, la conexión y su *ResultSet*, el objeto que representa el resultado de una consulta SQL y que retiene la conexión mientras se recorre, se cierran de inmediato y se liberan los puertos efímeros. Con la corrección, bajo la misma carga, los fallos desaparecieron y la latencia se redujo de forma notable, y el servidor pudo sostener los niveles de concurrencia más altos.

Esta corrección se aplicó por igual a la rama base y a la rama de la reingeniería, por lo que actúa como una constante del experimento y no como una variable. Las mediciones que siguen reflejan, por lo tanto, el comportamiento de la gestión de menús bajo concurrencia, una vez descartado el agotamiento de conexiones como factor de confusión.

Figura 4

Error "Too many connections" del servidor de base de datos durante las primeras pruebas de carga





5.2.3. Resultado de la Rama Develop

Resuelto el obstáculo de conexiones, se midió el rendimiento de la rama `develop` en los siete niveles de concurrencia, sobre los dos pasos del ciclo descrito en la sección 5.2.1. La Tabla 5 recoge los tiempos de la autenticación y la Tabla 6 los del render del menú; la Figura 5 contrasta ambos frente al nivel de concurrencia.

Tabla 5

Tiempos de respuesta de autenticación en la rama develop

Concurrencia	Requests	Fallos	Avg (ms)	Mediana (ms)	p95 (ms)	p99 (ms)
1 usuario	28	0.0%	108	110	120	150
5 usuarios	144	0.0%	110	110	130	140
10 usuarios	286	0.0%	112	110	140	160
20 usuarios	563	0.0%	118	110	140	240
50 usuarios	1387	0.0%	138	120	170	510
100 usuarios	2689	0.0%	175	160	240	450

Concurrencia	Requests	Fallos	Avg (ms)	Mediana (ms)	p95 (ms)	p99 (ms)
200 usuarios	3910	0.0%	840	750	1800	2200

Nota. Medición sobre la rama *develop* con el pool de conexiones aplicado.

Tabla 6

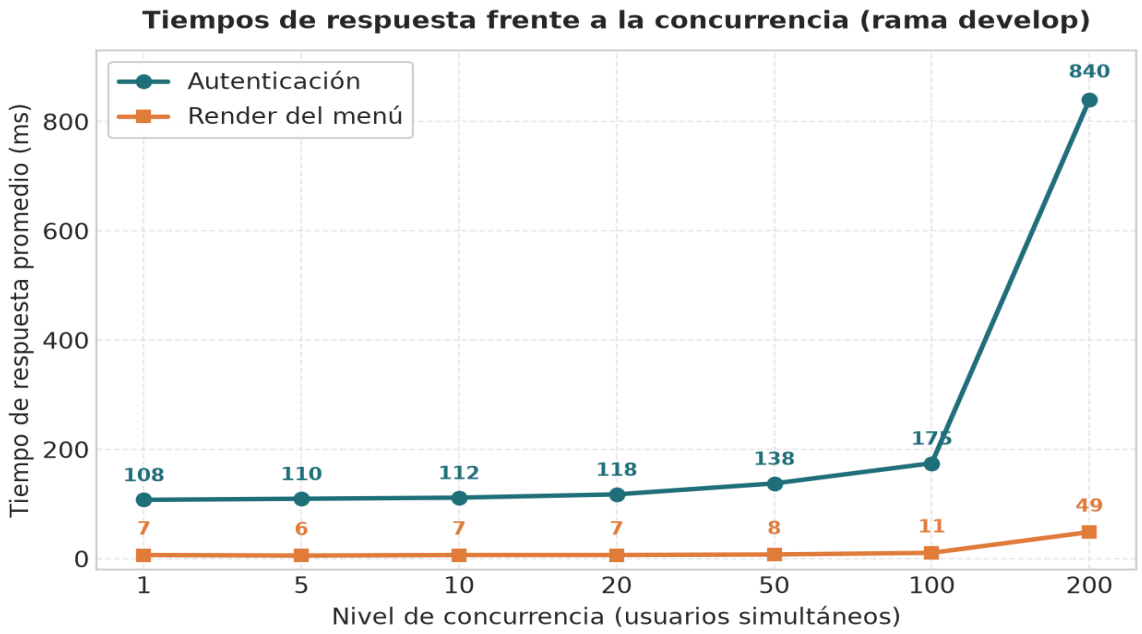
Tiempos de respuesta del render del menú (página de inicio) en la rama develop

Concurrencia	Requests	Fallos	Avg (ms)	Mediana (ms)	p95 (ms)	p99 (ms)
1 usuario	28	0.0%	7	7	8	9
5 usuarios	143	0.0%	6	6	7	9
10 usuarios	286	0.0%	7	6	8	12
20 usuarios	563	0.0%	7	6	9	16
50 usuarios	1386	0.0%	8	7	12	29
100 usuarios	2687	0.0%	11	9	19	34
200 usuarios	3905	0.0%	49	41	110	160

Nota. Medición sobre la rama *develop* con el pool de conexiones aplicado.

Figura 5

Error "Too many connections" del servidor de base de datos durante las primeras pruebas de carga



5.2.4. Análisis de los Resultados

Descartado el agotamiento de conexiones, los datos sitúan el cuello de botella del rendimiento en la autenticación.

La autenticación se degrada con la concurrencia**.** El tiempo de autenticación se mantuvo cerca de 110 ms hasta 20 usuarios, subió a 175 ms con 100 y saltó a 840 ms con 200, con un percentil 99 de 2.200 ms en el nivel más alto. La autenticación es el punto donde la plataforma construye el árbol de menús de la sesión mediante varias consultas a la base de datos; esa construcción, repetida en cada inicio de sesión, es la operación que no escala.

El costo se concentra en construir el menú. El render de la página de inicio se mantuvo entre 6 ms y 49 ms en todo el rango, muy por debajo de los 840 ms de la autenticación en el nivel

más alto. Ambos pasos recorren el mismo árbol de menús, así que la diferencia aísla el costo en su construcción, que se repite en cada inicio de sesión. El costo de la gestión de menús recae, por lo tanto, en la autenticación.

El sistema atendió toda la carga sin fallos. Con el pool de conexiones aplicado, los dos pasos registraron 0 % de fallos en los siete niveles, hasta 200 usuarios. El único efecto de la concurrencia fue el aumento de la latencia de la autenticación.

Este diagnóstico orienta la solución de la reingeniería. Como la construcción del árbol de menús se repite en cada inicio de sesión, compartir una única estructura entre las sesiones, mediante el gestor de menús de la Fase 3, debería reducir la latencia. La comparación cuantitativa entre la rama base y la rama de la reingeniería se presenta en la Fase 6.

5.3. Actividad 3: Análisis del Consumo de Memoria de Sesión

La Actividad 2 mostró que la latencia de la autenticación crece con la concurrencia y la atribuyó a la construcción del árbol de menús en cada inicio de sesión. Esta actividad cuantifica el costo en memoria de ese patrón. Se realizó en dos pasos: primero se analizó el código de la rutina de inicio de sesión para identificar qué estructuras guarda cada sesión y por qué se duplican; luego se capturó el estado del **heap** (la zona de memoria donde la máquina virtual Java mantiene los objetos mientras siguen en uso) de la rama *develop* bajo carga, para contar cuántas instancias de esas estructuras coexistían.

5.3.1. Análisis del Código: el Origen de la Duplicación

El primer paso fue revisar la rutina de inicio de sesión de `develop` para determinar qué se almacena en cada sesión. Al autenticarse un usuario, esa rutina arma sus menús y los deja en la sesión HTTP por dos vías:

- *beans.Permisos.menuprincipal* consulta los servicios autorizados del usuario en la tabla *TB_Menu* y devuelve una matriz de texto (*String[][]*), que se guarda en el objeto del usuario.
- *beans.ObjMenu.ArbolServicios arma con esos mismos servicios un árbol de objetos y lo guarda en la sesión bajo el atributo arbolServicios. Cada nodo del árbol es una instancia de beans.Servicio (un servicio de la plataforma) y el árbol entero es una instancia de beans.Menu que los agrupa.*

Dos rasgos de este código explican el consumo de memoria. Primero, *ObjMenu.ArbolServicios* reconstruye el árbol en cada llamada, sin una caché que se comparta entre sesiones, asi que **cada sesión autenticada guarda su propia copia**. Segundo, el mismo conjunto de servicios queda representado **dos veces** en la sesión, en memorias independientes: como la matriz de texto del usuario y como el árbol de objetos *beans.Servicio*.

Este hallazgo fijó qué medir: *beans.Servicio* y *beans.Menu* (el árbol replicado por sesión), *org.apache.catalina.session.StandardSession* (la sesión que lo contiene) y *java.lang.String* junto con los arreglos de bytes (*[B*), que dominan el consumo absoluto del **heap** y provienen en buena parte de los atributos de texto de los servicios y de las matrices.

5.3.2. Metodología de Captura

La medición se realizó sobre el mismo entorno SISIFO de la Actividad 2, con un componente de instrumentación que actúa sobre la máquina virtual Java del contenedor. El procedimiento fue:

1. Someter la rama a una carga de 200 usuarios concurrentes durante 60 segundos, con el mismo ciclo de reautenticación de la Actividad 2, en el que cada usuario limpia sus **cookies** y

vuelve a iniciar sesión en cada vuelta, lo que crea una sesión nueva cada vez. Así se acumula en el **heap** un número representativo de sesiones, que es lo que esta prueba busca medir.

2. Forzar una recolección de basura sobre la máquina virtual con *jcmd 1 GC.run*, para que en el **heap** permanecieran solo los objetos aún referenciados y se descartaran los transitorios

3. Capturar un histograma de clases con *jcmd 1 GC.class_histogram*, que reporta, para cada clase, su número de instancias y su tamaño superficial (**shallow size**), es decir, la memoria que ocupa el objeto en sí, sin contar los objetos que referencia.

El entorno se recreó desde cero antes de cada ejecución, sin que ninguna sesión de una prueba previa contaminara la siguiente. La métrica central de esta actividad es el número de instancias por clase, no solo los megabytes, porque lo que confirma la duplicación es cuántas copias del árbol coexisten en memoria.

5.3.3. Resultados del Histograma rama `Develop`

Con el pool de conexiones aplicado, la rama `develop` sostuvo la carga de reautenticación, con 5.067 inicios de sesión sin fallos y 5.096 sesiones activas (Tabla 7). En los 60 segundos de la prueba, el *heap* creció hasta 1,40 GB, porque cada inicio de sesión añade su copia del árbol de menús y las copias se acumulan mientras las sesiones siguen vivas.

Tabla 7

Contexto de la prueba de memoria sobre la rama develop

Métrica	Valor
Peticiones totales	15.241
Fallos	0 (0.0%)

Métrica	Valor
Logins exitosos	5.067
Sesiones activas	5.096

La Tabla 8 reúne las clases relevantes del histograma del **heap**.

Tabla 8

Instancias de las clases relevantes en el heap de la rama develop

Posición	Clase	Instancias	Tamaño superficial
1	*[B* (arreglos de bytes)	22.230.373	750,21 MB
2	*java.lang.String*	23.091.092	528,51 MB
9	*beans.Servicio*	1.480.608	45,18 MB
8	*beans.Usuarios*	10.184	1,94 MB
17	*org.apache.catalina.session.StandardSession*	5.096	437,94 KB
25	*muisca.seguridad.RSA*	5.096	159,25 KB
327	*beans.Menu*	5.088	79,50 KB

Nota. Tamaño superficial (**shallow**) capturado con *jcmd* tras forzar la recolección de basura. El **heap** total fue de 1.40 GB, con 47.350.736 instancias y 2.284 clases cargadas.

5.3.4. Análisis de los resultados

El histograma confirma la duplicación del árbol de menús por sesión, ahora a gran escala.

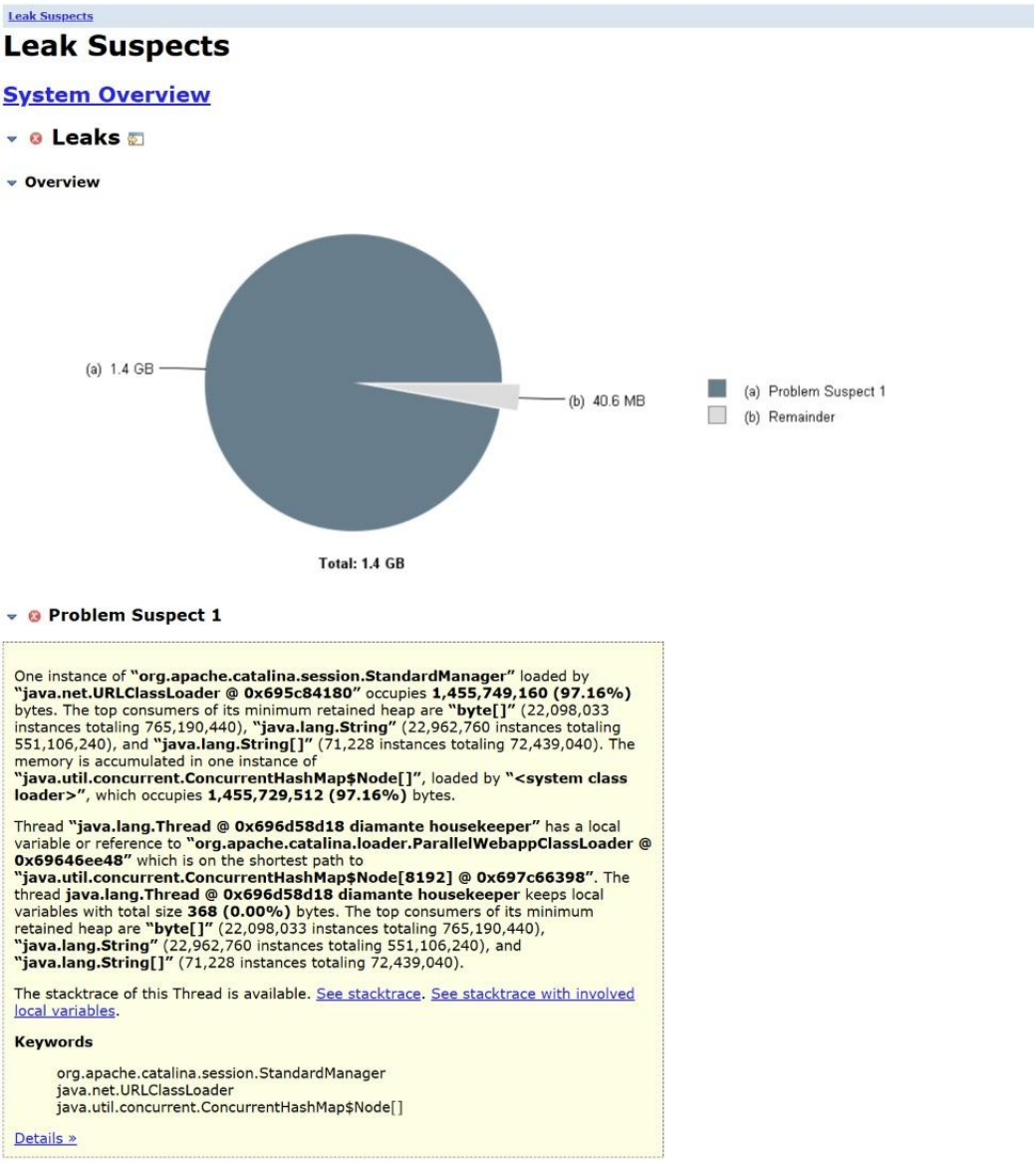
Cada sesión guarda su propia copia del árbol. La clase *beans.Servicio*, que representa un nodo del menú, presenta 1.480.608 instancias, porque el árbol de menús se reconstruye y se guarda entero en cada sesión, sin compartirse, con lo que los nodos se multiplican con el número de sesiones. Si las sesiones compartieran una sola estructura, bastaría un único conjunto de nodos para todas.

La duplicación del menú hace crecer el heap. El análisis de código de la sección 5.3.1 mostró que cada sesión guarda su menú como texto, en los campos de cada *beans.Servicio* y en la matriz de servicios del usuario. Esa estructura replicada es la que llena el **heap** de texto, donde las cadenas (528,51 MB) y los arreglos de bytes que las respaldan en la máquina virtual Java (750,21 MB) ocupan más de 1,2 GB de los 1,40 GB totales.

El peso recae por completo en las sesiones. Además del histograma, Eclipse MAT produce un informe de sospechosos de fuga, que señala al mayor retenedor del **heap.** Ese principal sospechoso es el administrador de sesiones de Tomcat (*StandardManager*), que conserva todas las sesiones activas y retiene 1,36 GB, el 97,16 % del **heap** (Figura 6). Su contenido es casi todo texto: 730 MB en arreglos de bytes y 525 MB en cadenas. Aquí el señalamiento es certero, pues lo que el administrador retiene son las copias del menú que cada sesión guarda, y casi todo el **heap** es memoria de las sesiones.

Figura 6

Distribución del tamaño retenido del heap por dominador (Eclipse MAT), rama develop



A la replicación del menú se suma otra fuente de estado por sesión, las 5.096 instancias de *muisca.seguridad.RSA*, que corresponden a un par de claves por sesión, generado al crearla y retenido durante toda su vida. No es el objeto de la reingeniería de menús, pero confirma que el modelo vigente acumula estado en cada sesión.

Estos resultados establecen la línea base de memoria contra la cual se comparará, en la Fase 6, la arquitectura reingenierizada, en la que el gestor de menús de la Fase 3 comparte una única instancia del árbol entre las sesiones y elimina la duplicación medida aquí.

5.4. Actividad 4: Análisis del modelo de control de acceso

Esta actividad diagnosticó la seguridad del control de acceso de COMA. Se revisó el código de la rama `develop` que decide los permisos en cada petición, para caracterizar cómo se autoriza el acceso a los servicios y qué debilidades presenta ese mecanismo. El análisis se centra en el control que decide a qué servicios accede cada usuario según su tipo y su clasificador, la categoría o el perfil al que pertenece; la lógica con que cada servicio trata sus datos, una vez concedido el acceso, pertenece a cada módulo y queda fuera de este alcance.

5.4.1. El patrón de control de acceso

El acceso a los recursos protegidos lo decide una rutina de validación que cada página incluye al inicio. En cada petición, la rutina lee al usuario desde la sesión, determina el servicio solicitado y comprueba que el usuario tenga permiso para ese servicio contra el árbol de menús de la sesión. Ese árbol procede de *Permisos.menuprincipal*, que une *TB_Menu*, la tabla que describe la navegación, con la tabla de autorización que define los permisos: *TR_Autorizacion_Usr* para los usuarios por categoría, incluido el invitado, y *TR_Autorizacion* para los perfiles administrativos. La misma estructura sirve, así, para navegar y para autorizar.

El patrón está disperso. La rutina se incluye al inicio de 969 páginas y, en muchas otras, su lógica se copió directamente en lugar de incluirse, hasta sumar alrededor de 1.540 páginas con la validación repartida por el sistema en varias versiones. La comprobación, además, se reparte entre

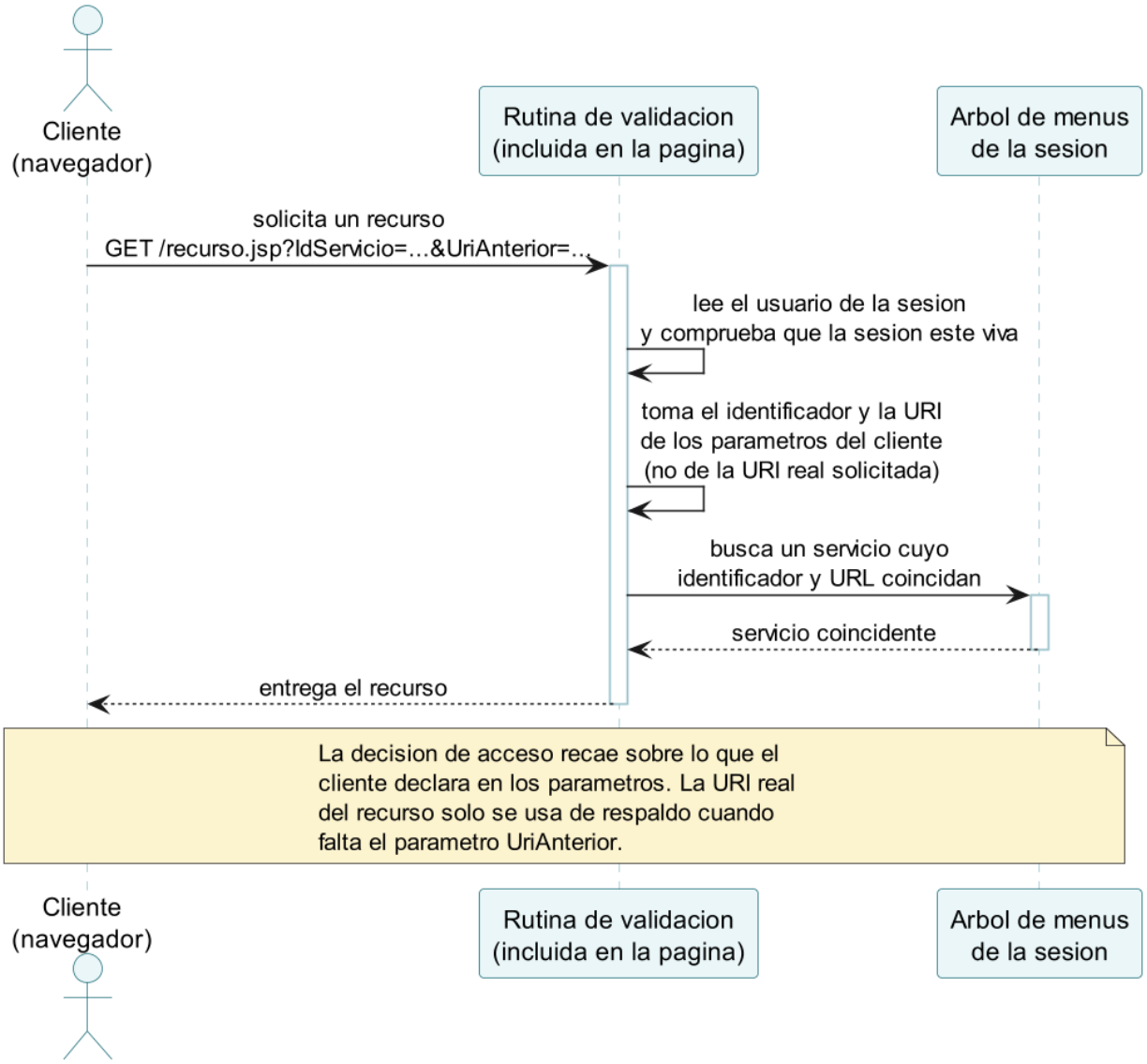
métodos distintos: *ValidaServicio* y *validaServicioGeneral* operan sobre el árbol de objetos de la sesión, y *ValidarServicio* sobre la matriz de servicios del usuario. No existe una única fuente de verdad para el control de acceso, lo que hace inviable corregirlo de forma consistente.

5.4.2. La Validación se Decide con los Parametros del Cliente

La revisión de la rutina reveló una vulnerabilidad de control de acceso. Los dos valores con los que decide el permiso, el identificador del servicio (por lo general del parámetro *IdServicio*) y la URI de referencia (del parámetro *UriAnterior*), se toman de la petición HTTP, que el cliente controla. La URI real del recurso solicitado solo se usa como respaldo, y solo cuando *UriAnterior* no viene en la petición. La decisión de seguridad recae, así, sobre el servicio que el cliente declara, no sobre el recurso que realmente se entrega. La Figura 7 muestra esta secuencia.

Figura 7

Validación de acceso decidida sobre los parámetros enviados por el cliente



5.4.3. Vector de Ataque

La vulnerabilidad se puede explotar sin credenciales. La rutina solo rechaza una petición cuando la sesión está muerta, es decir, cuando faltan los atributos del usuario. COMA nunca deja sin sesión a un visitante, porque al navegar por el portal público sin autenticarse el sistema crea

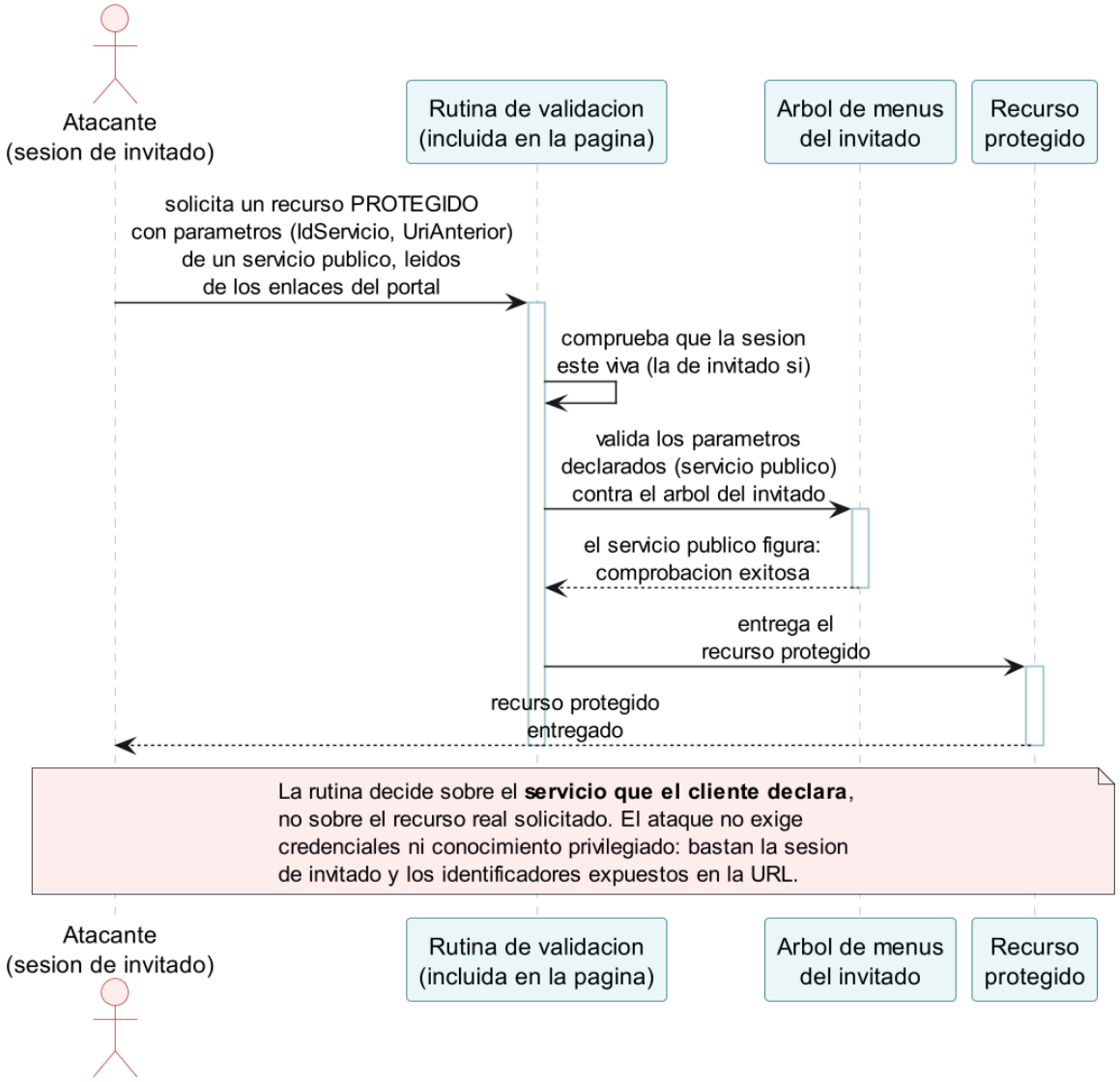
una sesión de invitado, asociada a un usuario público cuyo árbol de menus contiene solo recursos de acceso libre. Cualquier visitante obtiene así una sesión viva con al menos un servicio legítimo, que es la única condición que la rutina exige antes de validar.

El atacante necesita, además, el identificador de un servicio permitido, y obtenerlo es trivial, ya que los parámetros *IdServicio* y *UriAnterior* viajan en texto plano en la URL de cada enlace, y basta navegar como invitado a un contenido público para leerlo. Con una sesión de invitado y ese identificador, el atacante solicita un recurso protegido y le adjunta el identificador y la URI de un servicio público al que sí tiene acceso. Como la rutina valida sobre esos parámetros, la comprobación tiene éxito y el recurso protegido se entrega. La Figura 8 muestra este vector de ataque.

El ataque no exige credenciales ni conocimiento privilegiado, pues bastan la sesión de invitado y los identificadores expuestos en la URL. Esta deficiencia corresponde a la categoría de control de acceso roto (A01:2025, **Broken Access Control**) del OWASP Top Ten (OWASP, 2025a).

Figura 8

Vector de ataque que explota la validación por parámetros



5.4.4. Alcance de la Exposición

El diagnóstico encontró dos niveles de exposición con una misma causa, la dependencia total del control de acceso respecto de *TB_Menu*, una tabla concebida para describir la navegación y no la autorización.

En el primer nivel, la protección existe pero se puede eludir. Abarca las alrededor de 1.540 páginas que ejecutan la validación, sea por inclusión de la rutina o por copia de su lógica; en ellas hay comprobación, pero la validación por parámetros descrita arriba la sortea.

El segundo nivel es más grave y se descubrió al medir el alcance del primero. La plataforma tiene 1.893 páginas JSP, mientras que el árbol de menús sobre el que opera el control de acceso se construye solo con las filas de TB_Menu que corresponden a ítems del menú, las opciones que aparecen en el menú de navegación, un subconjunto mucho menor. La causa es arquitectónica. Una página no necesita estar en el menú para ser alcanzable, porque los recursos se enlazan entre sí con referencias directas, redirecciones, inclusiones y peticiones AJAX que llevan de una página a otra sin pasar por el menú principal. Muchas páginas internas resultan accesibles por la navegación, pero nunca se registraron como servicios. Un ejemplo es el trámite de trabajo de grado, que se inicia desde una opción del menú pero cuyas etapas posteriores transcurren por páginas de proceso que no figuran en él. Como la validación se hace sobre el servicio que el cliente declara, una página no registrada no define permisos propios, y su acceso queda gobernado por una herencia de permisos laxa que cualquier invitado satisface. La protección efectiva de una página dependía, entonces, de dos acciones manuales y propensas al olvido: incluir la rutina y registrar el recurso en TB_Menu.

5.4.5. Síntesis del diagnóstico

El diagnóstico estableció que las deficiencias de seguridad de COMA tienen una causa estructural común, el acoplamiento de la navegación, los servicios y los permisos alrededor de *TB_Menu* y de la rutina repetida en cada página. Esa raíz, de la que surgen los dos problemas anteriores, justifica las dos líneas de seguridad de la reingeniería: centralizar el control de acceso

en un middleware (Fase 2) y rediseñar la autorización en la Fase 3, para que el menú decida solo qué ve el usuario y el permiso se valide sobre el recurso que realmente se solicita, exista o no en el menú.

6. Fase 2: Rediseño de la rutina de inicio de sesión mediante middleware

La Fase 1 mostró que el control de acceso de COMA estaba disperso, pues cada página lo resolvía por su cuenta, con la rutina de validación incluida desde un archivo compartido o copiada dentro de la página (sección 5.4). La Fase 2 centralizó esa validación en un único punto, un middleware que intercepta las peticiones de toda la plataforma. Reprodujo el proceso de validación heredado sin modificar su lógica y dejó la corrección de la vulnerabilidad y del consumo de memoria para la Fase 3. La centralización se realizó en dos actividades: retirar las validaciones dispersas de los archivos (Actividad 5) y unificarlas en el filtro (Actividad 6).

6.1. Actividad 5: Eliminación de las rutinas de inicio de sesión independientes

En el primer paso de esa centralización, esta actividad retiró de los archivos de la plataforma la validación de acceso que cada página ejecutaba por su cuenta, para que ninguna verificación heredada quedara compitiendo con el filtro.

La intervención abarcó alrededor de 1.540 archivos JSP. La validación no seguía un patrón uniforme. En muchas páginas se incorporaba con una directiva de inclusión cuya ruta relativa cambiaba según la profundidad del archivo en el árbol de directorios, y tenía una variante aparte para las peticiones AJAX. En otras estaba copiada dentro de la página, a menudo delimitada por comentarios que la rotulaban y entremezclada con el código propio del servicio. El método de

comprobación tampoco era único, porque la rutina incluida validaba sobre el árbol de objetos de la sesión, mientras que algunas copias lo hacían sobre la matriz de servicios del usuario, y el rechazo se resolvía de distintas maneras. La Figura 9 muestra un ejemplo de cada forma.

Figura 9

Formas en que aparecía la validación de acceso antes de la centralización

informacionEgresadoEstudiante.jsp · la validación se incorpora con una sola directiva de inclusión

```jsp
<%@ page import="beans.Usuarios, beans.Menu" %>
<%@ page contentType="text/html; charset=iso-8859-1" language="java" %>
<jsp:useBean id="usuarios"  class="beans.Usuarios"  scope="session"/>
<jsp:useBean id="Permisos"  class="beans.Permisos"  scope="page"/>
<jsp:useBean id="Servicios" class="beans.Servicios" scope="page"/>
<%
    String tipoUsuario, servicio, Uri;
    String[] parametro = Parametrizacion.Parametros();
%>

<%-- (a) La validación se incorpora con UNA SOLA LÍNEA: --%>
<%-- la directiva de inclusión, mediante una ruta relativa --%>
<%@ include file="../Logueo/rutinaLogueo.jsp" %>

<html>
    <head>
        <title>Información del egresado</title>
    </head>
    <body>
        <%-- A partir de aquí, el contenido propio de la página --%>
        ...
    </body>
</html>
```

rutinaLogueo.jsp · rutina de validación copiada en línea dentro de la página

```jsp
<%-- (b) Rutina de validación COPIADA EN LÍNEA dentro de la página JSP --%>
<%
    // El servicio y la URI se toman de los PARÁMETROS del cliente
    servicio = "" + request.getParameter("IdServicio");
    Uri      = "" + request.getParameter("UriAnterior");
    if (Uri.equals("null") || Uri == null) {
        Uri = request.getRequestURI();              // la URI real solo de respaldo
    }

    // Usuario y árbol de menús guardados en la sesión
    usuarios           = (beans.Usuarios) session.getAttribute("usuario");
    tipoUsuario        = (String) session.getAttribute("tipoUsuario");
    Menu menuServicios = (Menu) session.getAttribute("menuPrincipal");

    if (usuarios == null || tipoUsuario == null || menuServicios == null) {
        // Sesión muerta: redirige al login
%>
<jsp:forward page="<%=Path%>"><jsp:param name="campo" value="caduca"/></jsp:forward>
<%
    } else {
        // Valida el servicio DECLARADO por el cliente contra el menú de la sesión
        if (menuServicios.checkPermissions(servicio)) {
            out.write(Servicios.EncabezadoDeServicio(servicio));   // acceso permitido
        } else {
%>
<jsp:forward page="<%=Path%>"><jsp:param name="campo" value="noAutorizado"/></jsp:forward>
<%      }
    }
%>
```

Esa heterogeneidad descartó el reemplazo automático, porque no había una cadena única ni un bloque uniforme que buscar y sustituir, así que un patrón general habría pasado por alto las copias atípicas o habría borrado código que la página todavía necesitaba. La eliminación se hizo, por lo tanto, de forma manual, archivo por archivo.

El criterio en cada archivo fue distinguir lo que pertenecía al mecanismo de autenticación de lo que era propio del servicio. Se retiraron solo las instrucciones de validación de sesión, tipo de usuario y permisos, con los objetos y las variables de sesión que únicamente les servían de apoyo, y se conservaron las que mostraban información, procesaban formularios o ejecutaban las acciones de la página.

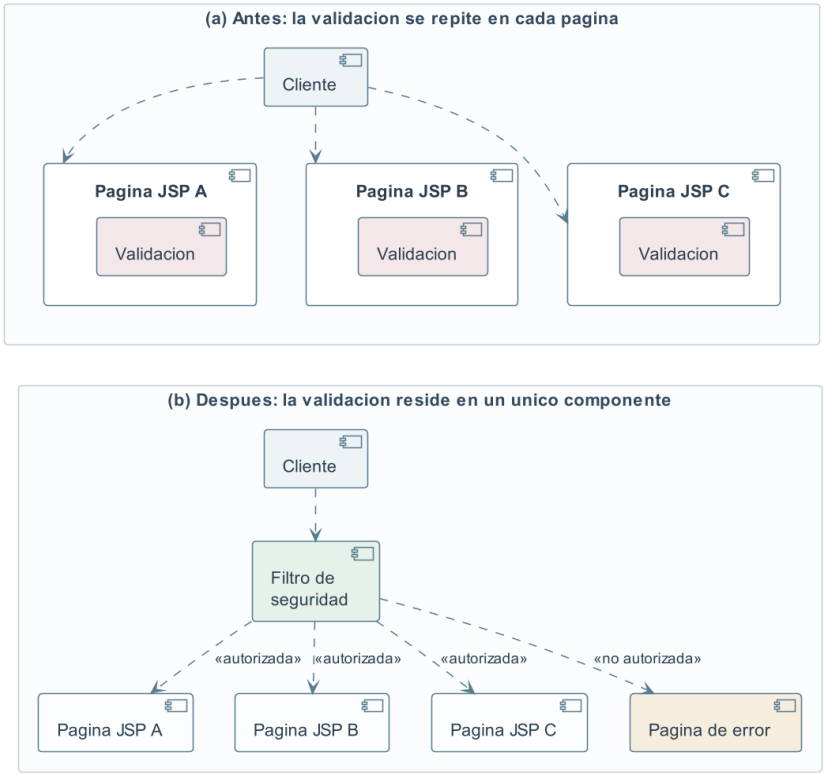
Con ello desapareció la duplicación de lógica de seguridad que la Fase 1 había señalado.

6.2. Actividad 6: Unificación de la Rutinas en el Middleware de Autenticación

La Actividad 6 reunió la validación que la Actividad 5 había retirado de las páginas en un solo componente, un filtro de servlet, el filtro de seguridad de la plataforma, que intercepta todas las peticiones antes de que alcancen el recurso solicitado. El control de acceso pasó de repetirse en cada archivo a resolverse en un único punto. La Figura 10 contrasta los dos esquemas.

Figura 10

El filtro como punto único de validación frente al esquema anterior repartido por página



El filtro reproduce el proceso heredado, sin modificar su lógica. En cada petición recupera de la sesión el árbol de menús del usuario, determina el servicio solicitado con los mismos parámetros del cliente que empleaba la rutina anterior y comprueba el permiso contra ese árbol. Si la comprobación falla, responde con un error de autorización; si no, deja pasar la petición hacia el recurso. El control de acceso conservó su comportamiento y solo cambió de ubicación, de las páginas al filtro.

Con la validación concentrada, el control de acceso ganó la fuente única de verdad que la Fase 1 había echado en falta (sección 5.4), y rediseñarla dejó de exigir cambios en las páginas. El

middleware conserva por ahora la lógica heredada, con lo que la vulnerabilidad y el costo de memoria que la Fase 1 diagnosticó persisten, a la espera del rediseño de la Fase 3.

7. Fase 3: Desarrollo del Gestor de Menús, Servicios y Permisos

La Fase 1 mostró que en COMA la navegación y el control de acceso vivían entrelazados en una sola estructura, el árbol de menús que cada sesión reconstruía al autenticarse y sobre el que, además, se decidía el permiso. La Fase 3 separa esas dos responsabilidades y las traslada a memoria para servir al middleware. En esta fase quedan resueltos por completo el consumo de memoria y la latencia que la Fase 1 había medido; la corrección de la seguridad comienza aquí, al validar el acceso sobre el recurso realmente solicitado y no sobre lo que el cliente declara, y se completa en las fases siguientes, al registrar todos los servicios (Fase 4) y separar el modelo en la base de datos (Fase 5).

7.1. Actividad 7: El Módulo de Gestión de Menús, Servicios y Permisos

El módulo de gestión se organizó en dos gestores con responsabilidades separadas: el gestor de menús, que construye la navegación, y el gestor de servicios y permisos, que decide el acceso. La división separa la navegación de la seguridad, y el menú se ocupa solo de qué ve cada usuario y la autorización de a qué puede acceder. Ambos gestores mantienen su información en memoria y la comparten entre las sesiones, que es el cambio del que dependen las mejoras de la fase.La separación también ordena el vocabulario. En el modelo anterior, cada entrada del menú era un servicio, y ese mismo servicio definía a la vez qué se mostraba en el menú y qué requería permiso. Al separar las dos responsabilidades, las entradas de navegación pasan a llamarse ítems

del menú y el nombre servicio queda reservado para los recursos sujetos a control de acceso. Un ítem del menú es una opción de navegación, con su nombre, su lugar en la jerarquía y la dirección a la que lleva; un servicio es un recurso que el control de acceso protege, figure o no en el menú.

7.1.1. El Gestor de Menús

El gestor de menús, en el paquete *muisca.menu*, construye la navegación y la conserva en memoria para todas las sesiones. Eliminar la duplicación del menú que cada sesión guardaba, la causa principal del consumo de memoria diagnosticado en la sección 5.3, es el objetivo central que cumple esta actividad. Para dimensionar el cambio conviene contrastar cómo se construía el menú antes y cómo se construye ahora.

7.1.1.1 El Esquema Anterior. En el modelo anterior, cada usuario obtenía su menú al autenticarse, cuando la rutina de inicio de sesión lo construía una vez y lo guardaba en la sesión; desde ahí lo leía cada página que lo mostraba. COMA distingue dos tipos de usuario: el usuario general y el administrador. Una misma cuenta administradora puede alternar entre los dos en una misma sesión, sin volver a autenticarse, para pasar de su acceso general al de administrador. Los permisos del menú no se asignan usuario por usuario, sino por grupos: los usuarios generales se reúnen en categorías y los administradores en perfiles. A ese grupo, la categoría o el perfil, se le llama el clasificador del usuario, y a él se conceden las entradas que cada uno puede ver. Esas entradas, los servicios de entonces, eran la materia con la que se armaba el menú.

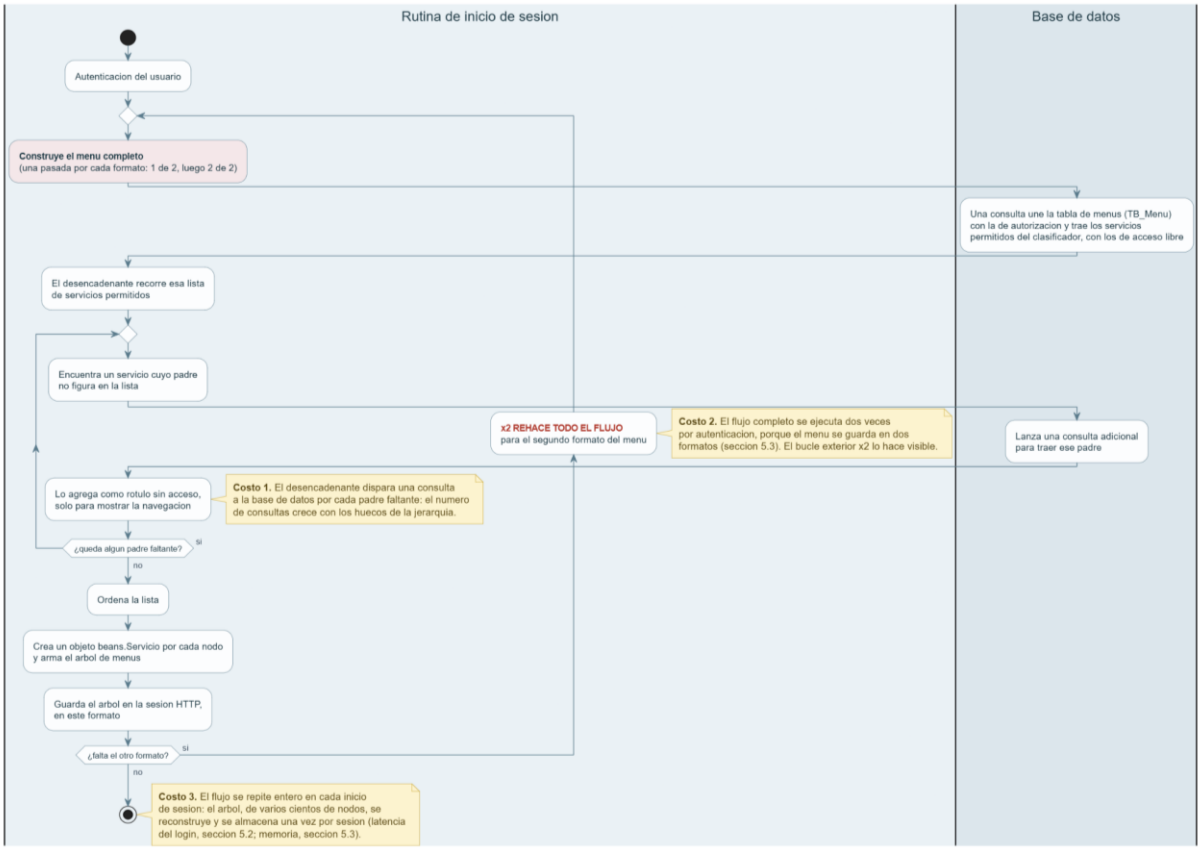
La construcción operaba en dos pasos. Primero, una consulta combinaba la tabla de menús con la tabla de autorización para traer los servicios que la categoría o el perfil del usuario tenía permitidos, junto con los de acceso libre. Como esa consulta solo traía los permitidos, faltaban los

servicios intermedios que enlazan cada opción con su raíz; un segundo paso, llamado desencadenante, los completaba para la navegación. Por cada servicio cuyo padre no estuviera en la lista, lo traía de la base de datos con una consulta aparte y lo agregaba como un rótulo, sin su dirección de acceso, para que el padre apareciera en el árbol y mostrara el camino hasta la opción permitida pero no quedara accesible. Completar la jerarquía era, así, una cuestión de navegación y no de permisos.

Ese encadenamiento de consultas, una por cada padre faltante, traía en fragmentos datos que ya residían en una sola tabla, y la construcción completa se ejecutaba dos veces por autenticación, porque el menú se guardaba en los dos formatos que describió la sección 5.3. Con la lista completa y ordenada, la rutina creaba un objeto *beans.Servicio* por cada uno y los reunía en el árbol que quedaba en la sesión. El resultado fue el que midió la Fase 1, un árbol de varios cientos de nodos por sesión, repetido en cada una, que llegó a dominar la memoria (sección 5.3), y un costo de construcción que recaía sobre la autenticación (sección 5.2).

Figura 11

Construcción del menú en el esquema anterior, repetida en cada inicio de sesión



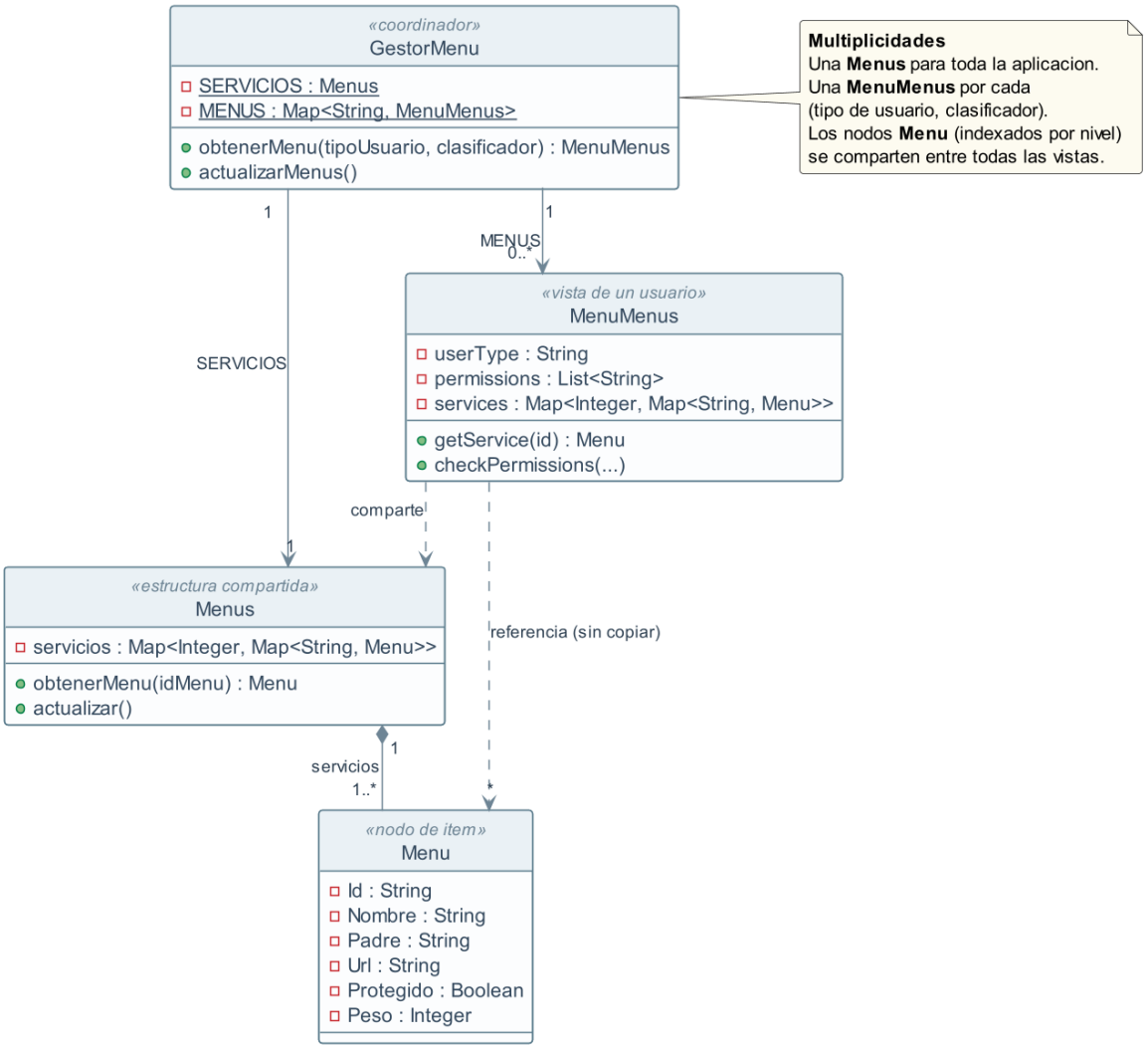
7.1.1.2 La Estructura Compartida de Memoria. El gestor sustituye ese esquema por una sola estructura en memoria, cargada una vez y compartida por toda la plataforma. Al arrancar, una consulta trae todos los ítems del menú activos, sin filtrarlos por usuario, y los organiza en una tabla por niveles, en la que cada nivel de la jerarquía tiene un índice de sus ítems por identificador. Cada ítem del menú es un nodo inmutable, de la clase *Menu*, que guarda su identificador, su nombre, el identificador de su padre, las URL que le corresponden, su peso para el orden y si está protegido.

Esa estructura es la única copia de los menús en la aplicación. No se filtra ni se guarda por usuario; las vistas que cada usuario ve se derivan de ella y la referencian en lugar de copiarla. Cuatro clases del paquete *muisca.menu* componen el gestor: el nodo *Menu* que se acaba de

describir, la estructura compartida *Menus* que guarda los nodos, la vista *MenuMenus* que se arma sobre ella para un usuario, y el coordinador *GestorMenu* que entrega y mantiene en caché esas vistas. La Figura 12 las modela.

Figura 12

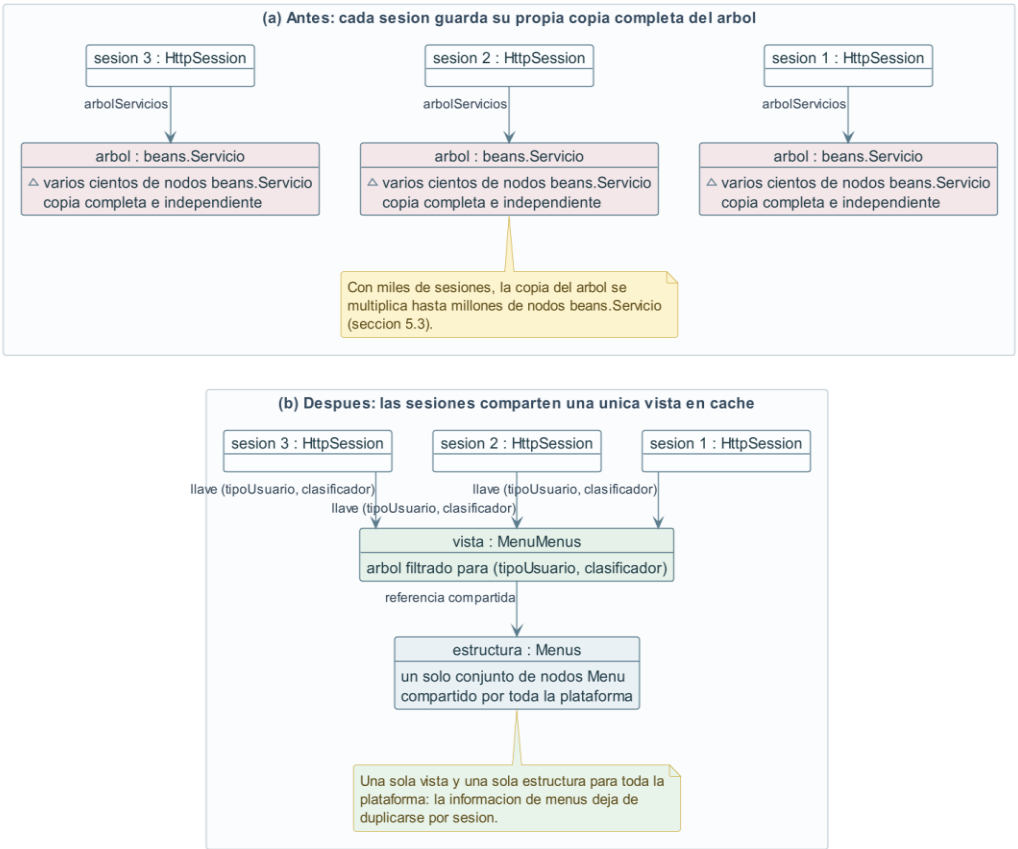
Estructura del gestor de menús



7.1.1.3 Compartición y Caché de las Vistas. El gestor guarda una vista por cada combinación de tipo de usuario y clasificador, no una por sesión, así que todas las sesiones con el mismo tipo de usuario y clasificador reciben la misma. La primera vez que se pide una vista, el gestor la arma y la guarda en su caché; en las peticiones siguientes la entrega ya armada, sin volver a construirla. Una sesión, por lo tanto, solo recibe una referencia a la vista compartida, en lugar de construir y guardar su propio menú (Figura 13).

Figura 13

De un árbol por sesión a una vista compartida



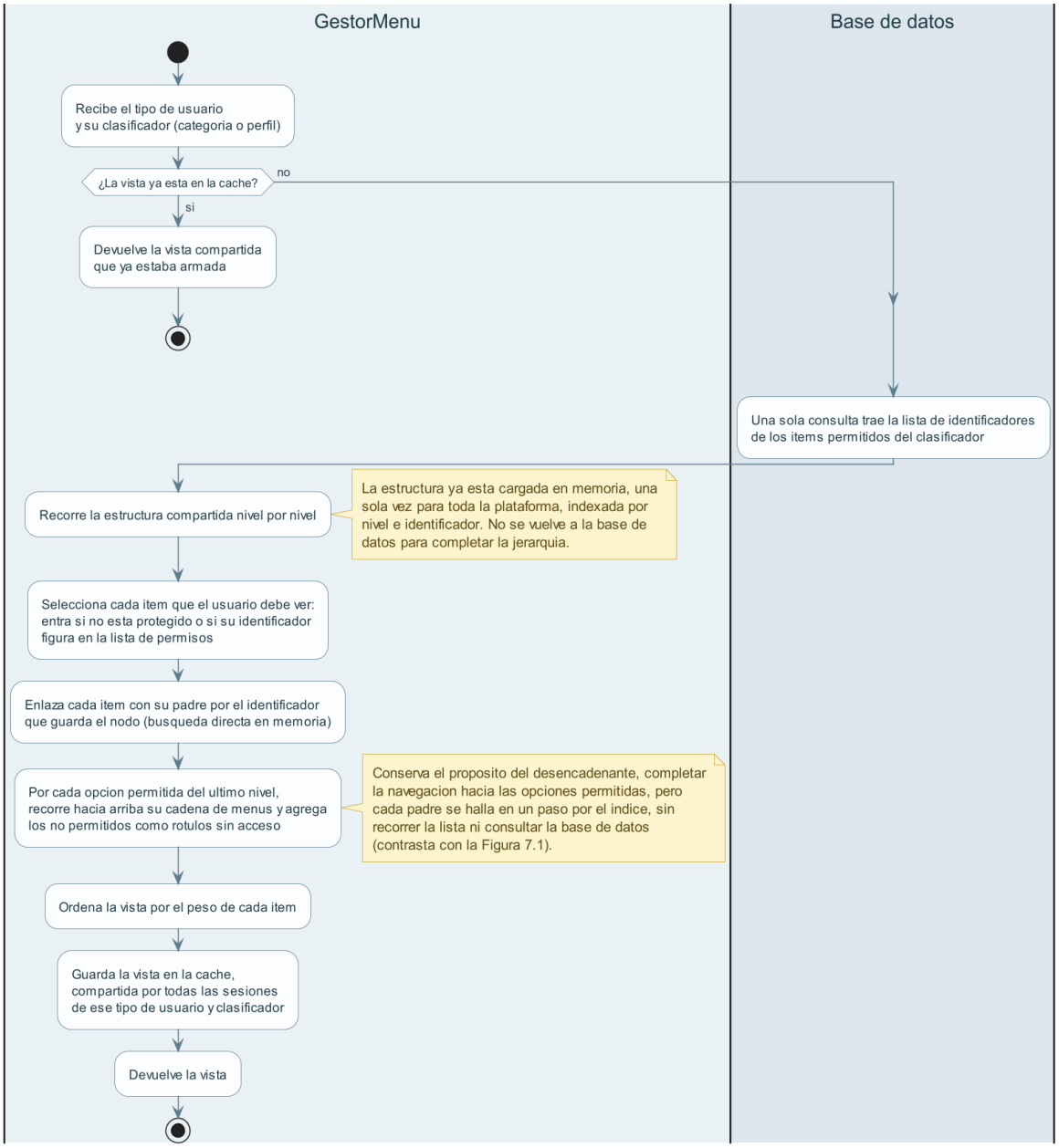
En el estado (b) las tres sesiones tienen el mismo (tipo de usuario, clasificador), por eso reciben la **misma** vista en cache, que a su vez referencia la estructura de menus compartida.

7.1.1.4 La Construcción del Menú de Cada Usuario. La vista que el gestor arma la primera vez que se pide se construye sobre la estructura compartida. Parte de una sola consulta que trae la lista de identificadores de los ítems que la categoría o el perfil del usuario tiene permitidos. Con esa lista, el gestor recorre la estructura nivel por nivel y selecciona los ítems que el usuario debe ver, los que no están protegidos o cuyo identificador figura en la lista de permisos. Cada ítem se enlaza con su padre a través del identificador que el nodo guarda, lo que reconstruye la jerarquía desde la estructura ya cargada, y el menú resultante se ordena por el peso de cada ítem.

A ese recorrido se suma el comportamiento del desencadenante descrito antes. Para cada opción permitida del último nivel, el gestor recorre hacia arriba la cadena de menús que la contiene, nivel por nivel, y muestra los que no estén permitidos, sin acceso, como rótulos, para que la opción quede alcanzable en el árbol. El gestor conserva así el propósito del desencadenante, completar la navegación hacia las opciones permitidas, pero lo resuelve de forma más eficiente. Como la estructura está indexada por identificador, encontrar cada menú de la cadena es una búsqueda directa en memoria. El desencadenante anterior, en cambio, buscaba cada padre recorriendo la lista entera, con un costo que crecía con el cuadrado del número de ítems, y traía de la base de datos los que faltaban (Figura 14).

Figura 14

Construcción de la vista de menú de un usuario en el nuevo gestor

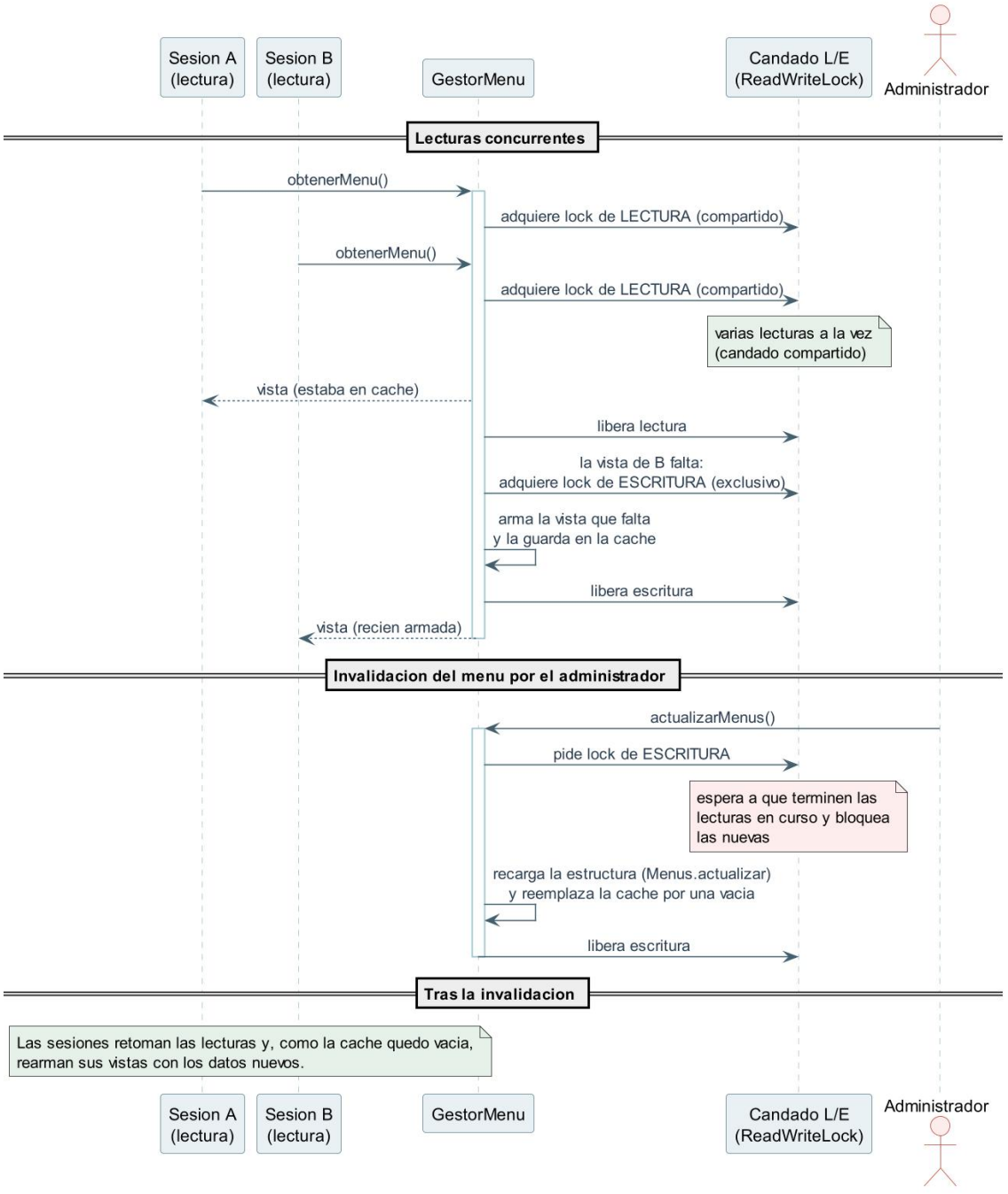


7.1.1.5 Concurrencia e Invalidación de la Caché. La estructura y la caché de vistas son una sola copia para todas las sesiones, así que el acceso concurrente se coordina con un candado de lectura y escritura. Obtener una vista toma el candado de lectura, que admite muchas lecturas a la vez; solo cuando la vista no está en la caché se toma el de escritura, para armarla y guardarla una vez.

La invalidación, que ocurre cuando un administrador cambia el menú o sus permisos, también toma el candado de escritura y, en exclusión, recarga la estructura desde la base de datos y reemplaza la caché por una vacía, mientras las vistas se rearman bajo demanda con los datos nuevos. Esa exclusión es la que hace seguro el cambio, porque sin ella una invalidación que reemplazara la estructura mientras una sesión la recorre la dejaría leyendo datos a medio cambiar. Como el reemplazo sustituye las referencias completas bajo el candado, una lectura concurrente ve la versión anterior o la nueva, nunca una a medias, y a lo sumo espera el breve instante de la recarga (Figura 15).

Figura 15

Lecturas concurrentes e invalidación de la caché bajo el candado de lectura y escritura



7.1.1.6 El Resultado. El paso de un árbol por sesión a una estructura compartida resuelve los dos problemas que la Fase 1 atribuyó a los menús. La memoria deja de crecer con el número de sesiones, porque los nodos *Menu* existen una sola vez, compartidos, más unas pocas vistas *MenuMenus* que los referencian, en lugar de una copia por sesión. La latencia de la autenticación baja al no repetirse en cada inicio de sesión la construcción del menú ni la consulta a la base de datos para completar la jerarquía. La Tabla 9 resume el contraste y la Fase 6 lo cuantifica.

Tabla 9

Construcción y almacenamiento del menú: esquema anterior y nuevo gestor

Aspecto	Esquema anterior	Gestor de menús
Momento de construcción	En cada inicio de sesión	Una vez al cargar; cada vista, una vez por combinación de tipo de usuario y clasificador
Almacenamiento	Un árbol por sesión	Una estructura compartida y una vista por combinación
Consulta a la base de datos	Dos construcciones por login; en cada una, una consulta por cada servicio padre faltante	Una para cargar los items del menú y una por combinación para los permisos
Comparación entre sesiones	Ninguna	Las sesiones de la misma combinación comparten la vista; los nodos se comparten
Reconstrucción de la jerarquía	Búsqueda del padre recorriendo la lista, con costo	Búsqueda directa del padre por índice, en memoria

Aspecto	Esquema anterior	Gestor de menús
	cuadrático y un consulta por cada faltante	

El menú que arma el gestor decide qué ve cada usuario, no a qué puede acceder. Esa comprobación corresponde al gestor de servicios y permisos, con lo que la navegación deja de ser la base de la seguridad.

7.1.2. El gestor de Servicios y Permisos

El gestor de servicios y permisos, la clase *GestorAutorizacion*, asumió el control de acceso que en el modelo anterior resolvía cada página por su cuenta. En cada petición decide si el usuario puede acceder al recurso solicitado, y lo hace en memoria, sin consultar la base de datos. El cambio de fondo está en la base de la decisión, el recurso que realmente se pide, en lugar de lo que el cliente declara.

7.1.2.1 El Esquema Anterior. En el modelo anterior, cada página resolvía su propio control de acceso al incluir la rutina de validación. Esa rutina decidía a partir de los parámetros de la petición, no del recurso real, y recorría el árbol de menús de la sesión en dos pasadas, una que exigía que el identificador de servicio y la URI declarados coincidieran con una entrada del árbol y otra que solo comprobaba la URI, mientras que la URI real solo se usaba como respaldo. La Fase 1 mostró dos problemas en ese diseño.

El primero, de seguridad, surge de que el identificador y la URI los pone el cliente, así que el control se podía eludir declarando los de un servicio permitido para alcanzar un recurso

protegido (sección 5.4). El segundo, de costo, está en que la comprobación recorría la copia del árbol que cargaba cada sesión y repetía en cada petición un recorrido lineal sobre la misma estructura cuyo peso en memoria documentó la sección 5.3.

7.1.2.2 La Comprobacion Sobre el Recurso Real. Sobre esa base, el gestor resuelve el control de acceso en memoria y a partir del recurso real. Toma el registro que ya existía, el del menú, donde cada ítem lleva su dirección y sus permisos, y arma con él una estructura indexada por esa dirección, cargada una vez al iniciar y compartida por toda la plataforma. En cada petición localiza en ella la URI que realmente se solicita y decide según lo que encuentra, con una sola búsqueda directa y sin volver a la base de datos. La comprobación deja de recorrer el árbol de navegación que cada sesión cargaba.

La decisión tiene cuatro desenlaces. Si la URI no está registrada, niega el acceso. Si el servicio está marcado como público, lo concede sin más. Si está protegido, concede el acceso cuando el clasificador del usuario está entre los que el servicio autoriza para su tipo de usuario, y lo niega cuando no. La regla por defecto es negar, y un recurso que no esté registrado, o cuyo permiso no incluya al usuario, no se entrega. La protección deja así de depender de que cada página recuerde validarse y de lo que el cliente afirme, y pasa a depender del recurso real y de un registro central (Figura 16). La Figura 17 contrasta punto por punto el control de acceso del esquema anterior, sobre los parámetros declarados por el cliente, y el del gestor, sobre la URI real.

Figura 16

Decisión de acceso del gestor de servicios y permisos

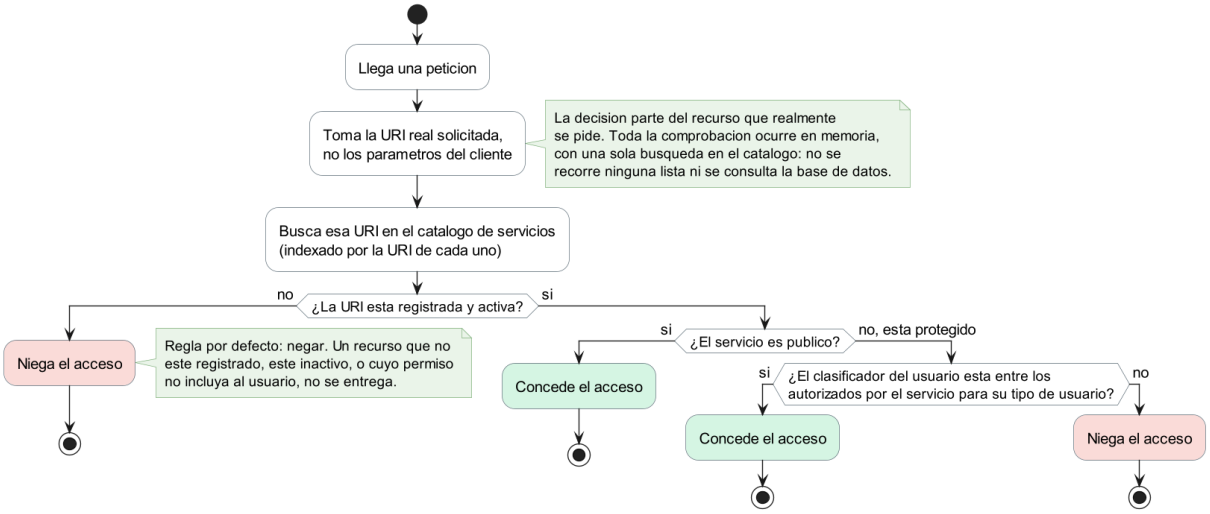
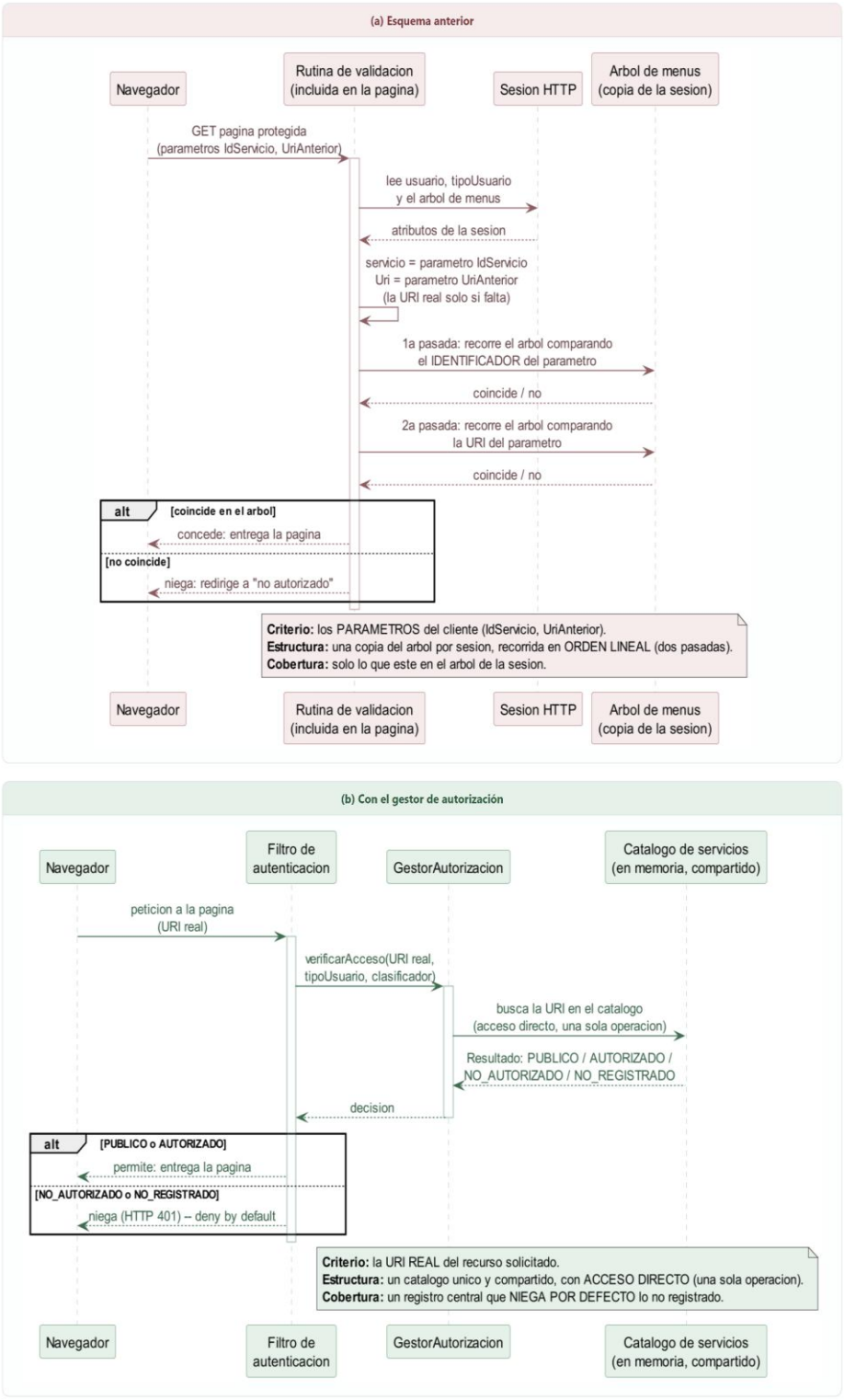


Figura 17

Decisión de acceso del gestor de servicios y permisos



7.1.2.3 Concurrencia e Invalidación. Compartir la estructura y los permisos entre todas las sesiones obliga a coordinar el acceso concurrente, para lo que el gestor usa el mismo candado de lectura y escritura que el de menús, con el que muchas peticiones leen a la vez y la exclusión se reserva para recargar. La invalidación se dispara cuando cambian los datos de acceso. Si un administrador modifica un permiso, el gestor recarga la estructura y los permisos desde la base de datos y reemplaza las copias en memoria, y la siguiente petición decide ya con los datos nuevos; el inventario de servicios de la Fase 4 lo actualiza de la misma forma.

7.1.2.4 Los Servicios Sueltos. Validar sobre el recurso real y negar por defecto cierra la vía por la que el control se eludía. Pero esa regla solo protege bien si el registro de recursos es completo, porque un recurso que no esté registrado se niega, aunque su acceso sea legítimo. En el modelo anterior, el único registro de recursos era el menú, y muchos recursos no figuran en él, por ser procesos internos y no opciones de navegación. A esos recursos que no están en el menú se les denomina servicios sueltos.

Esa laguna, los servicios sueltos sin registrar, fue lo que cerraron las dos fases siguientes. Encontrarlos y calcular sus permisos fue el objeto de la Fase 4; llevarlos, junto con los demás servicios, a un modelo de datos propio y separado del menú fue el de la Fase 5. El gestor decidía igual desde esta fase, validando sobre el recurso real; lo que esas fases completaron fue el registro sobre el que decide, no su forma de decidir.

8. Fase 4: Identificación de los Servicios Existentes

La Fase 1 mostró que el control de acceso solo conocía los servicios que figuraban en el menú, mientras que muchas páginas de la plataforma se alcanzan fuera de él (sección 5.4). El gestor de servicios y permisos de la Fase 3 valida cada petición sobre el recurso real, pero su registro sigue siendo el del menú; por la regla de negar por defecto, los servicios sueltos, los que ese registro no incluye, se rechazan aunque su acceso sea legítimo (sección 7.1.2).

La Fase 4 construye el catálogo de servicios que faltaba. Hace un barrido exhaustivo de la plataforma para localizar cada servicio, esté o no en el menú, y calcula los permisos de cada uno tomando como punto de partida la configuración de permisos del menú. El barrido y el cálculo los realiza un algoritmo propio, *cachama*, que se ejecutó sobre la plataforma real de la Escuela de Ingeniería de Sistemas y quedó integrado en el flujo de despliegue interno del equipo, Super Condor. Queda fuera del barrido el módulo Grupos, que tiene su propio sistema de seguridad.

8.1. Actividad 8: Definición de Criterios de Identificación de Servicios

Para localizar los servicios sin depender del menú, la fase modela la plataforma como un grafo dirigido, en el que cada página es un nodo y cada enlace de una página a otra es una arista. Cómo se reúnen los nodos y se detectan los enlaces es el objeto de la Actividad 9 (sección 8.2.1).

Sobre ese grafo se definen los tres criterios que distinguen a un servicio y determinan cuándo está aislado del control de acceso:

- Raíz. Un nodo al que ninguna página conduce pero que conduce a otras. Es un punto de entrada a la plataforma, desde donde arranca la navegación.

- Servicio del menú. Un nodo cuya página figura en *TB_Menu,* sea como página de un usuario o de un administrador. Trae consigo los permisos que el menú ya tiene definidos, y por eso es la semilla del cálculo.
- Servicio suelto. Un nodo que no figura en *TB_Menu.* Es una página que no define permisos propios, el caso que el control de acceso no cubría. El criterio para protegerlo es la herencia, que le transfiere los permisos de los servicios del menú desde los que se llega a él.

8.2. Actividad 9: Implementación del Análisis Estático - Herramienta *Cachama* Script

Cachama aplica esos criterios en una secuencia de pasos: construye el grafo de la plataforma, localiza las raíces, clasifica cada una contra el menú, propaga los permisos del menú por el grafo y escribe el catálogo resultante.

8.2.1. El grafo de Dependencias de la Plataforma

Construir el grafo tiene tres tareas: reunir las páginas que serán nodos, detectar los enlaces entre ellas y resolver cada referencia a la página concreta que señala. Las tres trabajan sobre el contenido de los archivos, sin ejecutar la plataforma.

8.2.1.1 Las Paginas como Nodos. El barrido recorre el directorio web de la plataforma. Toma como nodo cada una de sus páginas, y además las plantillas y los scripts que participan en algún enlace. De ese recorrido quedan fuera los módulos que tienen su propio sistema de seguridad, ajeno al control de acceso de COMA. Cada nodo se identifica por su ruta dentro de la plataforma.

8.2.1.2 La Deteccion de los Enlaces. Las aristas se hallan leyendo el contenido de cada archivo y reconociendo las formas en que una página nombra a otra. Una página no conduce a otra solo a través del menú, si no tambien al incluirla, al enlazarla, al enviarle un formulario, al redirigir hacia ella desde el servidor o al invocarla desde un script. Cada una de esas formas tiene una expresión característica en el texto, y el barrido reconoce más de veinte. La Tabla 10 las agrupa con un ejemplo de cada grupo.

Tabla 10

Formas de enlace entre páginas que reconoce el barrido, por grupo

Grupo	Ejemplo de la expresión
Inclusiones y envíos	*<%@ include file="Pagina.jsp" %>, <jsp:forward page="Pagina.jsp">*
Vínculos, formularios y marcos	*href="Pagina.jsp", action="Pagina.jsp", src="Pagina.jsp"*
Redirecciones del servidor	*response.sendRedirect("Pagina.jsp")*
LLamadas desde scripts	*window.open("Pagina.jsp"), url: "Pagina.jsp", $.getJSON("Pagina.jsp"), location.href = "Pagina.jsp"*
Asignaciones y concatenaciones	*URL = "Pagina.jsp", x = "Pagina.jsp" + parametros*

Nota. Los ejemplos usan un nombre de página genérico; las expresiones reales abarcan páginas, plantillas y scripts.

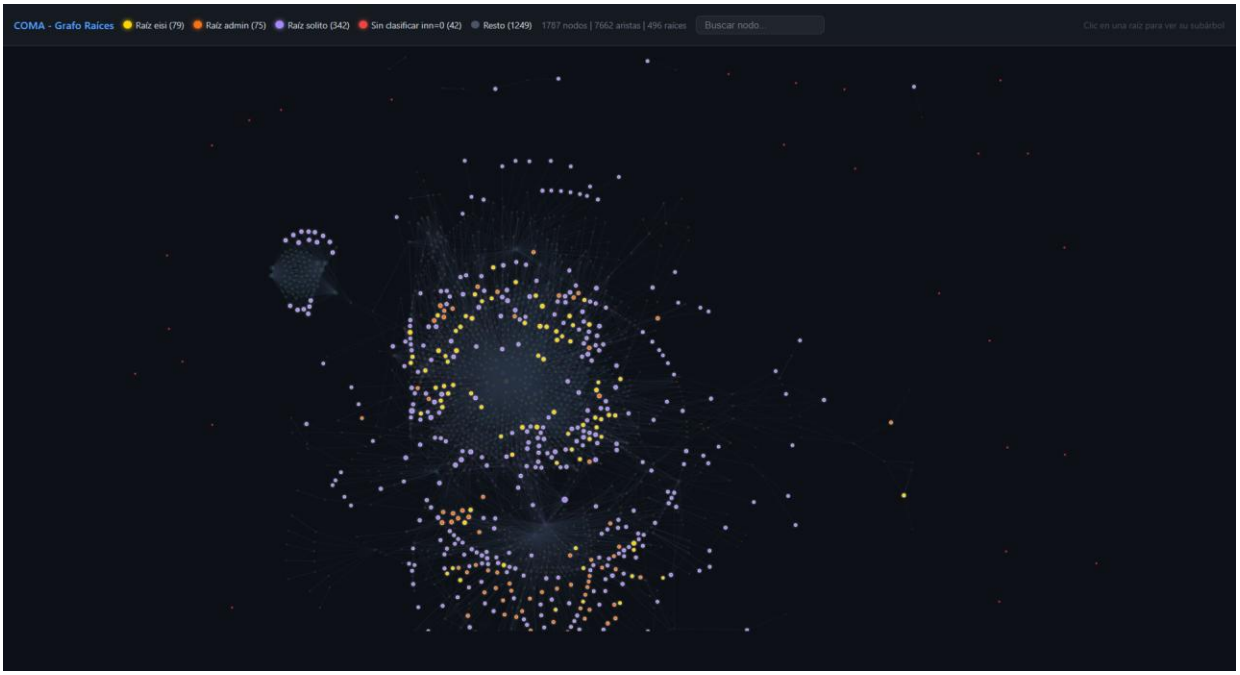
Dos casos piden un rodeo. El primero es el enlace que pasa por un script, cuando una página carga un script y este nombra otra página sin que el vínculo aparezca en la página de origen. El barrido lo sigue en dos tramos, de la página al script y del script a la página de destino, para no perderlo. El segundo es el reenvío cuyo destino es una variable. Cuando una página reenvía hacia lo que indica una variable en lugar de hacia una dirección fija, el barrido busca en el mismo archivo el valor que se le asigna y resuelve el destino con él.

8.2.1.3 Las Páginas como Nodos. Una misma página puede nombrarse de varias maneras: con su ruta completa desde la raíz del contexto, con una ruta relativa al archivo que la nombra, o solo con su nombre. Para que todas señalen el mismo nodo, el barrido resuelve cada referencia a una ruta única. Lo intenta en orden: la ruta completa del contexto, el directorio del archivo de origen, el directorio inmediatamente superior y la raíz web; si solo se da el nombre, busca el archivo que lo lleve, y cuando hay varios con ese nombre elige el más cercano por directorio. Las referencias que la aplicación arma en tiempo de ejecución, con expresiones del servidor o concatenando variables, no se pueden resolver de este modo y quedan fuera del grafo, una limitación que la Actividad 10 atiende.

Con los nodos y las aristas resueltos, el grafo queda armado, un mapa dirigido de qué página conduce a qué otra en toda la plataforma. La Figura 18 lo muestra.

Figura 18

El grafo de dependencias de la plataforma con las raíces resaltadas



8.2.2. La Clasificación de las Páginas

Sobre el grafo armado, el barrido localiza las raíces, los nodos a los que ninguna página conduce, que son los puntos de entrada de los que cuelga el resto. Cada raíz se busca en *TB_Menu* para clasificarla: del menú de usuario si su página figura como página de un usuario, del menú de administrador si figura como página de un administrador, y suelta si no figura en el menú. Sobre la Escuela de Ingeniería de Sistemas, el grafo arrojó 401 raíces: 73 del menú de usuario, 38 del de administrador y 290 sueltas. Esas 290 raíces sueltas, de las 401 en total, son la primera medida del problema, porque la mayoría de los puntos de entrada de la plataforma quedaba fuera del registro del menú.

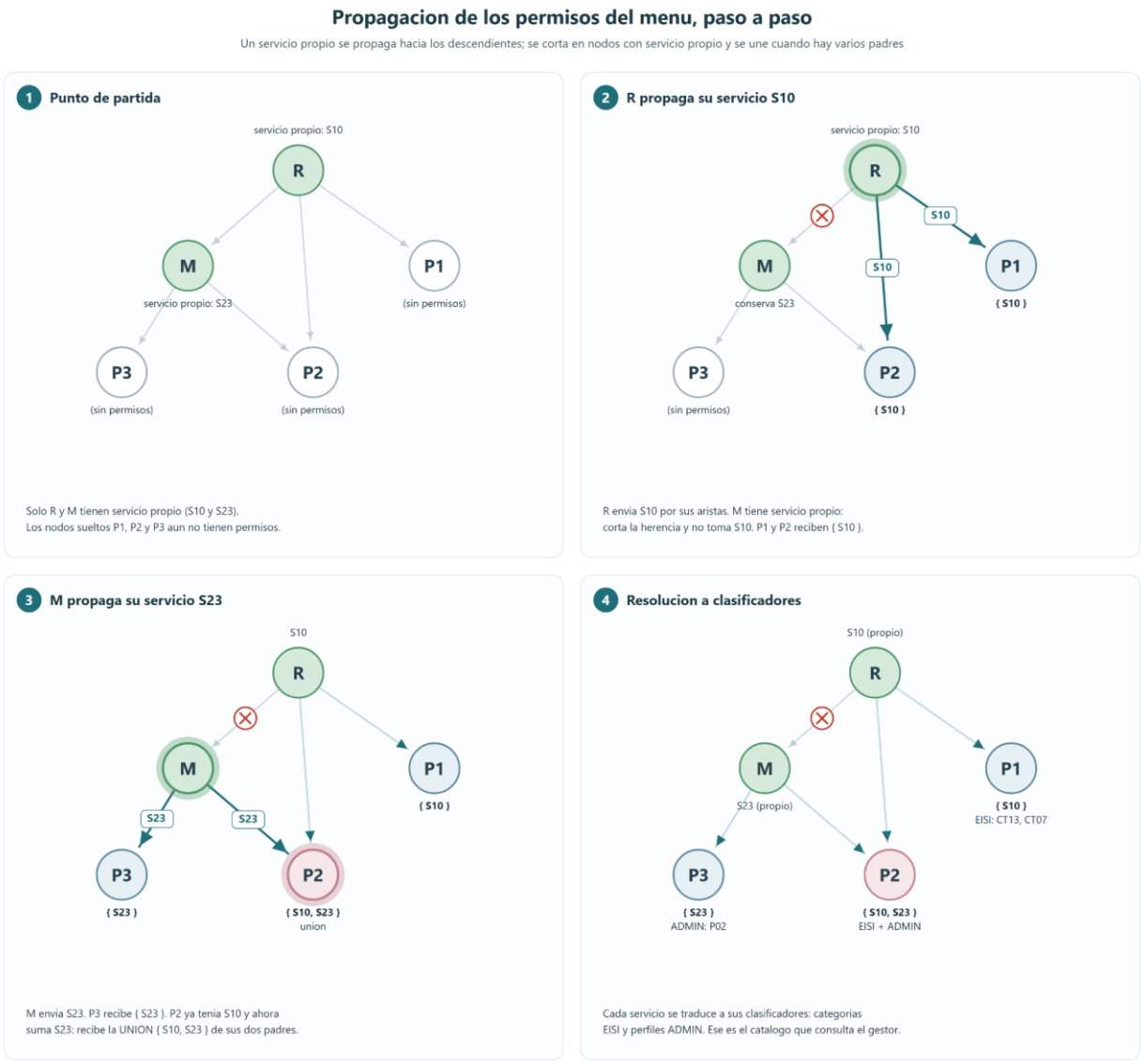
8.2.3. La Propagación de los Permisos del Menú

Con el grafo y las raíces clasificadas, cachama propaga los permisos del menú a todas las páginas. El recorrido va de las raíces hacia abajo, en orden topológico, y aplica una regla de herencia, por la que cada servicio suelto recibe la unión de los servicios de las páginas que conducen a él. Si una página tiene servicio propio, por figurar en el menú, propaga solo el suyo y corta ahí la herencia, así que sus descendientes sueltos heredan ese servicio y no los de niveles anteriores. Un servicio se identifica por la pareja de su identificador en el menú y su clasificación, de usuario o de administrador, porque la misma entrada de menú concede accesos distintos según figure como página de usuario o de administrador.

Heredar el servicio del menú es solo el primer tramo de la cadena. Cada servicio de menú heredado se resuelve después a los clasificadores que lo tienen concedido, leídos de la configuración de permisos del menú. Así, una página suelta termina asociada a las categorías y perfiles concretos que pueden acceder a ella, calculados a partir de quién podía ver los servicios de menú que la alcanzan (Figura 19). En el mismo paso se decide la visibilidad pública, según la cual una página es pública cuando todos sus servicios son de acceso libre y se mantiene protegida en cuanto uno de ellos lo está. Una página que no hereda ningún servicio se deja sin permisos, lo que el gestor traduce en negar por defecto.

Figura 19

Propagación de los permisos del menú sobre un subgrafo de ejemplo



8.2.4. El Catálogo Resultante

El recorrido produjo el catálogo de toda la plataforma. Sobre la Escuela de Ingeniería de Sistemas reunió 1.789 servicios. De ellos, 214 figuraban ya en el menú y los otros 1.575 eran sueltos; el menú registraba 236 páginas, de las que 22 no llegaron a aparecer como nodo del grafo.

Para cada servicio, el cálculo dejó sus permisos resueltos a clasificadores, 5.316 parejas de servicio y clasificador en total. De los 1.789 servicios, 1.063 quedaron sin permiso de acceso y se niegan por defecto; de los 726 restantes, 114 resultaron públicos, de acceso libre, y 612 protegidos con permisos para clasificadores concretos. La Tabla 11 reúne estas cifras y explica cada una.

Tabla 11

Inventario de servicios de la plataforma calculado en la Fase 4 (Escuela de Ingeniería de Sistemas)

Magnitud	Que representa	Valor
Páginas registradas en el menú	Las páginas que el menú ya conocía, la semilla del cálculo	236
Servicios en el catálogo	Todas las páginas de la plataforma, más las plantillas y scripts enlazados.	1798
Servicios de el menú	Del catálogo, los que ya figuraban en el menú	214
Servicios sueltos	Del catálogo, los que no figuraban en el menú	1575
Raíces	Puntos de entrada: ninguna pagina conduce a ellas, pero ellas conducen a otras	401
Raíces del menú de usuario	Raíces que figuran como página de usuario en el menú	73
Raíces del menú de administrador	Raíces que figuran como página de administrador en el menú	38
Raíces sueltas	Raíces que no figuran en el menú	290
Permisos calculados	Parejas de servicio y clasificador que el cálculo asignó	5316

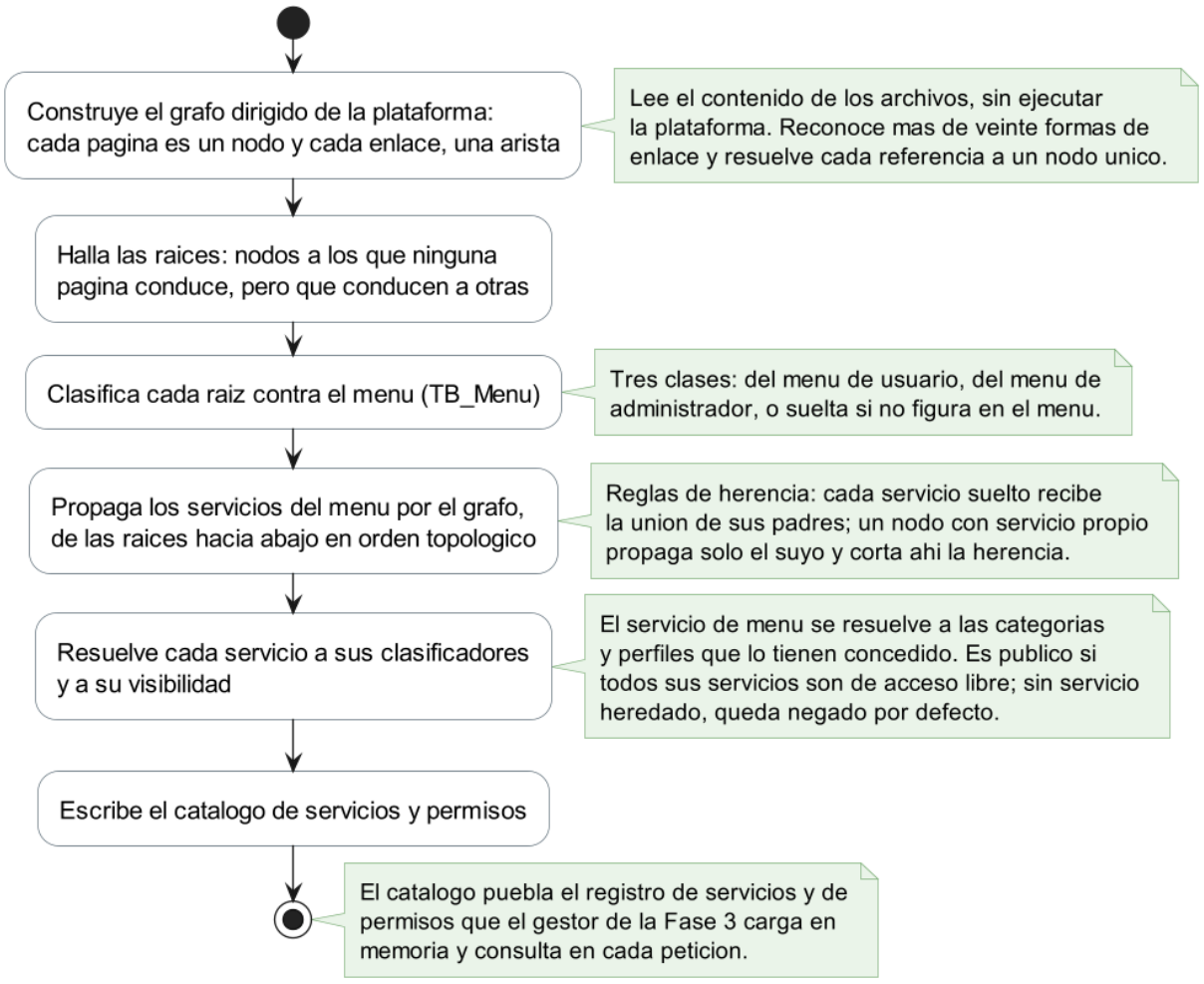
Magnitud	Que representa	Valor
Servicios publicos	De acceso libre, accesibles sin requerir permiso	114
Servicios protegidos con permisos	Accesibles sólo a los calificadores heredados del menú	612
Servicios negados por defecto	Sin permiso de acceso resuelto, inaccesibles salvo asignación manual	1063

Nota. Medición sobre la plataforma de la Escuela de Ingeniería de Sistemas. El desglose entre servicios del menú y sueltos se obtuvo cruzando el catálogo con las páginas del menú. Los servicios públicos, protegidos y negados reparten el total: 114 + 612 + 1.063 = 1.789.

Ese catálogo, una correspondencia de cada servicio con sus clasificadores y su visibilidad, es lo que alimenta el control de acceso. La Figura 20 resume la secuencia completa, del grafo al catálogo. La Fase 5 le da un modelo de base de datos propio y conecta con él al gestor de servicios y permisos, que hasta entonces decidía sobre el registro del menú.

Figura 20

Secuencia del algoritmo de inventario y propagación



8.3. Actividad 10: Depuración Manual del Inventario

El barrido se apoya en lo que las páginas declaran de forma literal, y por eso no alcanza todo. Cuando una página arma la dirección de destino en tiempo de ejecución, concatenando variables, o cuando el menú la carga de forma dinámica, el enlace no es visible para el análisis y la página de destino puede quedar fuera del grafo. Estas omisiones se trataron con una revisión

manual de la lista generada, que corrigió los registros mal resueltos y recuperó las páginas que el barrido no había enlazado, para que el catálogo final cubriera también esos casos.

La revisión dejó, además, un resultado útil más allá de la seguridad. El grafo deja a la vista las páginas que ya no reciben ningún enlace ni cuelgan de ninguna raíz viva, es decir, servicios que con el tiempo dejaron de usarse pero nunca se retiraron. Identificarlos permitió analizarlos y dejarlos listos para su deprecación, un aporte de mantenimiento para el grupo CALUMET que el barrido produjo de paso.

Con el catálogo completo y depurado, la plataforma cuenta por primera vez con un registro de todos sus servicios y de quién puede acceder a cada uno. La Fase 5 lleva ese registro a la base de datos y termina de conectarlo con el gestor que lo consume.

9. Fase 5: Reestructuración de la Base de Datos

Las fases anteriores dejaron los gestores en memoria (Fase 3) y el catálogo completo de servicios con sus permisos (Fase 4), pero apoyados todavía en un modelo de datos que mezclaba lo que debían separar. En la base de datos, una sola tabla, *TB_Menu*, describía a la vez la navegación y los recursos sujetos a control de acceso, porque cada fila era un ítem del menú y, al mismo tiempo, el servicio cuyos permisos se consultaban, y las tablas de permisos colgaban de esa misma fila. Es la raíz de lo que la sección 5.4 diagnosticó, que el control de acceso solo conocía lo que estaba en el menú.

La Fase 5 redefine el modelo para separar los tres elementos, menús, servicios y permisos, en tablas independientes que se administran por su cuenta. La separación cierra el arco que las Fases 3 y 4 habían dejado abierto, pues el catálogo de servicios que la Fase 4 construyó recibe aquí

su propio modelo, y el gestor de servicios y permisos pasa a decidir sobre él en vez del registro del menú con el que venía funcionando. Es lo que la sección 7.1.2 y el capítulo 8 anunciaron sin detallar.

9.1. Actividad 11: Separar las Tablas de Servicios y Menús en la Base de Datos

Antes, *TB_Menu* cargaba con tres responsabilidades. Sus columnas nombraban un servicio (*IdServ*, *PagServ*, *PagAdmin*), la jerarquía de navegación (*ServPred*, el padre) colgaba de las mismas filas, y las tablas de permisos del menú referían a ese identificador.

El nuevo modelo reparte esas responsabilidades en cinco tablas, que la Tabla 12 resume y la Figura 21 contrasta con el anterior. *TB_Menu* se queda solo para la navegación, con sus columnas renombradas para que digan lo que son: el identificador pasó de *IdServ* a *IdMenu*, las páginas de *PagServ* y *PagAdmin* a *PagMenu* y *PagMenuAdmin*, y el padre de *ServPred* a *MenuPred*, que ordena la jerarquía de navegación del menú; la capa Java conserva los nombres viejos mediante alias para no romper el código que aún los usa. Los permisos del menú se quedan en *TR_AutorizacionCategorias* y *TR_AutorizacionPerfiles*, las tablas que ya existían, renombradas desde *TR_Autorizacion_Usr* y *TR_Autorizacion*. Los servicios y sus permisos, en cambio, salen a dos tablas nuevas, *TB_Servicios* y *TR_AutorizacionServicios*, que la Fase 4 llena.

Tabla 12

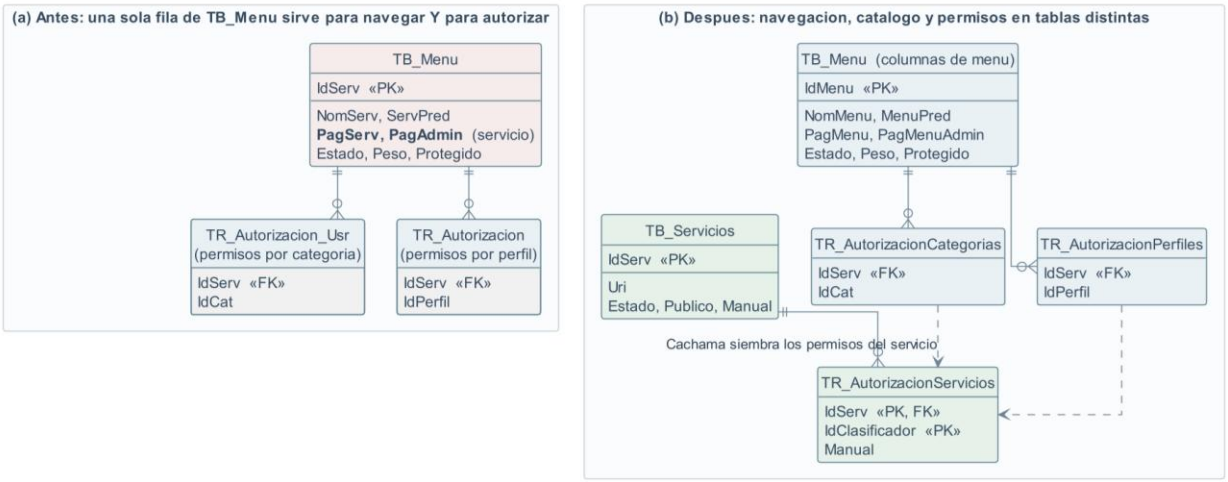
Modelo de datos separado de la Fase 5

Tabla	Qué guarda	Quién la usa
TB_Menu	La navegación: los ítems del menú y su jerarquía	El gestor de menús
TR_AutorizacionCategorias	Qué categorías de usuario tienen concedido cada	El gestor de menús; semilla

Tabla	Qué guarda	Quién la usa
	ítem del menú	de *cachama*
TR_AutorizacionPerfiles	Qué perfiles de administrador tienen concedido cada ítem del menú	El gestor de menús; semilla de *cachama*
TB_Servicios	El catálogo de servicios: la URI de cada recurso, su estado y su visibilidad	El gestor de servicios y permisos
TR_AutorizacionServicios	Qué clasificadores pueden acceder a cada servicio	El gestor de servicios y permisos

Figura 21

Modelo de datos antes y después de la separación



La separación se aplicó con un script de migración. COMA se levanta por escuela, y cada escuela tiene su propia base de datos, *diamante*, donde residen sus servicios y permisos; el script corrió sobre la *diamante* de todas las escuelas, renombrando las columnas y las tablas existentes y creando las dos tablas nuevas.

Para que el cambio no afectara a los administradores, los permisos del menú se conservaron tal como estaban, y las tablas por categoría y por perfil mantienen su estructura y la forma en que el administrador las configura. Esos permisos, que definen qué categorías y perfiles pueden ver cada ítem del menú, son además la semilla con la que cachama calcula los permisos de los servicios (capítulo 8). Reajustar los permisos de un servicio desde el menú, por lo tanto, se hace volviendo a ejecutar cachama, que los recalcula a partir de su configuración.

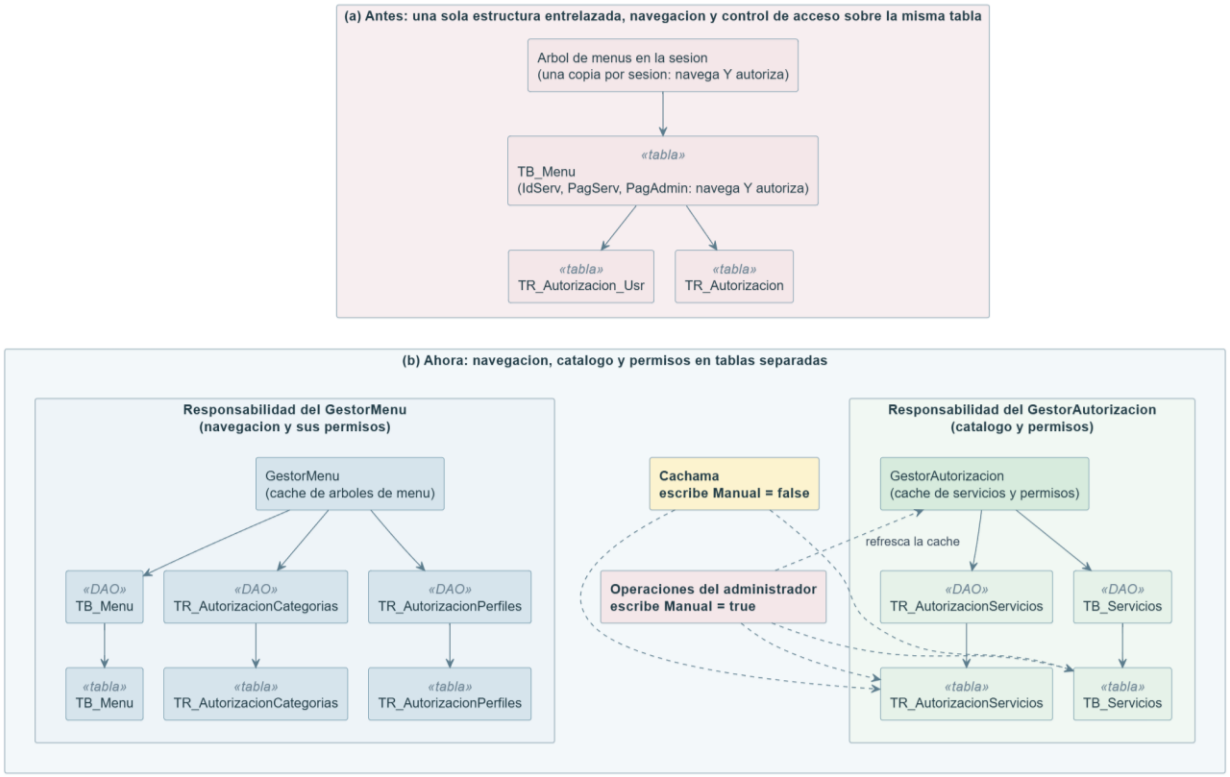
9.2. Actividad 12: Definición de Clases para Gestionar las Tablas en el Backend

Cada tabla del modelo nuevo recibió en el backend una clase de acceso a datos que reúne sus consultas y sus operaciones de escritura, siguiendo la convención del grupo CALUMET de dar a cada tabla la suya. Esas clases son el punto por el que la aplicación trabaja con las tablas, en lugar de repartir sentencias contra la base de datos, y existen por dos razones: para que los gestores lean por ellas el catálogo y los permisos que reemplazan al modelo anterior, y para que el administrador cambie por ellas un permiso o la visibilidad de un servicio. El catálogo de servicios y la tabla de permisos por servicio tienen las suyas, con las operaciones que el administrador necesita: marcar un servicio como público, o conceder y quitar a un clasificador el permiso sobre un servicio. Las tablas del menú conservan las clases que ya gestionaban sus lecturas y sus permisos, con las que el administrador sigue configurando qué categorías y perfiles ven cada ítem.

Cada uno de los dos gestores que la Fase 3 puso en memoria guarda en su caché lo que lee, una sola copia compartida por todas las sesiones (sección 7.1.1): el de menús, la navegación y sus permisos; el de servicios y permisos, el catálogo y los permisos por servicio. La Figura 22 ubica cada pieza.

Figura 22

Modelo de datos antes y después de la separación



La distinción que el modelo nuevo introduce es una marca, `Manual`, en el catálogo de servicios y en la tabla de permisos por servicio. Separa lo que calcula cachama de lo que fija el administrador a mano, con la marca en falso en las filas que escribe cachama y en verdadero en las del administrador. Importa porque cachama se vuelve a ejecutar y, en cada corrida, rehace sus propias filas, pero respeta las marcadas como manuales, y las decisiones del administrador sobreviven. Por eso las operaciones del administrador dejan la marca en verdadero y, tras cada cambio, refrescan la caché del gestor de servicios y permisos para que la siguiente petición decida con los datos nuevos.

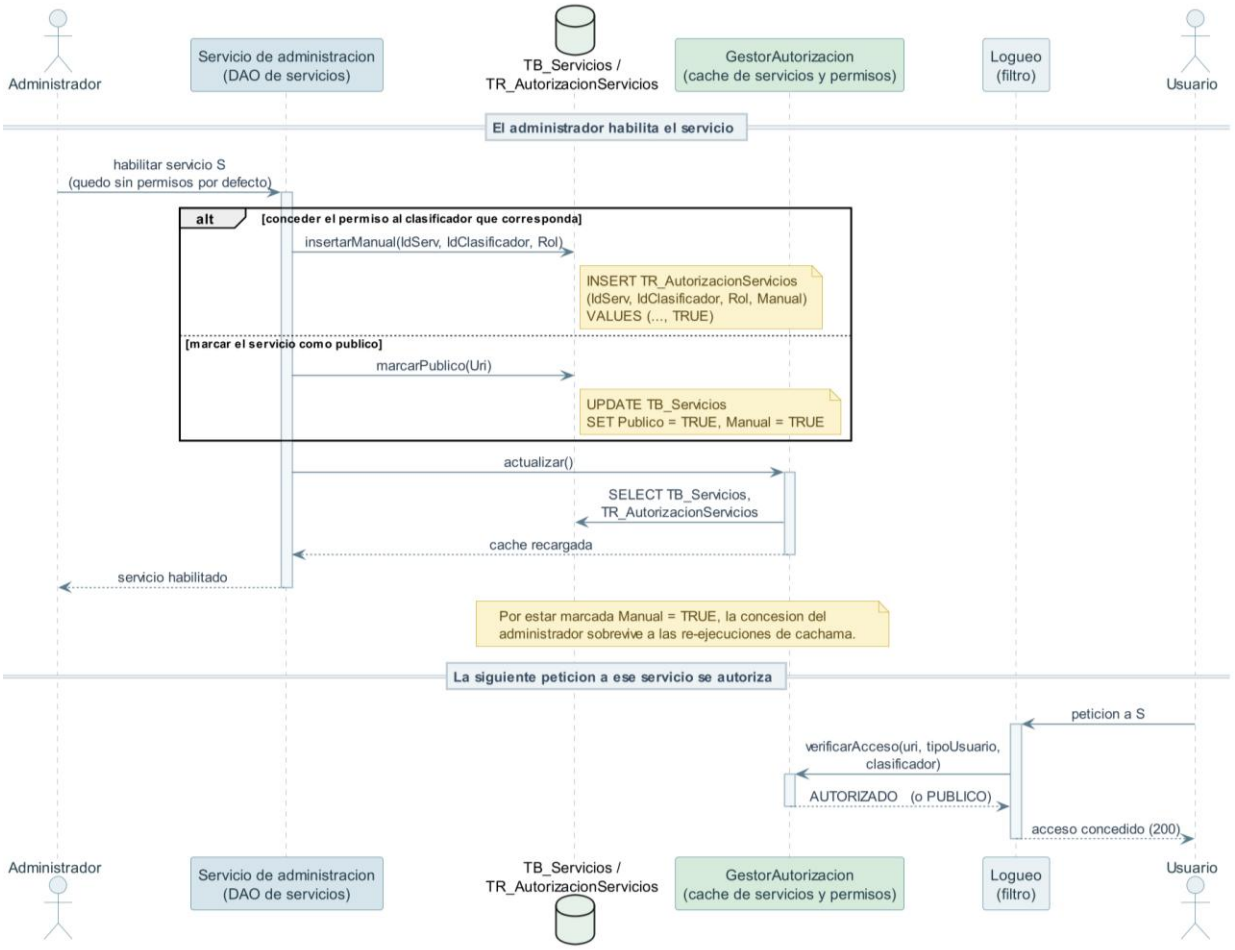
9.3. Actividad 13: Estructuración de la Lógica de Menús y Servicios Separada

Con el modelo y las clases en su sitio, la lógica de la aplicación queda partida en dos caminos que antes eran uno. En el modelo anterior, la navegación y el control de acceso se resolvían sobre la misma estructura, el árbol de menús que cada sesión cargaba (secciones 5.3 y 5.4). Ahora son caminos separados sobre tablas separadas: para mostrar el menú, el gestor de menús lee la navegación y sus permisos por categoría y perfil; para decidir el acceso, el gestor de servicios y permisos lee el catálogo y los permisos por servicio, las mismas tablas que *cachama* llenó.

La separación deja un caso por atender. Tras el inventario de la Fase 4, algunos servicios quedan registrados sin permisos para ninguna categoría o perfil, negados por defecto (capítulo 8), por ser recursos que no heredan permisos de ningún servicio del menú. No siempre es claro qué hacer con ellos. Unos deben seguir negados, pero otros son accesos legítimos que el cálculo no supo asociar, y que en el modelo anterior pasaban inadvertidos, porque el control no les exigía un permiso propio y cualquiera que llegara a ellos accedía (sección 5.4). Cuáles habilitar, y si el acceso debe abrirse a todos o solo a ciertos clasificadores, no se puede determinar de forma automática, pues exige conocer cada servicio. Por eso la decisión se dejó en manos del administrador. Para ello, sin rehacer todo el inventario, la administración cuenta con un servicio que, apoyado en las operaciones de la sección 9.2, concede a uno de esos servicios los permisos que le falten o lo marca como público. Como esas operaciones dejan la marca manual y refrescan la caché, la concesión persiste aunque *cachama* se vuelva a ejecutar y el servicio queda accesible desde la siguiente petición. La Figura 23 traza ese flujo.

Figura 23

Habilitación por el administrador de un servicio negado por defecto



Con esto se completa el arco que recorrió tres fases. La Fase 3 puso en memoria los gestores que separan la navegación del control de acceso y validan sobre el recurso real; la Fase 4 inventarió todos los servicios y calculó sus permisos a partir del menú; y la Fase 5 llevó esa separación al modelo de datos y conectó los gestores con él. La navegación, los servicios y los permisos quedan, por fin, en piezas distintas que se administran de forma independiente.

10. Fase 6: Evaluación Posterior a la Implementación

La Fase 1 midió el estado de COMA antes de la reingeniería y fijó la línea base de comparación (capítulo 5). La Fase 6 cierra el ciclo al repetir sobre la plataforma ya reingenierizada las mismas mediciones de rendimiento, memoria y control de acceso, que compara con esa línea base para cuantificar las mejoras alcanzadas.

Las medidas se tomaron con SISIFO (sección 5.2.1), el entorno de pruebas de estrés que la Fase 1 construyó para comparar dos ramas de COMA bajo la misma carga y atribuir las diferencias a los cambios de código. Aquí enfrenta la rama anterior a la reingeniería, que fija la línea base, con la rama reingenierizada. Para que la comparación aísle el efecto de la reingeniería, las dos se midieron con el arreglo del pool de conexiones aplicado por igual (sección 5.2.2), de modo que ese arreglo actúa como una constante del experimento y no como una variable.

10.1. Actividad 14: Medición de la Latencia de Inicio de Sesión bajo Concurrencia

Sobre la rama reingenierizada se repitió la prueba de velocidad de la sección 5.2, con el mismo ciclo de reautenticación de SISIFO en los siete niveles de concurrencia (1, 5, 10, 20, 50, 100 y 200 usuarios) y sesenta segundos por nivel. Como en la Fase 1, la medición separa la autenticación, en la que se construye el árbol de menús de la sesión, del render del menú, en el que se entrega la página que lo muestra; esa separación ubica el costo en su origen.

10.1.1. La Latencia Bajo Concurrencia

La autenticación es donde se concentra la mejora. La Tabla 13 contrasta su latencia en las dos ramas. En la anterior, el promedio crece poco a poco, de 108 a 175 ms hasta los 100 usuarios,

y se dispara a 840 ms con 200 usuarios concurrentes; en la reingenierizada se mantiene plano, entre 16,8 y 28,6 ms, en todo el rango (Figura 24). Con 200 usuarios, el promedio baja de 840,04 ms a 28,58 ms, una reducción del 96,6 %, y la cola se contrae en la misma proporción: el percentil 95 pasa de 1.800 ms a 64 ms y el percentil 99, de 2.200 ms a 130 ms.

Tabla 13

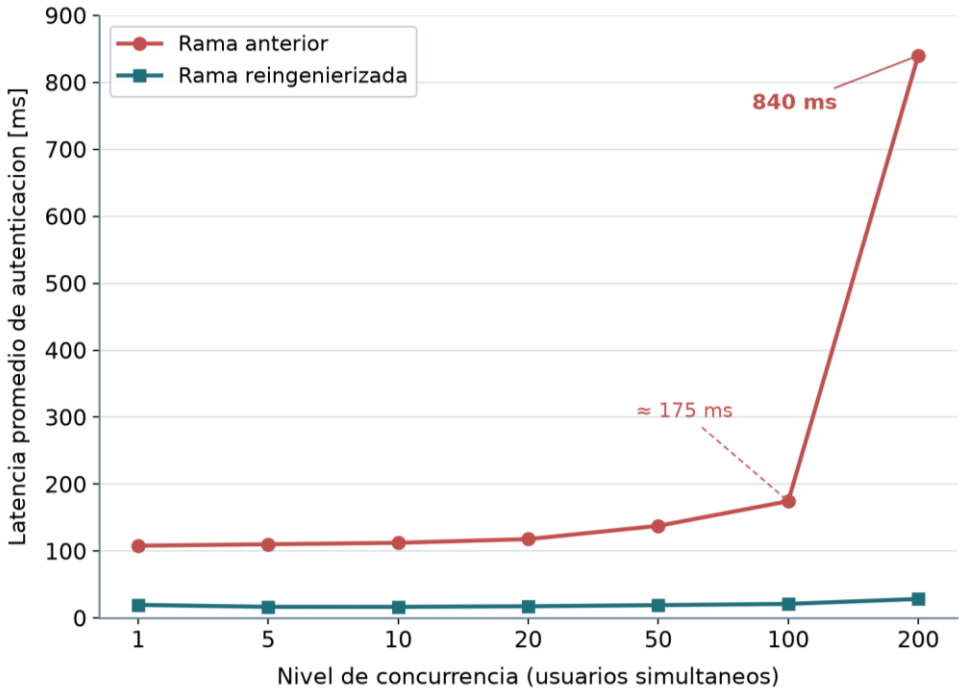
Latencia de la autenticación por nivel de concurrencia, antes y después de la reingeniería

Concurrencia (usuarios)	**Anterior, promedio (ms)**	**Anterior, p95 (ms)**	**Reingenierizada, promedio (ms)**	**Reingenierizad, p95 (ms)**
1	108,10	120	19,69	41
5	110,30	130	16,87	34
10	112,47	140	16,81	30
20	117,99	140	17,69	36
50	137,75	170	19,34	36
100	174,59	240	21,22	44
200	840,04	1800	28,58	64

Nota. Las dos ramas con el pool de conexiones aplicado (sección 5.2.2). Datos de SISIFO, sesenta segundos por nivel.

Figura 24

Latencia de la autenticación frente a la concurrencia, antes y después de la reingeniería



El render del menú, en cambio, cambió mucho menos, lo que confirmó que el cuello de botella estaba en la construcción del árbol y no en su presentación. En la Tabla 14, el render se mantiene en torno a 7 ms en las dos ramas hasta los 50 usuarios y sube poco a los 100 (10,90 ms en la anterior y 8,07 en la reingenierizada). Solo con 200 usuarios se separan, cuando la rama anterior salta a 49,25 ms, porque el servidor, saturado por el costo de la autenticación, degrada también la entrega, mientras que la reingenierizada se queda en 11,12 ms.

Tabla 14

Latencia del render del menú por nivel de concurrencia, antes y después de la reingeniería

Concurrencia (usuarios)	Anterior, promedio (ms)	Reingenierizada, promedio (ms)
1	6,72	6,70
5	6,42	6,67
10	6,57	6,64
20	6,76	6,93
50	7,69	7,97
100	10,90	8,07
200	49,25	11,12

Nota. Mismas condiciones que la Tabla 10.1.

La capacidad del servidor también mejora. En el nivel de 200 usuarios, y en los mismos sesenta segundos, la rama anterior completó 3.910 autenticaciones y la reingenierizada 5.751, un 47 % más, sin fallos en ninguna de las dos. Al no saturarse, el servidor reingenierizado atiende más peticiones en el mismo tiempo.

El origen de la diferencia es el gestor de menús de la Fase 3. En la rama anterior, cada autenticación reconstruía el árbol de menús de la sesión, un trabajo cuyo costo crece con el número de sesiones simultáneas y que, con 200 usuarios, llevaba la autenticación a 840 ms. El gestor de menús, en cambio, arma la navegación una sola vez y reutiliza una vista del menú por cada combinación de tipo de usuario y clasificador, en lugar de reconstruir el árbol en cada sesión, con

lo que el tiempo de la autenticación se vuelve casi independiente de la concurrencia. La latencia que la sección 5.2 había señalado como el síntoma más visible del diseño anterior queda, así, resuelta.

10.1.2. El Alcance Sobre las Conexiones

El efecto de la reingeniería llega más allá de la latencia, hasta la causa que había obligado a arreglar el pool de conexiones. Al construir el árbol de menús una sola vez en lugar de reconstruirlo en cada autenticación, el gestor hace muchas menos consultas a la base de datos por inicio de sesión, y con ello baja la presión sobre las conexiones. El estado de colapso que se midió sin el pool (sección 5.2.2) lo deja ver. La Tabla 15 compara las dos ramas en esa condición. Sin el pool, la rama anterior se hunde a medida que sube la concurrencia, hasta que con 200 usuarios el 91,8 % de las cargas de la página de login devuelve un error de servidor y falla el 77,7 % de las autenticaciones; la rama reingenierizada, en cambio, atraviesa los siete niveles sin un solo fallo y con la autenticación en torno a 28 ms, casi igual que con el pool aplicado.

Tabla 15

Comportamiento sin el pool de conexiones, por nivel de concurrencia

Concurrencia (usuarios)	Anterior: cargas de login fallidas (%)	Anterior: autenticaciones fallidas (%)	Reingenierizada: peticiones fallidas (%)
1	0,0	0,0	0,0
5	0,0	0,0	0,0
10	56,9	2,2	0,0
20	76,7	8,7	0,0

Concurrencia (usuarios)	Anterior: cargas de login fallidas (%)	Anterior: autenticaciones fallidas (%)	Reingenierizada: peticiones fallidas (%)
50	88,5	12,3	0,0
100	90,8	58,1	0,0
200	91,8	77,7	0,0

Nota. Estado sin el arreglo del pool de conexiones (sección 5.2.2). Datos de SISIFO. Los promedios de latencia de la rama anterior se omiten porque, con la mayoría de las peticiones fallidas, dejan de ser representativos; la rama reingenierizada mantuvo la autenticación entre 16 y 28 ms.

Sumadas las dos intervenciones, el recorrido de COMA va de un extremo al otro. Antes de este trabajo, la plataforma colapsaba bajo concurrencia sostenida, y el arreglo del pool de conexiones fue necesario solo para poder medirla. Con todo lo realizado, COMA sostiene 200 usuarios concurrentes a 28 ms y sin fallos, porque el pool cierra las conexiones que la plataforma no liberaba y libera los puertos efímeros, y la reingeniería reduce cuántas conexiones hacen falta. Son dos vías para una misma causa.

10.2. Actividad 15: Medición del Consumo de Memoria de Sesión

El árbol que cada sesión reconstruía tenía un segundo costo además del tiempo, la memoria. La sección 5.3 lo había medido sobre la rama anterior, donde cada inicio de sesión guardaba su propia copia del árbol de menús; bajo la reautenticación sostenida, el **heap** creció hasta 1,40 GB, retenido casi por completo por las sesiones. Sobre la rama reingenierizada se repitió la prueba, con el mismo entorno, la misma carga de 200 usuarios durante sesenta segundos y el mismo

procedimiento de la sección 5.3.2. Esta vez se grabó, además, un registro de vuelo de la máquina virtual Java, que permite ver cómo evoluciona la memoria y cómo trabaja el recolector de basura a lo largo de toda la prueba, no solo en la foto final del histograma.

Las dos ramas se midieron en una misma corrida y bajo la misma carga. La reingenierizada atendió incluso más inicios de sesión que la anterior, 5.685 frente a 5.067, sin fallos en ninguna. Con la carga así igualada, la diferencia entre las dos ramas está en la memoria que esa carga deja.

10.2.1. La Reducción del Heap y su Origen

El **heap** total bajó de 1,40 GB a 69 MB, una reducción del 95,2 %, y el número de objetos vivos cayó de 47,4 a 1,1 millones (Tabla 16). Normalizado por inicio de sesión, el consumo pasó de 289,59 KB a 12,45 KB, un 95,7 % menos. La memoria del menú dejó de crecer con las sesiones.

Tabla 16

Estado del heap bajo 200 usuarios concurrentes, antes y después de la reingeniería

Metrica	Anterior	Reingenierizada
Inicios de sesión exitosos	5.067	5685
Sesiones activas	5.096	5.688
Instancias vivas en el **heap**	47.350.736	1.145.366
Heap total	1,40 GB	69 MB
Heap por inicio de sesión	289,59 KB	12,45 KB

Nota. Prueba de memoria de la sección 5.3.2, con el pool de conexiones aplicado, sobre las dos ramas en una misma corrida.

El origen de la caída está en cómo se guarda el menú. En la rama anterior, los nodos del árbol, de la clase *beans.Servicio*, sumaban 1.480.608 instancias, la réplica del menú en cada sesión. En la reingenierizada esa clase desaparece del **heap**, porque el menú se guarda una sola vez, en 402 nodos de la clase *muisca.menu.Menu*, sobre una única estructura *muisca.menu.Menus* que el gestor de menús de la Fase 3 (sección 7.1.1) mantiene compartida por toda la aplicación, de la que cada combinación de tipo de usuario y clasificador deriva una vista *muisca.menu.MenuMenus* (Tabla 17). Cada sesión conserva un objeto de la clase *beans.Menus*, pero sin estado, pues pide el menú a la estructura compartida cuando lo necesita y no almacena nodos. Sus 5.688 instancias, una por sesión activa, ocupan en conjunto 88,88 KB, dieciséis bytes por instancia. Con la réplica desaparece el texto que arrastraba: las cadenas bajan de 528,51 MB a 7,29 MB y los arreglos de bytes, de 750,21 MB a 30,25 MB.

Tabla 17

Instancias de las clases que representan el menú, y el texto asociado, en el heap

Metrica	Anterior	Reingenierizada
Nodos del árbol por sesión (*beans.Servicio*)	1.480.608	0
Árbol por sesión (*beans.Menu*)	5.088	0
Nodos del menú compartido (*muisca.menu.Menu*)	0	402
Estructura compartida (*muisca.menu.Menus*)	0	1
Vistas del menú (*muisca.menu.MenuMenus*)	0	2
Accesor de menú por sesión (*beans.Menus*)	0	5.688

Metrica	Anterior	Reingenierizada
Cadenas (*java.lang.String*)	528,51 MB	7,29 MB
Arreglos de bytes (*[B*)	750,21 MB	30,25 MB

*Nota***.** Las seis primeras filas son número de instancias; las dos últimas, tamaño superficial. Mismas condiciones que la Tabla 16.

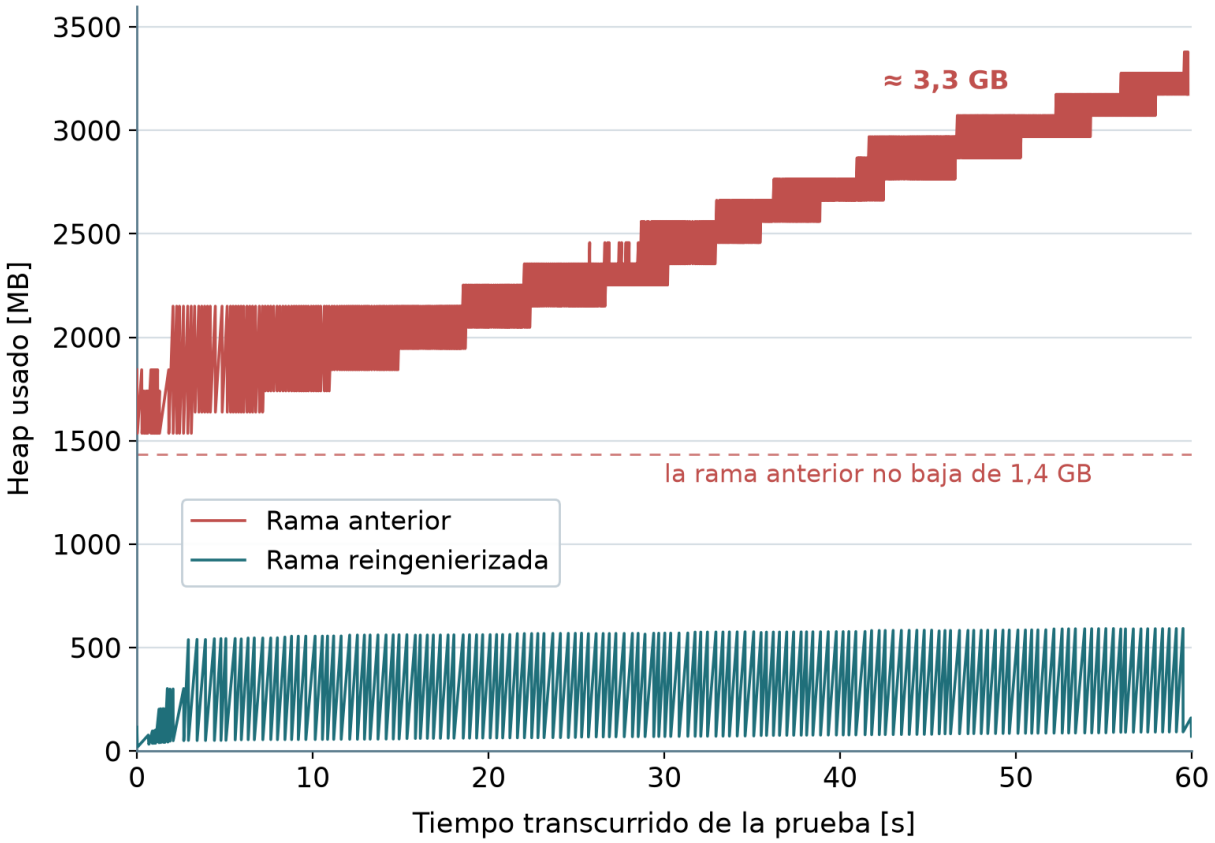
El análisis del tamaño retenido con Eclipse MAT confirma quién ocupa el **heap**. En la rama anterior, el administrador de sesiones de Tomcat (*StandardManager*) retiene 1,36 GB, el 97,16 % del **heap**, y casi todo son las copias del menú que cada sesión guarda. En la reingenierizada ese mismo administrador retiene 27 MB, el 40,14 % de un **heap** ya de por sí pequeño, y su contenido es estado normal de sesión, sin menús. Las sesiones dejan de concentrar la memoria.

10.2.2. El Heap y el Recolector a lo Largo de la Prueba

El registro de vuelo muestra la diferencia en movimiento (Figura 25). En la rama anterior, el **heap** se mantuvo en una banda alta, entre 1,4 y 3,3 GB, sin bajar de 1,4 GB ni siquiera tras cada recolección, porque las copias del menú no se pueden liberar mientras las sesiones siguen vivas, y trepaba hacia 3,3 GB a medida que estas se acumulaban. Durante la prueba, todas las recolecciones fueron menores, de hasta 30 ms, pero la presión se nota al vaciar ese **heap**, donde la recolección completa que el procedimiento de medición fuerza al terminar (sección 5.3.2) tardó 848 ms sobre la rama anterior, frente a 26 ms sobre la reingenierizada. En esta, el **heap** osciló por debajo de 600 MB y cada recolección lo devolvió a unas pocas decenas de megabytes.

Figura 25

Ocupación del heap durante la prueba de 200 usuarios, antes y después de la reingeniería



El recolector también trabajó mucho más en la rama anterior. En los mismos sesenta segundos ejecutó 733 recolecciones, frente a 156 de la reingenierizada, ya que generaba mucha más basura por inicio de sesión. Las pausas individuales fueron cortas en las dos ramas, de hasta 30 ms en la anterior y 32 ms en la reingenierizada, pero la anterior las pagó más de cuatro veces más a menudo y sobre un **heap** mucho mayor. Es trabajo de máquina que se suma al costo de latencia de la sección 10.1.

10.2.3. El Origen de la Asignación de Memoria

El registro también muestra de dónde sale la basura que el recolector recoge. Atribuida según la operación que la origina, la mayor parte de la asignación de un inicio de sesión en la rama anterior corresponde a la construcción del menú, el armado del árbol y sus consultas a la base de datos, que por sí sola supera la mitad. La generación de las claves RSA, común a las dos ramas, aporta cerca de un tercio, y el resto se reparte entre el armazón del inicio de sesión, la validación de la entrada y la consulta del usuario, que por sí misma apenas asigna alrededor de un 1 %.

La construcción del menú es la que la reingeniería elimina, pues en la rama reingenierizada el menú no se reconstruye en cada sesión, y esa parte de la asignación desaparece. Al generar mucha menos basura por inicio de sesión, la rama reingenierizada da mucho menos trabajo al recolector, que es la caída de recolecciones de la sección 10.2.2.

Esta asignación no es lo mismo que la ocupación del **heap**, porque una cosa es la memoria que se asigna y el recolector recoge en cada inicio de sesión, y otra la que queda retenida mientras la sesión vive. El menú pesaba en las dos, ya que se reconstruía en cada inicio de sesión, lo que generaba la basura recién descrita, y se guardaba copiado en cada una, lo que llenaba el **heap** (sección 10.2.1). Al compartirlo, la reingeniería corta las dos a la vez, y el consumo de memoria por sesión que la sección 5.3 había señalado como el segundo gran costo del diseño anterior queda resuelto. El menú deja de depender de la concurrencia, tanto en la memoria que retiene como en la que asigna.

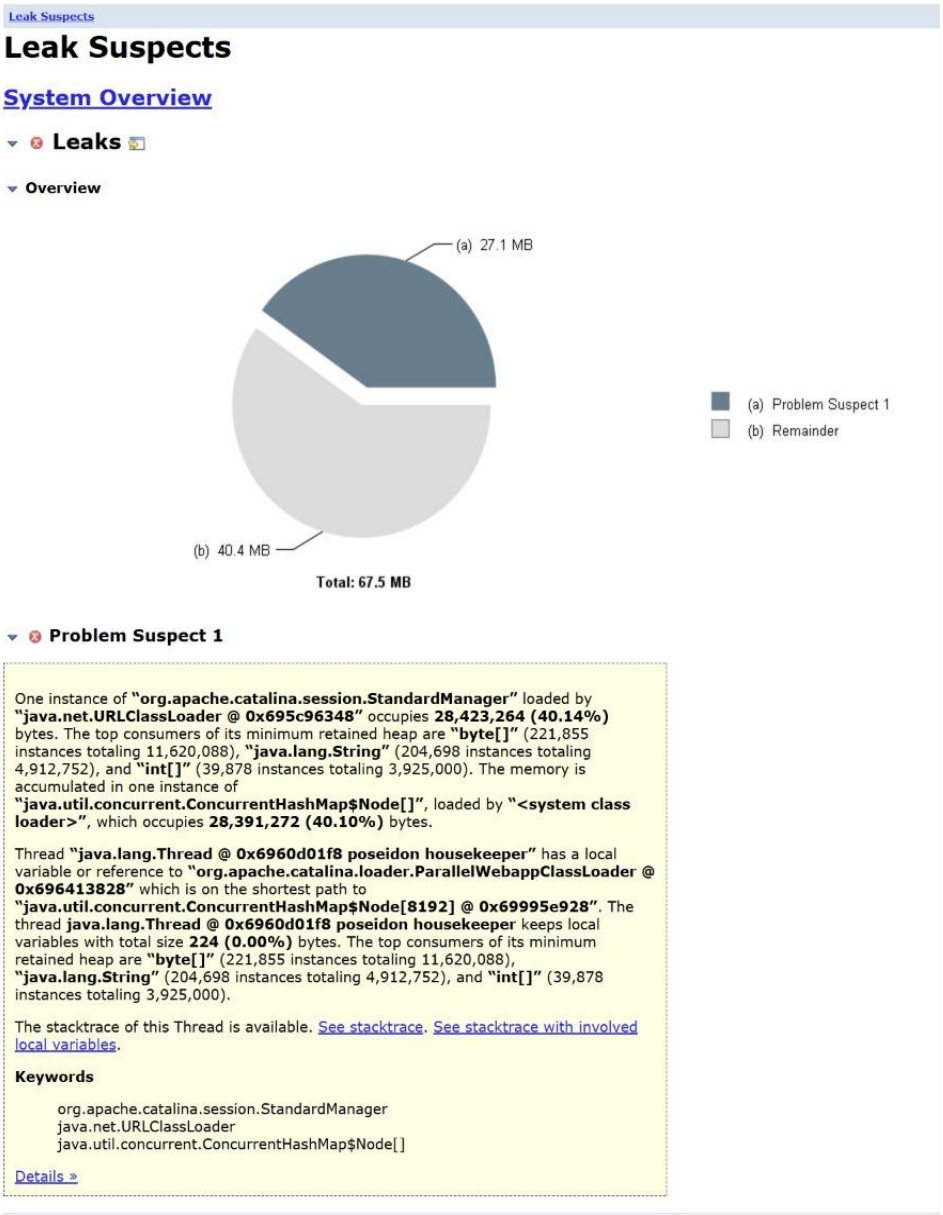
10.2.4. Retenedor de las Sesiones tras la Reingeniería

La sección 10.2.1 mostró que el administrador de sesiones de Tomcat (`StandardManager`) pasa de retener 1,36 GB a 27 MB. En la rama reingenierizada vuelve a encabezar el informe de

sospechosos de fuga de Eclipse MAT, como lo hacía en la anterior (sección 5.3); la Figura 26 lo muestra.

Figura 26

Tamaño retenido por el administrador de sesiones de Tomcat en la rama reingenierizada (EclipseMAT)



Esta vez, sin embargo, ese señalamiento no es una fuga.

Una fuga retiene objetos que ya no se usan y nunca se liberan, con un consumo que crece sin límite en el tiempo. Lo que el administrador retiene, en cambio, es el estado de las sesiones que seguían vivas en el momento de la captura, 5.688 en total, con unos cinco kilobytes cada una. Ese estado crece en proporción a las sesiones simultáneas, y Tomcat lo libera en cuanto cada sesión expira por inactividad.

El informe lo destaca porque el administrador, al custodiar todas las sesiones, es el mayor retenedor del **heap**, aunque no guarde nada que debiera haberse liberado ya.

Lo que esas sesiones retienen es estado propio de cada una, ya no la copia del menú. Su mayor componente es el par de claves RSA que el inicio de sesión genera y conserva para descifrar las credenciales que el cliente le envía cifradas; ese par de claves existe por igual en la rama anterior (sección 5.3) y no lo introdujo la reingeniería.

La diferencia entre las dos ramas no está en ese estado, sino en la copia del árbol de menús que la anterior sumaba a cada sesión, la clase *beans.Servicio* con 1.480.608 instancias, que en la reingenierizada desaparece del **heap** (sección 10.2.1).

10.3. Actividad 16: Verificación del Control de Acceso Tras la Reingeniería

La sección 5.4 diagnosticó una deficiencia de control de acceso en la rama anterior y la clasificó como control de acceso roto (A01:2025, **Broken Access Control**), la primera categoría del OWASP Top Ten (OWASP, 2025a).

Ese catálogo nombra los riesgos más frecuentes para concienciar sobre ellos, pero no fija requisitos comprobables.

Para verificar si el riesgo quedó cerrado, OWASP mantiene un documento distinto, el estándar de verificación de seguridad de aplicaciones (ASVS), que traduce cada riesgo en requisitos que se pueden comprobar uno a uno (OWASP, 2025b). Esta actividad verificó el cierre contra el capítulo de autorización de ese estándar, el V8, que corresponde al riesgo A01. Queda fuera el módulo de Grupos, que se rige por su propia autorización.

El cierre del riesgo es el resultado del arco de seguridad que recorren las Fases 2 a 5: el filtro único que centraliza el control (Fase 2), el gestor que decide sobre el recurso real (Fase 3), el inventario de todos los servicios y el cálculo de sus permisos que produjo cachama (Fase 4), y el modelo de datos que separa esos permisos del menú (Fase 5).

La Tabla 18 toma los dos requisitos del capítulo V8 del ASVS que corresponden al alcance de la reingeniería, la protección del acceso a los servicios según el tipo de usuario y su clasificador, y los cruza con la forma en que el diseño anterior los incumplía y con el resultado de ese arco.

Tabla 18

Requisitos de autorización del estándar OWASP ASVS 5.0 (V8) frente al diseño anterior y al reingenierizado

Requisito (ASVS 5.0, V8)	Diseño anterior (sección 5.4)	Diseño reingenierizado
El acceso se decide en una capa de servidor confiable, no sobre datos que el cliente pueda manipular (8.3.1)	El permiso se decidía sobre los parámetros *IdServicio* y *UriAnterior* que el cliente envía en la petición	El filtro y el gestor deciden en el servidor, sobre la URI real del recurso solicitado
El acceso a cada servicio se concede sólo a quien tiene	Las páginas fuera del menú no definían permisos propios y quedaban	Denegación por defecto sobre el inventario de servicios que cachama

Requisito (ASVS 5.0, V8)	Diseño anterior (sección 5.4)	Diseño reingenierizado
permiso explícito (8.2.1)	accesibles por una herencia laxa	calculó en la Fase 4 (capítulo 8): el acceso se concede sólo con el permiso explícito del clasificador

Nota. Ambos son requisitos de Nivel 1, el básico, del capítulo de autorización (V8) del ASVS 5.0 (OWASP, 2025b), y son los que atañen a la protección del acceso a los servicios, el objeto de esta reingeniería. Los demás requisitos de V8 corresponden a otras dimensiones de la autorización, como el acceso a los datos dentro de cada servicio (8.2.2), propio de la lógica de cada módulo. La columna del diseño anterior resume el diagnóstico de las secciones 5.4.1 a 5.4.4.

El resto de la sección verifica esos requisitos sobre el flujo reingenierizado: primero la decisión sobre el recurso real (8.3.1), que desactiva el vector de ataque; luego la cobertura de todos los servicios que hace efectiva la denegación por defecto (8.2.1); y, sobre ambas, la comprobación de que el control nuevo no rompió el acceso legítimo.

10.3.1. La Decisión Sobre el Recurso Solicitado

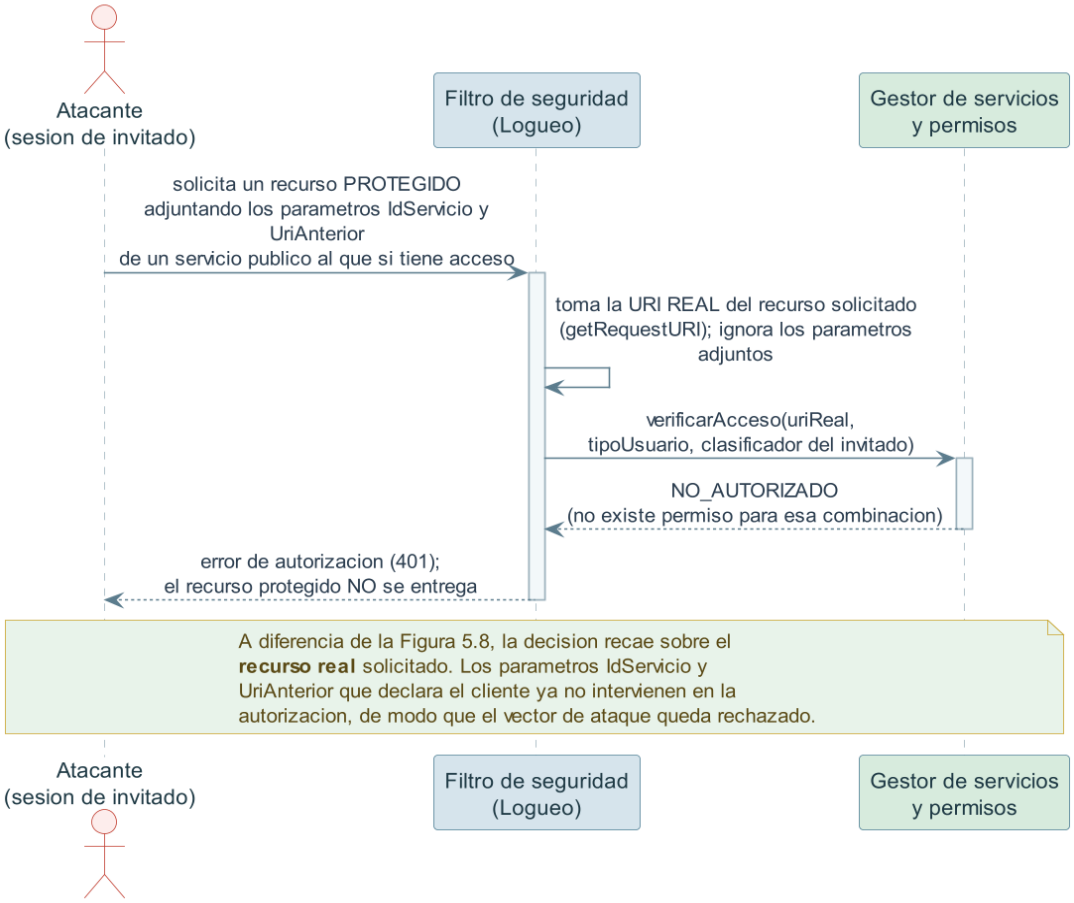
El requisito 8.3.1 exige que el control de acceso se imponga en una capa de servidor confiable. En la rama reingenierizada, toda petición pasa por un filtro de seguridad único (sección 6.2), que reemplaza la validación que el diseño anterior repartía por alrededor de 1.540 páginas, y delega la decisión en el gestor de servicios y permisos (sección 7.1.2). El gestor recibe la URI real del recurso solicitado, el tipo de usuario y su clasificador, y resuelve el permiso contra las autorizaciones precalculadas, en memoria. Los parámetros *IdServicio* y *UriAnterior*, que el diseño

anterior usaba para decidir, conservan un papel de compatibilidad y de presentación, pero ya no intervienen en la decisión de acceso.

Sobre ese flujo, el vector de ataque de la sección 5.4.3 deja de funcionar. Un visitante con sesión de invitado que solicita un recurso protegido y le adjunta el identificador y la URI de un servicio público ya no obtiene el recurso, porque el gestor evalúa la URI real del recurso protegido contra el clasificador del invitado, no halla permiso y deniega la petición. La Figura 10.4 reproduce el ataque de la Figura 27, ahora rechazado.

Figura 27

Vector de ataque de la Figura 5.8, rechazado por el control reingenierizado



10.3.2. La Cobertura de Todos los Servicios

El requisito 8.2.1 exige que el acceso a cada servicio se conceda solo a quien tiene permiso explícito, lo que supone que todos los servicios tengan sus permisos definidos. El segundo nivel de exposición de la sección 5.4.4, las páginas alcanzables que nunca se registraron como servicios, se cierra por la combinación de dos cambios. El gestor deniega por defecto cualquier URI que no esté registrada, y una página fuera del catálogo deja de quedar accesible por omisión. Y el inventario que construyó cachama en la Fase 4 registró los 1.789 servicios alcanzables de la plataforma, no solo los del menú de navegación, y calculó el permiso de cada uno: 114 públicos, 612 con acceso por clasificador y 1.063 sin acceso, denegados por defecto (capítulo 8). Las páginas internas que antes quedaban sin control efectivo pasan, así, a tener un permiso explícito.

10.3.3. La Conservación del Acceso Legítimo

Un control que deniega por defecto y decide sobre el recurso real es más estricto que el anterior, y la verificación incluyó comprobar que no bloqueara el acceso legítimo. La suite de pruebas de extremo a extremo descrita en la sección 5.1 se ejecutó sobre la rama reingenierizada, con el catálogo de servicios ya poblado por cachama, y las cuatro pruebas se completaron sin fallos (Tabla 19, Figura 28). El caso más exigente es el trámite de trabajo de grado, cuyas once etapas (sección 5.1) recorren los distintos actores del trámite, de los autores a los jurados, y se apoyan en páginas de proceso que se alcanzan desde una opción del menú pero no figuran en él, es decir, servicios sueltos. Que ese flujo se complete sin fallos confirma que cachama registró los servicios sueltos de los que depende y que el control nuevo no cerró el paso a los usuarios legítimos.

Tabla 19

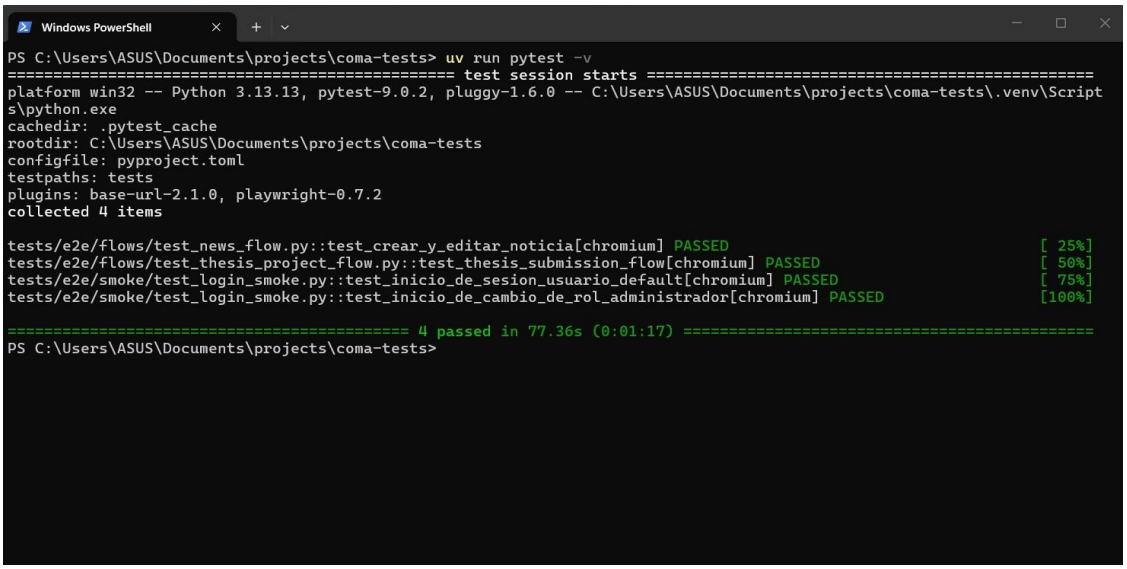
Resultado de las pruebas de extremo a extremo sobre la rama reingenierizada

Prueba de extremo a extremo	Qué verifica	Resultado
Inicio de sesión del usuario general	autenticación y carga del menú de la sesión	Sin fallos
Cambio de tipo de usuario a administrador	alternancia a la sesión administrativa	Sin fallos
Creación y edición de una noticia	flujo de gestión de noticias	Sin fallos
Trámite completo de un trabajo de grado	las once etapas, de varios actores, sobre los servicios sueltos del flujo (sección 5.1)	Sin fallos

Nota. Pruebas ejecutadas sobre la rama reingenierizada con el catálogo de servicios poblado por *cachama*; la corrida se muestra en la Figura 10.5. La misma suite sobre la rama anterior se presentó en la sección 5.1 (Figura 5.2). Las pruebas cubren los flujos principales de la plataforma, no la totalidad del catálogo de servicios.

Figura 28

Ejecución de la suite de pruebas de extremo a extremo sobre la rama reingenierizada



10.3.4. Sintesis

La causa estructural que la sección 5.4.5 señaló detrás de las dos deficiencias, la navegación, los servicios y los permisos acoplados alrededor de *TB_Menu* y de una rutina repetida en cada página, queda eliminada. La navegación y el control de acceso son ahora responsabilidades separadas: el menú decide qué ve cada usuario (sección 7.1.1) y el gestor de servicios y permisos decide qué recurso puede solicitar, sobre la URI real y exista o no en el menú (sección 7.1.2). Los requisitos de autorización del ASVS V8 que el diseño anterior incumplía quedan satisfechos, y el vector de ataque diagnosticado en la Fase 1 deja de tener efecto.

10.4. Actividad 17: Análisis Integral de los Resultados

Las tres actividades anteriores midieron por separado la latencia, la memoria y el control de acceso, y reportaron sus datos en las plantillas del plan de pruebas (Apéndice A). Esta actividad reúne esos resultados, los contrasta con la línea base de la Fase 1 y extrae de ellos la conclusión de la evaluación, que la plantilla de síntesis del plan recoge. Las conclusiones generales del trabajo se desarrollan en el capítulo de Conclusiones.

10.4.1. Resultados Consolidados

La Tabla 20 reúne las métricas de rendimiento y memoria de las dos ramas bajo 200 usuarios concurrentes, el nivel de mayor exigencia. Las mejoras son grandes y van todas en la misma dirección: la latencia de la autenticación y el consumo de memoria caen más del 95 %, la capacidad de atención sube casi a la mitad y el recolector de basura trabaja menos de una cuarta parte que antes (Figura 29).

Tabla 20

Resumen de las métricas de la Fase 1 y la Fase 6 bajo 200 usuarios concurrentes

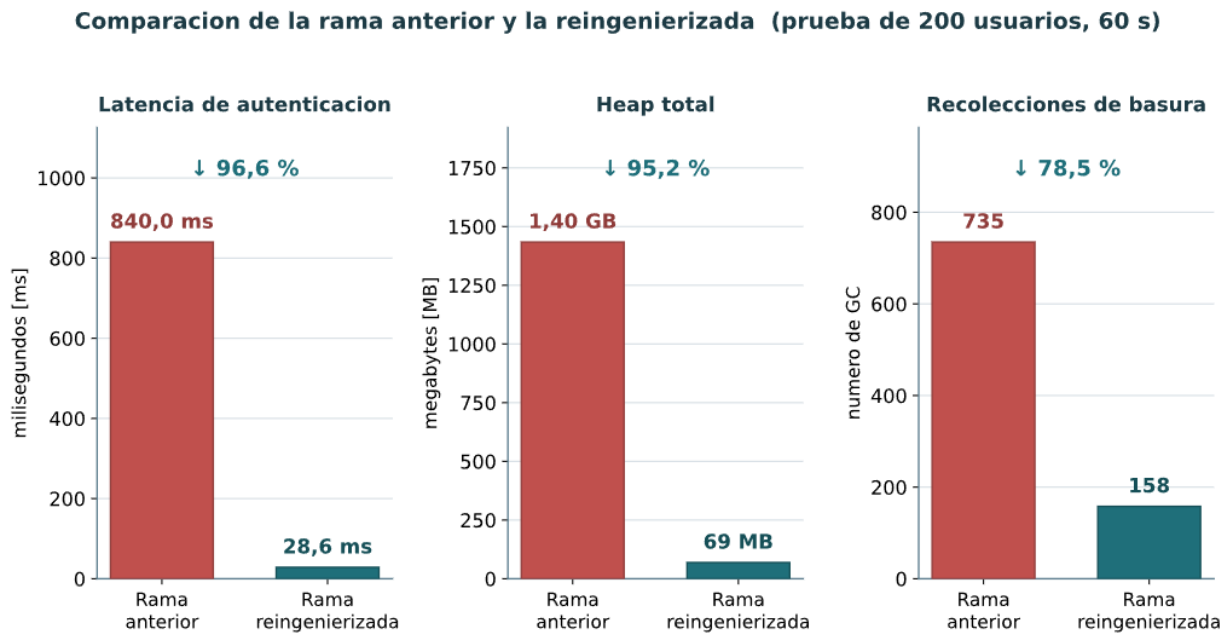
Métrica	Anterior	Reingenierizada	Variación
Latencia de la autenticación (ms)	840,04	28,58	-96,6%
Autenticaciones completadas en sesenta segundos	3.910	5.751	+47%
Heap total	1,40 GB	69 MB	-95,2%
Heap por inicio de sesión (KB)	289,59	12,45	-95,7%
Recolecciones de basura en sesenta segundos	733	156	-78,7%

Nota. Las dos ramas con el pool de conexiones aplicado (sección 5.2.2). Datos de SISIFO y del registro de vuelo de la máquina virtual Java; el detalle por nivel de concurrencia está en las Tablas 10.1 y 10.4 y en la sección 10.2.2.

El control de acceso, que no se expresa en cifras de rendimiento, mejoró en la misma dirección. El vector de ataque que la sección 5.4.3 había diagnosticado, solicitar un recurso protegido declarando un servicio público, deja de funcionar, porque la decisión recae sobre la URI real del recurso y no sobre los datos que el cliente declara (sección 10.3.1). Los 1.789 servicios alcanzables de la plataforma pasan a tener permiso explícito, con denegación por defecto para cualquier recurso no autorizado (sección 10.3.2), y las cuatro pruebas de extremo a extremo no muestran regresión en los flujos legítimos (Tabla 20).

Figura 29

Reducción de la latencia, la memoria y la recolección de basura tras la reingeniería



10.4.2. Una Raíz Común

Las mejoras de la Tabla 10.8 no son independientes. Detrás de las tres está el mismo diseño, el árbol de menús construido sobre la tabla *TB_Menu*, que el diseño anterior reconstruía en cada autenticación, replicaba en cada sesión y empleaba además como base de los permisos de acceso. La sección 10.3.4 señaló ese acoplamiento como la causa del control de acceso roto; es también la de la latencia y la de la memoria. La reconstrucción del árbol en cada inicio de sesión cargaba la latencia, que crecía con la concurrencia hasta los 840 ms (sección 10.1). La copia del árbol en cada sesión llenaba el **heap**, que llegaba a 1,40 GB retenido por las sesiones (sección 10.2). Y el permiso de acceso, resuelto a partir del menú y no del recurso solicitado, dejaba abierto el vector de ataque y las páginas ajenas al menú sin control (sección 10.3).

La reingeniería actuó sobre esa raíz, no sobre cada síntoma por separado. El gestor de menús arma la navegación una sola vez y la comparte entre todas las sesiones (sección 7.1.1), con lo que el árbol deja de reconstruirse y de replicarse, y ceden a la vez la latencia y la memoria. El gestor de servicios y permisos (sección 7.1.2) saca la decisión de acceso del menú y la lleva al recurso real, sobre el inventario de servicios que cachama calculó en la Fase 4 (capítulo 8), y el control de acceso deja de depender de la navegación. Separar lo que estaba acoplado es lo que hace que un solo cambio de diseño mejore tres dimensiones que parecían distintas.

10.4.3. Alcance de la Evaluación

Los resultados se leen dentro de las condiciones en que se midieron. Las dos ramas se ejecutaron bajo la misma carga y el mismo entorno, con el pool de conexiones aplicado por igual, que actúa como una constante y permite atribuir las diferencias a los cambios de código (sección 5.2.2). Las cifras corresponden al ciclo de reautenticación que SISIFO ejerce sobre el inicio de sesión y el armado del menú, el punto donde se concentraba el costo del diseño anterior, y no a una medición de toda la plataforma.

La verificación de seguridad corresponde al cierre del riesgo que la sección 10.3 comprobó, el control de acceso roto, categoría A01 del OWASP Top Ten. Dentro de ese alcance, los tres problemas que la Fase 1 había diagnosticado, la latencia, la memoria y el control de acceso, quedan resueltos.

11. Conclusiones

A lo largo de seis fases, este trabajo llevó a cabo la reingeniería del sistema de gestión de menús y permisos de la plataforma COMA, desde el diagnóstico del estado inicial hasta la evaluación posterior a los cambios. De ese recorrido se extraen las conclusiones que siguen.

11.1. El Origen Común de los Tres Problemas

Los tres problemas que la evaluación inicial atribuyó a la gestión de menús, la latencia del inicio de sesión, el consumo de memoria y el control de acceso roto, tenían una sola causa. El árbol de menús construido sobre la tabla *TB_Menu* se reconstruía en cada autenticación, se copiaba en cada sesión y servía además de base para decidir los permisos, y los tres se originaban en ese diseño (secciones 10.1 a 10.3). Por eso rehacerlo, con una única estructura compartida en memoria y la decisión de acceso tomada sobre el recurso que se solicita, mejoró las tres dimensiones.

El peso que la plataforma arrastraba venía de cómo almacenaba el menú. La misma navegación se sostiene con una fracción de la memoria y del tiempo cuando el menú deja de copiarse en cada sesión y pasa a compartirse; el gasto estaba en duplicar una estructura que podía ser única.

La seguridad del control de acceso se ganó en el diseño. El vector de ataque se cerró al decidir sobre el recurso que realmente se solicita (la dirección de la petición que llega al servidor) y exigir un permiso explícito para acceder a cada servicio, en lugar de depender de que cada página recordara validarse y de un identificador declarado por el cliente.

Más allá de COMA, el trabajo deja un caso de reingeniería en el que un rediseño estructural, guiado por un diagnóstico y verificado con métricas comparables, corrigió a la vez el rendimiento y la seguridad de un sistema en producción sin reescribir su funcionalidad.

11.2. El Logro de los Objetivos

Los cinco objetivos específicos trazaron el recorrido del trabajo. La evaluación del estado inicial diagnosticó las falencias del sistema, y ese diagnóstico orientó cuanto vino después: centralizar la arquitectura en un filtro de seguridad y dos gestores en memoria sin duplicación de código, rediseñar la gestión de menús sobre un almacenamiento de carga única, separar la lógica de permisos de la de menús en el código y en la base de datos, y suprimir los procesos redundantes al crear y consultar el menú. Con ello se dio cumplimiento al objetivo general de rediseñar la gestión de menús y permisos de COMA para mejorar la memoria, los tiempos de carga y la seguridad.

11.3. La Metodología y su Aporte

La metodología organizó el trabajo en seis fases sucesivas, y dos de sus rasgos resultaron decisivos. El plan de pruebas que fijó la Fase 1 (Apéndice A) hizo comparable el estado inicial con el final, porque medir ambos con el mismo procedimiento permite atribuir las diferencias a los cambios de código y no a la forma de medir. El orden de las fases, además, dejó la reestructuración de la base de datos para el final, después de construir los gestores en memoria (Fase 3) y de levantar el inventario de servicios con cachama (Fase 4). Al llegar a la base de datos con los gestores y el catálogo ya definidos, se supo de antemano qué tablas separar y qué columnas renombrar, y la intervención se limitó a lo necesario.

11.4. Las Herramientas que Quedan

La reingeniería obligó a construir sus propias herramientas, y cada una alcanza más allá de este trabajo. SISIFO, el entorno con que se midieron el rendimiento y la memoria, es un componente reutilizable que establece para el grupo CALUMET un modo de hacer pruebas de rendimiento, con dos versiones sometidas a la misma carga, útil para otros sistemas del grupo. La suite coma-tests es el primer conjunto de pruebas de extremo a extremo con que cuenta COMA, y el punto de partida de las que se automaticen en adelante. cachama es la menos reutilizable de las tres, atada al inventario de servicios de COMA, pero deja un precedente de cómo abordar un código heredado y acumulado por generaciones de desarrolladores, reconstruyendo el grafo de la plataforma para entenderlo y clasificarlo cuando ya no existe un mapa completo de lo que hay.

12. Recomendaciones

Además de corregir los problemas que la evaluación inicial había diagnosticado, el trabajo dejó abiertas varias líneas de continuación. Las recomendaciones que siguen empiezan por las dos áreas centrales del trabajo, el rendimiento y la operación de la plataforma y la seguridad, y cierran con el aprovechamiento de los instrumentos.

12.1. El Rendimiento y la Operación

Medir el rendimiento exigió primero resolver el agotamiento de conexiones a la base de datos, que hacía caer el servidor bajo carga moderada (sección 5.2.2). Se recomienda consolidar el manejo fiable de conexiones, con el pool y el cierre de sus recursos, como práctica estándar del acceso a datos, de forma que el agotamiento no reaparezca al crecer la plataforma.

El rendimiento de la plataforma se midió a lo largo del trabajo (Apendice A). Se sugiere instrumentarlo también en producción, la latencia, la memoria y los errores, para seguir su comportamiento bajo el uso real.

En la plataforma reingenierizada, el mayor retenedor de memoria que queda es el estado de sesión, y en particular las claves de cifrado que cada sesión guarda (sección 10.2.4), que son estado legítimo y se liberan al expirar la sesión. Conviene vigilarlo si el consumo de memoria vuelve a crecer, porque marca el siguiente límite natural de la plataforma.

12.2. La Seguridad

La seguridad fue una de las tres áreas del trabajo. El nuevo control de acceso decide sobre el recurso realmente solicitado y concede el paso solo con permiso explícito, y ese diseño se verificó contra el estándar OWASP ASVS (sección 10.3). Tres recomendaciones prolongan ese resultado.

Se recomienda conservar el estándar OWASP ASVS como marco de verificación del control de acceso conforme la plataforma evoluciona; así cada revisión se mide contra un criterio establecido.

Las pruebas de extremo a extremo ya comprueban que el control de acceso no rompe el acceso legítimo de los usuarios (sección 10.3). Conviene mantener esa comprobación en cada versión de la plataforma. Cada una confirma que el acceso se sigue decidiendo sobre el recurso solicitado y con permiso explícito.

Se recomienda, por último, sostener esos dos criterios, la decisión sobre el recurso solicitado y el permiso explícito con denegación por defecto, en el diseño de cada servicio nuevo, para que se integre al control de acceso central en lugar de resolver el acceso por su cuenta.

12.3. El Aprovechamiento de los Instrumentos

El plan de pruebas de la Fase 1 (Apendice A) midió el estado inicial y el final con el mismo procedimiento, y esa igualdad de método hizo comparables las dos medidas. Como las pruebas de rendimiento y de extremo a extremo están por completo automatizadas, se recomienda conservarlas como arnés de regresión e incorporarlas a un flujo de integración continua. Con ello, cada cambio se contrasta contra la misma referencia, y sus regresiones en el rendimiento, la memoria o el funcionamiento afloran al integrarlo.

La suite coma-tests cubre hoy los flujos principales de la plataforma, no la totalidad de sus servicios (sección 10.3). Conviene ampliarla de forma gradual hacia los demás flujos, de modo que la comprobación de que un cambio no rompe lo que ya funciona alcance una porción cada vez mayor de COMA.

SISIFO despliega la plataforma en contenedores y compara dos versiones bajo la misma carga (sección 5.2). Por ser un componente reutilizable, conviene aplicarlo a la evaluación de rendimiento de otros sistemas del grupo CALUMET, con el mismo método de contraste entre versiones.

El grafo de dependencias que cachama construye reveló, además del inventario que buscaba, servicios que con el tiempo dejaron de usarse pero nunca se retiraron (Fase 4). Se recomienda ejecutar cachama de forma periódica como herramienta de mantenimiento que localice esos servicios en desuso y los prepare para su retiro.

Referencias

Acumbamail. (s. f.). Template. Glosario de Acumbamail. https://acumbamail.com/glosario/template/

Calumet, G. (s. f.). Calumet Estándar. Escuela de Ingeniería de Sistemas e Informática, UIS. https://ingsistemas.uis.edu.co/eisi/Calumet/Estandar/

Calumet. (s. f.). CALUMET - Grupo de Innovación y Desarrollo. Escuela de Ingeniería de Sistemas e Informática, UIS. https://ingsistemas.uis.edu.co/eisi/grupo/calumet/

Carlos, R. D. L., & Angel, L. B. R. (2010). Administración, soporte a usuarios, mantenimiento del portal web eisiweb, análisis, diseño, desarrollo e implementación de nuevos servicios para el portal web de la escuela de ingeniería de sistemas e informática. Repositorio Institucional UIS. https://noesis.uis.edu.co/items/cb4e0fd5-8f24-423e-80ba-c151ad627b57

Grupo de Desarrollo de Software Calumet. (s. f.). Página oficial. Escuela de Ingeniería de Sistemas e Informática, UIS. https://ingsistemas.uis.edu.co/eisi/eisi.jsp

HTML: Lenguaje de etiquetas de hipertexto. (s. f.). MDN Web Docs. https://developer.mozilla.org/es/docs/Web/HTML

IBM. (s. f.). JavaServer Pages (JSP) technology. IBM Documentation. https://www.ibm.com/docs/es/dmrt/9.5

Interaction Design Foundation. (s.f.). What is User Experience (UX) Design? Recuperado de https://www.interaction-design.org/literature/topics/ux-design

Krekel, H., Oliveira, B., Pfannschmidt, R., Bruynooghe, F., Laugher, B., & Bruhin, F. (2024). *pytest* (versión 8.x). Recuperado de https://docs.pytest.org

MDN contributors. (2025, July 18). Responsive Web Design (RWD). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Glossary/Responsive_web_design

Microsoft. (2024). *Playwright for Python*. Recuperado de https://playwright.dev/python/docs/intro

Orlando, M. P. O. (2016). Mantenimiento del portal web, análisis, desarrollo e implementación de nuevos servicios para el sistema comunidad académica que soporta los portales web de las escuelas y facultades. Repositorio Institucional UIS. https://noesis.uis.edu.co/items/caa94822-bc42-48e6-97af-1336a0d991a5

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM, 21*(2), 120–126. https://doi.org/10.1145/359340.359342

Schade, A. (2014, May 4). Responsive Web Design (RWD) and user experience. Nielsen Norman Group. https://www.nngroup.com/articles/responsive-web-design-definition/

Socialmood. (2015, 1 de abril). ¿Qué es el Diseño Responsive? 40deFiebre. https://www.40defiebre.com/que-es/diseno-responsive

Tomcat. (s. f.). En Wikipedia, la enciclopedia libre. Recuperado el 31 de mayo de 2024 de https://es.wikipedia.org/wiki/Tomcat

Wikipedia. (2025, 31 de enero). JavaScript. Wikipedia, la Enciclopedia Libre. https://es.wikipedia.org/wiki/JavaScript

World Wide Web Consortium (W3C). (s. f.). *What is web accessibility?* Recuperado de https://www.w3.org/WAI/fundamentals/accessibility-intro/

World Wide Web Consortium. (s.f.). Cascading Style Sheets (CSS). Recuperado de https://www.w3.org/Style/CSS/

Yessenia, R. C. A. (2019). Análisis, mantenimiento, diseño, desarrollo e implementación de nuevas funcionalidades y mejoras relacionadas con los módulos de portal web de grupos y de trabajos de grado en la plataforma COMA comunidad académica. Repositorio Institucional UIS. https://noesis.uis.edu.co/items/b38282aa-4bed-4182-b60e-a4c67f1d6e4a