(05)

Casa abierta al tiempo

UNIVERSIDAD AUTÓNOMA METROPOLITANA

Auditoría de privacidad en PriServ

Idónea Comunicación de Resultados
para obtener el grado de
MAESTRO EN CIENCIAS
Presenta: Raziel Carvajal Gómez

Asesoras:
Dra. Elizabeth Pérez Cortés
Dra. Patricia Serrano Alvarado

25 de septiembre de 2011

Agradecimientos

El presente trabajo de investigación está dedicado a mis padres Neira y Lucio, les agradezco su apoyo incondicional, así como el amor y la comprensión que todos los días me brindan.

Fue muy grato que la Dra. Elizabeth Pérez Cortés y la Dra. Patricia Serrano Alvarado, me hayan guiado en este camino. Su sabiduría, profesionalismo y calidez humana lograron que me haya enamorado de la investigación, para mi siempre serán una fuente de inspiración.

A la Dra. Reyna Carolina Medina Ramírez y al Dr. Santiago Domínguez Domínguez, les expreso mi gratitud ya que sus comentarios fueron indispensables para enriquecer el presente trabajo.

Por el financiamiento que en dos años el Consejo Nacional de Ciencia y Tecnología me ofreció para mis estudios, muchas gracias.

Resumen

Cada día aumenta la cantidad de información que se comparte por diferentes sistemas de cómputo. En este contexto, la compartición de datos en los sistemas par a par (P2P acrónimo en inglés de *Peer-to-Peer*) puede divulgar datos privados, porque cada par (nodo) puede acceder a los datos del sistema y utilizarlos para diferentes propósitos. Para preservar la privacidad de los datos en estos sistemas, no basta con permitir que el propietario de datos especifique sus políticas de privacidad, sino también hay que proveerlos con herramientas de auditoría para verificar el cumplimiento de dichas políticas.

En el presente trabajo de investigación se desarrolló un servicio de auditoría de privacidad para sistemas P2P, el cual verifica el cumplimiento de las políticas de privacidad que PriServ (servicio de privacidad para sistemas P2P) asocia a los datos.

Para realizar el servicio de auditoría, en primer lugar se identificaron las acciones que violan las políticas de privacidad, posteriormente se realizó el diseño de un algoritmo de auditoría y del componente *Audit Manager* que ejecuta el algoritmo propuesto, finalmente se desarrolló un prototipo que emula un sistema P2P y se realizó la evaluación del algoritmo de auditoría mediante un caso de prueba. Al observar los resultados de evaluar el tiempo que requiere la auditoría, se observó que estos son consistentes con el análisis del orden del algoritmo.

Contenido

Lista de Figuras

Lista de Tablas

Capítulo 1
Introducción

La información personal que se encuentra distribuida en Internet, en diferentes servidores alrededor del mundo aumenta cada día. Desafortunadamente en muchos casos los sistemas de información comparten los datos que administran, sin considerar la privacidad de los usuarios. Sólo el propietario de datos tiene el derecho de especificar cuándo, cómo y qué información comparte con otros. Para estos fines, los sistemas de información deben permitir la especificación de políticas de privacidad para los datos y respetarlas.

Algunas propuestas para proteger la privacidad de los datos sugieren utilizar las legislaciones actuales de privacidad, como la Ley de privacidad de Estados Unidos o los lineamientos de la Organización para el Desarrollo y Cooperación Económica [1, 2], como base para la creación de políticas de privacidad. En su caso, el proyecto P3P (*Plataform for Privacy Preferences*) del W3C (*World Wide Web Consortium*) permite a los usuarios de sitios Web especificar sus políticas de privacidad usando archivos XML, para que al comunicarse con otros sitios Web no se divulguen sus datos privados [3].

En este proyecto, nos interesamos en la compartición de datos en los sistemas par a par (P2P acrónimo en inglés de *Peer-to-Peer*) porque puede considerarse como una fuente de riesgo para la privacidad. Un sistema P2P es un sistema distribuido, en donde los pares (nodos) que componen al sistema pueden tomar el rol de clientes o de servidores; algunas ventajas de los sistemas P2P son: compartir grandes cantidades de datos, ser descentralizados y auto-organizados. En principio, cada par puede acceder y utilizar los datos contenidos en el sistema para distintos propósitos, lo cual puede ocasionar el uso no autorizado de datos privados.

Se requiere el uso de estrategias que eviten la divulgación de datos privados en un sistema P2P. En particular, el servicio de privacidad PriServ [4] permite asociar políticas de privacidad a los datos de un sistema P2P y garantiza la entrega de datos solo a pares autorizados para los propósitos y las operaciones (e. g., lectura o escritura) especificadas. Sin embargo, no basta que un sistema proporcione mecanismos para especificar las políticas de privacidad de

los datos en un sistema P2P, sino que también es necesario verificar el cumplimiento de dichas políticas de privacidad, pues es posible que los pares realicen acciones para violarlas, guiados por intereses particulares. Considere por ejemplo que María se encuentra en un hospital para obtener los resultados de una prueba de embarazo, la cual es positiva. Algunos días después, María recibe un correo electrónico de una tienda departamental que ofrece descuentos especiales a mujeres embarazadas. El hospital violó la privacidad de María, porque divulgó el resultado de su prueba de embarazo. Al verificar el cumplimiento de políticas de privacidad, María puede conocer la política que fue violada y obtener información de cuándo y cómo fueron divulgados sus datos privados. El hospital también se puede beneficiar al identificar y corregir oportunamente, las acciones que violen la privacidad de los pacientes.

Una manera de verificar el cumplimiento de las políticas de privacidad es realizar una auditoría. **La auditoría es el proceso de evaluar que las acciones efectuadas sobre los datos cumplen con las políticas establecidas, mediante la documentación de dichas acciones.**

Los objetivos del presente trabajo de investigación son los siguientes:

- Estudiar y analizar las técnicas existentes de auditoría de datos.
- Proponer e implementar un algoritmo de auditoría de datos de acuerdo a políticas de privacidad individuales.

La metodología utilizada fue:

- Estudiar modelos de privacidad de datos y técnicas en auditoría de privacidad en dos áreas: sistemas de bases de datos y sistemas distribuidos.
- Identificar escenarios que violan las políticas de privacidad en un sistema P2P.
- Proponer un algoritmo de auditoría de privacidad, que identifique a los pares que violan políticas de privacidad y a los pares que accedieron a los datos de un propietario.
- Diseñar el componente de auditoría *Audit Manager*, que se comunica con los componentes de PriServ (servicio de privacidad utilizado) para ejecutar el algoritmo propuesto.
- Implementar, probar y evaluar el algoritmo de auditoría, utilizando la plataforma SPLAY.

Como resultado de construir un servicio de auditoría de privacidad para redes P2P, las aportaciones del presente trabajo de investigación son las siguientes:

- Una revisión del estado del arte en auditoría de privacidad en sistemas de bases de datos y en sistemas distribuidos.
- Cuatro escenarios que conducen a violaciones a las políticas de privacidad de datos en un sistema distribuido.
- Un algoritmo de auditoría de privacidad para sistemas P2P.
- Un prototipo del componente *Audit Manager* que realiza el algoritmo de auditoría propuesto y extiende la arquitectura del servicio de privacidad para sistemas P2P, PriServ.

El resto del documento se organiza de la siguiente manera. En el Capítulo 2 se presentan los conceptos básicos y las propuestas para realizar la auditoría de la privacidad en sistemas de bases de datos y sistemas distribuidos. Dado que el servicio de auditoría propuesto extiende a PriServ, en el Capítulo 3 se describe este servicio de privacidad. Posteriormente en el Capítulo 4, se detalla la problemática, la propuesta del servicio de auditoría, la metodología a seguir y el diseño del servicio de auditoría; mientras que el detalle de la implementación y los resultados obtenidos de evaluar el servicio de auditoría se describen en el Capítulo 5. Finalmente, en el Capítulo 6 se presentan las conclusiones y el trabajo futuro.

Capítulo 2
Revisión del estado del arte

En este capítulo se presentan los conceptos básicos para la realización de este proyecto, así como la literatura que guía esta investigación en auditoría de privacidad de datos. Principalmente se describen algunas técnicas de auditoría documentadas en dos áreas: sistemas de bases de datos y sistemas distribuidos.

2.1. Privacidad y auditoría

A continuación se enuncian las definiciones de privacidad y de auditoría, porque su comprensión es fundamental para el presente trabajo de investigación. Además, se describen dos tipos de auditoría.

La **privacidad** es: *"El derecho de los individuos a determinar cuándo, cómo y qué información relacionada con ellos puede ser revelada"* [1]. En el contexto de esta investigación una política de privacidad de datos, se entiende como una regla o lineamiento para especificar qué datos se pueden compartir, con quién y para qué propósito. El propósito representa el objetivo de uso para los datos, por ejemplo los datos asociados con un propósito educativo, sólo deben utilizarse como instrumento de enseñanza. Especificar políticas de privacidad permite que un propietario agregue privacidad a sus datos.

La **auditoría de privacidad** es: *"El proceso de evaluar que las acciones realizadas a los datos cumplen con las políticas de privacidad establecidas"*. La auditoría utiliza una bitácora donde se registra cada acción que se hace con los datos, la cual es revisada cuando se evalúa el cumplimiento de las políticas de privacidad. La auditoría se puede clasificar en función del momento en el cuál se realiza la evaluación de las acciones. Generalmente existen dos tipos de auditoría [5]:

- **Auditoría retroactiva (*offline audit*).** Un auditor retroactivo responde a la pregunta: *¿Las consultas pasadas pusieron en riesgo la privacidad de los datos?* Este tipo de auditoría verifica que las acciones realizadas en el pasado, no hayan violado las políticas

de privacidad de los propietarios. Busca en las bitácoras de manera periódica, acciones que violan las políticas de privacidad y las reporta pero no previene la divulgación de datos privados, pues las acciones identificadas ya ocurrieron.

- **Auditoría preventiva (*online audit*).** Un auditor preventivo responde a la pregunta: *¿La consulta q_{nueva} pone en riesgo la privacidad de los datos?* Este tipo de auditoría evita la divulgación de datos privados, al determinar si una consulta q_{nueva} debe ser rechazada o aceptada. Con la información de las consultas rechazadas y aceptadas registradas en la bitácora, la auditoría preventiva evalúa el riesgo de que una consulta q_{nueva} viole las políticas de privacidad.

2.2. Requerimientos para la auditoría

La auditoría de privacidad se puede realizar si se consideran dos requerimientos:

1. El uso de un modelo que asocie políticas de privacidad a los datos.
2. La gestión de bitácoras para registrar información de auditoría, sobre las consultas o acciones realizadas a los datos.

Además, la auditoría sólo debe evaluar el cumplimiento de los requerimientos que el modelo de privacidad satisface. Sin un modelo de privacidad no es posible evaluar cuándo una acción o consulta divulga datos privados, porque no se cuenta con una política de privacidad que se deba respetar. De esta manera, si un modelo de privacidad garantiza la integridad y control de acceso a datos, la auditoría sólo evalúa que no se corrompan los datos y se entreguen datos a usuarios autorizados.

Sin importar el tipo de auditoría que se lleve a cabo, se necesita consultar la información de auditoría registrada en las bitácoras, dicha información debe estar completa y no puede ser modificada, de no ser así, el auditor estaría utilizando información falsa.

2.3. Auditoría de privacidad en Sistemas de Bases de Datos (SBD)

Un SBD es un sistema computarizado que tiene como propósito mantener información y hacer que esté disponible cuando se solicite [6]. Sus funciones son administrar datos persistentes, permitir el acceso eficiente a datos, efectuar la gestión de transacciones y controlar el acceso a datos. También se puede definir a un SBD como una herramienta para el manejo de información.

Los SBD no tienen como objetivo ser responsables de la privacidad de los datos y como se mencionó en la Sección 2.2, la auditoría necesita considerar un modelo de privacidad, por lo que es necesario extender su diseño.

2.3.1. Base de Datos Hipocrática (BDH)

Inspirándose en el juramento de Hipócrates[1], una **Base de Datos Hipocrática** (BDH) es un modelo que es responsable de la privacidad de los datos que administra [1]. El diseño de una BDH cumple algunas regulaciones y lineamientos de privacidad internacionales, tales como la Ley de privacidad de Estados Unidos, El Modelo de Código Estándar de la Asociación Canadiense para la protección de Información Personal y los lineamientos de la Organización para el Desarrollo y Cooperación Económica.

Un SBD ofrece un ambiente que permite obtener y almacenar información de manera eficiente; una BDH debe realizar las actividades de un SBD, pero requiere de mecanismos que no comprometan la información privada. Por ejemplo, la eficiencia (que continúa siendo importante) puede no ser un objetivo central de una BDH. Una BDH opera bajo los siguientes diez principios:

1. **Especificación de propósitos.** Los datos tienen asociado un propósito para su registro.
2. **Consentimiento.** El propósito definido para la información personal, debe tener autorización del propietario.
3. **Colección limitada.** Colectar sólo la cantidad de datos necesaria para cumplir el propósito.
4. **Uso limitado.** Todas las consultas a la BD deben ser consistentes con el propósito.
5. **Limitación de la divulgación.** Los datos que se obtienen de la BD deben tener el consentimiento del propietario.
6. **Limitación de la retención.** Los datos se almacenarán sólo durante el tiempo especificado.
7. **Precisión.** La información del propietario debe ser legítima y estar actualizada.
8. **Seguridad.** La información del usuario debe estar protegida.

[1] *Guardaré silencio sobre todo aquello que dentro de mi profesión o fuera de ella, oiga o vea de la vida de los hombres que no deba ser público, manteniendo esas cosas de manera que no se pueda hablar de ellas* (fragmento del juramento de Hipócrates que se refiere a privacidad).

9. **Apertura.** El usuario será capaz de modificar y consultar su información.

10. **Cumplimiento.** El propietario de la información puede verificar el cumplimiento de todos los puntos anteriores.

En la Figura 2.1 se muestra la arquitectura de una BDH presentada en [1], el funcionamiento de sus componentes cumplen los diez principios de una BDH.

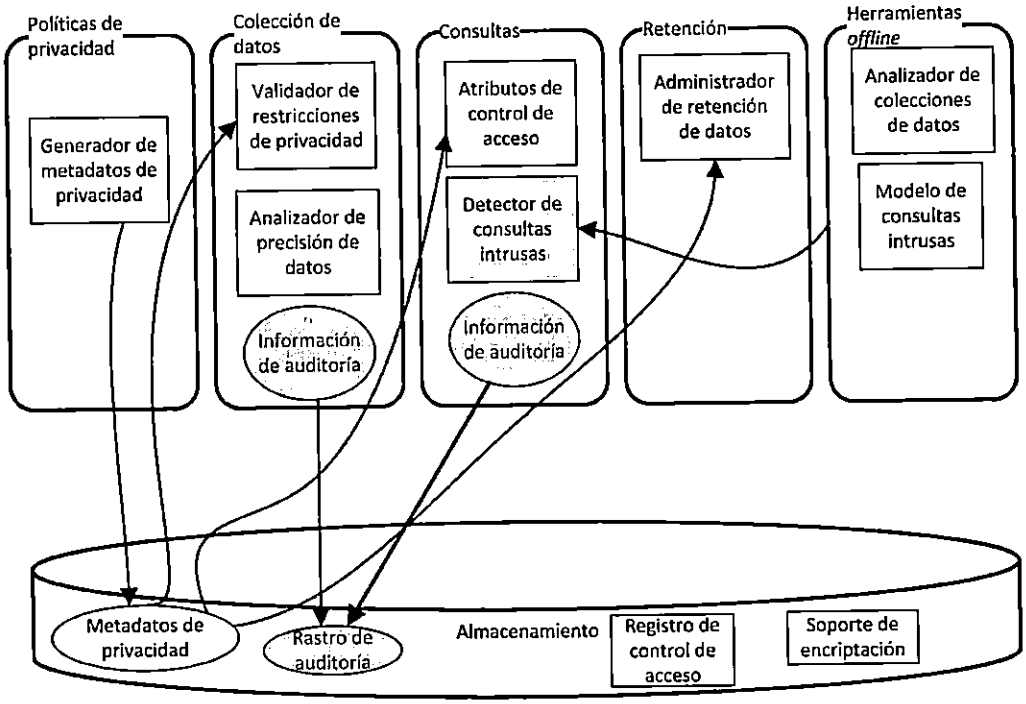

Figura 2.1: Arquitectura de una Base de Datos Hipocrática (BDH).

Para iniciar el funcionamiento de una BDH, hay que asociar las políticas de privacidad de un propietario a sus datos. El **Generador de Metadatos de Privacidad** (GMP) realiza una traducción automática de las políticas de privacidad de un propietario en metadatos de privacidad, asociados con las entidades (registros, columnas o tablas de una BD) correspondientes. Por ejemplo, la Tabla 2.1 representa metadatos de privacidad, en la que se observa que el atributo privado *nombre* de la tabla *Clientes* está asociado con el propósito *ventas* y la organización que tiene derecho a utilizar el atributo privado es la compañía *BanMx*.

Atributo	Tabla	Propósito	Recipientes-Externos
nombre	*Clientes*	*ventas*	*BanMx*

Tabla 2.1: Ejemplo de metadatos de privacidad para una BDH.

El GMP proporciona una conexión encriptada entre la BD y el propietario para modificar sus políticas de privacidad, dichas modificaciones se almacenan como información de auditoría.

Es posible que al tratar de cumplir las políticas de privacidad de un propietario y de una organización, se presenten conflictos. Por ejemplo, supongamos que por ley una organización tiene que retener los datos de Alberto al menos 3 meses; por otra parte Alberto sólo permite que una organización almacene su información por un mes. No hay consistencia entre ambas políticas y es posible que se viole la política de retención de Alberto. El **Validador de Restricciones de Privacidad** (VRP) verifica que las políticas de privacidad de una organización sean válidas para un propietario[2].

La gestión de consultas en una BDH tiene tres etapas:

1. **Antes de la ejecución.** Se comprueba que el usuario (persona, procedimiento, organización, etc.) que inició la consulta tiene derecho a realizarla. El **Control de Acceso de Atributos** comprueba que la consulta no obtenga información de campos que están fuera del propósito.

2. **Durante la ejecución.** El **Control de Acceso a Registros** mantiene visibles sólo los registros asociados con el propósito.

3. **Después de la ejecución.** El **Detector de Consultas Intrusas** identifica comportamientos inusuales de consultas con ayuda del **Modelo de Consultas Intrusas**, que analiza consultas realizadas en el pasado. Se mantiene información de auditoría de todas las consultas que se han realizado a la BD.

Cuando expira el tiempo de permanencia de los datos de un propietario el **Administrador de Retención de Datos** los elimina. Además se propone un **Analizador de Colecta de Datos**, que determina si la información de las consultas asociadas con algún propósito se almacena pero no se utiliza.

[2]En [1] no se explica si el VRP puede resolver los conflictos que puedan existir al tratar de hacer cumplir las políticas de privacidad de una organización y de un propietario.

2.3.2. Auditoría de una BDH

En [7] se presenta un *framework* para determinar si una BDH respeta sus políticas de privacidad. La auditoría inicia con la construcción de una Expresión de Auditoría (EA), la cual contiene los datos privados que serán auditados. Un componente de auditoría procesa la EA y obtiene todas las consultas (consideradas como sospechosas) que accedieron a los datos especificados. La auditoría tiene las siguientes propiedades:

- Sobrecarga mínima para el procesamiento de las consultas.
- Identifica de manera eficaz y eficiente las consultas que accedieron a los datos.
- Se audita a diferentes niveles de granularidad (desde un campo hasta columnas de distintas tablas).
- Se utiliza un lenguaje intuitivo y amigable para la EA.

La sintaxis de una EA es muy similar a una consulta SQL. Un ejemplo de una EA es:

AUDIT *enfermedad*
FROM *Cliente c, Tratamiento t*
WHERE *c.id* = *t.id* **AND** *c.edad* > *20*

La EA evaluará que no se haya divulgado el atributo privado *enfermedad* de pacientes mayores a 20 años.

Cada consulta a la base de datos se registra en la Bitácora de Consultas adjuntando un conjunto de metadatos al registro, tales como el usuario que solicitó la consulta, el tiempo de ejecución y el propósito de la consulta. La arquitectura del *framework* de auditoría se muestra en la Figura 2.2

Durante el proceso de auditoría cada EA se procesa por el **Generador de Consultas de Auditoría** (GCA) en dos etapas. En la primera etapa, a partir de la **Bitácora de Consultas**, se obtiene el conjunto de consultas Q que potencialmente pueden divulgar los valores de los atributos privados en la EA. En la segunda etapa, se crea una **consulta de auditoría** q_a en SQL y se ejecuta en la **Bitácora de Recuperación** por el GCA. Con ayuda del conjunto Q, se obtienen las consultas que tuvieron acceso a los atributos privados de la EA. Se utiliza un esquema de índices en la gestión de las bitácoras para no afectar el desempeño de la arquitectura de auditoría.

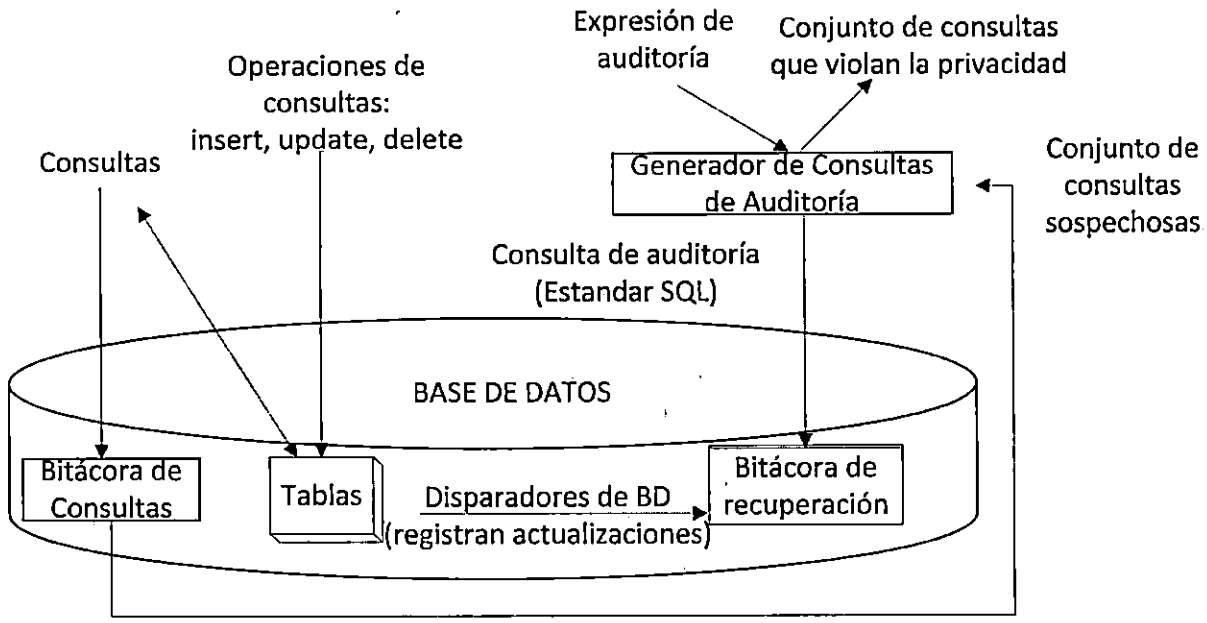

Figura 2.2: Framework para auditar las políticas de privacidad de una BDH.

Cabe mencionar que el *framework* de auditoría funciona para consultas SPJ (*Select-Project-Join*), con una cláusula JOIN y con una cláusula HAVING. Una consulta SPJ es una consulta SQL a la base de datos que está formada por un bloque simple SELECT-FROM-WHERE, sin ningún tipo de sub-consultas o funciones de agregación (operaciones matemáticas, como promedio, suma, etc.). En la cláusula WHERE sólo se permiten predicados de conjunción. La siguiente consulta es un ejemplo de una consulta SPJ:

SELECT *nombre, matrícula, grupo*
FROM *Alumnos*
WHERE *carrera = "Computación"*
AND *edad > 20*

2.3.3. Auditoría de consultas SQL

Dada una vista prohibida V (datos privados) de una base de datos, un conjunto Q de consultas realizadas y una definición de ser sospechoso, en [8] se propone un método para identificar las consultas que divulgan datos de V. Evitar la divulgación de los datos en V, se conoce como el problema de mantener la *privacidad perfecta* de V. Se proponen tres nociones de ser sospechoso para mantener la privacidad perfecta de V, las cuales son:

1. **Sospecha semántica.** Un conjunto de consultas Q es de sospecha semántica con respecto a una expresión de auditoría (descrita en la Sección 2.3.2) A, si: (i) existe una tupla t en las tablas comunes a Q y A que es indispensable[3] tanto para Q como para A y (ii) Q accede a todos los campos privados de A.

2. **Sospecha sintáctica fuerte.** Intenta auditar a cada $q \in Q$ sin ejecutarlas en la base de datos (sólo observando su estructura), se encontró que este problema es NP-difícil (véase la sección III de [8]). Sin embargo, se relaja el problema al proponer la sospecha sintáctica débil, la cual se puede resolver en tiempo polinomial.

3. **Sospecha sintáctica débil.** Un conjunto de consultas Q es de sospecha sintáctica débil con respecto a una expresión de auditoría A, si existe algún subconjunto $Q' \subseteq Q$ y una instancia de tablas I, tal que: (i) una tupla t (en las tablas comunes a Q y A) es indispensable tanto para A como para cada $q' \in Q'$ y (ii) toda $q' \in Q'$ accede al menos a un campo privado de A.

Una descripción más detallada de las nociones anteriores se puede encontrar en [8], donde también se presenta una clasificación sobre qué tan estrictas son las sospechas propuestas. Si se considera la expresión $A > B$ como la definición A más estricta que la definición B, se afirma que *Privacidad Perfecta* > *Sospecha Sintáctica Débil* > *Sospecha Sintáctica Fuerte* > *Sospecha Semántica.* Además se propone un algoritmo de auditoría para cada tipo de sospecha.

Para realizar la auditoría las consultas deben ser de tipo SPJ, sin el operador DISTINCT en la cláusula SELECT y pueden tener campos duplicados.

2.3.4. Auditoría y políticas de retención

Auditar cambios a una Base de Datos (BD) permite identificar comportamientos maliciosos, sin que se descuide el desempeño del sistema y la calidad de los datos. Algunas legislaciones de protección de datos obligan a mantener registros históricos que cumplen estrictas políticas de retención[4], porque dicha permanencia puede representar un riesgo para la privacidad de los datos. Resulta complejo que la retención de datos mantenga los objetivos de privacidad y al mismo tiempo la precisión del proceso de auditoría.

Existen algunas técnicas de auditoría que utilizan mecanismos para eliminar datos de los registros históricos, pero sigue presente la falta de efectividad en la protección. AuditGuard [9]

[3]Una tupla t es indispensable para X (que representa un conjunto de consultas o una expresión de auditoría) si el resultado de ejecutar X con t, es diferente al resultado de ejecutar X sin t.

[4]Una política de retención puede ser por ejemplo, que el número de la tarjeta de crédito del cliente debe permanecer a lo más dos días en la base de datos, después de una transacción.

tiene el beneficio principal de fortalecer políticas de retención, eliminando datos sensibles y al mismo tiempo, manteniendo cambios históricos para el proceso de auditoría. Los componentes principales de AuditGuard son:

- **Modelo de Datos Histórico.** Almacena estados que la BD tuvo en el pasado, mediante una *bitácora operacional* que contiene metadatos adicionales.
- **Políticas de Retención.** Gestión de políticas de retención que elimina o protege la divulgación de datos pasados (e. g., eliminar los registros de empleados del proyecto X que ha concluido).
- **Historia incompleta.** Gestión de *registros parciales incompletos* que se obtienen al aplicar las políticas de retención (es decir, eliminación de datos de manera física o lógica), respetando los objetivos de privacidad.
- **Consultando de manera incompleta.** Permite evaluar consultas de auditoría sobre historiales incompletos (resultado de aplicar las políticas de retención), ofreciendo respuestas parciales cuando una consulta de auditoría (completa) no se puede realizar.

2.4. Auditoría en Sistemas Distribuidos (SD)

Las técnicas de auditoría y la administración de bitácoras se utilizan para resolver diferentes problemas en el área de los SD. Un SD es un conjunto de computadoras autónomas interconectadas entre sí mediante una red y cuenta con software distribuido que permite la comunicación y coordinación entre computadoras. Su objetivo principal es compartir los recursos del sistema (*hardware*, *software*, datos).

Los aportes de la comunidad a la auditoría de privacidad en SD se enfocan en la gestión de políticas de privacidad en línea, los sistemas orientados a servicios y los sistemas P2P.

2.4.1. Gestión de políticas de privacidad en línea

En [2] se propone una arquitectura (Figura 2.3) que mantiene el ciclo de vida de las políticas de privacidad para su protección en línea, enfatizando la necesidad de utilizar aspectos no tecnológicos[5] en su mantenimiento.

La idea es preservar la privacidad de los datos que se intercambian en diferentes sitios Web, utilizando las leyes y tecnologías actuales de privacidad, desde dos puntos de vista:

[5]Por ejemplo, aportaciones de disciplinas como las Ciencias Sociales, Economía, Psicología, etc., que ayuden a reservar la privacidad de los datos.

(1) un usuario que especifica las políticas de privacidad y (2) la organización encargada de gestionar los datos. Cada organización tiene diferentes políticas que debe cumplir. La Figura 2.3 muestra la arquitectura propuesta, separando claramente lo que sucede en el lado del usuario, de lo que sucede del lado de la organización.

La capa **Sociedad, Leyes y Economía** contiene un análisis para hacer una traducción de las políticas de privacidad que se encuentran en lenguaje natural a un lenguaje formal. Se pretende evitar los conflictos entre políticas de diferentes organizaciones, mediante la regulación de las leyes actuales de privacidad.

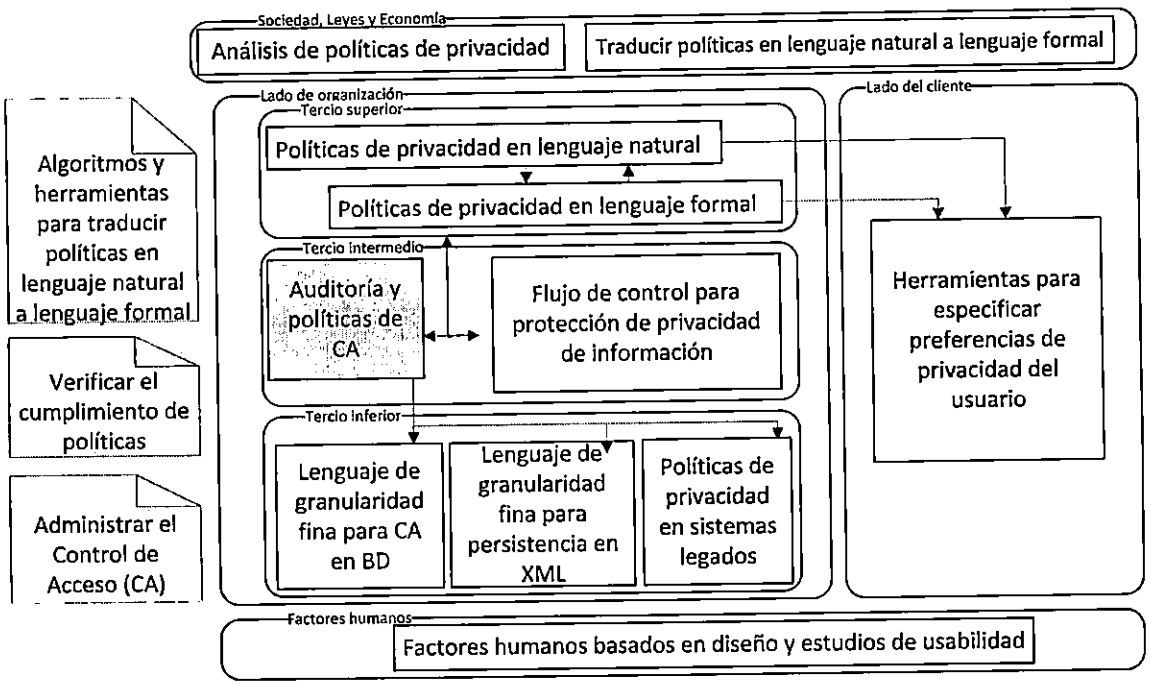

Figura 2.3: Arquitectura para la gestión de políticas de privacidad.

El **lado de la organización** está compuesto de tres tercios:

- **Tercio Superior** (***modelos y lenguajes***). Es responsable de coordinar los algoritmos y herramientas de privacidad, para generar las políticas de privacidad en un lenguaje formal. Este tercio recibe como entrada el análisis que se realizó en la capa Sociedad, Ley y Economía.
- **Tercio Intermedio.** Contiene herramientas para el control de acceso, protección (encriptación) de la información que proviene de las bases de datos y cumple las políticas

de privacidad del tercio superior. Cuenta con un componente para auditar sólo el control de acceso; a diferencia de los trabajos presentados en la Sección 2.3 que verifican el cumplimiento de las políticas de privacidad, la auditoría para el control de acceso verifica que los datos se entreguen sólo a los usuarios autorizados.

- **Tercio Inferior.** Administra los repositorios de los datos y de las políticas de privacidad, para tener un acceso eficiente y de granularidad fina (e. g., acceder a un campo particular de distintas tablas).

El **lado del cliente** permite a los usuarios expresar sus políticas de privacidad a través de una interfaz amigable y fácil de utilizar, después obtiene automáticamente una representación formal lo más cercana posible a lo especificado.

2.4.2. Auditoría de arquitecturas orientadas a servicios

Una arquitectura orientada a servicios se define como: *"Un framework para la integración de procesos de negocio, tiene una infraestructura para soportar tecnologías de la información (con bajo acoplamiento entre ellas), seguras y con componentes estandarizados llamados servicios. Los servicios pueden ser reutilizados y combinados para resolver otras prioridades de negocio"* [10].

El uso de los Sistemas Orientados a Servicios (SOS) permiten combinar las funciones de distintas organizaciones para obtener una nueva organización integrada y ágil, que es capaz de resolver necesidades de negocio más complejas. Sin embargo, los datos que se comparten con distintas organizaciones son vulnerables a tener violaciones a los derechos de privacidad o propiedad intelectual.

Los datos que se comparten requieren de un modelo que supervise su procesamiento distribuido, porque un propietario puede solicitar información de quién consulta y cómo se procesa su información. El procesamiento distribuido de los datos en un SOS requiere de colectar información para el registro de las acciones realizadas en el sistema. La falta de estándares para intercambiar los registros de acciones que se realizan en los diferentes ambientes de ejecución, ocasionan que la administración distribuida de dichos registros continúe siendo un reto. En [11] se presentan dos aportaciones:

1. **DIALOG (*DIstributed Auditing LOGs*):** método para la auditoría del procesamiento distribuido de datos en un SOS.

2. **Etiquetado de Registros (ER):** El ER monitorea el procesamiento de metadatos que se adjuntan a los registros de acciones, permitiendo la reconstrucción del procesamiento de datos, por quién y por qué, al seguir el método DIALOG.

En otras palabras, se provee un *middleware* genérico para la gestión de registros distribuidos.

2.4.3. Auditoría en redes P2P

En [12] una red P2P se define como: *"Una arquitectura de red distribuida en la que los participantes comparten parte de sus recursos de hardware (capacidad de procesamiento, almacenamiento, etc.). Estos recursos compartidos son necesarios para ofrecer servicios y contenido por la red (e. g., sistemas de archivos). Se puede acceder directamente a otros pares (nodos en la red), sin pasar por entidades intermedias."*

Cada par de una red P2P actúa como un *servent*, término compuesto por las primeras cuatro letras del término *server* (*serv*) y las últimas tres letras del término *client* (*ent*). Por lo tanto, cada par actúa como cliente o servidor (como proveedor o solicitante de servicios/recursos). Las redes P2P operan en la capa de aplicación de red, por lo que son conocidas como redes superpuestas. Atendiendo a su arquitectura, las redes P2P se clasifican principalmente en dos categorías: estructuradas y no estructuradas.

Red P2P estructurada

Una red P2P estructurada controla la topología de la red superpuesta y la ubicación de los datos. Se dice que forman una estructura porque sigue un protocolo específico para hacer un mapeo entre el contenido (archivo) y su localización (dirección IP del par), formando una tabla de ruteo distribuida.

El tipo más común de redes P2P estructuradas son aquellas que se basan en una Tabla de Dispersión Distribuida (DHT, acrónimo en inglés de *Distributed Hash Table*), las cuales utilizan el protocolo eficiente de localización *lookup*, porque localiza de manera eficiente el par que almacena un dato en particular. Para localizar un dato en específico, se asocia una llave *key* a dicho dato y se almacena la pareja llave/dato en el par correspondiente al mapeo de la llave *key*. Una DHT provee dos operaciones básicas:

- *put(key, data)* almacena la llave *key* y su dato *data* correspondiente, en la DHT.
- *get(key)* obtiene el dato asociado con la llave *key* en la DHT.

Red P2P no estructurada

Una red P2P no estructurada no provee un protocolo para la topología de la red superpuesta, cada par sólo conoce a sus vecinos pero no conoce los datos que administran; hay disponibilidad de datos gracias a la duplicación masiva. Para ubicar los datos se inunda la

red (*flooding*), lo cual consiste en el envío sucesivo de una solicitud a todos los vecinos de un par, hasta ubicar el par que almacena el dato solicitado.
Las redes P2P puras son no estructuradas y en [12] se definen como: *"Sistema P2P en el cual, cuando se escoge de manera arbitraria una entidad para ser removida de la red, no surge ninguna pérdida en el servicio de la red"*. Es decir, la red continúa ofreciendo los servicios tal y como lo hacía antes de remover la entidad seleccionada.

Técnica de auditoría para evitar el problema del *free-riding*

La investigación en [13] se enfoca en sistemas P2P puros (sistemas que no tienen nodos centrales), en los cuales no se predice el comportamiento de un par. En los sistemas P2P el problema del *free-riding* es muy serio, ocurre cuando los pares sólo se dedican a consumir el contenido sin aportar recursos. Como consecuencia, el tráfico de la red está completamente concentrado en sólo una minoría de los pares, aquellos que comparten contenido. La mayoría de los métodos que resuelven el problema del *free-riding* son vulnerables a que los pares determinados a explotar el sistema, realicen acciones maliciosas como: modificaciones de un par o limitar los flujos de los pares.

El objetivo de la investigación es fortalecer la compartición de recursos forzando a los pares a que contribuyan conforme a las solicitudes realizadas, mediante la utilización de un modelo económico de auditoría en un sistema P2P puro. El crédito representa la capacidad de un par para consumir un recurso. Los pares interesados son aquellos que observan el comportamiento de otro par, para detectar una acción maliciosa y dado que los pares observadores no son 100 % confiables, se utiliza un esquema de firmas digitales (MD5) para proteger la integridad de la información que viaja entre ellos. Se detectan eficientemente pares maliciosos con una probabilidad del 98 %, sin la necesidad de utilizar servidores centrales. En términos generales se considera que:

- Existen pares maliciosos, los cuales actúan siempre para su beneficio.
- Hay una estructura P2P (para localizar recursos un par realiza una simple consulta).
- Cada par tiene un identificador único.
- Cada par puede encontrar la llave pública de otro par y no es posible violar el mecanismo de encriptación. Es posible implementar un mecanismo distribuido para encriptar, sin la necesidad de nodos centrales.

Sobre el comportamiento de los pares:

- Se considera a cada par como egoísta (sólo contribuye si se beneficia a sí mismo y no necesariamente a la comunidad).

- Un par reporta el comportamiento de otros pares.
- En un sistema P2P justo, cada integrante almacena recursos y los comparte al máximo.
- Hay muchos pares que prefieren hacer el *free-riding*, pero pocos pares que estafan, al saber que los estafadores son castigados.
- Dado que el crédito es volátil, los pares pueden ahorrar tanto como puedan antes de que el valor de su crédito desaparezca.

Hay que mencionar que la información que se comparte, para contar y verificar contribuciones al sistema, es: valores de crédito, registros de transacciones y lista de pares interesados. Los tipos de estafas que pueden ocurrir son:

Créditos exagerados. Valor muy alto del crédito que se usa para evadir una detección.

Conspiración. Un par estafador que evade una detección con ayuda de sus colaboradores.

Transferencia de culpa. Un par estafador que trata de culpar a un par inocente para ocultar el mal comportamiento del estafador.

Ignorar a pares interesados. Los pares interesados son pares que observan el comportamiento de algún par P. Un estafador puede modificar la lista de pares interesados. No se considera una estafa a la acción de agregar a pares que no están interesados en observar a P.

Gestión de bitácoras en sistemas P2P

Las bitácoras de auditoría son partes invaluables de un sistema, porque registran los estados pasados y actuales del sistema; por esta razón son un objetivo atractivo para agresores en el sistema. Técnicas como el uso de *hardware* tolerante a errores y mantener un canal de comunicación entre el registrador (*logger*) y las *entidades confiables* no son prácticas, para los sistemas distribuidos actuales. Algunas medidas criptográficas no utilizan canales de comunicación dedicados o *hardware* tolerante a errores, y no son capaces de evitar que un agresor modifique los registros, después que el agresor ha comprometido el sistema; problema que se conoce como *fordward security*.

Los esquemas *Message Authentication Codes (MAC's)* y *Pseudo Random Number Generators (PRNG's)*, utilizan criptografía simétrica para evitar el problema de *fordward security*, pero su naturaleza impide la verificación de manera pública. Por otra parte, un esquema completamente simétrico de distribución implica una sobrecarga de almacenamiento y comunicación para el sistema. Los grupos de llaves criptográficas públicas requieren de almacenar

y retransmitir una etiqueta de autenticación para cada registro de entrada o periodo de registro. Se ha intentado minimizar el exceso de almacenamiento y comunicación de los registros, pero continúa siendo caro (computacionalmente hablando) para el registrador y aún más para el verificador.

El esquema de bitácoras *Blind-Aggregate-Fordward (BAF)* presentado en [14], resuelve las limitaciones antes mencionadas, con base en las siguientes propiedades:

- **Generación eficiente de registros.** El costo computacional para el registro de un dato es sólo de tres operaciones Hash criptográficas.
- **Eficiencia en almacenamiento y ancho de banda en el registrador.** Sin importar el número de periodos o elementos que sean firmados, el costo de almacenamiento es constante e igual a una sola firma compacta.
- **Eficiencia en registros de verificación.** El costo de verificar la entrada de un registro es una multiplicación escalar.
- **Verificación pública.** No se utiliza un esquema criptográfico completo para la distribución de llaves.
- **Verificación TTP (*Trusted Third Party*) fuera de línea (*offline*) e inmediata.** Se utiliza un diseño arquitectural simple que está libre de puntos de falla y no se requiere un soporte en línea TTP para realizar la verificación de registros.

2.5. Sumario de literatura revisada

Las Tablas 2.2 y 2.3 resumen las propuestas consultadas para la presente investigación en el área de sistemas de bases de datos y sistemas distribuidos, respectivamente. Se hace énfasis en los elementos de auditoría que cada propuesta considera.

Los SBD no están diseñados para preservar la privacidad de los datos, sin embargo deben seguir un modelo que permita a los propietarios asignar políticas de privacidad a sus datos. La gestión de los datos en un SBD debe respetar las políticas de privacidad de todos los propietarios, evitar contradicciones entre políticas y hacer que se cumplan cuando se comparten datos con otros. Si bien no todos los trabajos proponen una técnica para auditar las consultas a la BD, coinciden en utilizar un componente para administrar las bitácoras que contienen información de auditoría (datos consultados, hora y fecha de realización, usuario que solicito la consulta, etc.) sobre las consultas. También se hacen copias de las tablas de datos, para recuperar un estado de la BD, cuando se identifica una consulta que divulgó datos privados.

Las técnicas de auditoría se enfocan en encontrar consultas que divulgaron datos privados (auditoría retroactiva) y ofrecen algunas recomendaciones para realizar una auditoría preventiva. Además, se muestra al propietario una lista de las consultas que accedieron a sus datos, sin importar que sean o no maliciosas. Se utilizan técnicas de seguridad para proteger las bitácoras, con el objetivo de mantener la información de las bitácoras completa y fidedigna, para que la auditoría obtenga resultados confiables.

Por otro lado, los trabajos revisados en SD no sólo utilizan a la auditoría para verificar el cumplimiento de las políticas de privacidad sino para distintos fines, un ejemplo es la utilización de las bitácoras para reconstruir las acciones que se hicieron con los datos, en un sistema orientado a servicios. Independientemente del uso de las bitácoras se resalta la importancia para administrar eficientemente la información de auditoría, además de mantenerla segura y consistente entre los nodos del sistema.

Nombre de la propuesta	Base de Datos Hipocrática (BDH) [1]	Auditoría de una BDH [7]	Auditoría de consultas SQL[8]	Auditoría y políticas de retención [9]
Objetivo	SBD responsable de la privacidad.	Auditar las consultas de una BDH.	Auditar las consultas de una Base de Datos (BD).	Auditar una BD aunque se apliquen políticas de retención.
Aportaciones principales	(1)Asocia políticas de privacidad a los datos. (2)Arquitectura de una BDH.	(1)Expresión de Auditoría (EA). (2)Arquitectura para auditar una BDH. (3)Algoritmos para auditar consultas SPJ. (4)Procesamiento mínimo de consultas a diferentes niveles de granularidad.	(1)Auditoría retroactiva. (2)Definición de sospecha semántica y sintáctica para auditar. (3)Dos algoritmos de auditoría. (4)Bosquejo para realizar la auditoría preventiva.	Sistema AuditGuard que aplica políticas de retención sin afectar la auditoría.
Información de auditoría	(1)Cambios a las políticas de privacidad. (2)Registra consultas a la BD.	(1)Auditoría retroactiva. (2)Registra consultas a la BD. (3)Copias de tablas para recuperar su estado.	Registra consultas a la BD.	Registra el estado de una BD, cambios a políticas de privacidad y de retención.
¿Cómo se realiza la auditoría de privacidad?	No hay técnica para auditar, sólo se propone un componente para la gestión de bitácoras.	(1)Definir la EA y seleccionar consultas Q. (2)Con EA y Q obtener consultas sospechosas a la BDH.	(1)Vista prohibida V, consultas Q, sospecha S y EA. (2)Aplicar algoritmo de S a Q con EA. (3)Obtener consultas sospechosas en términos de S.	(1) Aplicar políticas de retención. (2)Etiquetar registros incompletos en las bitácoras. (3)Solicitar inf. de auditoría, AuditGuard obtiene datos consistentes y completos. (4)La auditoría se realiza.

Tabla 2.2: Técnicas para auditar la privacidad en los Sistemas de Bases de Datos.

Nombre de la propuesta	Gestión de políticas de privacidad en línea [2]	Auditoría en arquitecturas orientadas a servicios [11]	Auditoría y el problema del *free-riding* [13]	Gestión de bitácoras en sistemas P2P [14]
Objetivo	*Framework* para aplicar políticas de privacidad en línea.	Gestión de bitácoras en el uso de diferentes servicios, para proteger la privacidad de los datos.	Resolver el problema del *free-riding* en sistemas P2P con una técnica de auditoría.	Gestión eficiente de bitácoras en un sistema P2P puro.
Aportaciones principales	(1)Utilizar leyes y tecnologías actuales en privacidad. (2)Modelo para preservar la privacidad de los datos en línea.	(1)Reconstruir acciones que se hicieron con los datos en distintos servicios. (2)Adjuntar bitácora en cada solicitud de un servicio.	Modelo económico que audita a los pares.	(1) Administra eficientemente registros, almacenamiento y ancho de banda para el registrador. (2)Verificar que un registro no sea modificado, es constante. (3)Se verifica desde cualquier par.
Información de auditoría	Registra el control de acceso a los datos.	Registra operaciones (lectura, escritura, hacer copias) de los datos.	Registra los créditos de los pares cuando comparten datos al sistema.	Se considera una bitácora de propósito general.
¿Cómo se realiza la auditoría de privacidad?	No hay técnica para auditar, sólo se propone un componente para la gestión de bitácoras.	(1)Un usuario solicita la auditoría. (2)Con la información de la bitácora se muestran las operaciones que se hicieron con los datos.	(1)P un par, V sus vecinos y se desea auditar a P. (2)Cada par en V consulta su bitácora de créditos. (3)Si P solicita recursos los pares en V aprueban o rechazan. (4)Si P aporta recursos los pares en V actualizan el registro.	No se propone una técnica para auditar. Se garantiza que cada par recibirá registros íntegros y de manera eficiente.

Tabla 2.3: Técnicas de auditoría en los Sistemas Distribuidos.

Capítulo 3
PriServ: un servicio de privacidad para sistemas P2P

Como se describe en la Sección 2.2, para realizar una auditoría de privacidad primero se necesita elegir un modelo que preserve la privacidad de los datos y posteriormente evaluar el cumplimiento de las políticas de privacidad, conforme al modelo de privacidad establecido. Cuando se tomó la decisión de realizar el presente proyecto de investigación una de las premisas fue partir de PriServ, un servicio de privacidad para sistemas P2P y después desarrollar un servicio de auditoría. En este capítulo se describe PriServ.

3.1. Introducción

PriServ [4] es un servicio de privacidad para sistemas P2P el cual facilita la prevención de violaciones a la privacidad de los datos. Se utiliza el modelo de privacidad para redes P2P PriMod [15], el cuál tiene los siguientes objetivos:

- Permitir que el propietario de datos especifique sus preferencias de privacidad.
- Obligar a que los solicitantes especifiquen el propósito de acceso (como en las BDH) y la operación a realizar sobre los datos requeridos.

En PriMod se considera que un par viola las preferencias de privacidad cuando ocurre alguno de los siguientes comportamientos:

- **Mal uso de los datos.** No se cumple con el propósito (e. g., el nombre de un alumno se usa para un propósito comercial y no administrativo como se había establecido) o las condiciones de uso de los datos (e. g., no se deben hacer copias de los datos).
- **Divulgación no autorizada.** Obtener datos de un propietario sin autorización.
- **Ataques a la integridad de los datos.** Modificar los datos publicados en el sistema.

Para evitar los comportamientos anteriores, PriMod se basa principalmente en los principios de las bases de datos hipocráticas (véase la Sección 2.3.1 de [1]) y técnicas de reputación [16]. A continuación se describirán los elementos que componen el modelo de privacidad.

3.2. Descripción de PriMod

Los objetivos de PriMod son: (1) permitir que los propietarios de datos especifiquen sus preferencias de privacidad y (2) comprometer a los solicitantes a respetar dichas preferencias. PriMod se puede utilizar en sistemas P2P estructurados o no estructurados, los pares pueden unirse o abandonar el sistema en cualquier momento y se requiere que cada par tenga un identificador único. Hay tres tipos de roles para los pares en el sistema, el **propietario** que comparte sus datos, el **solicitante** que requiere datos y el **servidor** que almacena y provee los datos. PriMod permite que los propietarios especifiquen sus preferencias de privacidad a través de políticas de privacidad. El modelo de políticas de privacidad se muestra en la Figura 3.1:

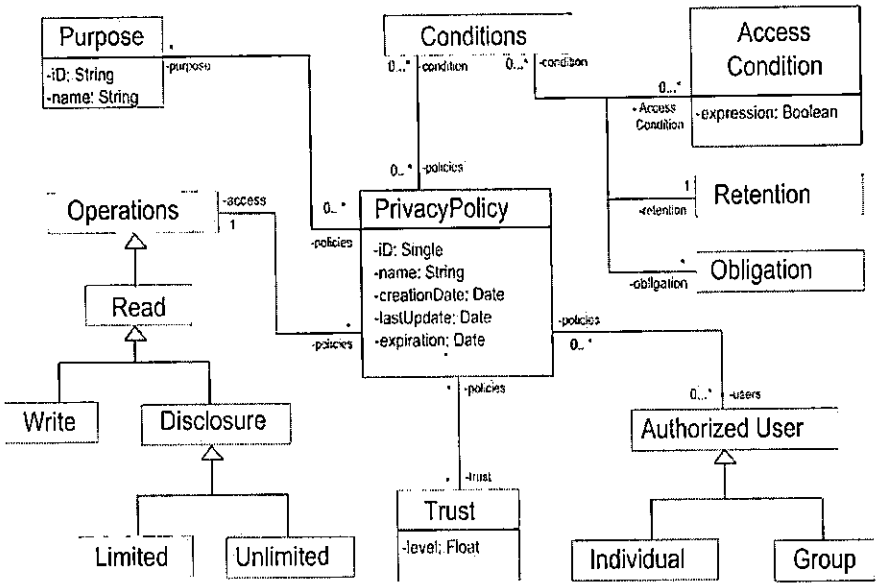

Figura 3.1: Modelo de políticas de privacidad [4].

Una política de privacidad incluye:

- **Usuarios autorizados.** Lista de pares autorizados a obtener los datos (ACL acrónimo de *Access Control List*).

- **Operación.** Acción que un par hace con los datos (lectura, escritura o divulgación). La divulgación representa el derecho a compartir los datos con otros pares.

- **Propósito.** Es el objetivo de acceso a los datos. Por ejemplo, un propósito académico puede significar que los datos serán utilizados para impartir clases.

- **Nivel de confianza.** Es una valoración de la probabilidad de que un par sea malicioso. Por ejemplo, un par es honesto cuando su nivel de confianza está en el intervalo [0.5, 1], maliciosos cuando está entre [0, 0.5) o desconocido, cuando no se conoce el valor del nivel de confianza.

Se utiliza un esquema de tablas relacional para efectos de descripción, pero el modelo es independiente del tipo de datos (ya sea relacional, XML, archivo, objetos, etc.). Se consideran tres tablas para asociar las políticas de privacidad con los datos de un propietario. Es importante que estas tablas cuenten con una llave primaria genérica para evitar divulgaciones de información personal (e. g., número de seguro social, RFC). La **tabla de datos** lista los datos que se desean compartir, la **tabla de políticas de privacidad** contiene las políticas de privacidad y la **tabla de datos privados** asocia las políticas de privacidad a los datos. Por ejemplo, en la Tabla 3.1 el propietario Marco lista los datos que comparte en el sistema; en la Tabla 3.2 la política de privacidad PP1 significa que sólo los profesores pueden leer los archivos, para el propósito de evaluación y después de hacer la evaluación enviarla al propietario; en la Tabla 3.3 el renglón DP-1 asocia el archivo *inversión* con la política de privacidad PP1.

Tabla de Datos (TD)			
ID (PK)	**Nombre**	**Tamaño**	**Tipo**
DAT-1	inversión	20 K	DOC
DAT-2	poema	10 K	TXT

Tabla 3.1: Archivos que el propietario comparte en el sistema.

Tabla de Políticas de Privacidad					
ID (PK)	**Usuario**	**Operación**	**Propósito**	**Condición**	**NivConf Mín**
PP-1	Profesor	Lectura	Evaluación	Enviar eval. al propietario	0.5
PP-2	Alumno	Escritura	Colaboración	–	0.8

Tabla 3.2: Políticas de privacidad del propietario.

Tabla de Datos Privados		
ID (PK)	Archivo	Política de Privacidad
DP-1	DAT-1	PP-1
DP-2	DAT-2	PP-2

Tabla 3.3: Asociación de datos con políticas de privacidad.

PriMod ofrece dos tipos de operaciones que respetan las políticas de privacidad de los propietarios al compartir sus datos, la primera de publicación y la segunda de solicitud de datos.

- **Publicación.** Hay dos maneras que un propietario puede utilizar para publicar datos, por referencia o cifrando los datos. Por ejemplo, un escritor puede publicar un libro de dos maneras, por su referencia (e. g., el título) o cifrando su contenido.
  La publicación por referencia se realiza con la operación *Boolean publishReference(data, PPID)*, el parámetro *data* es el dato del cual se publicará la referencia y el parámetro $PPID$ especifica la política de privacidad asociada; la función regresa verdadero si la publicación se realizó con éxito en el sistema, falso en otro caso.
  La publicación cifrando los datos se realiza con la operación *Boolean publishData(data, PPID)*, el parámetro *data* es el dato que se va a cifrar (se usa criptografía simétrica) y el parámetro $PPID$ especifica la política de privacidad asociada; la función regresa verdadero si la publicación se realizó con éxito en el sistema, falso en otro caso.
  La publicación por referencia permite al usuario controlar directamente el acceso a los datos, porque se almacenan de manera local. En la publicación cifrando los datos, éstos se almacenan en el sistema por algún servidor.

- **Solicitud.** Para solicitar datos los solicitantes utilizan la función *Data request(dataRef, purpose, operation)*. Un par utiliza esta función para solicitar datos (*dataRef*) con un propósito (*purpose*) específico (e. g., investigación, pedagógico, análisis) y realizar una operación (*operation*) particular (lectura, escritura o divulgación); la función regresa los datos si el solicitante cumple con la política de privacidad, en otro caso regresa nulo. La función *request*() obliga a que los pares respeten las políticas de privacidad y rechaza a pares que utilizan propósitos u operaciones desconocidas.

3.3. Arquitectura y funcionamiento de PriServ

PriServ es una implementación de PriMod y utiliza un sistema P2P que se basa en una tabla de dispersión distribuida (*Distributed Hash Table DHT*). Los sistemas P2P basados en una DHT (e. g., Chord [17], Pastry [18], etc.) implementan un protocolo de búsqueda

distribuido conocido como *lookup*, el cual busca de manera eficiente el par que almacena un dato en particular. Los datos y pares están asociados con llaves *key*, típicamente una llave se obtiene cuando se aplica a la dirección IP y al número de puerto de un par una función Hash consistente. El almacenamiento de la pareja *key/dato* se realiza en el par correspondiente a la llave *key* (hay un mapeo entre llaves y pares). La DHT ofrece dos operaciones básicas:

- ***put(key, data)*** almacena una llave *key* y su dato asociado en la DHT.
- ***get(key)*** obtiene el dato asociado con la llave *key* en la DHT.

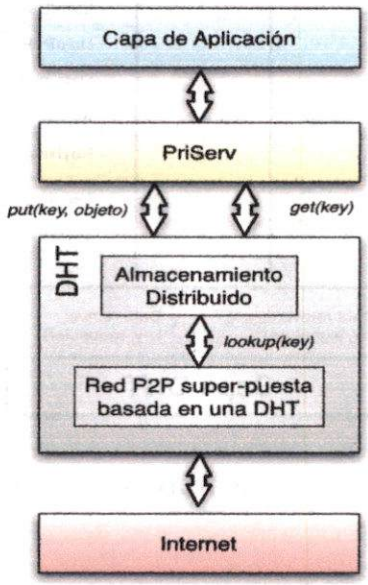

Figura 3.2: Arquitectura Global.

En la Figura 3.2 se muestra la arquitectura global a considerar. Sobre la capa de Internet se encuentra el sistema P2P donde la red super-puesta implementa la función *lookup* y administra las salidas, así como las llegadas de los pares al sistema. El almacenamiento distribuido administra la búsqueda de datos asociados con las llaves al implementar las funciones *put()* y *get()*. La implementación de PriServ se realiza con Chord por su eficiencia y simplicidad, pero cualquier sistema P2P basado en una DHT puede ser utilizado.

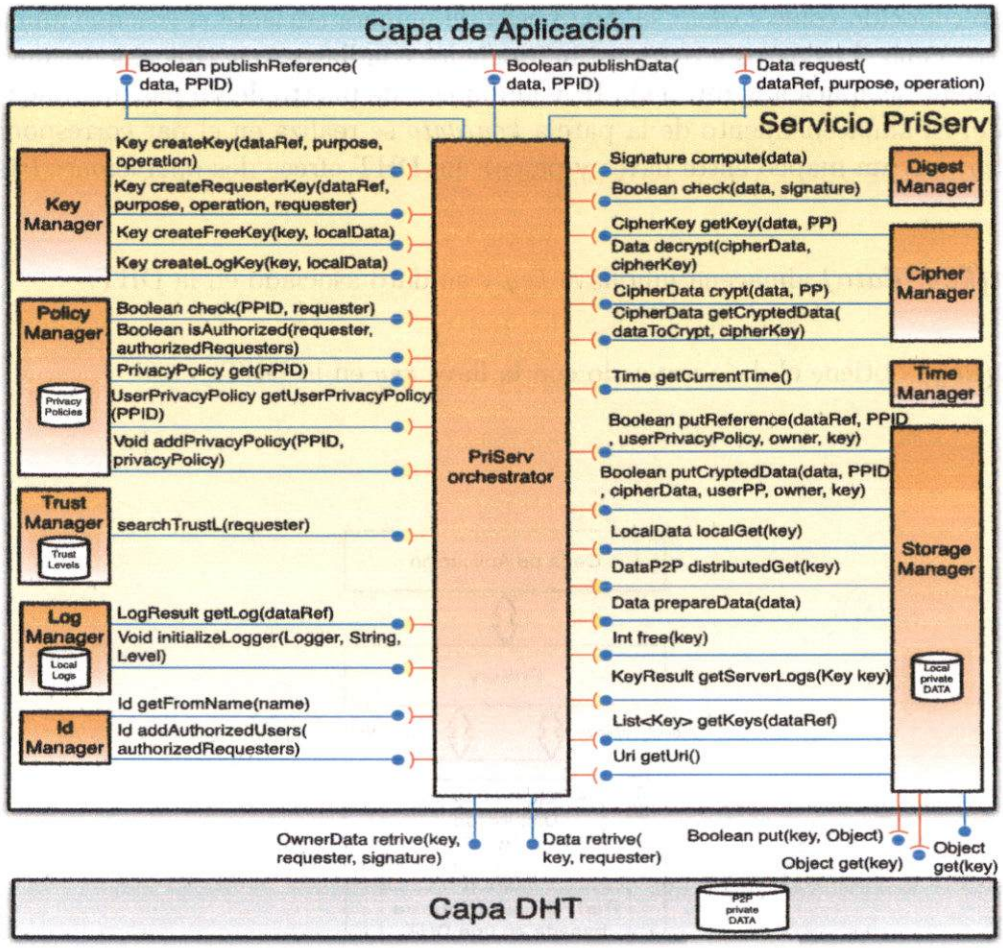

Figura 3.3: Arquitectura de PriServ.

Para fortalecer la privacidad de los datos, se propone que una llave *key* se genere al aplicar una función Hash consistente con la tupla *(dataRef, purpose, operation)*, donde *dataRef* es una referencia única a los datos *D*, *purpose* es el propósito de acceso y *operation* es la operación permitida, para el uso de *D*. La construcción de las llaves permite asociar diferentes propósitos de acceso y operaciones a un mismo dato; además se fortalece el control de acceso porque cuando un par solicite datos al sistema, debe especificar el propósito y la operación correctos.

En la Figura 3.3 se muestra la arquitectura de PriServ que se encuentra entre la capa de aplicación y la capa de la DHT. Se ofrecen las funciones de publicación (*publishReference()*, *publishData()*) y solicitud (*request()*) de datos.

El **orquestador de PriServ** (*PriServ orchestrator*, el cual está rodeado por varios componentes) es un componente principal que se encarga de instanciar tres diferentes flujos de ejecución, dependiendo del rol del par (propietario, servidor o solicitante).

- **Orquestador del propietario.** Coordina las funciones del propietario. Es responsable de la publicación por referencia o la publicación cifrando los datos. Responde a las peticiones de datos y llaves simétricas de los solicitantes. Tiene una interacción con la capa de aplicación en las publicaciones y con el orquestador del solicitante, al solicitar datos.

- **Orquestador del solicitante.** Coordina las solicitudes de los datos. Tiene una interacción con la capa de aplicación en las solicitudes y con el orquestador del propietario para recibir los datos.

- **Orquestador del servidor.** Coordina las funciones del servidor. Tiene una interacción con la capa DHT para almacenar y obtener, los datos del sistema P2P.

A continuación se describirá el resto de los componentes de la arquitectura de PriServ (Figura 3.3).

- **Policy Manager.** Realiza la gestión de las políticas descritas en la Tabla 3.2. Crea las condiciones de uso que los solicitantes deben respetar y obtiene una lista de control de acceso (*Access Control List* ACL), una ACL contiene los pares autorizados a obtener los datos.

- **Storage Manager.** Almacena y maneja todos los datos locales necesarios en PriServ, las Tablas 3.1-3.3, así como los datos de la DHT que le corresponden al par. Invoca las funciones *put()* y *get()* de la DHT, para la gestión de datos en el sistema P2P.
  Cuando un propietario hace una publicación por referencia se crea el objeto *DataP2P* que encapsula las condiciones de uso, el identificador del propietario y la ACL. Si la publicación es cifrando los datos se adjunta el cifrado de los datos al objeto *DataP2P*. En ambos casos, se envía cada objeto *DataP2P* al sistema de almacenamiento distribuido (DHT).

- **Trust Manager.** Administra los niveles de confianza de los pares. Si un par no conoce el nivel de confianza de un solicitante, se inicia un algoritmo para encontrar dicho valor (véase Sección III de [4]).

- **Cipher Manager.** Crea una llave simétrica para la pareja *datos/política de privacidad*, ofrece una función para encriptar y otra para desencriptar datos; se puede utilizar cualquier técnica para encriptar los datos.

PriServ utiliza encriptación pública de llaves para transmitir datos, cuando un par emite una solicitud adjunta su llave pública k y el propietario encripta los datos solicitados con k, de esta manera sólo un usuario autorizado obtiene los datos en claro, con su llave privada.

- **Data Signature Manager.** Administra las firmas digitales para verificar la integridad de los datos, se puede utilizar cualquier técnica para la creación de firmas digitales (e. g., *checksum*, Hash). La firma digital puede se utilizada por los propietarios para verificar si los servidores corrompen datos.

- **Key Manager.** Administra las llaves de datos, las llaves se obtienen al aplicar una función Hash a las referencias de datos, propósitos y operaciones.

- **Id Manager.** Identifica a los pares en función de sus identificadores (e. g. URI), es útil para el control de acceso.

- **Log Manager.** Administra las bitácoras de los pares, los registros contienen las acciones que hace un par con los datos, en función de su rol (descripción detallada en Sección 3.4).

- **Time Manager.** Obtiene el tiempo actual y es utilizado para sincronizar los relojes de todos los pares conectados.

A continuación se describirá de manera general el funcionamiento de los algoritmos de publicación y de solicitud de datos que implementa PriServ. Se considera que el propietario contiene información en las tablas de datos (Tabla 3.1) y políticas de privacidad (Tabla 3.3). La publicación por referencia con la función *publishReference(data, PPID)* de los datos *data*, con respecto al identificador de la política de privacidad $PPID$ de un propietario, funciona como sigue:

1. Obtener las condiciones de uso y la política de privacidad PP con $PPID$.

2. Crear la llave *key* con el componente *Key Manager* y los parámetros: *data.dataRef*, *PP.purpose* y *PP.operation*.

3. Almacenar los datos *data* y la asociación de *data* con PP, de manera local.

4. Crear el objeto *DataP2P* con los parámetros: condiciones de uso, identificador del propietario y la ACL.

5. Publicar la referencia de los datos con *DHT.put(key, DataP2P)*.

Si el propietario desea publicar sus datos cifrados utiliza la operación *publishData(data, PPID)*, que funciona como sigue:

1. Obtener las condiciones de uso y la política de privacidad PP con $PPID$.
2. Crear la llave *key* con el componente *Key Manager* y los parámetros: *data.dataRef*, *PP.purpose* y *PP.operation*.
3. Crear el cifrado *cipherData* con el componente *Cipher Manager* y los parámetros: *data* y PP.
4. Crear una firma digital de *cipherData* y almacenarla localmente.
5. Almacenar los datos *data* y la asociación de *data* con PP, de manera local.
6. Crear el objeto *DataP2P* con los parámetros: *cipher*, condiciones de uso, identificador del propietario y la ACL.
7. Publicar los datos con *DHT.put(key, DataP2P)*.

La solicitud de datos con la función *request(dataRef, purpose, operation)* para la referencia de datos *dataRef* y con respecto al propósito *purpose* y operación *operation*, funciona como sigue:

1. Crear la llave *key* con el componente *Key Manager* y los parámetros: *dataRef*, *purpose* y *operation*.
2. Obtener el objeto *DataP2P* con *DHT.get(key)*.
3. Si *DataP2P* es nulo el servidor identificó, gracias a la ACL correspondiente, que el solicitante no está autorizado a obtener los datos o simplemente *key* no existe. El solicitante no obtiene los datos y termina la solicitud.
4. Si *DataP2P* es una publicación de datos cifrados:

4.1 Obtener la firma digital *signature* con el componente *Data Signature Manager* y el parámetro *DataP2P.cipherData*.

4.2 Obtener los datos *ownerData* del propietario *DataP2P.owner* con la función *retrive(key, requester, signature)*. El propietario verifica (1) la ACL correspondiente, (2) el nivel de confianza del solicitante y (3) la firma digital (*signature*) del dato.

4.3 Si *ownerData* es nulo, el solicitante no cumple con las verificaciones de las políticas de privacidad 1 y 2 del paso 4.2. El solicitante no obtiene los datos y termina la solicitud.

4.4 Si *ownerData* contiene una llave de cifrado para key_{cipher}, el componente *Cipher Manager* del solicitante obtiene los datos originales *data* al decodificar *DataP2P.cipherData* con la llave key_{cipher}. El solicitante obtiene los datos *data* y termina la solicitud.

4.5 Si *ownerData* no contiene una llave de cifrado, el solicitante obtiene los datos de *ownerData.data* y termina la solicitud. Esto quiere decir que en 3 del paso 4.2 no hubo correspondencia entre la firma digital calculada por el propietario y la calculada por el solicitante sobre el dato cifrado enviado por el servidor.

5. Si *DataP2P* es una publicación por referencia:

5.1 Obtener los datos *data* del propietario *DataP2P.owner* con la función *retrive(key, requester, signature)*

5.2 Si *data* es nulo, el solicitante no cumple con las políticas de privacidad o simplemente *key* no existe. El solicitante no obtiene los datos y termina la solicitud.

5.3 Si *data* es distinto de nulo, el solicitante obtiene los datos y termina la solicitud.

En los puntos 4.2 y 5.1 de la función *request*() se observa que el solicitante siempre contacta al propietario de los datos requeridos, ya sea para obtener el dato (cuando se publicó con *publishReference*()) o para decodificar los datos cifrados (cuando se publicó con *publishData*()). Esto quiere decir que el propietario tiene el control sobre sus datos y quien accede a estos. También es interesante ver que para el solicitante es completamente transparente la manera de publicación de los datos.

3.4. Las bitácoras de PriServ

PriServ administra una bitácora distinta por cada tipo de rol (propietario, servidor o solicitante) que un par tiene en el sistema P2P, como se muestra en la Figura 3.4.

Cuando un par emite una solicitud con la función *request(dataRef, purpose, operation)* se agrega un registro en la bitácora *RequesterLog* con: referencia de los datos solicitados *dataRef*, llave *key*, fecha y hora de emisión de solicitud *date*, propósito *purpose*, operación *operation*, firma digital *signature* y el valor booleano *succes* (es verdadero cuando se entregaron los datos exitosamente y es falso cuando se rechazó la entrega).

A diferencia de la bitácora *RequesterLog* (que sólo agrega un tipo de registro), los registros que se agregan en las bitácoras *OwnerLog* y *ServerLog* son de dos tipos.

En la bitácora *OwnerLog* cuando un propietario:

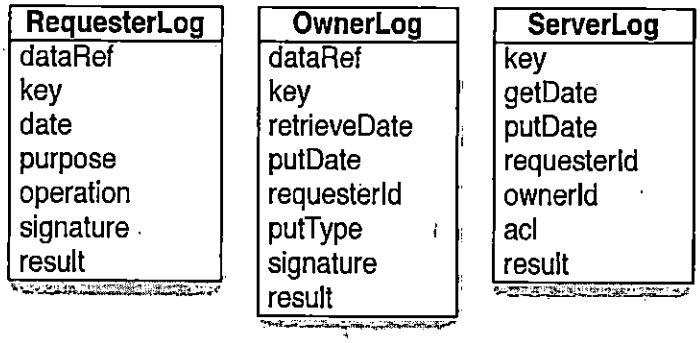

Figura 3.4: Bitácoras de PriServ.

- **emite una publicación con** *publishData*() **o** *publishReference*(), se agrega un registro con: referencia de los datos publicados *dataRef*, llave *key*, fecha y hora de publicación *putDate* y el tipo de publicación *putType*.
- **recibe una solicitud de datos,** se agrega un registro con: referencia de los datos solicitados *dataRef*, llave *key*, fecha y hora de recepción *retriveDate*, identificador del solicitante *requesterId*, firma digital *signature* (en caso de una publicación por cifrado de datos) y el valor booleano *result* (es verdadero cuando se entregaron los datos exitosamente y es falso cuando se rechazó la entrega).

En la bitácora *ServerLog* cuando un servidor:

- **recibe una publicación de datos,** se agrega un registro con: llave *key*, fecha y hora de recepción de solicitud *putDate*, identificador del propietario *ownerId* y la lista de control de acceso *acl*.
- **recibe una solicitud de datos,** se agrega un registro con: llave *key*, fecha y hora de recepción de solicitud *getDate*, identificador del solicitante *requesterId* y el valor booleano *result* (es verdadero cuando se entregaron los datos exitosamente y es falso cuando se rechazó la entrega).

El componente *Log Manager* administra las bitácoras de los pares.

3.5. Resumen

El uso de PriServ facilita la prevención de violaciones a las políticas de privacidad de los propietarios, que comparten sus datos con los pares en un sistema P2P. Sin embargo, no se

provee una manera para verificar que las políticas de los pares se respetaron. El requerimiento anterior es necesario, porque los pares en el sistema se pueden comportar de forma maliciosa aunque se comprometan a seguir un modelo de privacidad. Así pues, es deseable ofrecer a los propietarios la capacidad de evaluar a los pares para verificar el cumplimiento de sus políticas de privacidad.

Se necesita extender a la arquitectura de PriServ con un servicio de auditoría que utilice la información de las bitácoras, ejecute algoritmos de auditoría y se coordine con los componentes de PriServ, para evaluar el cumplimiento de las políticas de privacidad. En el siguiente capítulo se presenta una propuesta en esta dirección.

Capítulo 4
Servicio de auditoría para PriServ

El objetivo principal del presente trabajo de investigación es el diseño y la implementación de un servicio de auditoría para PriServ. La idea es ofrecer al propietario de datos una manera de verificar que los pares en el sistema P2P cumplen con las políticas de privacidad. En este capítulo se describe el protocolo de investigación (Sección 4.1), la identificación de escenarios maliciosos que pueden ocurrir en un sistema P2P (Sección 4.2), el algoritmo de auditoría propuesto (Sección 4.3) y el diseño del componente *Audit Manager* (Sección 4.4).

4.1. Protocolo de investigación

4.1.1. Problemática: comportamientos maliciosos

El uso de PriServ en un sistema P2P fortalece la privacidad de los datos que se comparten, al permitir que:

- Los propictarios asocien políticas dc privacidad a los datos publicados del sistema.
- Las publicaciones (por referencia y cifrando los datos) adjuntan información de las políticas de privacidad.
- Los solicitantes emitan solicitudes especificando el uso y la operación que se harán sobre los datos. Esto es un compromiso explícito que hacen los solicitantes.
- Los datos se entregan sólo a usuarios autorizados, según lo establecido en la política de privacidad.

Sin embargo, se identificó que los pares en el sistema pueden realizar acciones para violar las políticas de privacidad, a dichas acciones se les dará el nombre de comportamientos maliciosos. Los comportamientos maliciosos los pueden realizar los servidores o solicitantes, porque no tiene sentido que un propietario los realice.

Cuando un propietario publica datos en el sistema P2P crea un objeto *DataP2P* que encapsula el identificador del propietario (*ownerId*), la lista de control de acceso (*ACL*) y una lista de condiciones de uso (*usageConditions*) para los datos. Además, al publicar los datos cifrados se adjunta el cifrado de los datos (*cipherData*) a *DataP2P* (véase la Sección 3.3). Por lo tanto, el servidor que almacena los objetos *DataP2P* de diferentes propietarios en el sistema, puede efectuar los siguientes comportamientos maliciosos:

- Modificar los datos *ownerId*, *ACL* y *usageConditions* que encapsula un objeto *DataP2P*. Si el objeto *DataP2P* contiene datos cifrados, puede reemplazar el cifrado por otro (violación de la integridad del dato).

- Cuando un servidor recibe las peticiones de los solicitantes para obtener los objetos *DataP2P*, puede entregarlos a pares no autorizados o rechazar la entrega a pares autorizados, porque la ACL se puede modificar.

Los solicitantes también se pueden comportar de manera maliciosa, cuando utilizan el resultado de las solicitudes que envían al sistema para determinar el contenido de los datos privados. Por ejemplo, consideremos un sistema P2P para una comunidad médica en el cual, las secretarias conocen los nombres de los estudios médicos que se aplican a los pacientes. Una secretaria está autorizada a consultar los identificadores de los estudios médicos del paciente Javier; utilizando su experiencia laboral, es probable que ella determine la enfermedad de Javier sin que él haya autorizado que la secretaria conozca su enfermedad. Con base en la literatura revisada, particularmente en [8], es probable que los solicitantes conozcan datos privados distintos a los autorizados (si es el caso) con ayuda de las respuestas a las solicitudes realizadas.

Otros comportamientos maliciosos de los solicitantes son:

- Republicar los datos en el sistema. Esto es, los datos que se han obtenido correctamente los publican (con otro nombre, otra referencia o ligeramente modificados) como si ellos fueran los propietarios.

- Utilizar los datos para propósitos diferentes a los especificados en la solicitud.

La acción en el sistema P2P que implique alguno de los comportamientos maliciosos anteriores, viola la privacidad de los datos y debe ser identificada cuando se realice la auditoría. El servicio de auditoría propuesto ataca los comportamientos maliciosos de los servidores, con el objetivo de verificar el cumplimiento de las políticas de privacidad.

4.1.2. Descripción del servicio de auditoría

Se realizó el diseño, la implementación y la evaluación de un servicio de auditoría para PriServ, el cual realiza una evaluación del cumplimiento de las políticas de privacidad, con base en los siguientes supuestos:

- El contenido de las bitácoras de PriServ y el código del *Log Manager* (véase Sección 3.3), no pueden ser modificados por los pares del sistema P2P.
- En un sistema P2P no hay garantía de la permanencia de los pares.

De manera general, el funcionamiento del servicio de auditoría es el siguiente:

1. Un propietario inicia la auditoría de sus preferencias de privacidad.
2. Con base en las políticas de privacidad del propietario se busca en las bitácoras de los servidores (*SeverLog*), los registros que impliquen un comportamiento malicioso y los registros que accedieron a los datos privados.
3. El propietario recibe un reporte con los registros que realizaron alguna acción con sus datos y los registros maliciosos.

Los objetivos específicos del servicio de auditoría son:

- Utilizar las bitácoras de PriServ (*OwnerLog* y *ServerLog*, véase Sección 3.4) para verificar el cumplimiento de las políticas de privacidad.
- Auditar a todos los servidores que accedieron a los datos del propietario.
- Detectar los comportamientos maliciosos de los servidores.
- Utilizar el mínimo número de mensajes dirigidos a los pares durante la auditoría.

4.1.3. Metodología

La metodología que se siguió para construir el servicio de auditoría es la siguiente:

1. **Identificación de escenarios maliciosos.** Documentación de los escenarios que describen los comportamientos maliciosos de los servidores y solicitantes, para violar las políticas de privacidad.
2. **Diseño de un algoritmo de auditoría.** Propuesta de un algoritmo de auditoría para obtener los registros maliciosos y los registros que consultaron los datos de un propietario, registrado en las bitácoras de los servidores.

3. **Diseño del Administrador de Auditoría (*Audit Manager*).** Diseño del componente *Audit Manager* que extiende la arquitectura de PriServ, la documentación del componente se realiza utilizando diagramas UML (e. g., secuencia, paquetes, implementación). Este componente orquesta el algoritmo de auditoría propuesto y los componentes de PriServ que intervienen en el proceso de auditoría, tales como el *Storage Manager*, *Policy Manager*, *Log Manager* e *Id Manager*.

4. **Codificación del prototipo.** Codificación del sistema P2P estructurado (se implementó Chord [17]), componentes de PriServ (los necesarios para publicar y solicitar datos en el sistema P2P), el algoritmo de auditoría y el componente *Audit Manager*.

5. **Evaluación.** Se analizó el número de mensajes que requiere el algoritmo para auditar a los servidores de un sistema P2P. Además se emuló el sistema P2P y se creó un caso de prueba que inyecta registros maliciosos a las publicaciones de un propietario, para medir el tiempo que requiere la auditoría.

4.2. Identificación y selección de escenarios

Se identificaron cuatro escenarios que violan políticas de privacidad al buscar que acciones, realizadas por los pares, generan los tipos de comportamientos maliciosos que PriServ previene. Dichos comportamientos son:

- **Divulgación No Autorizada (DNA).** Ocurre cuando un servidor o solicitante divulga datos de un propietario a otro par no autorizado (conforme a las políticas de privacidad) a obtener dichos datos.

- **Ataque a la Integridad (AI).** Modificación de los datos publicados en el sistema, sin autorización del propietario.

- **Mal Uso de los Datos (MUD).** Cuando un par no cumple con el propósito o las condiciones de uso asociadas a los datos.

El primer escenario (Tabla 4.1), describe las acciones maliciosas que un servidor puede realizar. Una vez que el sistema almacena en un servidor la publicación de un propietario (pasos 1, 2), el servidor puede modificar el contenido de los datos (paso 3). Además, como es posible modificar la ACL (incluida en la publicación) el servidor puede entregar los datos a pares no autorizados o rechazar las solicitudes de pares autorizados (pasos 4, 5).

El segundo escenario (Tabla 4.2), muestra la posibilidad que tiene un solicitante para obtener datos distintos a los autorizados. Una vez que un propietario publica sus datos (paso 1), un par solicita los datos del propietario y obtiene las respuestas correspondientes a sus solicitudes (pasos 2 y 3), las cuales pueden ser la entrega de los datos o un rechazo a la solicitud.

Es probable que el solicitante conozca el contenido de datos privados no autorizados (paso 4).

La literatura en auditoría de bases de datos [5], muestra que aún cuando se utiliza alguna técnica para asociar políticas de privacidad a los datos, es probable la divulgación de datos privados cuando se responden ciertas consultas, sin importar que la respuesta sea el rechazo de algunas consultas. Esta afirmación es válida para las consultas SPJ (*Select Project Join*) realizadas a una base de datos. Por lo tanto, para determinar que el escenario Esc2-DNA puede ocurrir, hay que hacer una correspondencia entre las solicitudes de PriServ y una consulta SPJ.

ID del escenario	Esc1-DNA
Tipo de comportamiento malicioso	DNA y AI
Estímulo	Identificar cuando un servidor: ▪ Entregue datos a pares no autorizados ▪ Rechace solicitudes de pares autorizados ▪ Ataque la integridad de los datos publicados
Descripción	1. El propietario P publica el objeto $DataP2P_p$ en el sistema. 2. El servidor S almacena el objeto $DataP2P_p$. 3. S modifica la lista de usuarios autorizados, el identificador de P, las condiciones de uso o reemplaza el cifrado (si es el caso de una publicación por cifrado) de $DataP2P_p$. 4. El par autorizado $Peer_1$ solicita $DataP2P_p$ y S rechaza la solicitud. 5. El par no autorizado $Peer_2$ solicita $DataP2P_p$ y S acepta la solicitud.

Tabla 4.1: Escenario malicioso Esc1-DNA

Solicitud en Priserv:
request(referencia, propósito, operación)

Consulta SPJ:
SELECT *propietarioId.referencia.datos*
FROM *Tabla-de-datos*
WHERE *referencia.política-de-privacidad.propósito = "propósito"*
AND *referencia.política-de-privacidad.operación = "operación"*

Es posible realizar un mapeo entre una consulta SPJ y una solicitud con PriServ, los datos de la consulta se obtienen de las tablas de datos y políticas de privacidad del propietario. Por esta razón, el escenario Esc2-DNA puede ocurrir.

ID del escenario	Esc2-DNA
Tipo de comportamiento malicioso	DNA
Estímulo	Identificar cuando un par conoce datos privados distintos a los autorizados y cuando analiza el resultado de las solicitudes que emite al sistema.
Descripción	1. El propietario P publica el objeto $DataP2P_p$ en el sistema. 2. El par $Peer_1$ solicita el objeto $DataP2P_p$ y obtiene la respuesta R_1. 3. $Peer_1$ emite una solicitud diferente para obtener $DataP2P_p$ y obtiene la respuesta R_2. 4. Con R_1 y R_2 el solicitante deduce datos privados de P.

Tabla 4.2: Escenario malicioso Esc2-DNA

El tercer escenario (Tabla 4.3), describe cuando un par autorizado se adueña de datos de otro par. Una vez que un propietario publica sus datos en el sistema (paso 1) y un solicitante autorizado los obtiene (paso 2), el solicitante define una nueva política de privacidad para publicar datos que no le pertenecen, de esta manera se adueña de los datos sin autorización del propietario original (pasos 3 y 4).

Finalmente el cuarto escenario (Tabla 4.4), describe a un par autorizado que no cumple con las condiciones de uso de los datos obtenidos. Cuando un propietario publica sus datos en el sistema (paso 1) y un solicitante autorizado los obtiene (pasos 2, 3), el solicitante no

cumple con alguna de las condiciones de uso asociadas a los datos. Por ejemplo, una condición de uso puede ser: el solicitante tenía derecho a usar los datos para el objetivo de "análisis" y los usó para "marketing".

ID del escenario	Esc3-DNA
Tipo de comportamiento malicioso	DNA
Estímulo	Identificar cuando un solicitante autorizado obtiene datos del sistema y los publica a su nombre.
Descripción	1. El propietario P publica el objeto $DataP2P_p$ en el sistema. 2. El par $Peer_1$ solicita el objeto $DataP2P_p$. 3. Como $Peer_1$ es un par autorizado, obtiene $DataP2P_p$ de P. 4. $Peer_1$ asocia nuevas políticas de privacidad y publica el objeto $DataP2P$ como suyo.

Tabla 4.3: Escenario malicioso Esc3-DNA

Los escenarios maliciosos anteriores, son el resultado de buscar acciones en PriServ que violen las políticas de privacidad para ser identificadas por el servicio de auditoría. Sin embargo, el alcance del proyecto no permite resolver todos los escenarios propuestos. Por esta razón se describen por escenario, las causas por las cuales serán o no considerados por el servicio de auditoría.

El escenario Esc1-DNA si se considera dado que:

- Es posible proponer un algoritmo realista y eficiente que busque en las bitácoras de PriServ las acciones que violen políticas de privacidad.
- Se pueden listar las acciones que los pares realizan con los datos de un propietario, lo cual no es considerado por los sistemas P2P.
- El diseño, implementación y evaluación del servicio de auditoría, así como el algoritmo que identifica este escenario, no excede el tiempo límite del trabajo de investigación.

El escenario Esc2-DNA no se considera dado que:

ID del escenario	Esc4-MUD
Tipo de comportamiento malicioso	MUD
Estímulo	Identificar cuando un solicitante autorizado obtiene datos del sistema y no cumple con las condiciones de uso.
Descripción	1. El propietario P publica el objeto $DataP2P_p$ en el sistema. 2. El par $Peer_1$ solicita el objeto $DataP2P_p$. 3. Como $Peer_1$ es un par autorizado, obtiene los datos $DataP2P_p$ de P. 4. $Peer_1$ no cumple con al menos una de las condiciones de uso asociadas a los datos.

Tabla 4.4: Escenario malicioso Esc4-MUD

- La propuesta de un algoritmo que identifique las acciones de este escenario implica el estudio de otras líneas de investigación como Inteligencia Artificial, Métodos Formales y Minería de Datos; no se tiene experiencia en estas líneas de investigación y su estudio excede el tiempo programado para el proyecto.
- Antes de decidir que este escenario no formaría parte del servicio de auditoría, se realizó el esfuerzo de proponer un algoritmo para identificar las acciones de este escenario, estudiando principalmente las propuestas [7, 8].

Por otra parte una alternativa para identificar el escenario Esc3-DNA, es agregar a los datos de un propietario una marca, solamente visible por él, la cual no permita a otros pares adueñarse de datos ajenos. Antes de cada publicación es necesario verificar que los datos no le pertenezcan a otro par. Se propone que una solución para resolver este problema es el uso de marcas de agua para redes P2P [19], sin embargo, atender esta solución implicar realizar otro proyecto de investigación, dada la complejidad del problema.

Una política de privacidad contiene las condiciones de uso de los datos, las cuales pueden ser un política de retención o alguna obligación que un par deba cumplir al recibir los datos. Sin embargo, el alcance de PriServ es prevenir violaciones a la privacidad de los datos que se comparten y no al uso de los datos. Por ejemplo, un *Wiki* es un sistema orientado al uso de los datos para la creación de distintos tópicos (contenido agregado por la comunidad); en este contexto una auditoría de condiciones de uso, podría verificar que en un tópico de

Biología, no haya contenido de otro tipo (comercial, promocional, etc.). Dado que el servicio de auditoría no verifica el cumplimiento de las condiciones de uso, el escenario Esc4-MUD no se considera.

4.3. Auditoría de servidores

En esta sección se describe la auditoría de servidores, la cual detecta las acciones maliciosas que los servidores en PriServ pueden realizar, con respecto al escenario Esc1-DNA (Sección 4.2). El algoritmo de auditoría lo inicia un propietario y el resultado es un reporte con los registros maliciosos de los servidores y los registros de los pares que han solicitado los datos del propietario.

La auditoría se lleva a cabo usando tres funciones: una para obtener la información de las bitácoras de cada servidor (*getAuditInformation*()), una para verificar que en la información no hay registro de ninguna violación a la privacidad (*checkCompliance*()) y otra que orquesta el trabajo de los dos anteriores (*serversAuditing*()).

4.3.1. Función *getAuditInformation*()

Esta función (véase Algoritmo 1) obtiene de un servidor la información de auditoría $auditInfo$ para los datos publicados por el propietario $ownerId$. La información de auditoría está compuesta por: (i) la lista de llaves $keysList_{server}$, (ii) la lista de datos P2P $dataP2PList$ donde cada $dataP2P_j \in dataP2PList$ (que contiene el identificador del propietario, las condiciones de uso y la ACL) está asociado con la llave $key_j \in keysList_{server}$, así como por (iii) la lista de registros $logsList$ en la bitácora del servidor.

En primer término se obtienen las llaves de los datos publicados por el propietario; enseguida, se obtienen los datos de auditoría que deben ser verificados. Así pues, para cada llave key (línea 3) en $keysList_{server}$ (2), del componente $StorageManager$ en un servidor se obtiene:

- El identificador del propietario $ownerId_{server}$ (4), la lista de usuarios autorizados acl (5) y las condiciones de uso $usageConditions$ (6).
- La lista de registros $logList$ (7) que describen las solicitudes de los pares en el sistema, para obtener los datos asociados con key.

Posteriormente con los datos colectados, se llena el objeto $result$ (8) y se agrega a $auditInfo$ (9). Después, se adjunta a la información de auditoría el identificador del servidor $serverId$ (11) y la lista de llaves $keysList_{server}$ (12); finalmente $auditInfo$ se envía al

propietario (13).

Algoritmo 1: getAuditInformation(ownerId)

```
1  begin
2  |  keysList_server ⟵ ServerLog.getRequestedKeys(ownerId);
3  |  foreach key ∈ keysList do
4  |  |  ownerId_server ⟵ StorageManager.getOwnerId(key);
5  |  |  acl ⟵ StorageManager.getACL(key);
6  |  |  usageConditions ⟵ StorageManager.getUsageConditions(key);
7  |  |  logsList ⟵ ServerLog.getRequesterLogs(key);
8  |  |  result ⟵
   |  |  createAuditResult(key, ownerId_server, acl, usageConditions, logsList);
9  |  |  auditInfo.add(result);
10 |  end
11 |  auditInfo.addServerId(serverId);
12 |  auditInfo.addServerKeys(keysList_server);
13 |  return auditInfo
14 end
```

4.3.2. Función *checkCompliance*()

Esta función (Algoritmo 2) verifica que cada información de auditoría $auditInfo$ respete las políticas de privacidad del propietario y completa el reporte final de auditoría *auditReport*. Para cada llave *key* (línea 3) de *keysList* (2) en $auditInfo$, se obtiene la política de privacidad *privacyPolicy* (4) especificada por el propietario. Después se obtienen los valores del dato P2P (asociados con *key*), el identificador del propietario $ownerId_{server}$ (5), la lista de usuarios autorizados ACL (6) y las condiciones de uso *usageConditions* (7), para su evaluación. Si el servidor modificó:

- $ownerId_{server}$ (8), se agrega a *auditReport* el registro malicioso correspondiente (9).
- ACL (11), se agrega a *auditReport* el registro malicioso correspondiente (12).
- *usageConditions* (14), se agrega a *auditReport* el registro malicioso correspondiente (15).

Enseguida se recorre la lista de registros *logsList* (17). Para cada registro *log* ∈ *logsList* (18), si el servidor envió los datos (19) y el solicitante no pertenece a la ACL (20), se agrega a *auditReport* el registro malicioso correspondiente (21). En cambio, si se rechazó el envío (23) y el solicitante pertenece a la ACL (24), se agrega a *auditReport* el registro malicioso

correspondiente (25). Finalmente, se registra en el reporte de auditoría que la verificación de los datos anteriores se obtuvo del servidor *serverId* (30).

Algoritmo 2: checkCompliance(auditInfo, auditReport)

```
1  begin
2      keysList ⟵ auditInfo.getKeys();
3      foreach key ∈ keysList do
4          privacyPolicy ⟵ PolicyManager.getPP(key);
5          ownerId_server ⟵ auditInfo.getOwnerId(key);
6          ACL ⟵ auditInfo.getACL(key);
7          usageConditions ⟵ auditInfo.getUsageConditions(key);
8          if ownerId ≠ ownerId_server then
9              auditReport.addMaliciousRecord(OWNER − MOD, key, serverId);
10         end
11         if ACL ≠ privacyPolicie.ACL then
12             auditReport.addMaliciousRecord(ACL − MOD, key, serverId);
13         end
14         if usageConditions ≠ privacyPolicie.usageConditions then
15             auditReport.addMaliciousRecord(USCON − MOD, key, serverId);
16         end
17         logsList ⟵ auditInfo.getLogs(key);
18         foreach log ∈ logsList do
19             if log.envio() = ENVIADO then
20                 if log.getRequester() ∉ privacyPolicie.ACL then
21                     auditReport.addMaliciousRecord(NOT − AUT −
                       SEN, key, serverId);
22                 end
23             else
24                 if log.getRequester() ∈ privacyPolicie.ACL then
25                     auditReport.addMaliciousRecord(SEN − REJ, key, serverId);
26                 end
27             end
28         end
29     end
30     serverId ⟵ auditInfo.getServerId();
31 end
```

El reporte de auditoría resultante, contendrá las acciones que violaron las políticas de privacidad de un propietario.

4.3.3. Función *serversAuditing*()

Esta función (véase Algoritmo 3) la inicia un propietario para evaluar el cumplimiento de sus políticas de privacidad (línea 1). Para determinar qué servidores almacenan los datos del propietario, se obtiene la lista de llaves *keysList* publicadas exitosamente en el sistema. Para cada llave $key \in keysList$ (3), se obtiene una referencia al servidor *server* que almacena *key* (4). Posteriormente hay que solicitar a *server* la información de auditoría $auditInfo$ (5).

Para evitar que se repita el envío de solicitudes de información de auditoría a un servidor *server*, hay que actualizar la lista de llaves (6) con $keysList - auditInfo.getServerKeys() = keysList - keysList_{server}$; la razón es por el mapeo que existe entre llaves y pares. Finalmente, hay que detectar las acciones maliciosas del escenario Escl-DNA (véase sección 4.2) con la función *checkCompliance()* que existan en $auditInfo$ y actualizar el reporte final de auditoría *auditReport*, el cual se entregará al propietario (9).

Algoritmo 3: AuditReport serversAuditing()

```
1  begin
2  |  keysList ⟵ OwerLog.getKeys();
3  |  foreach key ∈ keysList do
4  |  |  server ⟵ IdManager.getServer(key);
5  |  |  auditInfo ⟵ server.getAuditInformation(ownerId);
6  |  |  keysList ⟵ keysList − auditInfo.getServerKeys();
7  |  |  checkCompliance(auditInfo, auditReport);
8  |  end
9  |  return auditReport
10 end
```

4.3.4. Características del algoritmo de auditoría

Se realiza una auditoría exhaustiva, porque se audita a todos los servidores que almacenan las publicaciones del propietario que inicia la auditoría. Si en el sistema P2P existen pares que se consideren como auditores externos (instancias legales del sistema), es posible que dichos pares inicien el algoritmo de auditoría y no sólo el propietario de datos.

El propietario debe pertenecer al sistema P2P y estar activo durante la auditoría. Cuando la función *serversAuditing*() termina el sistema P2P ha sido auditado, con respecto a las políticas de privacidad de un propietario.

El reporte final de auditoría puede estar incompleto porque los servidores se pueden au-

sentar. Si bien cuando un par P_A se anuncia su salida del sistema, se ejecuta un protocolo que asigna a otro par P_B las llaves almacenadas en P_A, el propietario posiblemente aún no conoce al nuevo servidor (P_B).

Sólo se solicita una vez la información de auditoría a los servidores auditados. Al considerar que un propietario tiene la lista de todas las llaves *keysList* que han sido publicadas en el sistema P2P y un servidor *server* almacena una sub-lista de llaves del propietario $keysList_{server}$, al actualizar *keysList* con $keysList = keysList - keysList_{server}$ (véase Algoritmo 1, línea 7), no existe alguna llave en *keysList* que esté almacenada en *server* y por lo tanto, cuando se solicite a la DHT el servidor asociado con alguna llave en *keysList* será distinto de *server*. Por esta razón no es posible solicitar dos veces la información de auditoría a un mismo servidor.

4.3.5. Análisis del orden del algoritmo

Si N es el número de pares en el sistema, se envían $O(log(N)+1)$ mensajes para solicitar y obtener la información de auditoría de un servidor, con la función $getAuditInformation()$ (véase la Sección 4.3.2), al utilizar la DHT. Cuando un propietario (interesado en auditar sus políticas de privacidad) solicita la información de auditoría, puede ocurrir alguno de los siguientes casos:

- **Mejor caso.** Cuando todas las publicaciones del propietario se almacenaron en un servidor, se envían $O(log(N)+1) = O(log(N))$ mensajes.
- **Peor caso.** Si cada servidor almacena al menos una de las publicaciones, se envían $O(S(log(N)+1))$ mensajes a S servidores.

Al analizar el tiempo que requiere el algoritmo de auditoría, el orden del algoritmo está en función del número de llaves publicadas, así como de los registros que hay en las bitácoras de los servidores. Si t_{get_i} es el tiempo para obtener la información de auditoría del servidor i y $1 \leq S \leq N$ es el número de servidores que almacenan las publicaciones de un propietario, la suma

$$f(N) = T_{get\_servers} = \sum_{i=1}^{S} t_{get\_i}$$

es el tiempo total para obtener la información de auditoría de todos los servidores. Sólo resta calcular el tiempo que tarda la función *checkCompliance*() (véase la Sección 4.3.3) para identificar las acciones maliciosas. Entonces, si t_{key_ev} es el tiempo que requiere la evaluación de los campos de un dato P2P para una llave *key* asociada, t_{log_ev} es el tiempo que toma evaluar a un registro *log* de la bitácora, $LOGS$ es el número de registros que se obtuvieron en la bitácora y $KEYS$ el número de llaves, que se almacenan en el servidor auditado, el tiempo total de evaluación de la información de auditoría es

$$\sum_{j=1}^{KEYS} t_{key\_ev} + \sum_{k=1}^{LOGS} t_{log\_ev} = KEYSt_{key\_ev} + LOGSt_{log\_ev}$$

Por lo tanto el orden del algoritmo es:

$$O(KEYSt_{key\_ev} + LOGSt_{log\_ev} + T_{get\_servers}) = O(KEYS + LOGS + T_{get\_servers}) \quad (4.1)$$

4.4. Diseño del componente Audit Manager

El componente Audit Manager extiende la arquitectura de PriServ para realizar una auditoría de privacidad a los pares del sistema P2P. Este componente se comunica con algunos componentes de PriServ y audita a los servidores (véase Sección 5.3) del sistema. El diseño del componente de auditoría lo conforman un conjunto de vistas que describen la arquitectura del componente, decisiones de diseño relevantes y algunos aspectos importantes de la comunicación con elementos de PriServ. Las vistas del diseño son la de casos de uso, arquitectura global, lógica y de procesos.

Se hace especial énfasis en identificar los componentes y operaciones que se agregan, así como los componentes de PriServ que el Audit Manager utiliza para su funcionamiento.

4.4.1. Vista de casos de uso

En esta vista se representan y documentan las funciones centrales (casos de uso) del Audit Manager, con base en los requerimientos funcionales para la auditoría de privacidad. En la Figura 4.1 se muestra el caso de uso Auditoría de Servidores, el cual realiza una auditoría de privacidad a los pares del sistema P2P.

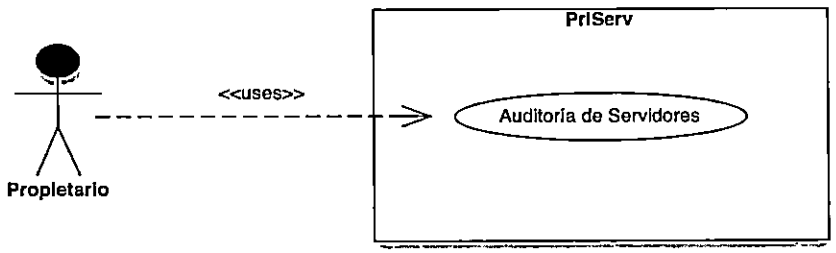

Figura 4.1: Diagrama de Casos de Uso

El caso de uso lo inicia el propietario de datos que desea verificar el cumplimiento de sus políticas de privacidad. El flujo principal del caso de uso es el siguiente:

1. El propietario inicia el caso de uso:
2. El Audit Manager inicia el algoritmo de auditoría.
3. El Audit Manager solicita información de auditoría a los servidores que almacenan datos del propietario.
4. Los servidores envían la información de auditoría al propietario.
5. El Audit Manager identifica comportamientos maliciosos con la información de auditoría de los servidores.

4.4.2. Vista de arquitectura global

En la Figura 4.2 se muestra la arquitectura global de PriServ con el componente Audit Manager incluido.

Las operaciones se clasifican de la siguiente manera:

- En color rojo está el Audit Manager y las nuevas operaciones que se agregaron.
- En color verde se encuentran las operaciones de PriServ que el Audit Manager utiliza, sin modificaciones.
- En azul se encuentran las operaciones restantes de PriServ.

Una vez ubicado el servicio de auditoría en la arquitectura global de PriServ, en la Figura 4.3 sólo se muestran las interfaces y los componentes de la arquitectura de PriServ, con los cuales el *Audit Manager* se comunica para realizar la auditoría.

La auditoría inicia en el orquestador del propietario cuando un par ejecuta la operación *startAudit()*. Las operaciones *serversAuditing()*, *getAuditInformation()* y *checkCompliance()*, corresponden a los Algoritmos 1, 2 y 3 respectivamente (descritos en la Sección 4.3). Se agregó la operación *getAuditResult()* al componente Log Manager para obtener los registros de la bitácora del servidor, asociados con una llave.

El Audit Manager se comunica con el componente:

- **IdManager** para obtener una referencia a los servidores que serán auditados.
- **StorageManager y LogManager** para obtener la información de auditoría.
- **PolicyManager** para obtener la política de privacidad y verificar el cumplimiento de dicha política.

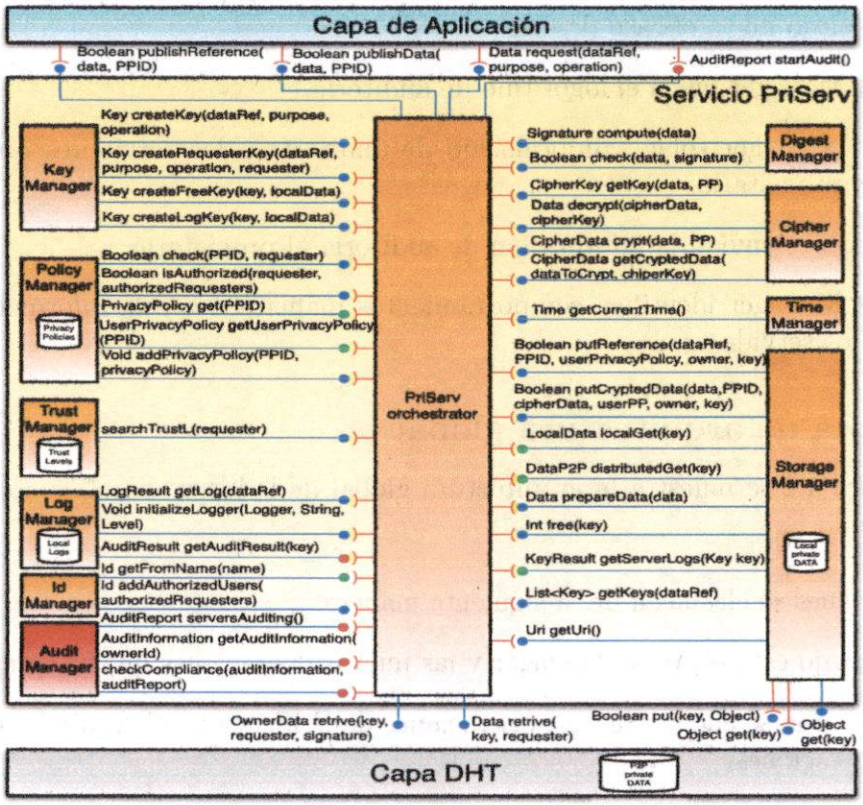

Figura 4.2: Arquitectura de PriServ extendida con el Audit Manager

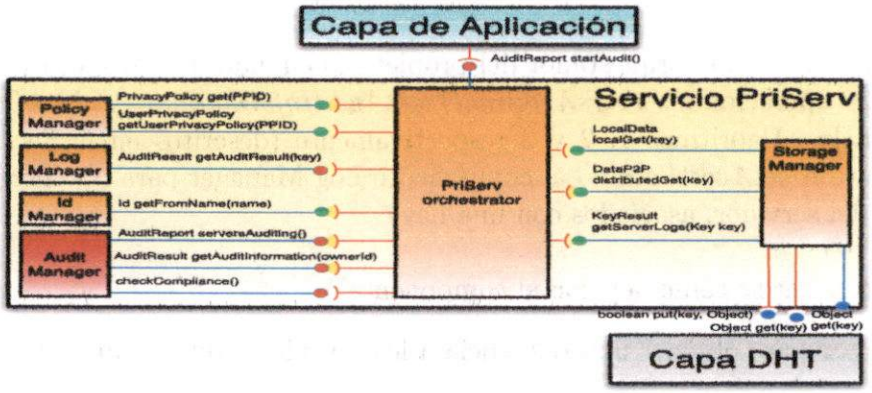

Figura 4.3: Componentes de PriServ que utiliza el Audit Manager

4.4.3. Vista lógica

La vista muestra los elementos relevantes del Audit Manager clasificados en sub-sistemas o paquetes. Para cada elemento se describen sus responsabilidades y relaciones con otros. Como se observa en la Figura 4.4 el Audit Manager está compuesto por los siguientes paquetes:

- **Interfaces.** Conjunto de interfaces que se clasifican en dos tipos. El primer tipo de interfaces son utilizadas por los pares para iniciar las operaciones de la auditoría. El segundo tipo ayudan a la persistencia de datos locales. El objetivo de este paquete es separar la implementación de la auditoría y la persistencia de datos.

- **Control.** Clases que coordinan los procesos de auditoría para comunicarse con los componentes de PriServ, realizar la persistencia de datos, implementar las operaciones de las interfaces, etc.; en general las clases de control crean objetos útiles para realizar la auditoría.

- **Implementation.** Clases concretas que implementan las operaciones de las interfaces.

- **Domain.** Clases útiles para administrar datos de la auditoría, como el reporte final, bitácoras, etc.

- **Persistence.** Clases que implementan la persistencia de los datos.

Las flechas que hay entre los paquetes representan el sentido de la comunicación entre las clases del Audit Manager y los componentes de PriServ. Una descripción global del protocolo de comunicación es la siguiente:

1. Los orquestadores del servidor y propietario (representados por el paquete priserv.api) inician las operaciones de auditoría que ofrecen las interfaces del Audit Manager.

2. Las clases de control implementan (con el paquete Implementation) las operaciones de las interfaces, para realizar el proceso de auditoría, el cual requiere:

   - La ejecución de las operaciones de los componentes de PriServ (StorageManager, LogManager, IdManager y PolicyManager).
   - Instanciar los objetos para la persistencia de datos, gestión de datos en memoria con el uso de clases en el paquete Domain.

3. Las clases de control entregan los resultados de la auditoría a los servidores o propietarios.

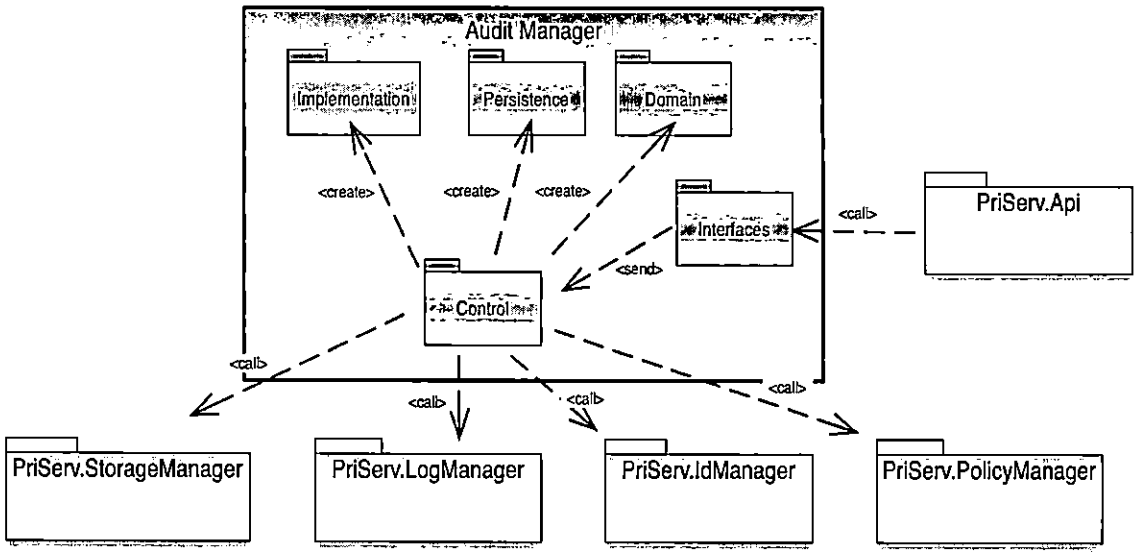

Figura 4.4: Paquetes del AuditManager y PriServ

4.4.4. Vista de procesos

La vista de paquetes describe de manera global la clasificación y comunicación entre las clases (en paquetes) del Audit Manager y algunos componentes de PriServ. Sin embargo, no se detalla la lógica del proceso de auditoría, el intercambio de mensajes o la ejecución de las operaciones específicas de cada componente. En la vista de procesos se utilizan diagramas de secuencia para detallar el protocolo de comunicación entre los componentes de PriServ y el Audit Manager, cuando se realiza la auditoría. (Figuras 4.5, 4.6 y 4.7)

En todos los diagramas de secuencia, los objetos en color amarillo son administrados por el propietario y los objetos en color verde por los servidores. Los tipos de objetos que contienen los diagramas de secuencia son los siguientes:

- *Interface.* Ofrece operaciones que son implementadas por otras clases. Se utilizan para separar la implementación y las operaciones de la interfaz (e. g., los objetos Auto y Avión tienen distintas implementaciones de la operación *avanzar*() de alguna interfaz).

- *Control.* Coordinan a otros objetos para encapsular y realizar una funcionalidad en particular (e. g., para implementar la operación *enciendeAuto*(), un objeto de tipo control crea una Persona y un Auto, indica a la persona que aborde el auto y encienda el motor).

- *Entity*. Objetos que son creados y utilizados por los objetos de tipo control.

El diagrama de secuencia de la Figura 4.5 describe el funcionamiento principal de la auditoría.

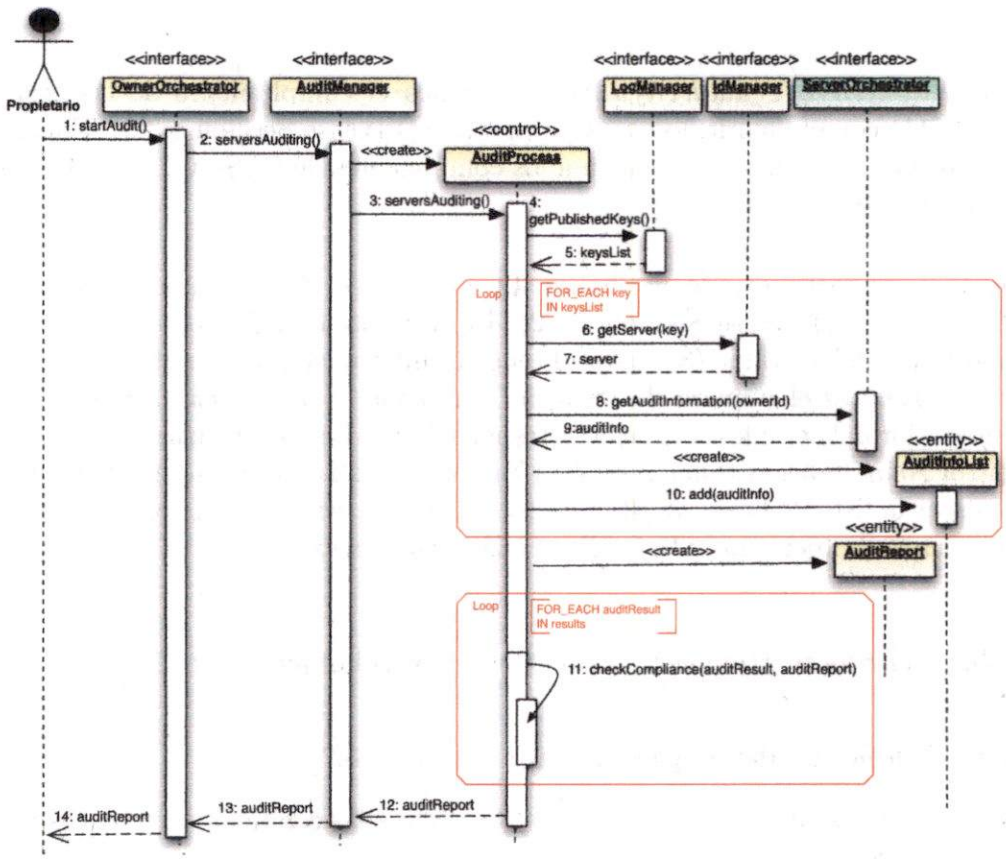

Figura 4.5: Diagrama de secuencia del algoritmo *serversAuditing*() (véase Sección 4.3.1).

Desde la capa de aplicación un propietario inicia la auditoría (1) con la función *startAudit*(). El orquestador del propietario envía un mensaje al componente *Audit Manager* para auditar a los servidores (2). El *Audit Manager* crea el control *AuditProcess* e inicia el algoritmo de auditoría (3) *serversAuditing*() (véase Sección 4.3.1). Se solicita al componente Log Manager la lista de llaves *keysList* (4 y 5), que el propietario ha publicado en el sistema. Para cada llave $key \in keysList$:

- Se solicita al componente Id Manager una referencia al servidor *server* que almacena *key* (6 y 7).

- Se solicita al orquestador del servidor *server* la información de auditoría *auditInfo* con el algoritmo *getAuditInformation*() (véase Sección 5.3.2) (8 y 9).

- Se agrega *auditInfo* a la lista de resultados de auditoría *auditInfoList* (10).

Para cada *auditInfo* $\in$ *auditInfoList*, se verifica el cumplimiento de las preferencias de privacidad (11) con el algoritmo *checkCompliance*() (véase Sección 4.3.3). Finalmente el reporte de auditoría *auditReport* se envía a los componentes de PriServ (12 y 13), hasta que se entregue al propietario (14).

El diagrama de secuencia de la Figura 4.6 describe el funcionamiento del algoritmo *getAuditInformation*() (véase Sección 5.3.2). El orquestador del propietario se comunica con el orquestador del servidor (8), para obtener la información de auditoría (se inicia con el número 8 porque es el número de línea de la función *serversAuditing*()) de la Figura 4.5. El orquestador del servidor le delega la responsabilidad al *Audit Manager* (8.1), el cuál crea el control *AuditProcess* e inicia el algoritmo *getAuditInformation*() (8.2). Se solicita al *Log Manager* la lista de llaves *keyList* (8.3 y 8.4) que el servidor almacena del propietario *ownerId* y se crea el objeto *auditInfo*. Para cada *key* $\in$ *keysList*:

- Se solicita al *Storer Manager* (valores obtenidos del objeto *dataP2P*):
  - El identificador del propietario *ownerId* (8.6 y 8.7)
  - Se obtiene la ACL en la variable *acl* (8.8 y 8.9) y las condiciones de uso *usageConditions* (8.10 y 8.11).
- Se solicita al *Log Manager* la lista de registros *logsList* de la bitácora del servidor (8.12 y 8.13).
- Se crea el objeto *result* (con *key, ownerId, acl y usageConditions*) y se agrega *result* al objeto *auditInfo* (8.14).

Finalmente se envía la información de auditoría a los componentes de PriServ (8.15 y 8.16), hasta que se entregue al orquestador del propietario (8.17).

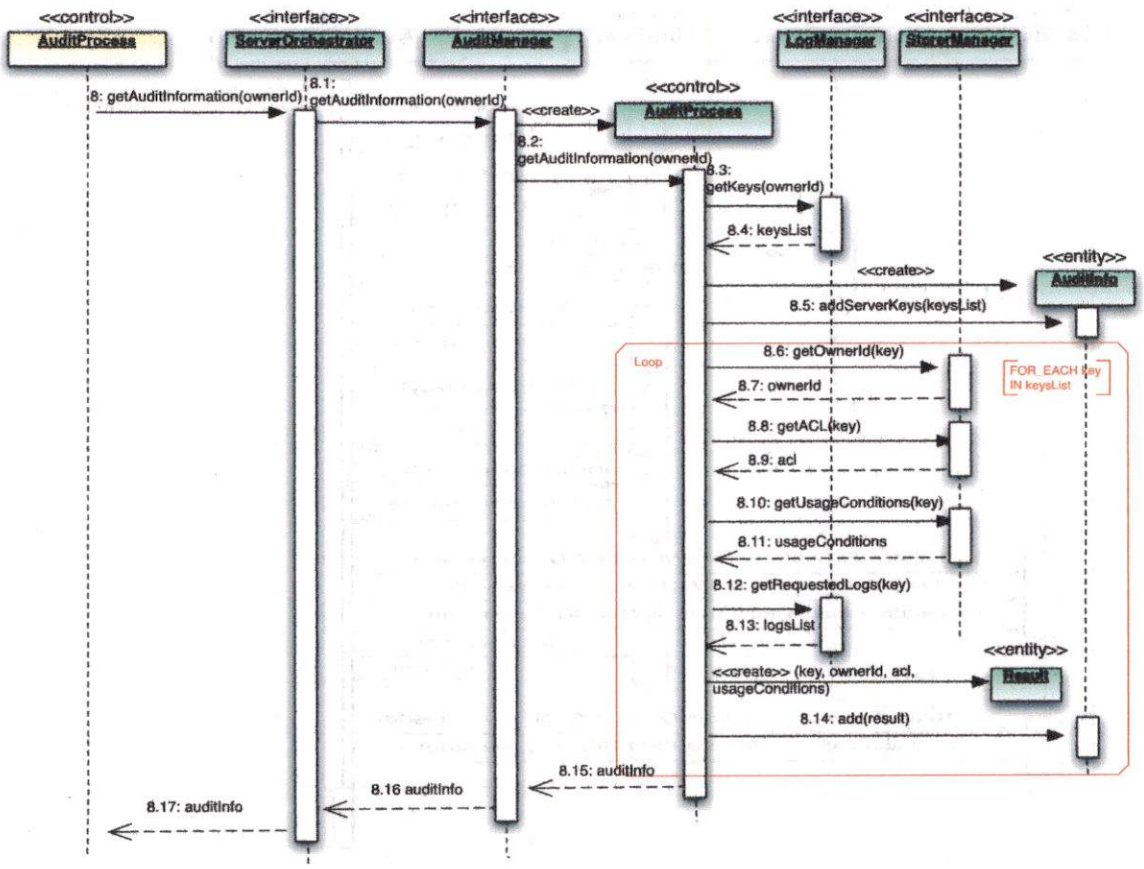

Figura 4.6: Diagrama de secuencia del algoritmo *getAuditInformation*() (véase Sección 4.3.2)

El diagrama de secuencia de la Figura 4.7 describe el funcionamiento del algoritmo *checkCompliance*() (véase Sección 5.3.3).

En el orquestador del propietario, el objeto *Audit Process* inicia el proceso (se inicia con el número 11 porque es el número de línea en la función *serversAuditing*() de la Figura 4.5). Se solicita al objeto $auditInfo$ la lista de llaves *keysList* (11.1 y 11.2) y el identificador del servidor *serverId* (11.3 y 11.4) del cual se obtuvo la información de auditoría. Para cada $key \in keysList$:

- Se solicita al *Policy Manager* la política de privacidad *privacyPolicie* asociada con *key* (11.5 y 11.6).
- Del objeto $auditInfo$ se obtiene:

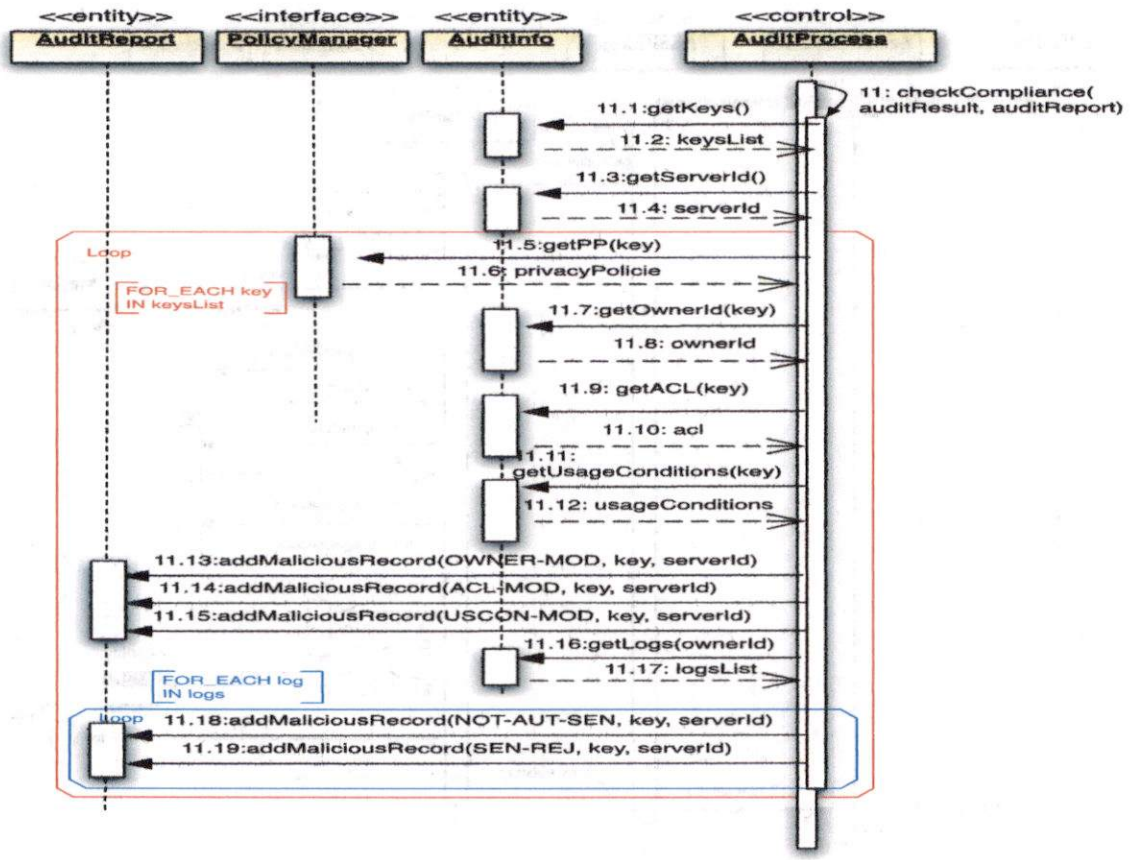

Figura 4.7: Diagrama de secuencia del algoritmo *checkCompliance*() (véase Sección 4.3.3)

- Se solicita al *Policy Manager* la política de privacidad *privacyPolicie* asociada con *key* (11.5 y 11.6).
- Del objeto $auditInfo$ se obtiene:
  - El identificador del propietario $ownerId$ (11.7 y 11.8)
  - La ACL *acl* (11.9 y 11.10)
  - Las condiciones de uso *usageConditions* (11.11 y 11.12)
- Si el servidor modificó el identificador del propietario, la ACL o las condiciones de uso, se agrega el registro malicioso correspondiente al reporte de auditoría (11.13, 11.14 y 11.15)

- Del objeto $auditInfo$ se obtiene la lista de registros $logsList$ (11.16 y 11.17). Para cada $log \in logsList$, si log registró una entrega de datos a un par no autorizado o rechazó el envío de datos a un par autorizado, se agrega el registro malicioso correspondiente al reporte de auditoría (11.18 y 11.19).

4.5. Resumen

En este capítulo se presentó el servicio de auditoría de privacidad para sistemas P2P, como parte del trabajo de investigación. Una vez identificada la problemática que nos interesa resolver (detectar comportamientos maliciosos que violan las políticas de privacidad), se enunciaron los objetivos del servicio de auditoría.

La metodología que se llevo a cabo fue en primer lugar, la identificación de escenarios maliciosos. En segundo lugar se diseño un algoritmo de auditoría que detecta los ataques de los servidores (véase el escenario Esc1-DNA), además se analizó el orden del algoritmo en términos del número de mensajes y el tiempo que se requiere para auditar a los servidores de un sistema P2P. Finalmente, se realizó el diseño del componente Audit Manager (mediante la documentación de vistas de diseño de software) que extiende la arquitectura del servicio de privacidad PriServ e implementa el algoritmo de auditoría.

Capítulo 5
Implementación y evaluación

En la etapa de implementación del servicio de auditoría, se desarrolló un prototipo del componente *Audit Manager* encargado de implementar el algoritmo para auditar a los servidores.

El prototipo del *Audit Manager* tiene una arquitectura por capas (ver Figura 5.1), para separar las clases de control, las interfaces y las clases para la persistencia de los datos. La clase *AuditProcess* (en la capa *Control*) es el orquestador del servicio: recibe las solicitudes de los orquestadores de PriServ durante la ejecución del algoritmo y después inicia las funciones de auditoría (*getAuditInformation*() y *checkCompliance*()) que están codificadas en la clase *AuditManagerImpl* (en la capa *Implementation*). La información de auditoría se obtiene del *LogManager* a través de la clase *SQLiteDaoFactory* (en la capa *Persistence*) que representa una implementación de las bitácoras en PriSery; la clase *DaoFactory* permite utilizar distintas implementaciones para la persistencia de datos. Finalmente en la capa *Domain* se encuentran las clases concretas *AuditReport*, *Result* y *AuditInfo*.

La implementación del servicio de auditoría se realizó con el sistema integrado SPLAY [20] que facilita el diseño, la implementación y pruebas de aplicaciones distribuidas. De PriServ se codificaron las funciones de publicación *publishReference()* y solicitud de datos *request()*, el prototipo del *Audit Manager* y los componentes que se muestran en la Figura 5.1.

Para evaluar el algoritmo de auditoría se analizó el número de mensajes que requiere la auditoría y se diseñó un caso de prueba que agrega registros maliciosos a las publicaciones de un sistema P2P, con base en las acciones del escenario Escl-DNA (véase Sección 4.2). El objetivo es comprobar que el algoritmo identifica los registros maliciosos, así como medir el tiempo que tarda la auditoría en dicho caso de prueba.

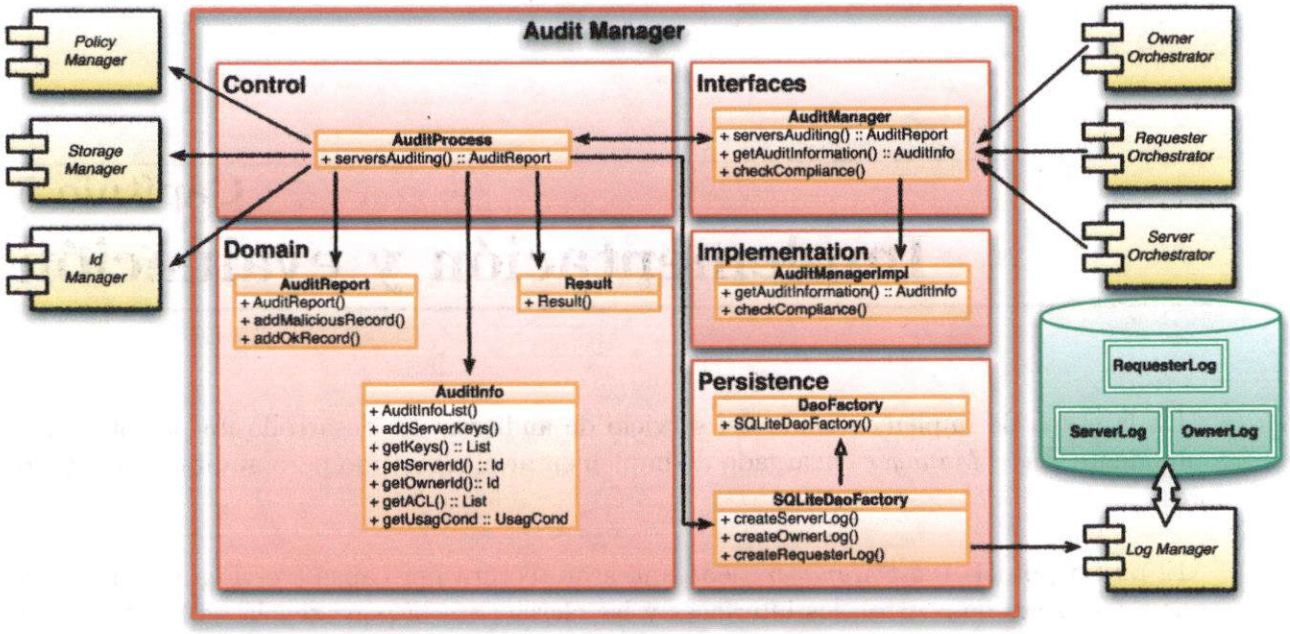

Figura 5.1: Arquitectura del componente Audit Manager.

5.1. Plataforma de experimentación: SPLAY

Una dificultad en el desarrollo de aplicaciones distribuidas, es la falta de herramientas integrales que permitan la construcción de prototipos, la implementación y la evaluación de algoritmos en ambientes reales. Usualmente hay que invertir un tiempo considerable en la configuración de ambientes de prueba (PlanetLab [21], ModelNet [22]) y utilizar diferentes herramientas en cada etapa del desarrollo (diseño, implementación, implantación, ejecución y monitoreo). Sin embargo, es necesario el uso de estas herramientas porque al evaluar aplicaciones distribuidas, es común encontrar discrepancias entre el análisis de los resultados esperados y lo resultados que se obtienen al hacer una implementación en una red real.

SPLAY [20] es un sistema integrado que simplifica la creación de prototipos, el desarrollo, la implementación y evaluación de aplicaciones distribuidas; además ofrece a los programadores un lenguaje independiente de plataforma (Lua [23]). SPLAY provee el monitoreo y control de varios experimentos a la vez, utilizando zonas seguras (*sandbox*) en cada nodo, sobre distintos ambientes de ejecución, como PlaneLab, ModelNet, redes no dedicadas o computadoras personales.

SPLAY cuenta con dos componentes principales:

- **splayctl (controlador)**: coordina la implementación y ejecución de aplicaciones distribuidas.
- **splayd (demonios)**: procesos que se ejecutan en cada nodo del ambiente de ejecución.

El controlador está implementado como un conjunto de procesos cooperativos que se comunican a una base de datos central, la cual contiene datos de los *host* (dirección IP y puertos, etc.), demonios (identificadores, ancho de banda y memoria, etc.) y aplicaciones (archivos fuente). Para ejecutar una aplicación distribuida, los demonios deben ser instalados y ejecutados por el administrador de cada nodo. Cuando un demonio inicia se registra en el controlador que el administrador haya especificado. Una vez registrados los demonios, el controlador envía los comandos para iniciar, detener y monitorear las aplicaciones distribuidas; es posible ejecutar diferentes aplicaciones y especificar a los demonios que participan en cada una (véase Figura 5.2). Las aplicaciones se comunican entre ellas mediante llamadas remotas.

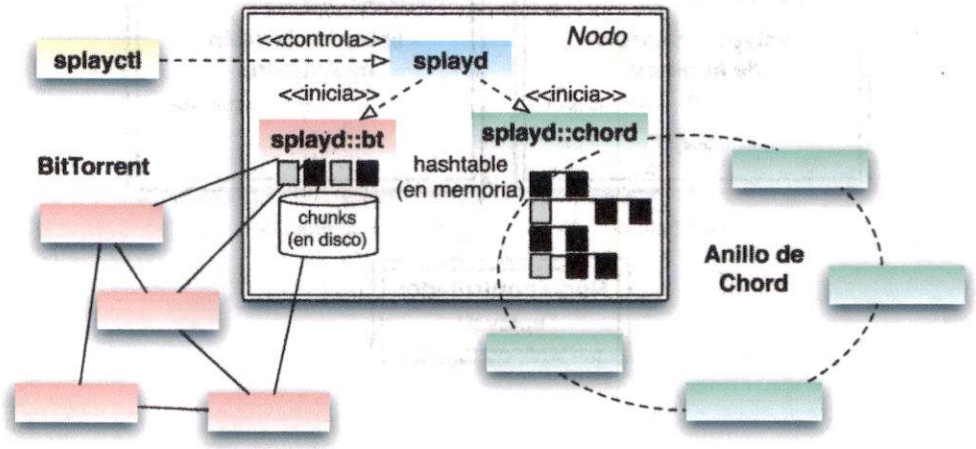

Figura 5.2: Ejecución de dos aplicaciones (BitTorrent, Chord) en distintos demonios y administrados por un controlador.

Es posible crear aplicaciones sencillas y probarlas en los demonios que se ejecutan en un nodo, con todas las bibliotecas de SPLAY. Las aplicaciones se escriben utilizando el lenguaje de programación Lua [23], por ser un lenguaje fácil de entender (su sintaxis es muy parecida al pseudocódigo que se muestra en algunos artículos de investigación), portable y eficiente al soportar diferentes procesos para un solo nodo.

5.2. Descripción general del prototipo

La Figura 5.3 muestra un ejemplo de cómo se utilizó SPLAY para la implementación del prototipo de auditoría y el sistema P2P. En un nodo se instaló el controlador de SPLAY, cada nodo (1 y 2) restante cuenta con un demonio que es administrado por el controlador. Los demonios ejecutan la aplicación distribuida, en este caso, se ejecuta el protocolo de la DHT y el servicio PriServ con el componente *Audit Manager*. Nótese que la Figura 5.3 representa una emulación de un sistema P2P de dos nodos que utiliza un servicio de privacidad, extendido con un componente de auditoría, para el intercambio de datos.

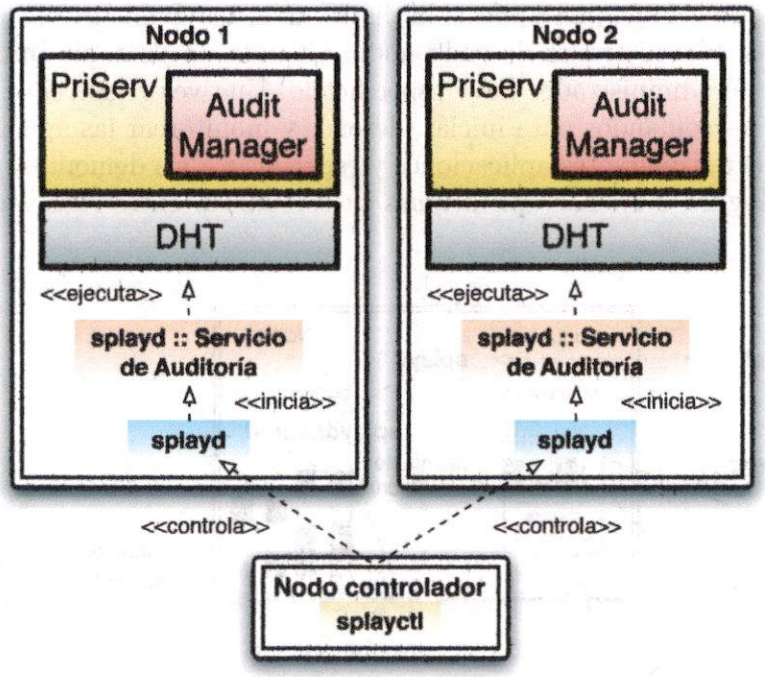

Figura 5.3: Implementación del prototipo de auditoría con SPLAY.

Es posible que cada nodo del ambiente de ejecución contenga más de un demonio. SPLAY implementa el protocolo TCP/IP para la comunicación en la red.

5.2.1. Representación del Sistema P2P

Cada par del sistema P2P está representado por un demonio (splayd) de SPLAY. Para iniciar el sistema P2P el controlador le indica a los demonios participantes que inicien el protocolo Chord. En el prototipo desarrollado se implementó el protocolo básico propuesto

en [17]. Este protocolo es capaz de reconstruir el anillo ante la salida, informada o no, de un nodo pero no considera la transferencia del contenido del nodo que sale.

Chord es un sistema P2P que se basa en una tabla de dispersión distribuida (DHT) que hace un mapeo entre llaves y pares del sistema, los pares utilizan su sección de la DHT (tabla de apuntadores con $log(n)$ entradas, con n el número de pares en el sistema) para encontrar el par actual que es responsable de una llave.

```
1 function join(n0)
2   predecessor = nil
3   finger[1] = call(n0, {'findSuccessor',
                          N.id})
4   call(finger[1], {'notify', N})
5 end

6 function stabilize()
7  local x = call(finger[1],'predecessor')
8  local y = call(finger[1],'N')
9  local z = call(x, 'N')
10  if x and between(z.id, N.id, y.id,
                     false, false) then
11      finger[1] = x
12  end
13  call(finger[1], {'notify', N})
14 end

15 function checkPredecessor()
16  if predecessor and not ping(predecessor) then
17      predecessor = nil
18  end
19 end

20 function fixFingers()
21  refresh = (refresh % M) + 1
22  finger[refresh] =
       findSuccessor((N.id + 2^(refresh - 1)) % 2^M)
23 end

24 function notify(n0)
25  if not predecessor then
26      predecessor = n0
27  else
28      local x = call(n0, 'N')
29      local y = call(predecessor, 'N')
30      if between(x.id, y.id, N.id,
                   false, false) then
31          predecessor = n0
32      end
33  end
34 end

35 function findSuccessor(id)
36  local x = call(finger[1], 'N')
37  if betweenInclude(id, N.id, x.id,
                      false, true) then
38      return finger[1]
39  end
40  local n0 = closestPrecedingNode(id)
41  return call(n0, {'findSuccessor', id})
42 end

43 function closestPrecedingNode(id)
44  local i
45  for i = M, 1, -1 do
46      if finger[i] then
47          local x = call(finger[i], 'N')
48          if between(x.id, N.id, id,
                       false, false) then
49              return finger[i]
50          end
51      end
52  end
53  return N
54 end
```

Figura 5.4: Funciones para mantenimiento y construcción de Chord.

Cuando un par se une al sistema recibe un identificador único (típicamente al aplicar una función Hash a su dirección IP y número de puerto) que determina su posición; los pares se organizan en un anillo conforme a sus identificadores. Cada par es responsable por las llaves que se encuentran entre él (incluyéndose) y su predecesor (sin incluirlo).

En la Figura 5.4 se muestra el código para la construcción y el mantenimiento del protocolo Chord. La función *join*() permite que un nodo se una al anillo de Chord, con ayuda del nodo $n0$, el cuál forma parte del anillo. Las funciones *stabilize*(), *checkPredecessor*() y *fixFingers*() se ejecutan de manera periódica para actualizar el par sucesor, la tabla de

apuntadores y monitorear que el par predecesor no haya fallado, respectivamente. La función *notify*() determina quién será el predecesor. Finalmente, la función *findSuccessor*() es la función de localización (*lookup*()) que determina el par sucesor de un identificador dado.

La Figura 5.5 muestra el código para inicializar el sistema P2P. Se inicializa el sistema con un anillo de tamaño dos (líneas 2 - 9). Posteriormente, el resto de los pares se unen al anillo (con la función *join*()) e inicializan las variables del protocolo P2P (líneas 10 -15). Todos los pares ejecutan de manera periódica las funciones para el mantenimiento de Chord.

```
M = 5              --[[2^M pares y llaves con
                   identificadores de longitud M]]--
N = job.me         --[[Par actual {identificador,
                   dirección IP, número de puerto}]]--
TIMEOUT = 5        --Frecuencia de estabilización
refresh = 1        --[[Índice de la tabla de apuntadores
                     que será actualizado]]--
N.id = job.position --Identificador del par

1 function p2PsystemInitialization()
2   if N.id == 1 then
3       predecessor = job.nodes[2]
4       finger = {[1] = job.nodes[2]}
5   end
6   if N.id == 2 then
7       predecessor = job.nodes[1]
8       finger = {[1] = job.nodes[1]}
9   end
10  if N.id > 2 then
11      predecessor = nil
12      finger = {[1] = N}
13      n0 = job.nodes[1]
14      join(n0)
15  end
16  events.periodic(stabilize, TIMEOUT)
17  events.periodic(checkPredecessor, TIMEOUT)
18  events.periodic(fixFingers, TIMEOUT)
19 end
```

Figura 5.5: Función que inicializa el anillo de Chord y variables globales.

5.2.2. Funciones de PriServ

Para auditar el sistema P2P se requiere que exista el registro de publicaciones y solicitudes de datos, respetando el modelo de PriServ. Por lo tanto se codificaron las operaciones *publishReference*() y *request*(), descritas en la Sección 3.3. Sólo se describe el código de algunas funciones relevantes para la publicación y solicitud de datos.

Función *publishReference*()

La Figura 5.6 (a) muestra el código de la función de publicación por referencia. Se recibe el identificador de la política de privacidad y el dato a publicar (línea 1). El *Policy Manager*

obtiene la política de privacidad y el *Key Manager* crea una llave, al aplicar la función Hash SHA-1 a la concatenación de la referencia de los datos, el propósito y la operación permitidos (2 y 3). El *Storage Manager* almacena los datos de manera local y distribuida con la función *localPut*() (4). Se registra la publicación en la bitácora del propietario con la referencia del dato (*dataRef*), la llave (*key*), la fecha y hora del registro (*os.date*()) y un valor booleano (*result*) que significa el éxito o rechazo de la publicación (5). La función *localPut*() agrega un registro a la tabla de datos y a la tabla de datos privados (véase la Sección 3.2), respectivamente (8 y 9). Finalmente se crea el dato P2P (10) y se almacena en el sistema P2P (11).

Cuando un servidor recibe una publicación (consecuencia de la función *DHT.put*()), almacena los datos con la función *putDataP2P*(). El servidor almacena el dato P2P (línea 14) y registra la publicación en su bitácora (15) con la llave (*key*), la fecha y hora del registro (*os.date*()), el identificador del propietario (*ownerId*) y la lista de usuarios autorizados (*acl*).

(a)

```
1 function publishReference(data, ppId)
2   privacyPolicy = PolicyManager.get(ppId)
3   key = KeyManager.createKey(data['dataRef'],
          privacyPolicy['purpose'],
          privacyPolicy['operation'])
4   result = StorageManager.localPut(data, ppId,
          privacyPolicy['conditions'],
          privacyPolicy['acl'], N.id, key)
5   LogManager.insertIntoTable('OwnerLog',
          data['dataRef'], key, os.date('%c'),
          result)
6 end
```

(b)

```
7 function localPut(data, ppId, conditions,
                    acl, ownerId, key)
8   StorageManager.insertIntoTable('Data',
              data['dataRef'],data['info'])
9   StorageManager.insertIntoTable('PrivateData',
              data['id'], ppId, key)
10  dataP2P = StorageManager.createDataP2P(
              conditions, acl, ownerId)
11  DHT.put(key, dataP2P)
12 end
```

(c)

```
13 function putDataP2P(key, dataP2P)
14   StorageManager.insertIntoTable('DataP2P',
          key, dataP2P['ownerId'], dataP2P['acl'],
          dataP2P['conditions'])
15   StorageManager.insertIntoTable('ServerLog',
          key, os.date('%c'), dataP2P['ownerId'],
          dataP2P['acl'])
16 end
```

Figura 5.6: Funciones involucradas en la publicación por referencia.

Función *request*()

En la Figura 5.7 (a) se muestra el código de solicitud de datos. El solicitante especifica la referencia, el propósito y la operación (línea 1), posteriormente el *Key Manager* crea la llave y se solicita al sistema el dato P2P (4 y 5 respectivamente). Como el dato P2P es una referencia, se solicita al propietario el dato correspondiente (6). Se registra la solicitud en la bitácora del solicitante (7) con la referencia (*dataRef*), la llave (*key*), la fecha y hora de registro (*os.date*()), el propósito, la operación y un valor booleano (*result*) que significa el éxito o rechazo de la solicitud. Finalmente se regresa el dato al solicitante (8).

En la Figura 5.7 (b) está el código de la función *retrieve*() que ejecuta un propietario cuando recibe una solicitud. El *Storage Manager* obtiene los datos asociados con la llave (*key*) y el *Policy Manager* verifica si el solicitante tiene derecho a recibir los datos (líneas 11 y 12 respectivamente). Se registra la solicitud en la bitácora del propietario con la referencia (*dataRef*), la llave (*key*), la fecha y hora de registro (*os.date*()), el identificador del solicitante (*requesterId*) y un valor booleano (*result*) que significa el éxito o rechazo de la entrega de datos (13). Finalmente, se regresa el dato y el resultado de la solicitud (14).

Cuando un servidor recibe una solicitud (consecuencia de la función *DHT.get*()), verifica si el solicitante está autorizado a obtener los datos con la función *requesterEvaluation*(), que se muestra en la Figura 5.7 (c). Se obtiene (de manera local) el dato P2P asociado con la llave (línea 18). Después se verifica si el solicitante está en la lista de usuarios autorizados (19). Finalmente, se registra la solicitud en la bitácora del servidor con la llave (*key*), la fecha y hora de registro (*os.date*), el identificador del solicitante (*requesterId*) y un valor booleano (*result*) que significa el éxito o rechazo de la solicitud (20).

(a)

```
1 function request(dataRef, purpose, operation)
2   data = nil
3   result = false
4   key = KeyManager.createKey(dataRef,
            purpose, operation)
5   dataP2P = DHT.get(key)
6   data, result = call(dataP2P['ownerId'],
                  {'retrieve', key, N.id})
7   LogManager.insertIntoTable('RequesterLog',
        dataRef, key, os.date('%c'), purpose,
        operation, result)
8   return data
9 end
```

(b)

```
10 function retrieve(key, requesterId)
11   data = StorageManager.localGet(key)
12   result = PolicyManager.check(data['ppId'],
                requesterId)
13   LogManager.insertIntoTable('OwnerLog',
       data['dataRef'], key, os.date('%c'),
       requesterId, result)
14   return data, result
15 end
```

(c)

```
16 function requesterEvaluation(requesterId, key)
17   result = false
18   dataP2P = StorageManager.localGet(key)
19   result = contains(requesterId, dataP2P['acl'])
20   LogManager.insertIntoTable('ServerLog', key,
                os.date('%c'), requesterId, result)
21   return result
22 end
```

Figura 5.7: Funciones involucradas en la solicitud de datos.

5.2.3. Funciones del Audit Manager

En las Figuras 5.8 (a) y 5.8 (b) se muestra el código de las funciones (descritas en la Sección 5.3) *serversAuditing*() y *getAuditInformation*() respectivamente, que auditan a los servidores del sistema.

Función *serversAuditing*()

El *Log Manager* obtiene todas las llaves del propietario publicadas en el sistema P2P (línea 2). Para cada llave (líneas 3 y 4), el *Id Manager* obtiene una referencia al servidor que almacena el dato P2P correspondiente (5). Después se le solicita al servidor toda la información de auditoría del propietario, con la función *getAuditInformation*() (6). Posteriormente se detectan los registros maliciosos (7), con base en el escenario Escl-DNA (véase Sección 5.1) con la función *checkCompliance*() (véase la Sección 5.3.3). Finalmente, se actualiza la lista de llaves para evitar que se audite más de una vez a un servidor (8).

Función *getAuditInformation*()

Al solicitar la información de auditoría con la función *getAuditInformation*(), el *Log Manager* obtiene la lista de llaves del propietario (línea 12). Para cada llave el *Storage Manager* obtiene el dato P2P asociado y el *Log Manager* obtiene las solicitudes (registradas en la bitácora del servidor) del dato P2P por los pares del sistema (15 y 16 respectivamente). El resultado de auditar a un servidor consta de una tabla Hash (que por cada llave almacena los campos del dato P2P y el registro de sus solicitudes, en la líneas 17-19), el identificador del servidor (21) y la lista de llaves del propietario (22).

(a)

```
1 function serversAuditing()
2   keysList = LogManager.getOwnerKeys()
3   while containsItems(keysList) do
4       key = choseFirst(keysList)
5       server = IdManager.getServer(key)
6       auditInfo = call(server,
                    {'getAuditInformation', N.id})
7       checkCompliance(auditInfo)
8       keysList = markItems(keysList,
                    auditInfo['keysList'])
9   end
10 end
```

(b)

```
11 function getAuditInformation(ownerId)
12  keysList = LogManager.getKeysFromServer(ownerId)
13  for i = 1, # keysList, 1 do
14      key = keysList[i]
15      dataP2P = StorageManager.getDataP2P(key)
16      logsList = LogManager.getRequesterLogs(key)
17      auditResult['dataP2P'] = dataP2P
18      auditResult['logsList'] = logsList
19      auditInfo[key] = auditResult
20  end
21  auditInfo['keysList'] = keysList
22  auditInfo['serverId'] = N.id
23  return auditInfo
24 end
```

Figura 5.8: Funciones de auditoría.

5.3. Características del experimento

Se utilizó un *cluster* local de 5 nodos, cada nodo está equipado con un procesador AMD de 64 bits a 2.13 Ghz, 2 GB en memoria y cada uno ejecuta el sistema operativo GNU/Linux 2.6.9.

En un nodo se instaló la versión 1.02 del controlador de SPLAY y en los 4 nodos restantes, se instaló la versión 1.0 del *splayd* [24] (para la ejecución de los demonios).

Se instaló la versión 5.1.4 del lenguaje de programación Lua, la versión 5.1.14 de mysql y la versión 1.8.7 de Ruby (los últimos dos para uso exclusivo del controlador de SPLAY).

Se utilizaron cuatro nodos para correr siete demonios de SPLAY en cada nodo y en el quinto nodo se inicia el controlador de SPLAY. Se emuló un sistema P2P de 28 pares (un par por demonio), cada par se configuró con las siguientes características:

- 12 Mb en memoria
- 1 Gb en disco
- Ancho de banda 10Mbps (envío y recepción)

Cada par administra una base de datos que contiene las tablas de PriServ y sus bitácoras (véase Sección 3.2 y 3.4, respectivamente), dependiendo del tipo de par (propietario, servidor o solicitante). La implementación de las bases de datos se realizó con el gestor de bases de datos SQLite [25]. Para tener acceso a las diferentes tablas desde la aplicación con Lua, se utilizó el API LuaSQLite [26].

5.4. Evaluación

Se diseñó un caso de prueba para comprobar que el algoritmo de auditoría encuentra todos los registros maliciosos y medir el tiempo que tarda en hacerlo. Para este caso de prueba se considera una red estática de 28 pares. Los pares del sistema están identificados con el número 1 hasta el 28. El propietario tiene el identificador 1 y cuenta con 27 servidores (del 2 al 28) que serán auditados.

Variable	Descripción	Valor
M	Bits en la llave	5
N	Número máximo de pares	2^5
P	Propósitos	10
PP	Políticas de privacidad	30
F	Referencias de datos que se publicaron	50

Tabla 5.1: Parámetros de emulación del caso de prueba.

El caso de prueba consta de tres etapas:

1. *Publicación:* Para que el propietario pueda publicar, debe registrar políticas de privacidad (pp) en la tabla (véase la Sección 3.2) correspondiente. Dado que el modelo PriMod (véase la Sección 3.2) permite tres operaciones (a saber, divulgación, lectura y escritura), se utilizan 10 propósitos, se cuenta con $10 \times 3 = 30$ pp; cada pp tiene registrada la lista $\{2, 3, 4, ..., 28\}$ como lista de pares autorizados. Se consideraron 50 referencias de datos (Tabla 5.1) que se asociarán con cada pp, por lo tanto, el propietario realizará $30 \times 50 = 1,500$ publicaciones con la función *publishReference*().

2. *Solicitud:* En esta etapa se realizaron solicitudes legítimas (de pares autorizados) de los datos publicados en la etapa uno. Se eligió el 70 % (1,050) de las publicaciones con una distribución uniforme, para ser solicitados por los 27 pares (del 2 al 28) que también se eligieron de manera uniforme.

3. *Acciones maliciosas:* Como se describió en el escenario Escl-DNA (véase la Sección 5.1) un registro es malicioso si ocurre al menos una de las siguientes acciones.

   Modificación en el dato P2P[1] en:

   i Lista de pares autorizados ACL

   ii Condiciones de uso

   iii Identificador del propietario

   Se identifica en los registros de la bitácora:

   iv El rechazo de una solicitud a un par autorizado

   v La entrega de datos a pares no autorizados

Para representar un registro malicioso se crea un vector binario de cinco entradas, con distribución uniforme. Cada posición del vector corresponde al número de acción maliciosa (i, ii, iii, iv y v) que se puede o no presentar, es decir, la primera entrada del vector corresponde a la ACL, la segunda a las condiciones de uso, la tercera al identificador del propietario, la cuarta al rechazo de solicitud y la quinta a la entrega no autorizada. Por ejemplo, el vector [1, 1, 0, 1, 0] representa un registro malicioso en el cuál se modificó la ACL, las condiciones de uso y se rechazó la entrega a un par autorizado. No se permite el vector [0, 0, 0, 0, 0].

Se eligió el 70 % (735) de las solicitudes realizadas en la etapa 2, con una distribución uniforme y por cada elección se crea un vector binario. Posteriormente, se realizaron las modificaciones en el dato P2P y/o en la solicitud registrada en la bitácora del servidor, tal como

[1] No se considera ataque a la integridad del dato cifrado puesto que solo se implementó la publicación por referencia.

lo especifica el vector binario para inyectar el registro malicioso.

En la Sección 4.3.5 se analizó el número de mensajes que el algoritmo necesita para auditar a N servidores, en el mejor caso sólo se utilizan $log(N)$ mensajes y el peor caso es cuando se envían $O(N(log(N)+1))$ mensajes. Se realizaron tres rondas del caso de prueba (I, II y III), la primera y la segunda ronda para el mejor y peor caso en el envío de mensajes de auditoría, respectivamente; en la tercer ronda se pemitió que la función Hash dispersara las publicaciones. Cada ronda se dividió en cinco fases para agregar Registros Maliciosos (RM) al sistema, en la primera fase se agregó el 20 % de RM, en la segunda etapa se agregó el 40 % y así sucesivamente, hasta alcanzar el 100 % de RM inyectados en el sistema. Finalmente, para cada ronda del caso de prueba se midió el tiempo que requiere la auditoría para encontrar los RM que se agregaron en las cinco fases; se realizaron 25 mediciones en cada fase con un intervalo de confianza al 95 % de nivel de confianza.

Para implementar el mejor caso en el envío de mensajes de auditoría, todas las publicaciones de la ronda I del caso de prueba se almacenaron en un solo servidor. La gráfica de la Figura 5.9 muestra los resultados obtenidos para la ronda I.

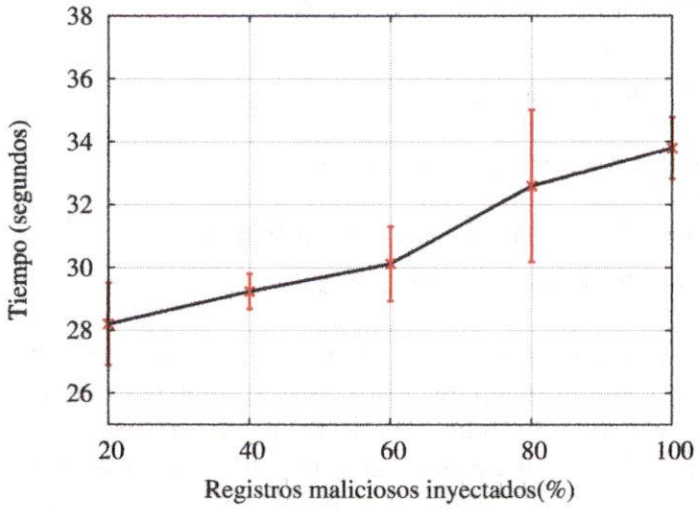

Figura 5.9: Gráfica para el mejor caso en el envío de mensajes de auditoría.

Para el peor caso, se decidió que los servidores con identificador entre el rango [2, 27] almacenan 56 publicaciones y el servidor con el identificador 28 almacena 44 publicaciones (lo cual corresponde a las 1500 publicaciones del caso de prueba). La gráfica de la Figura 5.10 corresponde a la ronda II.

La gráfica de la Figura 5.11 corresponde a la ronda III, se observó que la función Hash almacenó las publicaciones en doce servidores.

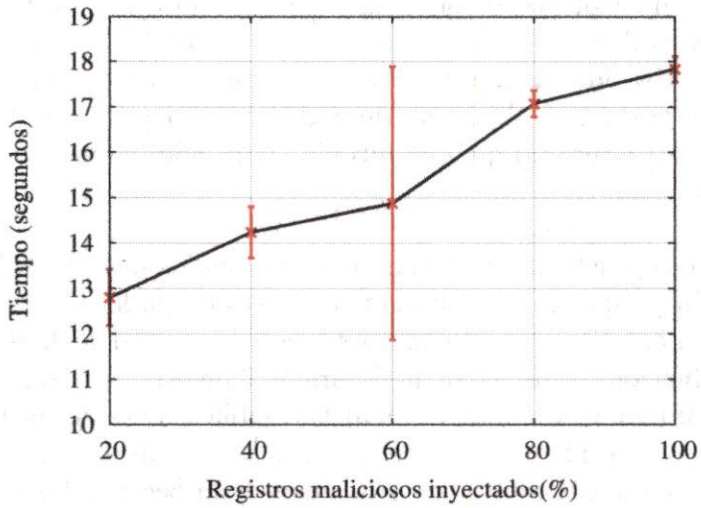

Figura 5.10: Gráfica para el peor caso en el envío de mensajes de auditoría.

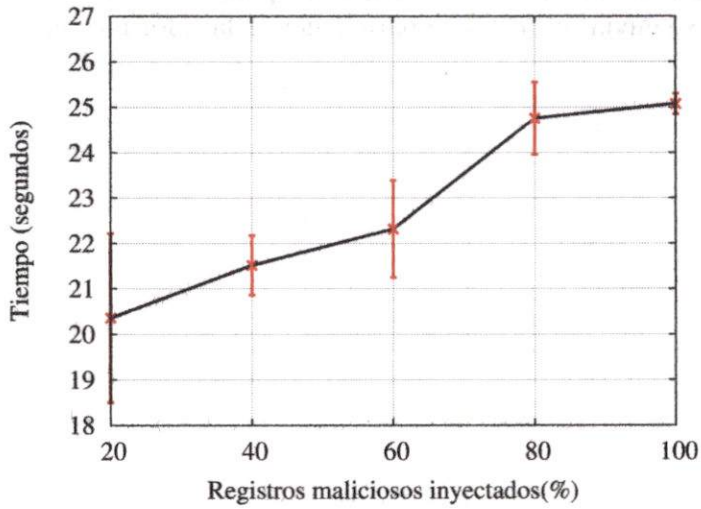

Figura 5.11: Gráfica del envío de mensajes de auditoría a los servidores que eligió la función Hash.

5.5. Resumen

En este capítulo se utilizó el sistema integrado SPLAY [20] para el desarrollo, la implementación y la evaluación de un prototipo del servicio de auditoría. Los elementos del prototipo que se desarrollaron fueron el componente Audit Manager, el algoritmo de auditoría, el servicio de privacidad PriServ (sólo los componentes que requiere la auditoría), un caso de prueba para evaluar el algoritmo de auditoría y el protocolo básico de Chord [17] (como sistema P2P estructurado) que permitió la emulación de un sistema P2P de 28 pares (en un *cluster* local de cinco nodos).

Se aplicó un caso de prueba para evaluar el tiempo que requiere la auditoría de servidores, se tomó esta decisión porque no se contaban con datos de publicaciones y solicitudes de un sistema P2P que utilice a Priserv [4] como servicio de privacidad de datos. Al analizar los resultados de la evaluación, la gráfica de la Figura 5.10 muestra que el algoritmo de auditoría alcanza un tiempo mínimo (17.84 s), cuando las publicaciones se encuentran distribuidas equitativamente en los servidores del sistema P2P. Los resultados de medir el tiempo que requiere la auditoría son consistentes con el análisis de la Sección 4.3.5, porque el orden del algoritmo (en términos del tiempo que requiere la auditoría de servidores) depende de la cantidad de publicaciones que un propietario tenga en el sistema y de los registros que haya en las bitácoras de los servidores. Lo cual es una ventaja para los sistemas P2P basados en DHT's, pues algunas funciones Hash (como el caso de la función SHA-1) consideran balance de carga [17]. En cambio, si en un servidor se almacenan todas las publicaciones de un propietario, la gráfica de la Figura 5.9 muestra los peores resultados (33.8 s valor máximo), a pesar de que sólo se envía un mensaje para solicitar la información de auditoría.

Capítulo 6
Conclusiones y trabajo futuro

El uso indistinto de los datos privados almacenados en los sistemas informáticos se debe evitar, pues sus propietarios pueden verse severamente afectados (e. g., en el caso de la divulgación de un historial médico). Cada vez más países, entre ellos México, cuentan con leyes para la protección de datos personales y, en general las tecnologías de la información no pueden ser ajenas a estas leyes.

Los sistemas informáticos deben considerar un modelo de privacidad en la gestión de datos para permitir que los propietarios de la información especifiquen por quién, cuándo, cómo y para qué, pueden ser usados sus datos. El presente trabajo de investigación se centró en la auditoría de la privacidad, es decir, en el problema de verificar el cumplimiento de las políticas de privacidad en un sistema P2P.

Los objetivos particulares del proyecto, a saber: estudiar y analizar las técnicas existentes de auditoría de datos y proponer e implementar un algoritmo de auditoría de datos de acuerdo a políticas de privacidad individuales, se cumplieron satisfactoriamente para un escenario preciso.

En primer término, se revisó la literatura relativa a la auditoría de datos en sistemas distribuidos y en bases de datos. Esta revisión nos permitió identificar las técnicas de auditoría desarrolladas, su ámbito de aplicación y los elementos a incluir en un sistema para poder auditarlo. Enseguida, teniendo como contexto específico un sistema P2P que cuenta con un servicio de privacidad de datos (PriServ), se identificaron los escenarios de violación potencial de la privacidad y se acotó el alcance de este trabajo a auditar los servidores para detectar: la entrega de datos a pares no autorizados, la negación del acceso a los datos a pares autorizados y la modificación de la información almacenada.

En segundo término, se realizó el diseño de un servicio de auditoría de privacidad para sistemas P2P. El servicio puede ser utilizado tanto para sistemas P2P estructurados como no estructurados, pues no depende de las implementaciones ni del almacenamiento distribuido

ni tampoco del tipo de red super-puesta. Sin embargo, depende fuertemente del modelo de privacidad utilizado en PriServ. Además se propuso un algoritmo para auditar a los servidores de un sistema P2P con base en PriServ y permite que los propietarios identifiquen violaciones a sus políticas de privacidad.

Finalmente, se realizó la emulación de un sistema P2P en donde se implementaron los componentes de PriServ involucrados en la publicación por referencia y solicitud de datos y el servicio de auditoría. Durante la emulación se ejecutó un escenario de prueba para publicar, solicitar e inyectar registros maliciosos y se midió el tiempo requerido para el mejor caso, el peor caso y un caso intermedio. Los resultados obtenidos son consistentes con el orden del algoritmo de auditoría (en términos del tiempo que se requiere), porque el orden depende de la cantidad de publicaciones que un propietario tenga en el sistema y de los registros que haya en las bitácoras de los servidores.

Enseguida se describe el trabajo futuro identificado hasta el momento:

- Reducir el tiempo para obtener y evaluar la información de auditoría que se obtiene de los pares. Una posible solución es aplicar un esquema de índices para ordenar los registros de las bitácoras.
- Trabajar en los escenarios de violación de privacidad identificados pero no atendidos en este trabajo, es decir, detectar cuando:
  a) algún par se apropia de datos ajenos. En este caso, se propone explorar las técnicas de marcas de agua en sistemas P2P [19].
  b) diferentes tipos de consultas divulgan datos privados. Para este caso, la propuesta es un algoritmo de auditoría de consultas SQL similar a las técnicas en bases de datos [5].
  c) algún par no cumple con las condiciones de uso de los datos.
- Explorar la aplicación de técnicas de minería de datos para la realización de la auditoría.

Referencias

[1] R. Agrawal, J. Kiernan, R. Srikant, y Y. Xu. "Hippocratic Databases", en *Proceedings of the 28th international Conference on Very Large Data Bases*, 2002, págs. 143-154.

[2] A. I. Antón, E. Bertino, N. Li, y T. Yu "A Roadmap for Comprehensive Online Privacy Policy Management". *Commun. ACM*, vol. 50, 7, págs. 109-116, Julio 2007.

[3] W3C. Platform for Privacy Preferences (P3P) project; www.w3.org/P3P/

[4] M. Jawad, P. Serrano-Alvarado, P. Valduriez, y S. Drapeau. "A Data Privacy Service for Structured P2P Systems", en *Mexican International Conference on Computer Science (ENC09)*, 2009, págs. 45-56.

[5] S. U. Nabar, K. Kenthapadi, N. Mishra, y R. Motwani. "A Survey of Query Auditing Techniques for Data Privacy". *Privacy Preserving Data Mining.* E. A. K. Elmagarmid, C. C. Aggarwal y P. S. Yu. Springer, vol. 34, págs. 415-431, 2008.

[6] C. J. Date. *Introducción a los Sistemas de Bases de Datos.* Wilmington, Delaware, E. U. A.: Addison-Wesley Iberoamericana, 1993, págs. 5-8.

[7] R. Agrawal, R. Bayardo, C. Faloutsos, J. Kiernan, R. Rantzau, y R. Srikant. "Auditing Compliance with a Hippocratic Database", en *Proceedings of the 30th international Conference on Very Large Data Bases*, 2004, págs. 516-527.

[8] R. Motwani, S. U. Nabar y D. Thomas. "Auditing SQL Queries", en *Proceedings of the 2008 IEEE 24th international Conference on Data Engineering*, 2008, págs. 287-296.

[9] W. Lu y G. Miklau. "AuditGuard: a System for Database Auditing under Retention Restrictions", en *Proceedings of the VLDB Endowment*, 2008, págs. 1484-1487.

[10] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Upper Saddle River, NJ.: Pearson Education, 2005, pág. 37.

[11] C. Ringelstein y S. Staab. "DIALOG: Distributed Auditing Logs", en *IEEE International Conference on Web Services*, 2009, págs. 429-436.

[12] R. Schollmeier. "A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications", en *Proceedings of the 1st International Conference on Peer-to-Peer Computing*, 2001, págs. 101-103.

[13] M. Ham y G. Agha. "ARA: A Robust Audit to Prevent Free-Riding in P2P Networks", en *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, 2005, págs. 125-132.

[14] A. Altay y P. Ning. "BAF: An Efficient Publicly Verifiable Secure Audit Logging Scheme for Distributed Systems", en *Proceedings of the 2009 Annual Computer Security Applications Conference*, págs. 219-228, 2009.

[15] M. Jawad. "Data Privaci in P2P Sysmtems". Phd Thesis, Université de Nantes, Francia, 2011.

[16] S. D. Kamvar, M. T. Schlosser y H. Garcia-Molina. "The Eigentrust Algorithm for Reputation Management in P2P networks", en *ACM World Wide Web Conference* 2003, págs. 640-651.

[17] I. Stoica, R. Morris, D. Liben-Nowwel, D. Karger, M. Kaashoek, F. Dabek y H.Balakrishnan. "Chord: a Scalable Peer-to-Peer Lookup Protocol for Internet Applications", *IEEE/ACMTrans. Netw.*, vol. 11, 1, págs. 17-32, Febrero 2003.

[18] I. T. Rowstron y P. Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems", en *ACM/IFIP/USENIX Middleware Conference*, 2001, págs. 329-350.

[19] D. Tsolis, S. Sioutas y T. Papatheodorou. "Digital Watermarking in Peer-to-Peer Networks", en *Proceedings of the 16th international conference on Digital Signal Processing*, 2009, págs. 1086-1090.

[20] L. Leonini, É. Rivière y P. Felber. "SPLAY: Distributed Systems Evaluation Made Simple (or how to turn ideas into live systems in a breeze)", en *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, 2009, págs. 185-198.

[21] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink y M. Wawrzoniak. "Operating system support for planetary-scale network services" en *In Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, 2004, págs. 19-19.

[22] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kosti, J. Chase y D. Becker. "Scalability and Accuracy in a Large-Scale Network Emulator". *SIGOPS Oper. Syst.*, rev. 36, págs. 271-284. Diciembre 2002.

[23] R. Lerusalimschy, L. Defigueiredo and W. Celes. "The Implementation of Lua 5.0", *J. of Univ. Comp. Sc.*, vol. 11, 7, págs. 1159-1176, 2005.

[24] L. Leonini. "SPLAY Distributed Aplications Made Simple". Internet: http://www.splay-project.org/, 2010, [Mayo 30 2011].

[25] "SQLite Small. Fast. Reliable. Choose any three". Internet: http://www.sqlite.org [Julio 1 2011]

[26] D. Currie y T. Dionizio. "LuaForge: LuaSQLite". Internet: http://www.luaforge.net/projects/luasqlite, Enero 12 2011 [Julio 6 2011]