

Proyecto Final de carrera

Seguridad en Cloud Computing

Curcio Pablo
Fermani Mauro



UNIVERSIDAD NACIONAL DEL SUR
DEPARTAMENTO DE CIENCIAS E INGENIERÍA DE LA COMPUTACIÓN

índice

1. Introducción
2. ¿Qué es Cloud Computing?
 - a. Atributos del Cloud Computing
 - b. Tecnologías relevantes
 - c. Modelos de distribución de servicios de Cloud Computing
 - d. Modelos de despliegue de Cloud Computing
3. Seguridad en Cloud Computing
 - a. Beneficios de seguridad
 - b. Riesgos de seguridad
 - c. Vulnerabilidades
4. Implementación de un software como servicio
 - a. Tecnologías utilizadas
 - i. Git
 - ii. Lenguaje de programación Ruby
 - iii. Framework web Ruby on Rails
 - iv. Librerías Qt
 - v. Truecrypt
 - b. Funcionamiento del programa
 - c. Ventajas y desventajas
5. Conclusión
6. Anexos

Introducción

Este trabajo final de carrera de grado es una extensión al trabajo final de la materia Seguridad en Sistemas, en el cual se trataron los conceptos de cloud computing, nombrando sus ventajas y desventajas, sus riesgos y vulnerabilidades.

En este informe retoma lo mencionado anteriormente como marco teórico del trabajo final y detalla la implementación de un sistema de software representativo de cloud computing.

Estructura del informe

El informe comienza con conceptos básicos de Cloud Computing, las características que dieron lugar al inicio de este tipo de sistemas y los diferentes modelos que existen.

Luego sigue con una sección donde se tratan temas relacionados a la seguridad de los sistemas de Cloud Computing. Nombra los beneficios y riesgos que conlleva a una empresa tener sus sistemas informáticos en una estructura de este tipo, y las vulnerabilidades a las que queda expuesta.

La siguiente sección trata la implementación de un sistema de software llamado Rubox, cuya funcionalidad principal es compartir archivos entre diferentes usuarios, además de brindar un nivel de seguridad de encriptación de los datos del usuario y versionar los archivos para poder obtener copias viejas de lo hecho. Además incluye las tecnologías utilizadas para llevar a cabo el desarrollo junto a sus ventajas y desventajas.

El informe cierra con una conclusión sobre lo tratado en las secciones anteriores.

¿Qué es Cloud Computing?

Cloud Computing es un modelo de servicio por demanda, para la provisión de tecnología de información. Utiliza como soporte tecnologías de virtualización y de cómputo distribuido.

En su libro “The Big Switch” Nicholas Carr plantea una analogía entre el surgimiento del Cloud Computing y la red eléctrica. En un principio las organizaciones debían proveerse de su propia energía eléctrica. Luego, con la llegada de la red eléctrica, solo necesitaban conectarse a la misma para abastecerse, pudiendo concentrarse en su actividad principal y adquirir la energía como un servicio. Carr argumenta que el Cloud Computing es en realidad el principio de un cambio similar, donde las organizaciones pasan de generar y mantener sus propios sistemas de cómputo a adquirirlos como servicio a través de esta tecnología.

Atributos del Cloud Computing

El Cloud Computing se puede definir en base a los siguientes atributos:

- Alta escalabilidad: significa que puede adaptarse a una demanda creciente de recursos de cómputo que no está acotada, por ejemplo ancho de banda o almacenamiento. En el modelo tradicional, escalar el sistema puede implicar la necesidad de agregar hardware y soporte de tecnología de la información.
- Flexibilidad: permite a los usuarios incrementar o disminuir rápidamente los recursos de cómputo, pudiendo manejar picos de procesamiento, es decir, adaptar la cantidad de recursos disponibles a la demanda.



- Recursos compartidos: mediante virtualización, se abstraen los recursos físicos compartidos en recursos lógicos dedicados, a nivel de red, host y aplicación.
- Pay as you go: se paga en función de qué recursos se utilizaron y cuánto tiempo. Las aplicaciones tradicionales están basadas en un modelo con

grandes costos de licencia que no están directamente relacionados con la utilización que se le da a la aplicación.

- Autoabastecimiento de recursos: los usuarios pueden modificar su capacidad de procesamiento, almacenamiento, ancho de banda y adquirir software por sí mismos.

Tecnologías relevantes

El Cloud Computing en realidad no es una tecnología nueva, sino una combinación de varias preexistentes. Estas tecnologías que evolucionaron individualmente se han unido para soportar el concepto de Cloud Computing. Algunas de ellas son:

- Dispositivos de acceso a la nube: cada vez existen más dispositivos con capacidad de acceder a internet.
- Browsers y clientes finos: los usuarios pueden acceder a aplicaciones e información desde cualquier lugar capaz de cargar un browser. Se tiene una mayor disponibilidad de servicios que pueden ser accedidos a través de un cliente fino, en lugar de instalar una aplicación dedicada (cliente grueso) para acceder al mismo.
- Conexiones de alta velocidad: hoy en día se tiene gran disponibilidad de ancho de banda, que permite la interconexión de diferentes componentes y la creación de nuevos modelos de cómputo.
- Data centers: Los servicios basados en Cloud Computing necesitan gran capacidad de cómputo por lo que se hostean en data centers, que pueden abarcar varias ubicaciones interconectadas, proveyendo capacidad de distribución de cómputo y entrega de servicios.
- Dispositivos de almacenamiento: las nuevas tecnologías han reducido los costos y aumentado la flexibilidad de los dispositivos de almacenamiento. Las redes de área de almacenamiento (SANs) permiten la integración de varios dispositivos y asignar espacio por demanda.
- Virtualización: posibilita la creación de un pool escalable de recursos accesibles a través de métodos estandarizados, y la abstracción de los recursos físicos en lógicos ofreciendo una vista de recursos dedicados a los usuarios de la plataforma.
- Interfaces de programación de aplicaciones (APIs): Las APIs facilitan a los usuarios el autoabastecimiento y manejo de los recursos y servicios del cloud, enmascarando la complejidad.

Modelos de distribución de servicios de Cloud Computing

Los modelos de distribución de servicios de Cloud Computing, comúnmente referidos como SPI (Software - Plataform - Infraestructure) se clasifican en las siguientes categorías:

- Software as a Service (SaaS): Aplicaciones en la nube con tarifa mensual. Por ejemplo Google Docs, MobileMe, Zoho.
- Plataform as a Service (PaaS): Una plataforma que habilita a los desarrolladores a escribir aplicaciones que corran sobre la nube. Por ejemplo Microsoft Azure, Google App Engine, Force.com.
- Infraestructure as a Service (IaaS): Una infraestructura de cómputo compartida, redundante y escalable accesible a través de tecnologías de internet. Por ejemplo: Amazon EC2, Sun's cloud services, etc.

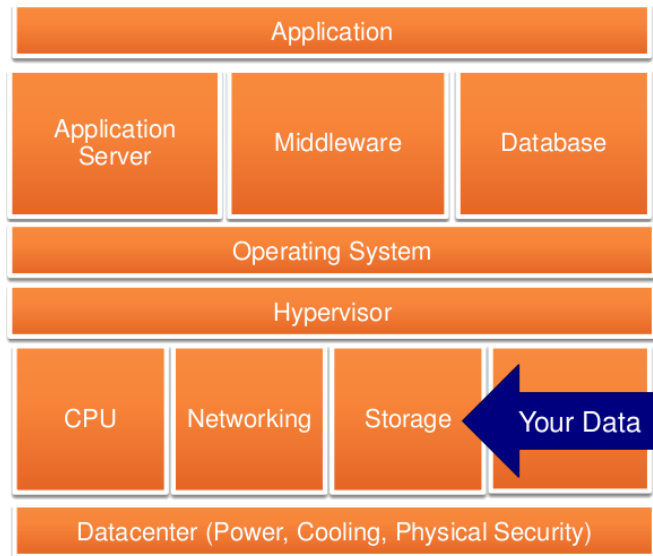
Software as a Service

La forma tradicional de adquirir software consistía en que los clientes compraran las licencias de las aplicaciones, servicios de soporte o mantenimiento y provean además el hardware donde las aplicaciones se ejecutarían.

En el modelo SaaS, el cliente paga por utilizar las aplicaciones del proveedor, que se ejecutan en la infraestructura del cloud. Generalmente el servicio comprado es completo en cuanto a hardware, software y soporte. Desde el punto de vista de la arquitectura, en el modelo SaaS la infraestructura de hardware es compartida por diferentes clientes, pero a nivel lógico es única para cada uno.

Entre las características ofrecidas por el SaaS se encuentran:

- El hosting y manejo de las aplicaciones puede ser realizado por terceros, reduciendo costos de licencias, servidores y otra infraestructura y personal necesarios para realizar el hosting en forma interna.
- El vendedor del software tiene mayor control sobre la copia y distribución del mismo, y se facilitan las actualizaciones.
- El acceso a la aplicación SaaS se realiza a través de un browser, aunque pueden existir casos en que los vendedores provean su propia interface para soportar características específicas de la aplicación.
- Generalmente es posible utilizar la infraestructura de acceso a internet existente, sin requerir hardware adicional, aunque puede ser necesario cambiar reglas en el firewall y realizar configuraciones.



En el modelo de servicio SaaS el cliente proporciona sus datos (azul) mientras que el proveedor es responsable del resto de los componentes (naranja).

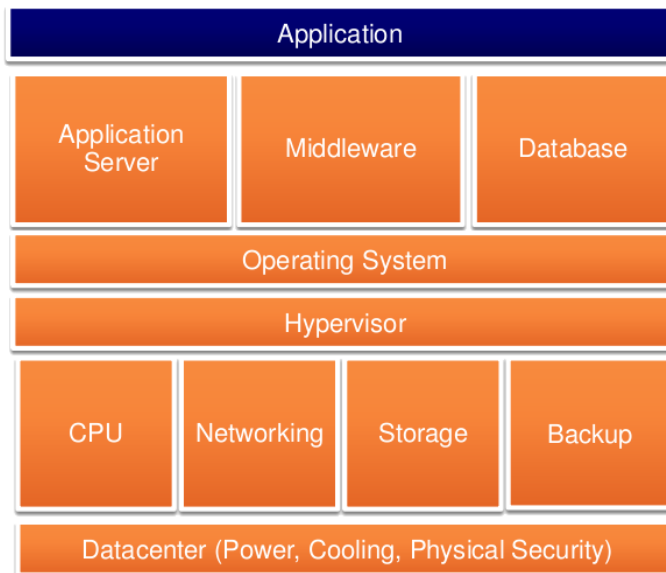
Plataform as a Service

Es una variante del SaaS en la que el proveedor ofrece un entorno de desarrollo de aplicaciones como servicio, que es accedido por los desarrolladores a través de un browser. El entorno está hosteado en la infraestructura del cloud.

Las aplicaciones creadas por los usuarios pueden ser accedidas por sus clientes a través de la plataforma provista. El PaaS habilita a los desarrolladores a crear aplicaciones sin instalar herramientas en su computadora y a desplegarlas sin poseer conocimientos avanzados de administración de sistemas, evitando el costo y la complejidad de comprar servidores e instalarlos. El bajo costo inicial y las facilidades de desarrollo y despliegue permiten una rápida propagación de las aplicaciones.

Algunas de las características que debería ofrecer una alternativa PaaS son:

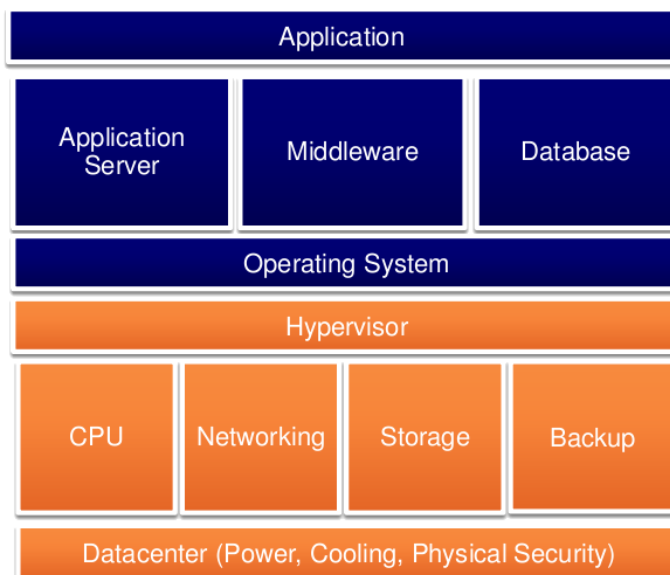
- Escalabilidad, confiabilidad, seguridad y multi-tenancy (capacidad de una aplicación de particionar el estado y los datos automáticamente para servir a un número arbitrario de usuarios) sin requerir desarrollo adicional, configuración u otros costos por parte de los usuarios.
- Integración con bases de datos y servicios web externos.
- Monitoreo de la aplicación y de la actividad de los usuarios para ayudar a los desarrolladores a hacer mejoras.
- Permitir la colaboración entre los desarrolladores durante el ciclo de vida del software y al mismo tiempo mantener la seguridad del código fuente y la propiedad intelectual asociada.
- Soporte de facturación de acuerdo al modelo pay-as-you-go.



En el modelo de servicio PaaS el proveedor se encarga de la infraestructura subyacente a la aplicación que desarrolla el cliente.

Infrastructure as a Service

En el modelo de servicio IaaS se provee la infraestructura necesaria para que el cliente desarrolle y ejecute software arbitrario, incluyendo procesamiento, almacenamiento, servicios de red y otros recursos de cómputos fundamentales. El cliente no administra ni controla la infraestructura del cloud subyacente al hypervisor, pero tiene control sobre los sistemas operativos, almacenamiento, las aplicaciones y algunas características de los componentes de red (por ejemplo, firewall de los sistemas operativos).



En el modelo de servicio IaaS el proveedor gestiona la infraestructura del Cloud necesaria para que el cliente desarrolle y ejecute software arbitrario.

Las principales características de un sistema típico de IaaS incluyen:

- Escalabilidad: tiene la capacidad de escalar la infraestructura, por ejemplo memoria y almacenamiento, basado en los requerimientos de uso.
- Pay as you go: capacidad de adquirir la cantidad exacta de infraestructura requerida en un momento dado.
- Acceso a mejores soluciones de tecnología a una fracción de su costo.

Modelos de despliegue de Cloud Computing

Independientemente del modelo de servicio utilizado (SaaS, PaaS, IaaS) existen cuatro modelos de implementación de Cloud Computing.

Nubes públicas o externas

Las nubes públicas son hospedadas, operadas y manejadas por terceros desde uno o más Data Centers. Los servicios son ofrecidos a múltiples clientes sobre una infraestructura compartida.

En una nube pública, la gestión de la seguridad y las operaciones del día a día recaen en los proveedores del servicio. Por lo tanto, el cliente tiene un bajo grado de control y supervisión de los aspectos de seguridad física y lógica de una nube pública.



Nubes privadas o internas

En las nubes privadas la infraestructura de red, cómputo y almacenamiento está dedicada a una sola organización y no es compartida con ninguna otra. Solo es accesible desde la red privada.

La organización tiene el control sobre la infraestructura lo que favorece la seguridad de los datos, el gobierno corporativo y la confiabilidad. Sin embargo, no se capitalizan los beneficios del pay-as-you-go ya que se debe pagar el costo de la infraestructura total para soportar los picos de uso del sistema.

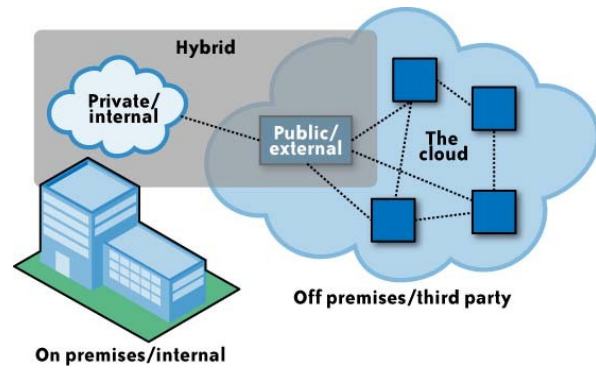
En virtud de este modelo de gobierno directo, un cliente de una nube privada tiene un alto grado de control y supervisión sobre los aspectos lógicos y físicos de la seguridad de la infraestructura, tanto del hypervisor como de los sistemas operativos virtualizados. Debido a esto, es más fácil para el cliente cumplir con los estándares de seguridad establecidos por la empresa, con las políticas y regulaciones.

Nubes híbridas

Las nubes híbridas son una combinación de las públicas y las privadas, donde los datos y aplicaciones críticas se mantienen en la nube privada mientras que el resto en la pública.

Nubes de comunidad

La infraestructura del cloud es compartida por varias organizaciones y soporta una comunidad específica con intereses comunes (objetivos, requerimientos de seguridad, regulaciones, etc). Puede ser manejada por las organizaciones o por terceros.



Seguridad en Cloud Computing

Los controles de seguridad para Cloud Computing son en general similares a los aplicados en otros ambientes de IT. Sin embargo los riesgos presentes pueden ser diferentes debido a las tecnologías involucradas y a los modelos de servicio y de despliegue.

Los controles se implementan en las distintas capas del modelo de servicio, que abarcan la seguridad física, la seguridad de red, del sistema y de las aplicaciones. Dependiendo del modelo de servicio varían las responsabilidades de seguridad que tienen el cliente y el proveedor. En el modelo SaaS el cliente sólo es responsable de utilizar la aplicación en forma segura, mientras que el proveedor es quien debe garantizar la seguridad desde la capa física hasta la de aplicación. En cambio, en el modelo IaaS, el proveedor asegura desde la capa física hasta el hypervisor, dejándole al usuario la responsabilidad de las capas superiores .

Beneficios de seguridad

A continuación se presentarán los principales beneficios de seguridad que ofrece el modelo de Cloud Computing.

Seguridad y beneficios de escala

Las medidas de seguridad son más económicas cuando se implementan a gran escala. Entonces, la misma inversión en seguridad permite comprar una mejor protección. Esto incluye una variedad de defensas como filtros, manejo de parches, instancias de máquinas virtuales e hipervisores reforzados, recursos humanos y su gestión, redundancia de hardware y software, mecanismos de autenticación fuertes, entre otras. Otros beneficios de escala son:

- Los proveedores de Cloud Computing tienen generalmente los recursos económicos para replicar contenido en distintas ubicaciones incrementando la redundancia, lo que permite una mayor tolerancia y recuperación ante fallas.
- La posibilidad de almacenar, procesar y entregar los datos más cerca de los extremos de la red incrementa la confiabilidad y la calidad del servicio, y reduce la propagación de los problemas de red.
- Se tiene mayor capacidad para responder a incidentes en forma eficaz y eficiente.

- Los proveedores de Cloud Computing pueden afrontar el costo de contratar personal altamente capacitado para tratar cuestiones de seguridad específicas.

Seguridad como un diferencial en el mercado

La seguridad es una prioridad para muchos de los clientes de Cloud Computing. La reputación en relación a cuestiones de confidencialidad, integridad, tolerancia a fallas y los servicios de seguridad ofrecidos hacen la diferencia al momento de elegir un proveedor. Esto incentiva a los proveedores a competir para brindar mejor seguridad.

Interfaz estándar para servicios de gestión de seguridad

Los servicios de gestión de seguridad (MSS) incluyen asesoramiento, respuesta a incidentes, manejo del perímetro remoto, monitoreo y penetration tests periódicos.

Los proveedores de Cloud Computing pueden ofrecer una interfaz abierta y estandarizada a los proveedores de MSS que les dan servicio a sus clientes, facilitando y reduciendo los costos de cambiar de proveedor de MSS.

Escalamiento de recursos

Entre los recursos que se pueden escalar rápidamente y por demanda se encuentran el almacenamiento, los ciclos de CPU, la memoria, los requerimientos de servicio web e instancias de máquinas virtuales.

Un proveedor de Cloud Computing tiene la capacidad de reasignar recursos dinámicamente, permitiendo dedicar más recursos a soportar medidas defensivas, como control de tráfico, filtros o mecanismos de cifrado, en el momento en que se produce o es probable que ocurra un ataque. Si esto se combina con métodos apropiados de optimización de recursos, el proveedor de Cloud Computing puede ser capaz de limitar el efecto que algunos ataques provocan en la disponibilidad de los recursos utilizados por los servicios legítimos, como así también limitar el efecto de asignar más recursos a los mecanismos defensivos en el momento de un ataque.

Auditoría y recolección de evidencia

El modelo de Cloud Computing soporta la clonación de imágenes u otros componentes virtuales para realizar un análisis forense fuera de línea en caso de que se sospeche que la seguridad fue comprometida, lo que permite disminuir el tiempo en que el sistema está fuera de servicio para realizar el análisis.

Pueden crearse múltiples copias y paralelizar las actividades de análisis para reducir el tiempo de investigación, lo que aumenta la probabilidad de rastrear a los atacantes y posibilita corregir antes las vulnerabilidades, llevando a un menor tiempo de exposición.

A su vez, el modelo provee un almacenamiento más flexible y rentable, lo que permite guardar logs más compresibles sin comprometer la performance y simplifica la realización de ajustes para satisfacer los requerimientos de logs de auditoría futuros. Esto hace más eficiente el proceso de identificación de los incidentes de seguridad a medida que ocurren.

Updates y defaults

Las imágenes de máquinas virtuales y los módulos de software usados por los clientes pueden estar previamente reforzados, configurados y actualizados para mejorar la seguridad. También es posible crear imágenes de la infraestructura virtual y compararlas regularmente con un patrón para detectar irregularidades (por ejemplo, para asegurar que las reglas del firewall no han cambiado). Además, contar con una plataforma homogénea permite desplegar las actualizaciones con mayor rapidez, reduciendo el tiempo de exposición. En el caso de los modelos SaaS y PaaS, es probable que las aplicaciones hayan sido reforzadas para ejecutarse fuera del entorno empresarial, haciéndolas más robustas que sus equivalentes en el modelo de software tradicional.

Auditorías y SLAs obligan a un mejor manejo de riesgos

La necesidad de cuantificar las penalidades para varios escenarios de riesgo en SLAs (acuerdos de nivel de servicio) y el posible impacto de las brechas de seguridad en la reputación, motiva a desarrollar procedimientos internos de auditoría y evaluación de riesgos más rigurosos. Además, las frecuentes auditorías impuestas sobre los proveedores de Cloud Computing tienden a exponer riesgos que de otra manera podrían no haber sido descubiertos, con la consecuente mejora en la seguridad.

Beneficios de concentración de recursos

Aunque tiene desventajas obvias, la concentración de recursos facilita el establecimiento de un perímetro y el control del acceso físico a los recursos. También hace más sencilla y económica la aplicación de procedimientos relacionados con la seguridad como el control sobre el manejo de datos, actualizaciones, mantenimiento e incidentes.

Riesgos de seguridad

A continuación se describen los principales riesgos que se presentan en el modelo de Cloud Computing.

Pérdida de gobierno

En el uso de infraestructuras cloud, el cliente cede necesariamente al proveedor un número de cuestiones que pueden afectar la seguridad. Por ejemplo,

el proveedor le puede impedir al usuario que realice escaneo de puertos, evaluación de vulnerabilidades y penetration testing.

Los acuerdos de nivel de servicio (SLA) pueden no obligar al proveedor a proporcionar el nivel de seguridad requerido por el cliente, dejando una brecha en las defensas de seguridad. Otra cuestión que lleva a la pérdida de gobierno ocurre cuando los terceros contratados por el proveedor influyen en la seguridad de cliente y no brindan las mismas garantías que el proveedor.

La pérdida de gobierno y control podría comprometer los requerimientos de seguridad, la confidencialidad, integridad y disponibilidad de datos, y causar un deterioro del rendimiento y calidad del servicio. Esto conlleva un impacto considerable sobre la estrategia de la organización y, por consiguiente, en la capacidad para cumplir con sus objetivos.

VULNERABILIDADES
ROLES Y RESPONSABILIDADES POCO CLAROS
CUMPLIMIENTO DEFICIENTE DE LAS DEFINICIONES DE LOS ROLES
ATRIBUCIÓN ERRÓNEA DE RESPONSABILIDADES AL PROVEEDOR
AUDITORÍAS O CERTIFICACIONES NO DISPONIBLES POR PARTE DEL PROVEEDOR
DEPENDENCIA DEL PROVEEDOR HACIA TERCEROS
FALTA DE SOLUCIONES Y TECNOLOGÍAS ESTANDARIZADAS
FALTA DE INFORMACIÓN SOBRE LA UBICACIÓN DE DATOS
RESTRICCIONES EN LOS PROCESOS DE EVALUACIÓN DE VULNERABILIDADES
FALTA DE CLARIDAD O COMPLETITUD DE LOS TÉRMINOS DE USO

Lock-In

Si la portabilidad de los datos y servicios es baja, es difícil para el cliente cambiar de proveedor o volver a un ambiente de cómputo tradicional, lo que introduce una dependencia del cliente hacia el proveedor. Si el proveedor quiebra y el costo de realizar la migración de las aplicaciones y el contenido a otro entorno es muy grande, o no se dispone del tiempo suficiente para realizarla, el cliente se vería seriamente afectado.

La adquisición del proveedor de Cloud Computing por parte de otra firma puede tener un efecto similar, debido al posible cambio en las políticas de servicio y las condiciones que no estén reguladas en contratos.

Los proveedores pueden tener un incentivo para no ofrecer buena portabilidad, aunque ofrecerla también podría significar una ventaja competitiva.

El lock-in puede manifestarse de diferentes maneras de acuerdo al modelo de servicio de Cloud Computing en cuestión:

- En el modelo SaaS, los datos de los clientes suelen estar almacenados en un esquema de base de datos que varía de proveedor a proveedor. Generalmente se dispone de funciones en la API que permiten leer registros de datos. Con esas funciones, en caso que el proveedor no ofrezca facilidades adicionales de exportación de datos, el cliente necesitará crear sus propias rutinas de exportación, que lean y transformen sus datos a una representación adecuada para importarlos a otro entorno. El cliente podría pagarle al proveedor nuevo para asistirlo en esta tarea.

El cambio de proveedor también impacta en los usuarios finales de las aplicaciones SaaS, que pueden tener que ser recapitados, con el consecuente costo para el cliente, especialmente si los usuarios son numerosos. Además, si se contaba con programas que interactuaban directamente con la API del proveedor anterior, por ejemplo para integrar diferentes aplicaciones, los mismos deberán ser reescritos para adecuarlos a la API nueva.

- En el modelo PaaS, los clientes desarrollan sus aplicaciones basándose en la API que les ofrece el proveedor. El lock-in se da a nivel de la API, debido a que la misma generalmente varía de proveedor a proveedor. También pueden existir diferencias en los soportes de runtime, modificados y configurados para operar en forma segura en un entorno cloud, quedando a cargo del cliente entender y tomar en cuenta estas diferencias. En el modelo PaaS, al igual que en el SaaS, se puede dar el lock-in a nivel de datos, pero en este caso, es el cliente el responsable de crear las facilidades de exportación de datos apropiadas.
- En el modelo IaaS, el lock-in puede deberse a la incompatibilidad en las tecnologías de virtualización o a la falta de portabilidad de datos.

VULNERABILIDADES
FALTA DE SOLUCIONES Y TECNOLOGÍAS ESTANDARIZADAS
ELECCIÓN INADECUADA DEL PROVEEDOR
FALTA DE CLARIDAD O COMPLETITUD DE LOS TÉRMINOS DE USO

Falla de aislamiento

La abstracción de recursos físicos compartidos en recursos lógicos dedicados mediante virtualización, es una característica fundamental del Cloud Computing. Esta clase de riesgo consiste en la falla de los mecanismos de separación de recursos entre los diferentes usuarios de la infraestructura compartida.

Los ataques al hypervisor y las inyecciones SQL que exponen datos de múltiples clientes son algunos de los ataques que pueden originar fallas de aislamiento.

La probabilidad de ocurrencia de estos incidentes depende del modelo de despliegue en cuestión, siendo menor en el caso de las nubes privadas que en las públicas.

Las consecuencias de una falla de aislamiento pueden incluir pérdida de datos críticos, daños en la reputación e interrupción del servicio tanto de los proveedores de Cloud Computing como de los clientes.

VULNERABILIDADES
VULNERABILIDADES EN EL HYPERVISOR
FALLA EN EL AISLAMIENTO DE LOS RECURSOS
IMPACTO DE ACTIVIDADES DE UN CLIENTE EN LA REPUTACIÓN DE OTRO
CHEQUEOS ILEGALES DE RED Y RESIDENCIA

Problemas de cumplimiento

Algunas organizaciones pueden haber realizado importantes inversiones para obtener determinada certificación, ya sea para lograr una ventaja competitiva o para cumplir con una regulación o estándar de la industria. Estas inversiones pueden verse en riesgo al migrar a la nube si el proveedor de Cloud Computing no es capaz de ofrecer evidencia de que cumple con los requerimientos relevantes o si no permite auditorías por parte del cliente.

Algunas certificaciones no pueden alcanzarse utilizando una infraestructura de nube pública, debiendo emplearse otras alternativas para brindar servicios que necesiten dichas certificaciones.

VULNERABILIDADES
AUDITORÍAS O CERTIFICACIONES NO DISPONIBLES POR PARTE DEL PROVEEDOR
FALTA DE SOLUCIONES Y TECNOLOGÍAS ESTANDARIZADAS
FALTA DE INFORMACIÓN SOBRE LA UBICACIÓN DE DATOS
FALTA DE CLARIDAD O COMPLETITUD DE LOS TÉRMINOS DE USO

Interfaz de administración comprometida

Las interfaces de administración en las nubes públicas son accesibles a través de Internet y permiten controlar más cantidad de recursos que en los ambientes de cómputo tradicionales, por lo que suponen un mayor riesgo, especialmente si se combinan con vulnerabilidades en los browsers y en el acceso remoto. Esto hace necesario utilizar mecanismos de autenticación fuertes, por ejemplo autenticación de dos factores.

Pueden verse afectadas tanto las interfaces de los clientes que controlan cierto número de máquinas virtuales, como las de los proveedores que controlan el sistema completo.

VULNERABILIDADES
VULNERABILIDADES AAA (AUTHENTICATION, AUTHORIZATION AND ACCOUNTING)
ACCESO REMOTO DESDE PUNTOS INSEGUROS
ERRORES DE CONFIGURACIÓN
VULNERABILIDADES EN LOS SISTEMAS OPERATIVOS
VULNERABILIDADES EN LAS APLICACIONES Y MANEJO INADECUADO DE ACTUALIZACIONES

Intercepción y filtrado de datos

Al tratarse de un sistema distribuido, en el modelo de Cloud Computing existe un mayor tráfico de datos que en el sistema tradicional, debido a que se tiene una mayor transferencia de datos entre la infraestructura Cloud y los clientes remotos.

Se deben tener en cuenta ataques como sniffing, spoofing, man-in-the-middle, canales ocultos, replay, entre otros.

Además, en algunos casos los SLAs no ofrecen cláusulas de confidencialidad para garantizar la protección de la información del cliente y no especifican cómo circularán los datos dentro de la nube.

VULNERABILIDADES
VULNERABILIDADES AAA (AUTHENTICATION, AUTHORIZATION AND ACCOUNTING)
VULNERABILIDADES EN LA ENCRIPCIÓN DE DATOS
CHEQUEOS ILEGALES DE RED Y RESIDENCIA
FALTA DE CLARIDAD O COMPLETITUD DE LOS TÉRMINOS DE USO
VULNERABILIDADES EN LAS APLICACIONES Y MANEJO INADECUADO DE ACTUALIZACIONES
BORRADO DE DATOS INSEGURO

Atacante interno

Las actividades de un atacante interno pueden impactar sobre la confidencialidad, integridad y disponibilidad de los datos, los servicios, y por lo tanto, en la reputación de la organización. Esto se puede considerar especialmente importante en el caso de Cloud Computing debido a que estas arquitecturas necesitan ciertos roles que suponen un riesgo muy alto (ejemplo: administrador de sistemas del proveedor).

VULNERABILIDADES
ROLES Y RESPONSABILIDADES POCO CLAROS
CUMPLIMIENTO DEFICIENTE DE LAS DEFINICIONES DE LOS ROLES
PRINCIPIO DE NEED-TO-KNOW NO APLICADO
VULNERABILIDADES AAA (AUTHENTICATION, AUTHORIZATION AND ACCOUNTING)
VULNERABILIDADES EN LOS SISTEMAS OPERATIVOS
PROCEDIMIENTOS DE SEGURIDAD FÍSICA INADECUADOS
VULNERABILIDADES EN LAS APLICACIONES Y MANEJO INADECUADO DE ACTUALIZACIONES

Otros riesgos

Distributed denial of service: se puede llevar a cabo generando un gran flujo de información desde varios puntos de conexión. La forma más común de realizar un DoS es a través de una botnet.

Economic denial of service: los recursos de un cliente de cloud son usados por un atacante y causan un impacto económico en el cliente. Por ejemplo, robo de identidad, el proveedor no limita el uso de los recursos pagos, un atacante utiliza un canal público a fin de agotar los recursos limitados del cliente, etc.

Pérdidas de claves de cifrado: esto involucra la revelación de las claves secretas (SSL, cifrado de archivos, claves de los clientes particulares, etc) a terceros maliciosos, la pérdida o corrupción de las llaves, o su uso no autorizado para la autenticación y no repudio (firma digital).

Ataques de ingeniería social: las vulnerabilidades explotadas por los atacantes pueden ser la falta de concientización sobre la seguridad, vulnerabilidades en el cifrado de las comunicaciones, procedimientos inadecuados de seguridad física, etc.

Logs comprometidos: incluye tanto a los logs operacionales como a los de seguridad.

Vulnerabilidades

A continuación se detallan las principales vulnerabilidades nombradas anteriormente que dan lugar a los riesgos mencionados anteriormente. Algunas de ellas son específicas del modelo de Cloud Computing mientras que otras también están presentes en el modelo de cómputo tradicional.

Roles y responsabilidades poco claros

Esta vulnerabilidad consiste en la especificación inadecuada de los roles y responsabilidades dentro de la estructura del proveedor de Cloud Computing.

Cumplimiento deficiente de las definiciones de los roles

Una falla en la separación de los roles en el proveedor de Cloud Computing puede dar lugar a roles excesivamente privilegiados, haciendo el sistema vulnerable. Por ejemplo, una sola persona no debería poder acceder a la nube entera.

Principio de need -to-know no aplicado

Consiste en que en la definición de roles y responsabilidades se otorguen mayores privilegios de los que se necesitan para desarrollar las tareas encomendadas.

Atribución errónea de responsabilidades al proveedor

Los clientes de Cloud Computing pueden no estar al tanto de las responsabilidades que tienen asignadas en los términos de servicio y atribuir las erróneamente al proveedor. Por ejemplo, el cliente podría considerar que el proveedor debe encriptar sus archivos, aún cuando no está especificado en el contrato.

Falta de información sobre la ubicación de datos

Almacenar los datos en mirrors para utilizarlos en redes periféricas o guardarlos en forma redundante sin que el cliente disponga de información de su ubicación en tiempo real introduce cierto grado de vulnerabilidad.

Falla en el aislamiento de los recursos

El uso de recursos por parte de un cliente puede afectar la utilización de recursos de otro. En el modelo de Cloud Computing los recursos físicos son compartidos por múltiples máquinas virtuales y por lo tanto por múltiples clientes. Las vulnerabilidades en el hypervisor pueden llevar al acceso no autorizado de estos recursos compartidos. Esto podría ocurrir si por ejemplo las máquinas virtuales de dos clientes tienen sus discos virtuales almacenados en la misma LUN (logic unit number) de un SAN y uno de ellos logra mapear el disco virtual del otro en su máquina virtual obteniendo los permisos para ver y utilizar sus datos.

En el modelo IaaS el proveedor brinda a sus clientes una interfaz para gestionar sus recursos. Una vulnerabilidad en esta interfaz puede permitir el acceso no autorizado a los recursos del cliente, posibilitándole al atacante provocar denial of service, compromiso y fuga de datos, o daños financieros.

Vulnerabilidades en la encriptación de datos

Estas vulnerabilidades consisten en la posibilidad de leer datos en tránsito utilizando ataques como man-in-the-middle, explotando mecanismos de autenticación pobres o aceptando certificados firmados por entidades inválidas.

Se incluyen además en esta clase de vulnerabilidades las fallas en la encriptación de los datos almacenados en archivos, bases de datos, imágenes de máquinas virtuales no montadas, datos e imágenes forenses, logs críticos, entre otros.

Borrado de datos inseguro

Cada vez que un cliente cambie de proveedor, los recursos se reduzcan o el hardware se reubique, entre otros casos, los datos pueden quedar disponibles más allá del tiempo de vida especificado en la política de seguridad. Podría ser imposible llevar a cabo los procedimientos especificados por la política de seguridad, ya que la eliminación completa de todos los datos sólo es posible mediante la destrucción de un disco que también almacena los datos de otros clientes.

Además, las APIs ofrecidas por el proveedor pueden no soportar procedimientos de borrado seguro de datos.

El cifrado de los datos en el disco reduce considerablemente la posibilidad de explotar esta vulnerabilidad .

Implementación De Un Software Como Servicio

En esta sección se presentará la implementación de un sistema de software como servicio, se explicarán las herramientas utilizadas y se dará una conclusión del sistema en referencia a lo introducido anteriormente.

Rubox

El sistema implementado consiste en una aplicación de software como servicio orientada a nubes privadas para la gestión de archivos compartidos por múltiples usuarios, con soporte para archivos versionados. Tiene como objetivo un funcionamiento parecido a otros sistemas ya implementados (como Dropbox, Google Drive, iCloud, entre otros) en donde se pueden almacenar archivos de los usuarios y compartirlos pero con ciertas particularidades:

- Orientado a nubes privadas, es decir, los archivos no necesitan ser almacenados a través de internet.
- Versionado. La idea es que se pueda acceder a versiones anteriores de los archivos.
- Sincronización entre computadoras y publicación de los cambios a cargo del usuario.

Las carpetas compartidas en Rubox se llaman proyectos, un proyecto es un conjunto de archivos de un usuario en particular, el cuál puede dar permisos de lectura o escritura a los diferentes usuarios del sistema.

El sistema consiste a grandes rasgos en dos partes. Un servidor desde el cual se puede gestionar los usuarios y los permisos que se le otorgan sobre los diferentes proyectos y un cliente fino, que se instala en la pc de cada usuario, que es el encargado de la sincronización local de los proyectos con el servidor.

Tecnologías Utilizadas

En este capítulo se tratarán las diferentes tecnologías utilizadas para llevar a cabo el software.

Git

Git es un software de control de versiones open source, diseñado para pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente.

Control de versiones:

Un sistema de control de versiones (Version Control System o VCS) registra los cambios de un archivo o un conjunto de archivos en el tiempo con el fin de poder revertir cambios específicos y recuperar versiones anteriores, revertir el proyecto entero a un estado anterior, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que puede estar causando un problema, quién introdujo un error y cuándo, y mucho más. Ejemplos de otros software de control de versiones son, entre otros, CVS, Subversion, Mercurial, Perforce.

Un sistema de control de versiones debe proporcionar:

- Mecanismo de almacenamiento de los elementos que deba gestionar (ej. archivos de texto, imágenes, documentación...).
- Posibilidad de realizar cambios sobre los elementos almacenados (ej. modificaciones parciales, añadir, borrar, renombrar o mover elementos).
- Registro histórico de las acciones realizadas con cada elemento o conjunto de elementos (normalmente pudiendo volver o extraer un estado anterior del producto).

Terminología

La terminología empleada puede variar de sistema a sistema, pero a continuación se describen algunos términos de uso común:

- **Repositorio:** lugar en el que se almacenan los datos actualizados e históricos de cambios, a menudo en un servidor. A veces se le denomina depósito o depot.
- **Revisión ("version"):** una versión determinada de la información que se gestiona. Hay sistemas que identifican las revisiones con un contador (Ej. subversion). Hay otros sistemas que identifican las revisiones mediante un código de detección de modificaciones (Ej. Git usa SHA1). A la última versión se le suele identificar de forma especial con el nombre de HEAD.
- **Rotular ("tag"):** Los tags permiten identificar de forma fácil revisiones importantes en el proyecto. Por ejemplo se suelen usar tags para identificar el contenido de las versiones publicadas del proyecto.
- **Línea base ("Baseline"):** Una revisión aprobada de un documento o fichero fuente, a partir del cual se pueden realizar cambios subsiguientes.
- **Branch:** Un branch o rama es una bifurcación en un instante de tiempo de forma que, desde ese momento en adelante se tienen dos copias (ramas) que evolucionan de forma independiente siguiendo su propia línea de desarrollo. La ventaja es que se puede hacer un "merge" de las modificaciones de ambas ramas, posibilitando la creación de "ramas de prueba" que contengan código para evaluación, si se decide que las modificaciones realizadas en la "rama de prueba" sean preservadas, se hace

un "merge" con la rama principal. Son motivos habituales para la creación de ramas la creación de nuevas funcionalidades o la corrección de errores.

- **Desplegar ("Check-out", "checkout", "co"):** Un despliegue crea una copia de trabajo local desde el repositorio. Se puede especificar una revisión concreta, y por defecto se suele obtener la última.
- **"Publicar" o "Enviar"("commit", "check-in", "ci", "install", "submit"):** Un commit sucede cuando una copia de los cambios hechos a una copia local es escrita o integrada sobre repositorio.
- **Conflicto:** Un conflicto ocurre en las siguientes circunstancias:
 1. Los usuarios *X* e *Y* despliegan versiones del *archivo A* en que las líneas *n1* hasta *n2* son comunes.
 2. El usuario *X* envía cambios entre las líneas *n1* y *n2* al *archivo A*.
 3. El usuario *Y* no actualiza el *archivo A* tras el envío del usuario *X*.
 4. El usuario *Y* realiza cambios entre las líneas *n1* y *n2*.
 5. El usuario *Y* intenta posteriormente enviar esos cambios al *archivo A*.

El sistema es incapaz de fusionar los cambios. El usuario *Y* debe resolver el conflicto combinando los cambios, o eligiendo uno de ellos para descartar el otro.

Formas de colaborar

Para colaborar en un proyecto usando un sistema de control de versiones lo primero que hay que hacer es crearse una copia local obteniendo información del repositorio. A continuación el usuario puede modificar la copia. Existen dos esquemas básicos de funcionamiento para que los usuarios puedan ir aportando sus modificaciones:

- De forma exclusiva: En este esquema para poder realizar un cambio es necesario comunicar al repositorio el elemento que se desea modificar y el sistema se encargará de impedir que otro usuario pueda modificar dicho elemento. Una vez hecha la modificación, ésta se comparte con el resto de colaboradores. Si se ha terminado de modificar un elemento entonces se libera ese elemento para que otros lo puedan modificar. Este modo de funcionamiento es el que usa por ejemplo SourceSafe. Otros sistemas de control de versiones (Ejemplo subversion), aunque no obligan a usar este sistema, disponen de mecanismos que permiten implementarlo.
- De forma colaborativa: En este esquema cada usuario modifica la copia local y cuando el usuario decide compartir los cambios el sistema automáticamente intenta combinar las diversas modificaciones. El principal problema es la posible aparición de conflictos que deban ser solucionados manualmente o las posibles inconsistencias que surjan al modificar el mismo fichero por varias personas no coordinadas. Los sistemas de control de versiones subversion o Git permiten implementar este modo de funcionamiento.

Arquitecturas de almacenamiento

Podemos clasificar los sistemas de control de versiones atendiendo a la arquitectura utilizada para el almacenamiento del código:

- **Centralizados:** existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos). Se facilitan las tareas administrativas a cambio de reducir flexibilidad, pues todas las decisiones fuertes (como crear una nueva rama) necesitan la aprobación del responsable. Algunos ejemplos son CVS y Subversion.
- **Distribuidos:** Cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales. Ejemplos: Git y Mercurial.

Ventajas de sistemas distribuidos :

- Necesita menos veces estar conectado a la red para hacer operaciones. Esto produce una mayor autonomía y una mayor rapidez.
- Aunque se caiga el repositorio remoto la gente puede seguir trabajando.
- Al hacer los distintos repositorio una réplica local de la información de los repositorios remotos a los que se conectan, la información está muy replicada y por tanto el sistema tiene menos problemas en recuperarse si por ejemplo se quema la máquina que tiene el repositorio remoto. Por tanto hay menos necesidad de backups. Sin embargo los backups siguen siendo necesarios para resolver situaciones en las que cierta información todavía no haya sido replicada.
- Permite mantener repositorios centrales más limpios en el sentido de que un usuario puede decidir que ciertos cambios realizados por él en el repositorio local, no son relevantes para el resto de usuarios y por tanto no permite que esa información sea accesible de forma pública.
- El servidor remoto requiere menos recursos que los que necesitaría un servidor centralizado ya que gran parte del trabajo lo realizan los repositorios locales.
- Al ser los sistemas distribuidos más recientes que los sistemas centralizados, y al tener más flexibilidad por tener un repositorio local y otro/s remotos, estos sistemas han sido diseñados para hacer fácil el uso de ramas (creación, evolución y fusión) y poder aprovechar al máximo su potencial. Por ejemplo se pueden crear ramas en el repositorio remoto para corregir errores o crear funcionalidades nuevas. Pero también se pueden crear ramas en los

repositorio locales para que los usuarios puedan hacer pruebas y dependiendo de los resultados fusionarlos con el desarrollo principal o no.

Ventajas de sistemas centralizados

- En los sistemas distribuidos hay menos control a la hora de trabajar en equipo ya que no se tiene una versión centralizada de todo lo que se está haciendo en el proyecto.
- En los sistemas centralizados las versiones vienen identificadas por un número de versión. Sin embargo en los sistemas de control de versiones distribuidos no hay números de versión, ya que cada repositorio tendría sus propios números de revisión dependiendo de los cambios. En lugar de eso cada versión tiene un identificador al que se le puede asociar una etiqueta (tag).

Sistema de Versión Git

Git es un sistema de versión con arquitectura de almacenamiento *distribuida*, con un esquema de funcionamiento de aporte de modificaciones *colaborativo*.



El diseño de Git resulta de la experiencia del diseñador de Linux, Linus Torvalds, manteniendo una enorme cantidad de código distribuida y gestionada por mucha gente, que incide en numerosos detalles de rendimiento, y de la necesidad de rapidez en una primera implementación.

El origen de Git:

El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande. Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos. En 2002, el proyecto del núcleo de Linux empezó a usar un DVCS propietario llamado BitKeeper.

En 2005, la relación entre la comunidad que desarrollaba el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida gratuitamente. Esto impulsó a la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de Linux) a desarrollar su propia herramienta basada en algunas de las lecciones que aprendieron durante el uso de BitKeeper. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- Velocidad
- Diseño sencillo
- Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido

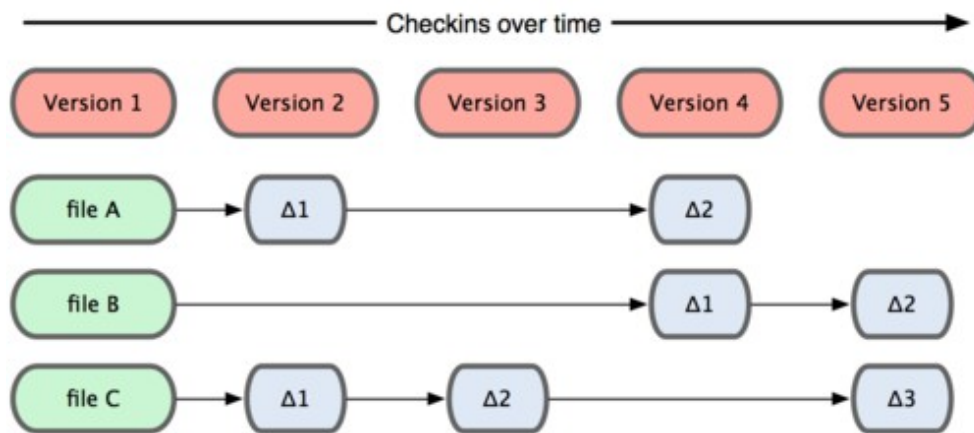
- Capaz de manejar grandes proyectos como el núcleo de Linux de manera eficiente (velocidad y tamaño de los datos)

Desde su nacimiento en 2005, Git ha evolucionado y madurado para ser fácil de usar y aún así conservar estas cualidades iniciales. Es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un sistema de ramificación para desarrollo no lineal muy bueno e intuitivo.

Bases de Git

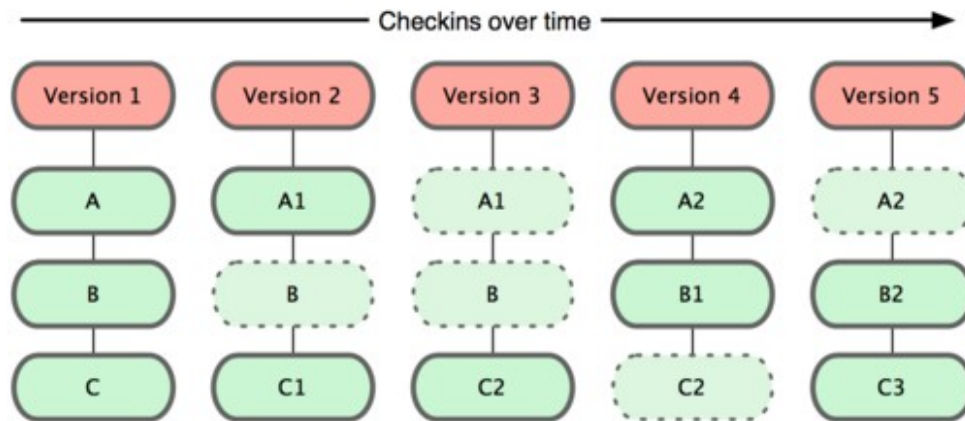
Snapshots, no diferencias:

La principal diferencia entre Git y cualquier otro VCS (Subversion y compañía incluidos) es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo, como ilustra la siguiente figura



Git no modela ni almacena sus datos de este modo. Sino que modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, básicamente se realiza una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo (sólo un enlace al archivo anterior idéntico que ya tiene almacenado). Git modela sus datos como la siguiente figura

Esta es una distinción importante entre Git y prácticamente todos los demás VCSs. Hace que Git reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace a Git más como un mini sistema de archivos con algunas herramientas tremendamente potentes construidas sobre él, más que como un VCS.



Casi todo es una operación es local

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar, por lo general no se necesita información de ningún otro ordenador de tu red. Como toda la historia del proyecto está en el disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita salir al servidor para obtener la historia y mostrártela, simplemente la lee directamente de tu base de datos local. Esto significa que la historia del proyecto se puede consultar casi al instante. Los cambios introducidos entre la versión actual de un archivo y ese archivo hace un mes se pueden consultar al hacer un cálculo de diferencias localmente.

Integridad

Todo en Git es verificado mediante un checksum antes de ser almacenado, y es identificado a partir de ese momento mediante esa cadena de caracteres. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que el sistema lo detecte.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula en base a los contenidos del archivo o estructura de directorios en Git.

24b9da6552252987aa493b52f8696cd6d3b00373

Git guarda todo no por nombre de archivo, sino en la base de datos de Git por el valor hash de sus contenidos.

Generalmente sólo añade información

Casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear

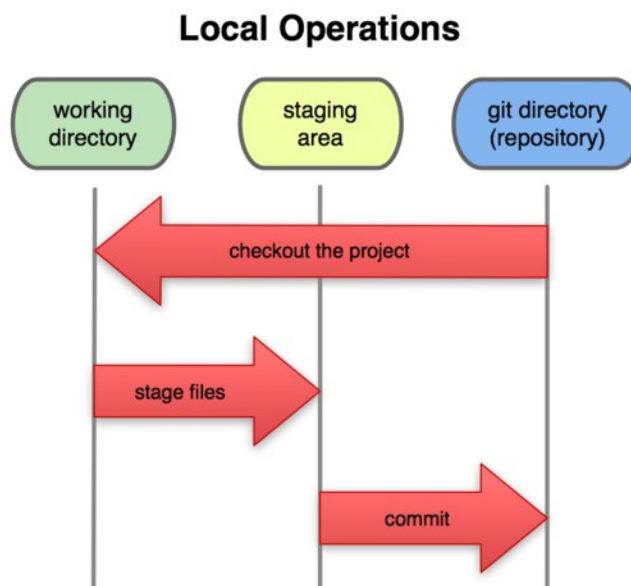
cambios que no has confirmado todavía; pero después de confirmar una instantánea en Git, es muy difícil de perder, especialmente si envías (push) tu base de datos a otro repositorio con regularidad.

Los tres estados

Git tiene tres estados principales en los que se pueden encontrar tus archivos: *confirmado* (committed), *modificado* (modified), y *preparado* (staged).

Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: el directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).



El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiendo instantáneas de ellos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esa instantánea de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio,

pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

Ruby

Ruby es un lenguaje de programación interpretado, reflexivo y orientado a objetos, creado por el programador japonés Yukihiro "Matz" Matsumoto, quien comenzó a trabajar en Ruby en 1993, y lo presentó públicamente en 1995. Combina una sintaxis inspirada en Python y Perl con características de programación orientada a objetos similares a Smalltalk. Comparte también funcionalidad con otros lenguajes de programación como Lisp, Lua, Dylan y CLU. Ruby es un lenguaje de programación interpretado en una sola pasada y su implementación oficial es distribuida bajo una licencia de software libre.



Características de Ruby

Viendo todo como un objeto

En Ruby, todo es un objeto. A cada variable se le puede asignar propiedades y acciones. En muchos lenguajes los números y otros tipos primitivos no son objetos, Ruby sigue las influencias del lenguaje Smalltalk pudiendo asignarle métodos y variables a todos sus tipos de datos. Esto facilita el uso de Ruby, porque las reglas que se aplican a los objetos se aplican de igual manera a todos los tipos. Esto se puede demostrar en un simple código

```
5.times { print "Rubox" }
```

La flexibilidad de Ruby

Ruby es considerado un lenguaje flexible, ya que permite a sus usuarios alterarlo libremente. Las partes esenciales de Ruby pueden ser quitadas o redefinidas a placer. Se puede agregar funcionalidad a partes ya existentes. Ruby intenta no restringir al desarrollador.

Por ejemplo, la suma se realiza con el operador suma (+). Pero si prefieres usar la palabra sumar, puedes agregar un método llamado sumar a la clase Numeric que viene incorporada.

```
class Numeric
  def sumar(x)
    self.+(x)
  end
end
```

```
end
y = 5.sumar 6
# ahora y vale 11
```

Los operadores de Ruby son simples conveniencias sintácticas para los métodos. Se pueden redefinir como y cuando sea necesario.

Bloques

El desarrollador puede anexar una cláusula a cualquier método, describiendo cómo debe actuar. La cláusula es llamada bloque y es vista como una fuente de gran flexibilidad. Los bloques están inspirados por los lenguajes funcionales como Lisp.

```
motores_de_busqueda =
  %w[Google Yahoo MSN].map do |motor|
    "http://www." + motor.downcase + ".com"
  end
```

En este código, el bloque está descrito entre la construcción *do ... end*. El método *map* aplica el bloque a la lista de palabras provista. Muchos otros métodos en Ruby dejan abierta la posibilidad al desarrollador para que escriba su propio bloque describiendo los detalles de qué debe hacer ese método.

Modelo de Objetos

A diferencia de otros lenguajes de programación orientada a objetos, Ruby se caracteriza por su intencional herencia simple. Sin embargo, Ruby incorpora el concepto de módulos (llamados categorías en Objective-C), que son colecciones de métodos.

Ruby implementa únicamente herencia simple, para logra comportamientos similares al de la herencia múltiple, utiliza el mecanismo de *Mixin*, que permite incluir módulos dentro de la definición de una clase. Por ejemplo, cualquier clase que implemente el método *each* puede incorporar el módulo *Enumerable*, que le agrega un conjunto de métodos que usan *each* para recorrer sus elementos.

```
class MiArray
  include Enumerable
end
```

Variables en Ruby

Ruby no necesita declaraciones de variables. Se utilizan convenciones simples para nombrar y determinar el alcance de las mismas.

- `var` puede ser una variable local.
- `@var` es una variable de instancia.

- \$var es una variable global.

Estos detalles mejoran la legibilidad permitiendo que el desarrollador identifique fácilmente los roles de las variables. También se hace innecesario el uso del molesto self. como prefijo de todos los miembros de instancia.

Otras características

- ✓ Manejo de excepciones, como Java y Python, para facilitar el manejo de errores.
- ✓ Garbage Collection con algoritmo *mark-and-sweep*, no es necesario mantener contadores de referencias en bibliotecas externas.
- ✓ Insertar código Ruby en programas hechos en C se puede realizar de manera sencilla utilizando una API.
- ✓ Bibliotecas de extensión dinámicamente si lo permite el sistema operativo.
- ✓ Manejo de hilos independiente del sistema operativo. De esta forma, tienes soporte multi-hilo en todas las plataformas en las que corre Ruby, sin importar si el sistema operativo lo soporta o no.
- ✓ Ruby es fácilmente portable: se desarrolla mayoritariamente en GNU/Linux, pero corre en varios tipos de UNIX, Mac OS X, Windows 95/98/Me/NT/2000/XP, DOS, BeOS, OS/2, etc.

Ruby On Rails

Ruby on Rails, también conocido como RoR o Rails, es un framework de aplicaciones web de código abierto escrito en el lenguaje de programación Ruby, siguiendo el paradigma de la arquitectura Modelo Vista Controlador (MVC). Trata de combinar la simplicidad con la posibilidad de desarrollar aplicaciones del mundo real escribiendo menos código que con otros frameworks y con un mínimo de configuración.



Creado por David Heinemeier Hansson, liberado en abril del 2004. La versión 1.0 fue liberada a principios del 2006 y ha sido referencia en la creación y evolución de otros frameworks webs escritos en otros lenguajes.

La filosofía de Rails se basa los siguientes principio:

- No te repitas (Don't Repeat Yourself / DRY): propone no escribir el mismo código una y otra vez, los componentes están integrados de manera que no hace falta establecer puentes entre ellos. Por ejemplo, en ActiveRecord (clase de la que heredan los modelos), las definiciones de las clases no necesitan especificar los nombres de las columnas; se pueden averiguar a

partir de la propia base de datos, definirlos tanto en el código como en el programa sería redundante.

- Convención sobre configuración: Rails hace suposiciones acerca de lo que se quiere hacer y como se quiere hacer en lugar de configurar muchos archivos que indiquen esto, esto significa que el programador sólo necesita definir aquella configuración que no es convencional.

Arquitectura Model View Controller (MVC)

La arquitectura MVC es un patrón o modelo de abstracción de desarrollo de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica que implementa la funcionalidad en tres componentes distintos.

Características del MVC:

- Aislamiento entre la lógica de la aplicación y la interfaz de usuario.
- Facilidades para mantener el código DRY.
- Legibilidad entre los diferentes tipos de códigos hace mas fácil el mantenimiento del sistema.

Descripción del patrón:

Modelo

Representa los datos de la aplicación y las reglas para manipular a los mismos. En el caso de Rails, los modelos se usan principalmente para definir las reglas de utilización de la base de datos, en la mayoría de los casos cada tabla en la base de datos se corresponde con un modelo en la aplicación. La mayor parte de las operaciones sobre la base de datos se concentra en los modelos.

En Rails, las clases del Modelo son gestionadas por `ActiveRecord`. Por lo general, lo único que tiene que hacer el programador es heredar de la clase `ActiveRecord::Base`, y el programa averiguará automáticamente qué tabla usar y qué columnas tiene.

Las definiciones de los modelos también detallan las relaciones entre clases con sentencias de mapeo entre diferentes modelos. Por ejemplo, si la clase `Imagen` tiene una definición `has_many:comentarios`, y existe una instancia de `Imagen` llamada `a`, entonces `a.comentarios` devolverá un arreglo con todos los objetos `Comentario` cuya columna `imagen_id` (en la tabla `comentarios`) sea igual a `a.id`.

Las rutinas de validación de datos (por ejemplo `validates_uniqueness_of: checksum`) y las rutinas relacionadas con la actualización (por ejemplo `after_destroy: borrar_archivo`, `before_update: actualizar_detalle`) también se especifican e implementan en la clase del modelo.

Vista

Representan la interfaz del usuario. En Rails, las vistas son generalmente archivos html con código Ruby embebido que realizan tareas solamente relacionadas a los datos que llegan a través del controlador.

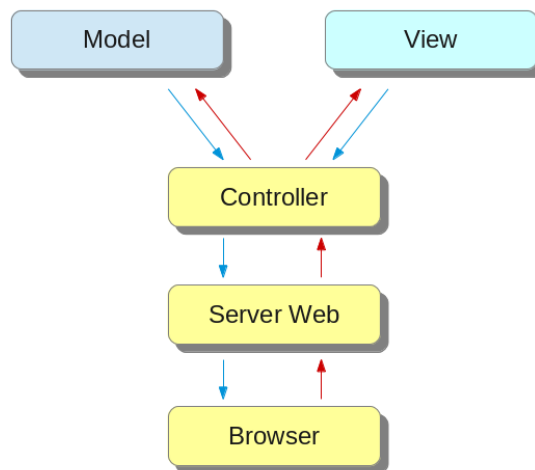
Existen en la actualidad muchas maneras de gestionar las vistas. El método que se emplea en Rails por defecto es usar Ruby Empotrado (archivos.rhtml, desde la versión 2.x en adelante de RoR archivos.html.erb), que son básicamente fragmentos de código HTML con algo de código en Ruby, siguiendo una sintaxis similar a JSP. También pueden construirse vistas en HTML y XML con Builder o usando el sistema de plantillas Liquid.

Es necesario escribir un pequeño fragmento de código en HTML para cada método del controlador que necesita mostrar información al usuario. El "maquetado" o distribución de los elementos de la página se describe separadamente de la acción del controlador y los fragmentos pueden invocarse unos a otros.

Controlador

Une las vistas y modelos. Un controlador en Rails se encarga de procesar solicitudes, generalmente provenientes de un navegador web, realizar consultas a los modelos de la aplicación y pasar estos datos a las vistas.

Un controlador hereda de la clase ApplicationController y define las acciones necesarias como métodos, que pueden ser invocados por el usuario a través de una URL.



Base de datos

El acceso a la base de datos es totalmente abstracto desde el punto de vista del programador, esto se debe a que todo acceso a los datos, ya sea para obtener, modificar o eliminar datos, se puede realizar a través de los métodos

proporcionados por los modelos (aunque, si se necesita, se pueden hacer consultas directas en SQL).

Rails intenta mantener la neutralidad con respecto a la base de datos, la portatibilidad de la aplicación a diferentes sistemas de base de datos y la reutilización de bases de datos preexistentes. Sin embargo, debido a la diferente naturaleza y prestaciones de los sistemas de gestión de bases de datos el framework no puede garantizar la compatibilidad completa. Se soportan diferentes sistemas de gestión de bases de datos, incluyendo MySQL, PostgreSQL, SQLite, IBM DB2 y Oracle.

La base de datos a usar se especifica en un archivo de configuración `config/database.yml`. El archivo contiene secciones de tres ambientes diferentes en los que Rails se pueden ejecutar de forma predeterminada:

- El entorno de desarrollo se utiliza en el equipo local mientras se interactúa manualmente con la aplicación.
- El entorno de prueba se utiliza al ejecutar pruebas automatizadas.
- Y por último el entorno de producción.

Ejemplo de archivo de configuración:

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

test:
  adapter: mysql2
  encoding: utf8
  reconnect: false
  database: rubox_test
  pool: 5
  username: root
  password:
  socket: /var/lib/mysql/mysql.sock

production:
  adapter: mysql2
  encoding: utf8
  reconnect: false
  database: rubox_production
  pool: 5
```

```
username: root
password:
socket: /var/lib/mysql/mysql.sock
```

Migraciones en Rails

Las migraciones son una herramienta de Rails que alteran la base de datos de una forma organizada y estructurada. Al editar la estructura de la base de datos a mano, queda en responsabilidad del programador propagar los cambios de manera correcta hacia las bases de datos de producción y testing. Permiten describir cambios a la base de datos mediante código Ruby. Esto es lo que da la funcionalidad de que los modelos sean independiente del motor de base de datos, es decir, no se especifica mediante el lenguaje de definición de motor de la base de datos.

Una migración es una clase que hereda de `ActiveRecord::Migration` que implementa algunos de los siguientes métodos:

- `add_column`
- `add_index`
- `change_column`
- `change_table`
- `create_table`
- `drop_table`
- `remove_column`
- `remove_index`
- `rename_column`

Este es un ejemplo de una clase de migración

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :login, :null => false
      t.string :name, :null => false
      t.string :email, :null => false
      t.string :password_hash, :null => false
      t.string :password_salt, :null => false

      t.timestamps
    end
  end
end
```

Los tipos soportados por el framework son:

- :binary
- :boolean
- :date
- :datetime
- :decimal
- :float
- :integer
- :primary_key
- :string
- :text
- :time
- :timestamp

Para gravar estos cambios en la base de datos se ejecuta el siguiente comando en la carpeta raíz el proyecto rails:

```
$rake db:create #crea la base de datos si es que no está creada
$rake db:migrate
```

Este comando ejecuta todas las migraciones que estén pendientes. Para poder revertir las migraciones existe otro comando que vuelve atrás los cambios de una migración específica:

```
$rake db:rollback
```

Truecrypt

TrueCrypt es una aplicación para cifrar y ocultar datos que el usuario empleando para ello diferentes algoritmos de cifrado como AES, Serpent y Twofish o una combinación de los mismos. Permite crear un volumen virtual cifrado en un archivo de forma rápida y transparente.

The logo for TrueCrypt, featuring the word "TRUECRYPT" in a light blue, sans-serif font, centered within a solid blue rectangular background.

Lo que hace TrueCrypt es crear un volumen que consiste en un archivo que puede tener cualquier nombre y que TrueCrypt puede montar como una unidad de disco, con su identificación respectiva, según el sistema operativo utilizado. El contenido de ese archivo tiene su propio sistema de archivos y todo lo necesario para operar como una unidad común de almacenamiento. Lo que se grabe en esa unidad virtual se cifra usando tecnología y la potencia de cifrado que el usuario elija. Cuando se "monta" la unidad a través de TrueCrypt, se pide la contraseña que el usuario escogió al momento de crear este archivo secreto.


Funcionamiento del programa

En este capítulo se abarcará la manera en que las distintas tecnologías nombradas en el punto anterior se relacionan entre sí para poder implementar el software.

Rubox

Como se introdujo anteriormente Rails es un framework para aplicaciones web. Es muy intuitivo de usar y por su modelo de diseño se pueden realizar aplicaciones web complejas en un tiempo relativamente corto.

En Rubox, Rails se utiliza como gestor de usuarios y permisos. Una de las cosas que se puede hacer desde la interfaz web es dar de alta usuarios. La siguiente pantalla muestra la interfaz para dar de alta un usuario:

El formulario es una tarjeta roja con el título "Nuevo" en blanco. Contiene cinco campos de entrada blancos con etiquetas rojas: "Nombre", "Login", "Email", "Contraseña" y "Confirme contraseña". En la parte inferior hay dos botones: "Crear usuario" (rojo) y "Cancelar" (gris).

Además, un usuario se puede loguear y gestionar sus proyectos. Esto incluye, crear y borrar proyectos y otorgar permisos de escritura o lectura a otros usuarios. Cuando un usuario inicia sesión en la página web, puede crear un proyecto nuevo. Esto creará un repositorio git en el servidor con la propiedad bare y se le asignará como owner el usuario logueado.

```
$ git init --bare "/var/cache/bare/proyecto.git"
```

Un repositorio git normal contienen archivos y carpetas que conforman el denominado `working git` y una carpeta llamada `.git` en donde se encuentran los objetos que llevan la administración e historia del repositorio. Cuando se crea un repositorio tipo bare, no se crea el `working dir`, es decir en el directorio principal se encuentran todos los objetos que, en un repositorio normal se encuentran en la carpeta `.git`. En este tipo de repositorios se pueden aplicar el comando `push`, para

poder agregar desde otro repositorio commits hechos localmente, pero no se puede trabajar editando los archivos, ya que no se dispone de un `working dir`.

Luego de crear el repositorio, se copia el hooks denominado `update`, el cual es un script escrito en ruby que se ejecuta cada vez que se realiza un `push` al repositorio y controla que el autor del commit entrante tenga permisos de escritura sobre el repositorio. El script realiza consultas a la base de datos mediante el ambiente que proporciona Rails para tener un acceso mas uniforme a la base de datos. A continuación se especifica el script

```
#!/home/usuario/.rvm/rubies/ruby-1.9.3-p327/bin/ruby

output = 0
begin
  $stdout.reopen("/tmp/out.txt", "a")
  $stderr.reopen("/tmp/err.txt", "a")
  require "/home/usuario/workspace/rubox/config/environment"

  pathLog = Dir.pwd + "/log"
  Dir.mkdir(pathLog) if (!File.exists?(pathLog))
  logFile = File.open(pathLog+"/out.log", "a")

  logFile.puts "-----\n"
  logFile.puts "[" + Time.new.to_s + "] \n"
  refname=ARGV[0]
  oldrev=ARGV[1]
  newrev=ARGV[2]

  #autor del commit y el mensaje del commit entrante
  gitLog = `git show --pretty="Autor: %an\nCommit: %s" #{newrev} | sed -n
1,2p`
  #lo escribo en el log
  logFile.puts gitLog.to_s
  #obtengo el login. Formato: [<login>] <name>
  split = gitLog.split("\n")
  split[0]["Autor: "] = ""
  split[1]["Commit: "] = ""
  login = split[0]
  #Obtengo el usuario de la base de datos
  user = User.where("login='" + login + "'").first

  path = Dir.pwd
  pName = File.basename(path, ".git")
  project = Project.where("name = '" + pName.to_s + "'").first

  if (user == nil || project == nil)
```

```

    logFile.puts "Error de validacion de usuario y proyecto"
    output = 1
  else
    pers = Permission.where("user_id = " + user.id.to_s
                          + " and project_id = " + project.id.to_s).first
    if (pers == nil || pers.type.description == "read")
      logFile.puts "Error. El usuario no tiene permisos"
      output = 1
    else
      logFile.puts "Cambios aceptados"
    end
  end
end

rescue Exception => e
  output = 1
  logFile.puts (e.message)
ensure
  logFile.flush
  logFile.close
end

exit(output)

```

El usuario podrá también listar el los proyectos que tiene compartidos y sobre estos otorgar privilegios de escritura o lectura, según pueda.

La política implementada para otorgar privilegiados es:

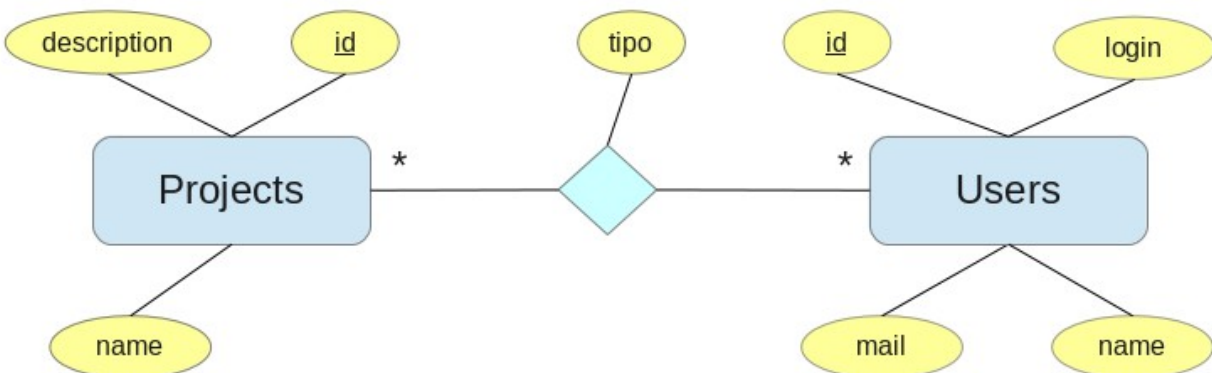
- El dueño (owner) puede otorgar privilegios de lectura y escritura. Pero no podrá ceder su propiedad de dueño del proyecto a otra persona. Esto es para mantener un responsable por cada proyecto.
- Un escritor (writer) puede otorgar privilegiados de lectura a otros usuarios.
- Un lector (reader) no puede otorgar privilegiados de ningún tipo.



El servidor de Rails sirve también como interfaz de comunicación entre el cliente fino instalado en las computadoras de los usuarios y los repositorios Git. Por ejemplo, el sistema de logueo que se usa por medio de la interfaz web es el mismo que utiliza el Drubox, o las consultas sobre los permisos que tiene sobre los diferentes proyectos también son consultas que se hacen a través del servidor Rails.

Modelos y estructuras

La estructura de base de datos empleada se puede ver reflejada en el siguiente diagrama entidad relación:



Este sencillo diagrama es el que se mapea a los modelos de Rails.

Los tres principales modelos son los siguientes:


```

class Project < ActiveRecord::Base
  attr_accessible :description, :name

  validates :name, :uniqueness => true
  validates :description, :presence => true

  has_many :permissions, :dependent => :delete_all
  has_many :users, :through => :permissions
  has_many :types, :through => :permissions
end

```

```

class User < ActiveRecord::Base
  attr_accessible :login, :name, :email

  validates :login, :uniqueness => true, :presence => true

  has_many :permissions
  has_many :projects, :through => :permissions
  has_many :types, :through => :permissions
end

```

```

class Permission < ActiveRecord::Base

  belongs_to :type
  belongs_to :user
  belongs_to :project

end

```

```

class Type < ActiveRecord::Base

  attr_accessible :description

end

```

Uno de los tres pilares del framework Rails son los modelos, en ellos se definen las clases de los objetos que se usan en los controladores y vistas. En los modelos se especifican los campos de cada clase, validaciones y relaciones entre modelos.

En el caso del modelo Project, cuyos campos son description y name, las validaciones sobre estos campos son que el nombre tiene que ser único (uniqueness) y que la descripción no pueda ser vacía (presence). Con especificar

estas validaciones, el framework las tendrá en cuenta siempre que se quiera modificar o agregar proyectos desde cualquier lugar.

El modelo Permission representa la relación entre los usuarios y sus proyectos junto a los permisos. Es por eso que no tiene especificado ningún atributo, solo se declara la relación entre los demás objetos.

Además, en los modelos se declaran los métodos que se aplican directamente sobre los objetos de estos modelos. Para el caso del modelo User, el método de validación de usuario y password se declara de la siguiente manera:

```
def self.authenticate(login, password)
  user = find_by_login(login)
  if user && user.password_hash ==
    BCrypt::Engine.hash_secret(password, user.password_salt)
    user
  else
    nil
  end
end
```

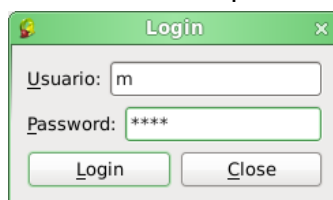
Esta función utiliza un método proporcionado por la librería bcrypt, en donde se verifica si la autenticación es correcta, la encriptación depende de una cadena de caracteres que se genera y almacena en la base de datos al crearse el usuario que se usa como semilla de la encriptación del password.

Drubox

Drubox es un programa escrito en ruby que su función es sincronizar los archivos en las máquinas locales con la versión que está en los repositorios centrales. Utiliza las librerías Qt para realizar la interfaz con el usuario y la gema Git para realizar las operaciones sobre los archivos.

Funcionamiento

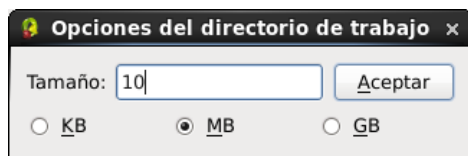
Al abrirse el programa se encuentra una pantalla de logueo como la siguiente:



La acción de logueo implica la validación del usuario frente al servidor rails utilizando los mismos métodos que cuando se realiza por la página web de Rubox. Una vez realizada la validación, se prosigue al montado del un volumen encriptado mediante truecrypt, esto agrega un nivel de seguridad y protección de los archivos

de los usuario, ya que se trabaja sobre un volumen encriptado, el cual al desmontarse, los datos que permanezcan almacenados en el disco local quedan inaccesible si no se posee la contraseña del usuario.

Al iniciar sesión por primera vez, el volumen no está creado para este usuario, por lo tanto se solicita al usuario el tamaño del espacio de trabajo que quiere reservar y se genera, mediante `truecrypt` un volumen en la carpeta oculta `.hd` que se encuentra dentro de una carpeta llamada `Rubox` en el home del usuario. Luego monta ese volumen en el directorio `Rubox/<login>`.



Una vez accedido al sistema y montado el volumen, se cargará en un combo todos los proyectos que el usuario tenga permiso de lectura o escritura. Cuando se selecciona un proyecto, si éste no está localmente en un repositorio, Drubox lo clona automáticamente desde el repositorio central en el directorio de trabajo.











Una vez seleccionado el proyecto, se cargará en la parte central del programa el árbol de archivos del proyecto y quedará una pantalla como la siguiente:

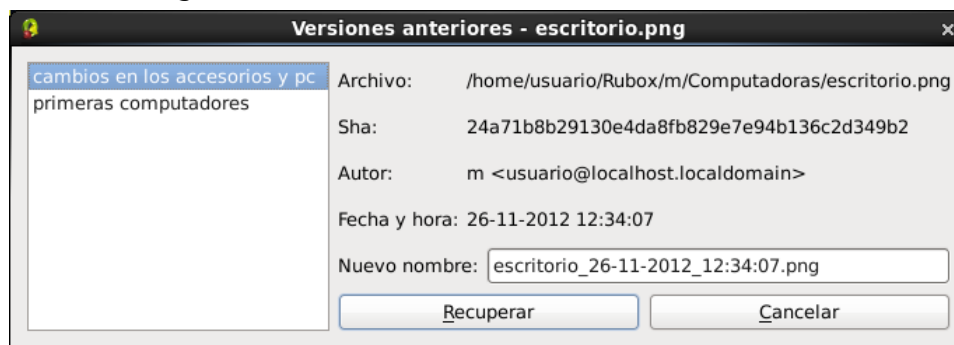


Menu

El primer botón del menú sirve para loguarse, es útil cuando el usuario cierra sesión desde Rubox → Cerrar Sesión. Al lado de este botón se encuentra el combo de los proyectos que el usuario logueado tiene acceso, ya sea de lectura o escritura.

Botones de acción sobre el repositorio:

-  Agrega un archivo a la carpeta que está seleccionada en el árbol. Si el archivo existe, pregunta si se desea reemplazar.
-  Agrega una carpeta dentro de la carpeta que está seleccionada en el árbol. Si la carpeta existe, pregunta si se desea reemplazar.
-  Elimina el archivo o carpeta seleccionado en el árbol. Pide antes una confirmación.
-  Pull. Este botón realiza la acción de sincronización con el repositorio central. Esto implica, realizar un commit en el repositorio local para salvar los cambios y luego hacer un pull. Si hay cambios se pide un mensaje para el commit a realizar, si el working dir está limpio se procede directamente a descargar los cambios.
-  Push. La acción que realiza es la de subir los cambios locales al repositorio principal. Antes de hacer esto realiza la misma acción que el botón anterior, esto se hace para obtener primero todos los cambios, actualizando primero el repositorio local.
-  Actualiza la columna estado del árbol con los siguientes iconos según corresponda:
 -  Archivo añadido al repositorio local. Quiere decir que el en repositorio central no existe.
 -  Archivo modificado. Cuando el usuario realiza modificaciones al archivo. Por ejemplo si es un archivo de texto y le agregó, modificó o borró líneas, o si es una imagen y se reemplazó por otra.
 -  Archivo eliminado en el repositorio local. Existe en el repositorio central, por lo tanto cuando se suban estos cambios se eliminará del repositorio.
-  Versiones anteriores. Con este botón el usuario tiene la posibilidad de recuperar versiones anteriores de sus archivos. Al hacer click se abre el siguiente diálogo:



En este diálogo se muestra a la izquierda un listado de todos los commits en los que fue incluido el archivo seleccionado en el árbol, y a la derecha información del commit seleccionado en la lista. Al hacer click en Recuperar se agregará al working dir la versión anterior del archivo con el nombre indicado en el diálogo.

Cuando el usuario termina de trabajar con drubox y cierra sesión, se desmonta el volumen de truecrypt del usuario y se realiza la acción de deslogueo en el servidor rails al igual que cuando cierra sesión desde la página web.

Ventajas y desventajas

Las ventajas frente a un sistema tradicional para compartir archivos, en donde los usuarios tienen carpetas compartidas, se transfieren archivos a través de la red o mediante pendrives son las siguientes:

- Se tiene control de acceso a los datos compartidos.
- Los cambios realizados por los usuarios quedan registrados en los logs lo que hace que la recolección de evidencia sea más sencilla.
- El control de actualizaciones concurrentes proporcionado por git permite trabajar en forma distribuida sobre los mismos sin posibilidad de corromperlos.
- Se cuenta con un control de versiones de los diferentes archivos que posibilita retroceder los cambios realizados y obtener versiones anteriores de los archivos.
- Al contar con repositorios locales y remotos se tienen una mayor distribución de los datos por lo que se obtiene una mayor tolerancia a fallas en caso de pérdidas de datos.

La principal desventaja es que si bien se tienen los datos distribuidos en los repositorios locales, al tener los datos concentrados en el repositorio central, el riesgo del atacante interno es mayoría que puede obtener mayor información de un único lugar.

Al tratarse de una nube privada se evitan los riesgos asociados a las nubes públicas que involucran a un proveedor del servicio, como la pérdida de gobierno, lock-in y problemas de cumplimiento ya que la empresa es la encargada de administrar sus propios datos. No obstante, los beneficios que provee una nube pública como seguridad a escala y escalamiento de recursos no son aprovechados ya que la inversión en licencias y recursos la realiza en su totalidad la empresa.

Conclusión

En este informe se presentó el concepto de Cloud Computing como un mecanismo para adquirir tecnologías de información como servicio, junto a sus principales características, modelos de servicio y de despliegue. Sobre esta definición se trataron los diferentes aspectos en relación a la seguridad, considerando los principales beneficios y riesgos.

En el modelo de Cloud Computing público la organización le cede al proveedor el control de un número de cuestiones que hacen a la seguridad del sistema. Esto resulta beneficioso si la organización no cuenta con la capacidad de manejar la seguridad en forma adecuada ya que se la deja en manos de terceros, aprovechando así las ventajas del modelo de Cloud Computing entre las que se destacan la economía de escala, la seguridad como diferencial en el mercado y la resistencia de los sistemas ante ataques. Sin embargo, las organizaciones que sí son capaces de garantizar el nivel de seguridad que desean, pueden inclinarse por los ambientes de cómputo tradicionales o modelos de Cloud Computing privados para evitar exponerse a los riesgos propios de una nube pública, como la pérdida de gobierno, el lock-in o las fallas de aislamiento.

Por lo tanto, cuando se considera adoptar el modelo de Cloud Computing es necesario evaluar los riesgos y beneficios que ofrece y compararlos con los presentes en el sistema actual.

Anexos

Instalar Git

Instalando desde código fuente

En general es útil instalar Git desde código fuente, porque se obtiene la versión más reciente. Cada versión de Git tiende a incluir útiles mejoras en la interfaz de usuario, por lo que utilizar la última versión es a menudo el camino más adecuado si te sientes cómodo compilando software desde código fuente. También ocurre que muchas distribuciones de Linux contienen paquetes muy antiguos; así que a menos que estés en una distribución muy actualizada o estés usando backports, instalar desde código fuente puede ser la mejor apuesta.

Para instalar Git, se necesita tener las siguientes librerías de las que Git depende: curl, zlib, openssl, expat, y libiconv. Por ejemplo, si estás en un sistema que tiene yum (como Fedora) o apt-get (como un sistema basado en Debian), puedes usar uno de estos comandos para instalar todas las dependencias:

```
$ yum install curl-devel expat-devel gettext-devel \
    openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
    libz-dev
```

Cuando estén todas las dependencias necesarias, se descarga la versión más reciente de Git desde su página web (<http://git-scm.com/download>)

Luego compila e instala:

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Una vez hecho esto, también puedes obtener Git a través del propio Git para futuras actualizaciones:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Instalando en Linux

Si se quiere instalar Git en Linux a través de un instalador binario, en general es posible hacerlo a través de la herramienta básica de gestión de paquetes que trae tu distribución.

Si estás en Fedora, puedes usar yum:

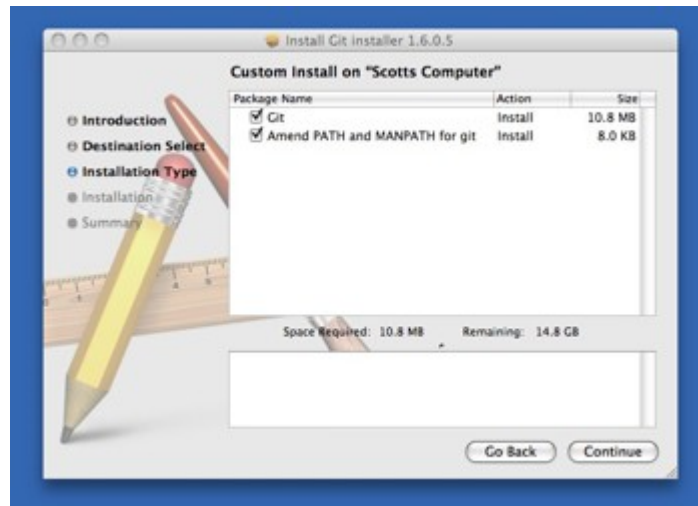
```
# yum install git-core
```

O si estás en una distribución basada en Debian como Ubuntu:

```
# apt-get install git-core
```

Instalando en Mac

Hay dos maneras fáciles de instalar Git en un Mac. La más sencilla es usar el instalador gráfico de Git, que puedes descargar desde la página de Google Code (<http://code.google.com/p/git-osx-installer>)



La otra manera es instalar Git a través de MacPorts (<http://www.macports.org>). Si tienes MacPorts instalado, instala Git con:

```
# sudo port install git-core +svn +doc +bash_completion +gitweb
```

Instalando en Windows

El proyecto msysGit tiene uno de los procesos de instalación más sencillos. Simplemente descarga el archivo exe del instalador desde la página de Google Code, y se ejecuta (<http://msysgit.github.com/>)

Este paquete incluye tanto la versión de línea de comandos (incluido un cliente SSH) como la interfaz gráfica de usuario estándar.

Intalar Rubox en Linux

Ruby y RoR

Para instalar ruby en nuestro sistema linux recomendamos hacerlo mediante Ruby Version Manager (RVM), ya que es un gestor de versiones fácil de usar y de instalar.

Aquí están los pasos para instalar en la distribución de Linux CentOS 6.

Instalar las dependencias necesarias:

```
# yum install curl-devel expat-devel gettext-devel \
    openssl-devel zlib-devel curl
```

Instalar RVM y Ruby

```
$ \curl -L https://get.rvm.io | bash -s stable --ruby
```

RVM Requeriments

```
$ rvm requirements
# yum install -y gcc-c++ patch readline readline-devel \
    zlib zlib-devel libyaml-devel libffi-devel \
    openssl-devel make bzip2 autoconf automake \
    libtool bison iconv-devel
```

Instalamos sqlite y mysql

```
# yum install sqlite sqlite-devel
# yum install mysql mysql-server libmysql-ruby \
    libmysqlclient-dev mysql-devel
# chkconfig --level 2345 mysqld on
```

Reinstalamos ruby

```
$ rvm reinstall 1.9.3
```

Instalamos Rails con RVM

```
$ \curl -L https://get.rvm.io | bash -s stable --rails
```

Intalación de Truecrypt

Descarga del tar desde <http://www.truecrypt.org/downloads>

```
# tar -zxvf truecrypt-7.1a-linux-console-x86.tar.gz
# ./truecrypt-7.1a-setup-console-x86
```

Para montar volúmenes con usuario normal hacer lo siguiente

```
# groupadd truecrypt
# visudo
```

```
-- Agregar al final --
# Users in the truecrypt group are
# allowed to run TrueCrypt as root.
%truecrypt ALL=(root) NOPASSWD:/usr/bin/truecrypt

-- Agrego los usuarios normales al grupo --
gpasswd -a USER_1 truecrypt
```

Instalación de Qt

```
# yum install qt qt-demos qt-designer qt4 qt4-designer cmake
$ gem install qtbindings
```

Instalación de gema Git

```
$ gem install git
```

Rubox

Se puede descargar Rubox desde el repositorio de desarrollo en GitHub. También está disponible el código fuente en un cd anexo al informe impreso.

```
$ mkdir ~/workspace
$ cd ~/workspace
$ git clone https://github.com/maurofermani/Rubox.git
$ cd Rubox
```

Instalación de gemas adicionales

```
$ bundle config build.mysql2 \
    --with-mysql-config='/usr/bin/mysql_config'
$ bundle install
```

Creación de la base de datos y ejecución de las migraciones

```
$ rake db:create
$ rake db:migrate
$ rake db:seed
```

Ejecución del servidor rails

```
$ rails server
```

Drubox

Drubox, al igual que rubox, se puede descargar los fuentes del GitHub o está disponible en el cd anexo.

```
$ mkdir ~/workspace
```

```
$ cd ~/workspace
$ git clone https://github.com/maurofermani/drubox.git
$ cd drubox/src
$ ruby druboxGUI.rb
```

En el archivo `src/config/environment.yml` se encuentra la configuración de la ubicación del directorio de trabajo, dirección host y puerto del servidor rails y el nombre del archivo del log.

Implementaciones futuras

Autenticación de usuarios frente a un servidor LDAP

Una de las principales mejoras que hay para implementar es la autenticación de los usuarios ante un servidor LDAP y no contra la base de datos de usuarios de Rubox.

Con esto se lograría una mejor integración con el sistema informático de una empresa, además de poder implementar mayor seguridad en una capa inferior. Si los usuarios de Rubox fueran los usuarios del sistema en general, se podría implementar seguridad mediante ssh, con sus llaves públicas y privadas para ser mas transparente, dando una transferencia de datos encriptadas y un mejor control de acceso a los repositorios.

Robustes en el servidor

Para hacer el sistema mas robusto se puede agregar redundancia en el servidor rails y distribuir el repositorio central a diferentes servidores sincronizados a través de los hooks de git cuando se reciben los push realizados por los usuarios.

Además se puede agregar que la conexión al servidor rails se realice mediante el protocolo https para mejorar la autenticación del usuario.

Portabilidad a otros sistemas operativos

El sistema fue desarrollado y testeado para sistemas linux. Otra extensión consiste en adaptar el código de DRubox para que pueda correr el cliente fino en diferentes sistemas operativos como Windows y Mac OS.

Acceso web a los repositorios

Agregar la funcionalidad al server Rails de poder trabajar sobre los datos de los proyectos cuando el usuario accede mediante la aplicación web.

Referencias

Referencias sobre Cloud Computing

- Cloud Security and Privacy – Oreilly
- [Cloud Computing Security Risk Assessment – ENISA](#)
- [Cloud Security Alliance](#)
- [Cloud Computing Security - iSEC Partners](#)

Referencias de programación

Ruby on Rail:

- http://es.wikipedia.org/wiki/Ruby_on_Rails
- <http://gwolf.org/files/rails.pdf>
- http://es.wikipedia.org/wiki/Modelo_Vista_Controlador
- <http://guides.rubyonrails.org/>
- <http://es.asciicasts.com/episodes/250-autenticacion-desde-cero>

Ruby

- <http://es.wikipedia.org/wiki/Ruby>
- <http://www.ruby-lang.org/en/about/>
- <http://git.rubyforge.org/>

Git

- <http://es.wikipedia.org/wiki/Git>
- <http://git-scm.com/doc>
- http://es.wikipedia.org/wiki/Control_de_versiones

Truecrypt

- <http://es.wikipedia.org/wiki/TrueCrypt>
- <http://www.irongeek.com/i.php?page=backtrack-3-man/truecrypt>
- <https://wiki.archlinux.org/index.php/TrueCrypt>