UNIVERSIDAD ORT URUGUAY
FACULTAD DE INGENIERIA

Arquitectura de Software en entornos de Cloud Computing

José Chiarino – 157586
Tutor: Gastón Mousqués

2013

Abstract

La computación en la nube es un paradigma que ha transformado radicalmente la forma en que el software es desarrollado, entregado y licenciado. Las nuevas tecnologías disponibles en la nube permiten:

- Despliegue rápido de aplicaciones. Solo existe una copia de la aplicación instalada en un entorno controlado, esto permite liberar versiones en todo momento.
- Accesibilidad de la información y sistemas desde cualquier dispositivo. La aplicación y los datos se almacenan en una infraestructura distribuida accesible desde cualquier lugar y momento.
- Disponibilidad de la aplicación. La gestión de disponibilidad y continuidad se transfiere del departamento de TI del usuario al proveedor de servicios en la nube.
- Ausencia de riesgo financiero. Solo se paga por los recursos utilizados.
- Escalabilidad. Al utilizar Cloud Computing la capacidad crece junto con la empresa, sin frenar su crecimiento y sin mayor esfuerzo.

Debido a las diferencias tecnológicas y su gran dinamismo, la computación en la nube tiene un alto impacto sobre el diseño arquitectónico de las aplicaciones. Las aplicaciones deben ser construidas para proveer servicio flexibles, deben ser capaces de escalar agresivamente bajo demanda, y permitir consumir funcionalidad a través de redes no confiable entre otras características.

Durante el desarrollo del presente documento se realiza una introducción al concepto de computación en la nube, mencionando cada uno de los modelos de servicio y despliegue.

Posteriormente se realiza un análisis profundo acerca del diseño de arquitecturas para la nube, cuales son los objetivos buscados, y como alcanzarlos.

Finalmente se presentan algunos de los patrones más importantes para el desarrollo de aplicaciones en la nube, los cuales permiten alcanzar los objetivos buscados.

Palabras clave

Computación en la nube. La computación en la nube es un paradigma de computación distribuida, gobernada por economías de escala, que permite acceso global, inmediato y conveniente a un conjunto abstracto, virtualizado y configurable de recursos de cómputo.

Nodo. Una aplicación generalmente ejecuta en un número variable de nodos. Cada nodo pose asociado recursos de hardware. Existen varios tipos de nodos: nodos de cómputo (donde ejecutan las aplicaciones), nodos de datos (donde se almacenan datos) y otros. Un nodo puede ser parte de un servidor físico (máquina virtual), un servidor físico entero, o un clúster de servidores.

Configuración topológica. Conjunto especifico de asociaciones entre componentes y conectores de la arquitectura de un sistema.

Modelo de referencia. Descomposición de un problema conocido en partes que cooperan para resolverlo. Surgen de la experiencia y son indicadores de dominio (del problema) madurez.

Arquitectura de referencia. Es un modelo de referencia mapeado a elementos de software y los flujos de datos entre ellos.

Estilo de arquitectura. Es una colección de decisiones de arquitectura que tienen un nombre y que son aplicables en un determinado contexto, restringen dentro de ese contexto las decisiones de arquitectura para un sistema, y proveen cualidades que son beneficiosas en cada sistema en el cual se aplica

Patrón de arquitectura. Un patrón es una colección de decisiones de arquitectura que son aplicables a un problema de diseño recurrente y parametrizables para contemplar distintos contextos en el cual el problema aparece.

Contenido

1. Computación en la nube

La computación en la nube es un paradigma de computación distribuida, gobernada por economías de escala, que permite acceso global, inmediato y conveniente a un conjunto abstracto, virtualizado y configurable de recursos de cómputo (redes, servidores, almacenamiento, aplicaciones o servicios) que pueden ser rápidamente adquiridos y liberados con un mínimo esfuerzo de gestión con el proveedor de servicio. Los recursos son entregados a demanda de clientes externos a través de la red, generalmente internet [1], [2].

El paradigma incluye un conjunto de características esenciales, tres modelos de servicio, y dos de despliegue los cuales serán tratados en el resto del capítulo.

1.1. Características esenciales

Las características esenciales describen el modelo de computación en la nube, y a su vez constituyen las principales fuerzas que justifican su adopción. A continuación se las presentan.

Autoservicio bajo demanda. Un usuario puede adquirir recursos de computación, tales como tiempo en servidor, acceso a la red o almacenamiento, cuando lo requiera y sin necesidad de interacción humana con el proveedor de servicios [1].

Acceso a través de la red. Los recursos son accesibles mediante conexiones de red usando mecanismos estándares, los cuales permiten el acceso a diferentes tipos de clientes, por ejemplo dispositivos móviles, tabletas, laptops o estaciones de trabajo [1].

Agrupación de recursos. El proveedor agrupa y administra los recursos disponibles para dar servicio a un gran número de clientes, como consecuencia los recursos físicos y virtuales son dinámicamente asignados de acuerdo a la demanda de cada uno [1].

Independencia de la localización. El cliente generalmente no tiene conocimiento ni control acerca de la ubicación exacta de los recursos que utiliza. Sin embargo, en algunos casos es posible especificar un país o región [2].

Elasticidad. La capacidad puede ser elásticamente adquirida o liberada, generalmente de forma automática, para escalar rápidamente según lo exija la demanda. Para el cliente, la capacidad disponible suele aparecer como ilimitada y puede ser asignada en cualquier cantidad en cualquier momento [1].

Uso del servicio. Los sistemas en la nube proveen instrumentos para medir su utilización, los cuales permiten monitoreo, control y reporte aportando transparencia tanto para el cliente como para el proveedor [1].

Economía de escala. En general los modelos de negocios en la nube son de alto volumen y bajo margen [2].

1.2. Virtualización y Cloud Computing

La virtualización es la abstracción de algún recurso tecnológico. Puede ser una plataforma de hardware, un sistema operativo, un dispositivo de almacenamiento u otros recursos de red [3].

A través de la virtualización, cada aplicación y sistema operativo ejecuta dentro de un contenedor de software llamado máquina virtual (VM). Debido a que cada VM es independiente y está completamente aislada, muchas de ellas pueden funcionar en simultáneo dentro de una sola computadora física (host). Una capa de software llamada *Hypervisor* desacopla las VMs del host, y asigna dinámicamente los recursos a cada VM cuando estos se necesitan [4].

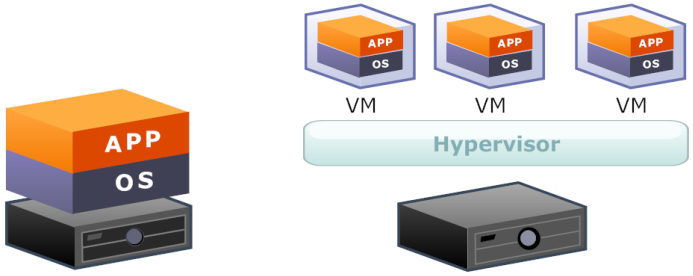

Ilustración 1 Arquitectura tradicional vs. Arquitectura con virtualización

La virtualización permite:

- **Ejecución de múltiples aplicaciones en cada servidor.** Como cada VM encapsula una computadora completa, muchas aplicaciones y sistemas operativos pueden funcionar en un host al mismo momento.
- **Maximización del uso de cada servidor.** Cada máquina física es usada a capacidad completa, esto permite hacer un uso eficiente de los recursos. Se reducen los costos al disminuir la cantidad de servidores necesarios.
- **Despliegue de aplicaciones y asignación de recursos fácil, rápido y flexible.** Como archivos de software auto contenidos, las VMs pueden ser manipuladas muy fácilmente. Esto permite simplicidad y rapidez en la administración de aplicaciones. Las máquinas virtuales incluso pueden ser transferidas de un servidor físico a otro mientras ejecutan, a través de un proceso conocido como *live migration*.

Esta tecnología es fundamental para el desarrollo de Cloud Computing, ya que posibilita al proveedor de servicios la flexibilidad necesaria para mover, reservar y asignar los recursos de computación requeridos por una aplicación, a aquellas maquinas físicas que estén disponibles de forma instantánea.

Toda aplicación o servicio en la nube ejecuta dentro de una máquina virtual (alojada en un host). Cuando la demanda de una aplicación aumenta la VM es replicada, y un balanceador de carga distribuye el trabajo entre las distintas instancias disponibles. De esa forma se logra la elasticidad característica de la nube.

1.3. Modelos de servicio

Los servicios de computación en la nube son entregados de acuerdo a distintos modelos. Los modelos se distinguen según el nivel de abstracción de los recursos facilitados, cada nivel oculta los detalles de niveles inferiores. A mayor nivel de abstracción, menor serán los esfuerzos de gestión necesarios por parte del usuario para hacer uso del servicio, como contraparte su flexibilidad será menor. La oferta es diversa e incluye todo el abanico de posibilidades desde arrendar recursos de procesamiento, hasta el licenciamiento de software a demanda.

1.3.1. Infraestructura como servicio (IaaS)

Se provee al consumidor la capacidad de adquirir bajo demanda recursos de procesamiento, almacenamiento, red, etc. donde se puede desplegar y ejecutar software arbitrario (incluye sistemas operativos y aplicaciones) [2], [5].

El consumidor no maneja la infraestructura de *cloud* directamente pero tiene control sobre las instancias, sistemas operativos, almacenamiento, aplicaciones y red. Como consecuencia, IaaS resulta muy versátil. No existe ningún impedimento para ejecutar aplicaciones o sitios web existentes, paquetes estándar de aplicaciones o máquinas virtuales para desarrollo y prueba de aplicaciones.

En este modelo los desarrolladores crean, configuran y mantienen máquinas virtuales junto con todo el software requerido para ejecutar sus aplicaciones.

Por ejemplo, para desplegar una aplicación sencilla con base de datos los pasos requeridos son:

1. Se escoge una imagen para el DBMS (*Database Management System*) y se lo configura.
2. Se escoge una imagen para los servidores web y de aplicaciones. Se crean las VMs y luego se configuran.
3. Se crea la base de datos, y se instalan los esquemas (estructura e información).
4. Se despliega la aplicación.
5. Se configura el balanceador de carga, el cual distribuye la carga entre los servidores disponibles.

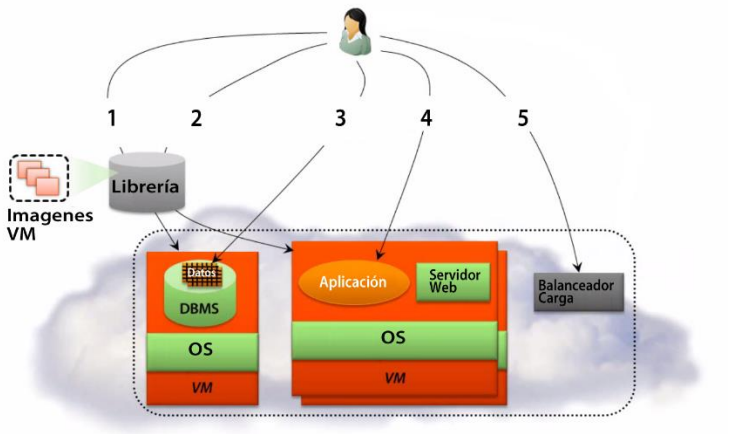

Ilustración 2 Esquema de despliegue IaaS

1.3.2. Plataforma como servicio (PaaS)

Se provee al consumidor la capacidad de desplegar en una infraestructura en la nube, aplicaciones creadas o adquiridas utilizando lenguajes de programación, librerías, servicios y herramientas soportadas por el proveedor del servicio [2], [5].

La plataforma existe y es administrada por el proveedor, el consumidor no maneja la infraestructura de *cloud* incluyendo red, servidores, sistemas operativos y almacenamiento. Sin embargo, puede tener control sobre las aplicaciones desplegadas. Como consecuencia existe un ahorro de tiempo, dinero y trabajo que permite concentrar los esfuerzos en el desarrollo de la aplicación.

PaaS resulta adecuado para aplicaciones nuevas que puedan ser diseñadas para explotar las características de la plataforma.

En este esquema el desarrollador provee una aplicación que la plataforma ejecuta, a alto nivel los pasos requeridos son:

1. Se crean las bases de datos, y se instalan los esquemas (estructura e información).
2. Se despliega la aplicación.

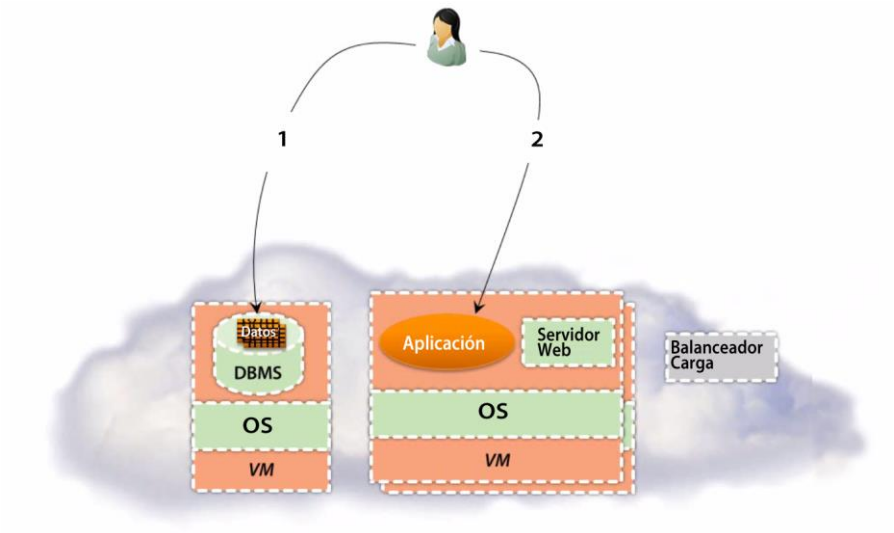

Ilustración 3 Esquema de despliegue PaaS

1.3.3. Software como servicio (SaaS)

Es un modelo de despliegue de software, donde un proveedor licencia (a demanda) a un consumidor el uso de una aplicación que ejecuta en infraestructura de la nube [2], [5].

Las aplicaciones pueden ser usadas desde diferentes tipos de dispositivos, livianos como un navegador web (email), o a través de programas. El consumidor no gestiona ni controla la funcionalidad de la aplicación ni la infraestructura en la nube donde ésta ejecuta.

La historia de SaaS comenzó con *SalesForce CRM*. Ellos son las personas que demostraron al mundo que SaaS realmente funciona como tecnológica y como modelo de negocios. En el siguiente diagrama se muestran algunos de los proveedores más importantes de software como servicio:

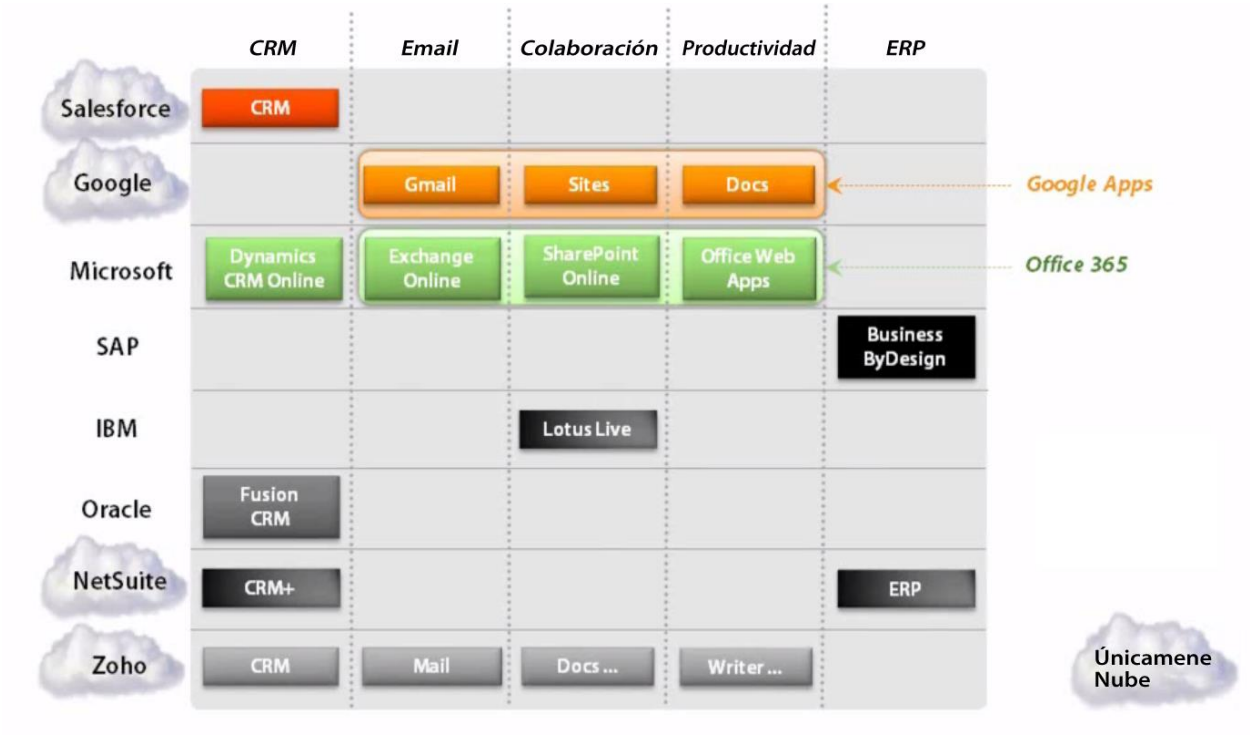

Ilustración 4 Proveedores de software como servicio

1.3.4. Proveedores y modelos de servicio

Existen tres proveedores de servicios en la nube que se destacan por su escala y trayectoria: Amazon, Microsoft y Google. Cada uno provee distintos modelos de servicio junto con un conjunto de tecnologías relacionadas.

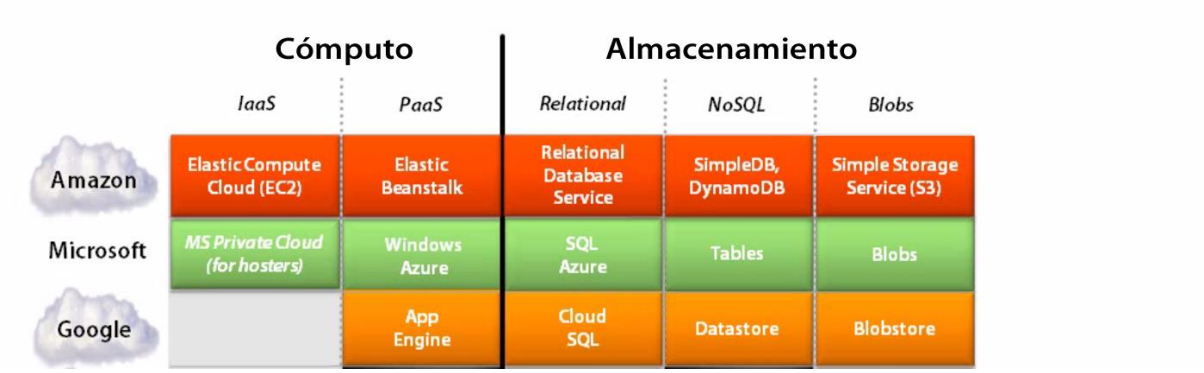

Ilustración 5 conjunto de tecnologías por proveedor

1.3.4.1 Amazon Web Services

Amazon Web Services (AWS) ofrece un conjunto completo de servicios de infraestructuras y aplicaciones que permiten ejecutar prácticamente cualquier tipo de aplicación en la nube. Desde aplicaciones empresariales y proyectos de *big data* hasta juegos sociales y aplicaciones móviles [6].

AWS tradicionalmente provee infraestructura como servicio, sin embargo en los últimos tiempos ha incorporado nuevas tecnologías para proveer plataforma como servicio. A continuación se describen algunas de las tecnologías más relevantes del proveedor:

Elastic compute cloud (infraestructura como servicio). Servicio web que permite lanzar y gestionar instancias de servidores de Linux/UNIX y Windows en centros de datos de Amazon [7].

Elastic Beanstalk (plataforma como servicio). Servicio que permite desplegar y administrar aplicaciones, sin la necesidad de gestionar la infraestructura donde ejecutan las mismas [8].

Relational Database Service. Se trata de un servicio web que permite de una forma sencilla configurar, operar y escalar una base de datos relacional en la nube [9].

SimpleDB / DynamoDB. Se trata de un almacén de datos no relacional, escalable, confiable y flexible que permite almacenar y consultar información utilizando servicios web [10].

Simple Storage Service. Es un servicio de almacenamiento que permite guardar y recuperar cualquier cantidad de datos en cualquier momento [11].

1.3.4.2 Windows Azure

Windows Azure es una plataforma en la nube abierta y flexible que permite compilar, desplegar, ejecutar y administrar aplicaciones rápidamente, en una red global de centros de datos administrados por Microsoft [12].

Windows Azure provee distintos modelos de servicio según las necesidades del cliente:

Máquinas virtuales (infraestructura como servicio). Permite lanzar y gestionar instancias de servidores de Windows y Linux en centros de datos de Microsoft [13].

Web Sites (plataforma como servicio). Permite construir aplicaciones web de cualquier tamaño utilizando herramientas y tecnologías conocidas. Las aplicaciones no requieren ser diseñadas específicamente para la nube, aunque hacerlo es recomendable. No es necesario la administración de la infraestructura, y es posible escalar hasta cierto grado la aplicación [14].

Cloud Services (plataforma como servicio). Permite ejecutar aplicaciones que requieren alto grado de disponibilidad y escalabilidad. Las aplicaciones requieren ser diseñadas específicamente para la nube. Una vez desplegada la aplicación, la plataforma administrará los detalles de implementación, desde el aprovisionamiento y equilibrio de carga hasta el seguimiento del estado, para ofrecer disponibilidad continua [15].

Al igual que en *Web Sites*, esta modalidad permite concentrarse en el desarrollo de la aplicación. El mantenimiento de la infraestructura subyacente es responsabilidad del proveedor.

A continuación se describen algunas de las tecnologías más relevantes del proveedor:

SQL Azure. Es un servicio de base de datos relacional muy completo y totalmente administrado que ofrece una experiencia de alta productividad, incorpora tecnología de SQL Server y ofrece funcionalidad de clase empresarial [16].

Tables. Las tablas ofrecen funcionalidad NoSQL para las aplicaciones que requieren almacenamiento de grandes cantidades de datos no estructurados. Las tablas son un servicio administrado que se pueden escalar automáticamente para satisfacer un rendimiento y volumen masivos [16].

Blobs: es un modo sencillo de almacenar grandes cantidades de texto no estructurado o datos binarios tales como vídeo, audio e imágenes. Los blobs son un servicio administrado que se pueden escalar automáticamente para satisfacer un rendimiento y volumen masivos [16].

1.3.4.3 Google App Engine

Google App Engine permite ejecutar aplicaciones web en la infraestructura de Google. Las aplicaciones son fáciles de crear, mantener y escalar en la medida que la demanda así lo requiera [17].

Google App Engine provee un modelo de plataforma como servicio. A continuación se describen algunas de las tecnologías más relevantes del proveedor:

Cloud SQL. Es un servicio web que permite crear, configurar y usar bases de datos relacionales almacenadas en la infraestructura de Google. Se trata de un servicio totalmente gestionado por el proveedor, que ofrece las características de una base de datos MySQL [18].

***Datastore*.** Se trata de un almacén de datos para objetos, carece de esquema, y es sumamente robusto y escalable [19].

Blobstore: permite almacenar grandes cantidades de datos binarios tales como vídeos, audio e imágenes. En general se utiliza para almacenar elementos que exceden el tamaño permitido en el *datastore* [20].

1.4. Modelos de despliegue

Según la ubicación de los recursos, y el acceso que se provea a los mismos se distinguen los diferentes modelos de despliegue.

1.4.1. Public Cloud

La infraestructura en la nube es provista para uso del público en general. Puede ser administrada u operada por una compañía, una institución educativa, una organización del gobierno o cualquier combinación de ellos. La infraestructura se ubica en las instalaciones del proveedor [1], [2].

1.4.2. Private Cloud

La infraestructura en la nube es provista para uso exclusivo de una organización. Puede ser administrada u operada por la organización o un tercero. La infraestructura generalmente se ubica en las instalaciones de la organización [1], [2].

En las organizaciones sin *private cloud*, el proceso para solicitar recursos de infraestructura se puede describir de la siguiente forma:

1. Un usuario de IT necesita la asignación de un recurso informático (en general una máquina virtual). Para ello solicita su acceso al administrador del centro de cómputo. La solicitud puede ser realizada utilizando distintos canales por ejemplo email.
2. Se verifica que la solicitud es válida. En organizaciones pequeñas este proceso puede ser resuelto rápidamente, incluso por el administrador del centro de cómputo. En organizaciones grandes, para que una VM pueda ser usada en producción es necesario mucho tiempo y la aprobación por distintos roles dentro de la organización.
3. El administrador de las VMs utiliza una herramienta de administración para crear manualmente el recurso a ser utilizado.

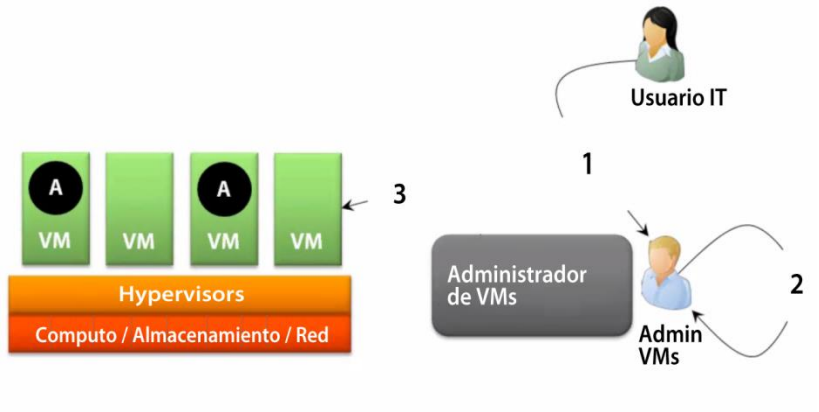

Ilustración 6 Proceso de asignación de recursos

Al utilizar *private cloud*, se importan las tecnologías de gestión de recursos disponibles en la nube pública a los data centers internos a una organización. De esta forma se logra reducir considerablemente los tiempos requeridos para la verificación de solicitudes. A continuación se describe el nuevo proceso:

1) El administrador de los recursos informáticos dispone de un conjunto de herramientas de gestión de máquinas virtuales y *cloud*. Haciendo uso de las mismas crea por adelantado un conjunto de *private clouds*.

2) Cuando un usuario de IT necesita recursos, los solicita directamente a través de un portal de self-service. Como los usuarios, permisos y *quotas* están predefinidos consigue su VM inmediatamente.

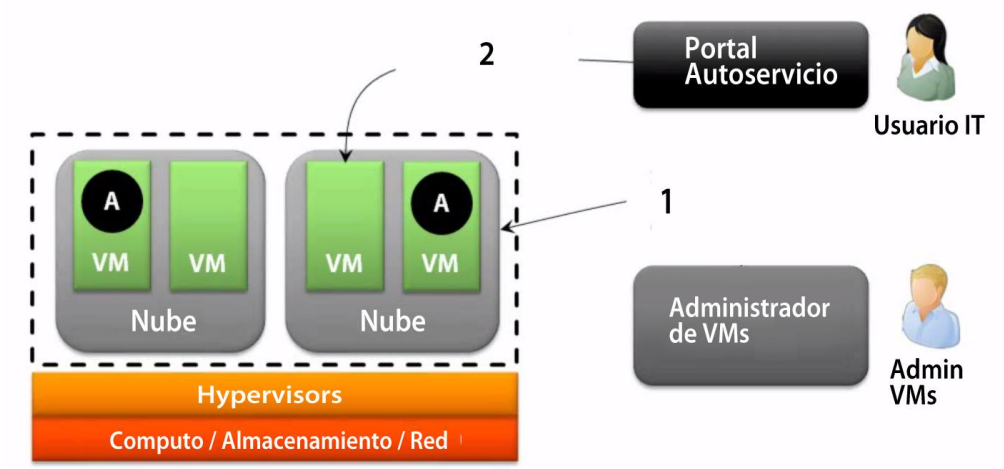

Ilustración 7 Proceso de asignación de recursos utilizando private cloud

Beneficios de *private cloud* para uso interno a una organización.

- Despliegue más rápido. Se trata de un proceso automático, se reduce el número de personas intermediarias. Existe una reducción de gastos administrativos.
- Menor cantidad de errores en el despliegue. Generalmente las personas tienden a cometer mayor número de errores que el software.
- Manejo de costos. Es posible saber para las organizaciones que unidades del negocio están utilizando los recursos, en que cantidad y por cuando tiempo.

1.4.3. Hybrid Cloud

La infraestructura está dada por una composición de infraestructuras de nube pública y privada. Cada una es administrada por entidades separadas, pero están conectadas a través de distintas tecnologías (estandarizadas o propietarias) que permite el balance de carga entre nubes [1], [2].

2. Arquitectura para la nube

Durante el desarrollo de este capítulo se introducen conceptos y definiciones que ayudan al lector a comprender los mecanismos que se emplean para diseñar aplicaciones optimizadas para la nube. Muchas de las técnicas serán retomadas y descriptas en detalle durante el desarrollo del capítulo de patrones.

Una aplicación optimizada para la nube es aquella cuyo diseño arquitectónico extrae el máximo beneficio de la infraestructura, y plataformas disponibles en la nube.

Estas aplicaciones en general tienen las siguientes características [21]:

- Emplean comunicación asincrónica no bloqueante. Los componentes comunicados están débilmente acoplados.
- Escalan horizontalmente agregando recursos (nodos) cuando la demanda incrementa y liberándolos cuando disminuye.
- Están optimizadas para ejecutar eficientemente, sin desperdiciar recursos ni dinero.
- Manejan cambios de escala sin tiempo fuera de servicio ni degradación de la experiencia del usuario.
- Manejan los fallos transitorios sin degradación de la experiencia de usuario.
- Distribuyen el contenido geográficamente para minimizar la latencia en las comunicaciones de red.
- Escala automáticamente de forma proactiva y reactiva.
- Monitorea los *logs* de la aplicación, incluso cuando los nodos son agregados y quitados dinámicamente.

2.1. Arquitectura de referencia

La arquitectura de un sistema puede estar compuesta por uno o más estilos, pero generalmente predomina uno.

Un estilo es una colección de decisiones de arquitectura que tienen un nombre y que son aplicables en un determinado contexto, restringen dentro de ese contexto las decisiones de arquitectura para un sistema, y proveen cualidades que son beneficiosas en cada sistema en el cual se aplica [22].

En aplicaciones desarrolladas para este contexto, la escalabilidad es un atributo de calidad fundamental. Por este motivo el procesamiento generalmente se distribuye en distintos nodos, dando lugar a una división en capas físicas: una capa web, una capa de negocios y una capa de datos. Cada capa puede crecer elásticamente a demanda.

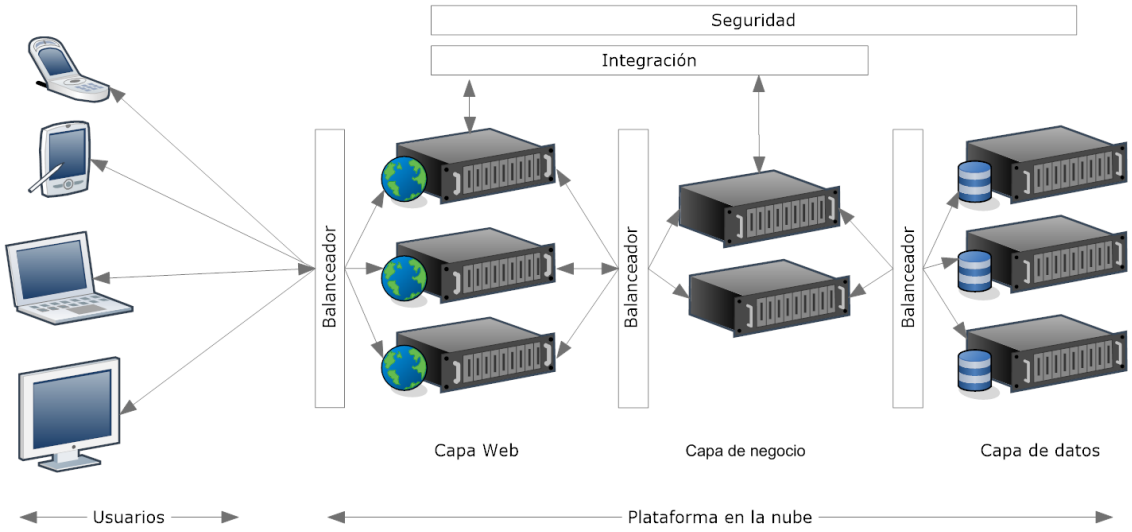

Ilustración 8 Arquitectura de referencia

2.1.1. Capa web o de presentación

La funcionalidad de la aplicación es entregada a los usuarios a través de una capa de presentación. Esta capa despliega los datos al usuario y permite operaciones para manipular e ingresar datos [23], [24].

El modelo típico de interacción con la nube es a través del uso de un navegador. HTTP es el protocolo de uso más común y HTML es un estándar que permite presentación consistente en distintos tipos de cliente. Sin embargo, estas tecnologías no garantizan una experiencia de usuario óptima. Por ello, otras técnicas se han desarrollado para mejorar el rendimiento, la capacidad de respuesta de las aplicaciones, y permitir comunicación en tiempo real. Ejemplos de estas tecnologías son:

- Javascript. El único lenguaje de programación soportado por la mayoría de los exploradores de internet.
- Ajax. (*Asynchronous JavaScript and XML*). Tecnología que permite, a través de la ejecución de código JavaScript del lado del cliente, continuamente adquirir contenido del servidor con el cual se puede actualizar la pantalla de forma transparente para el cliente.
- Tecnologías *Push*. Mecanismo de envío de datos desde un servidor web hacia un navegador web. Existen diversas herramientas para su implementación, entre ellas:
    - **Long Polling:** en esta modalidad el cliente inicia una conexión al servidor, si el servidor no posee información disponible para el cliente en lugar de enviar una respuesta vacía, guarda la petición y espera hasta que alguna información esté disponible. Cuando la información esté disponible el servidor envía una respuesta completa al cliente. Posteriormente el cliente inicia nuevamente una petición al servidor para que éste siempre tenga una petición en espera.
    - **WebSockets:** provee un canal de comunicación bidireccional, full-*duplex* utilizando un único socket TCP. La API de *websockets* está siendo estandarizada por W3C y IETF.

Aun así, a menudo existen razones para recurrir a tecnologías fuera del navegador. Los dispositivos móviles tienen un conjunto de características y limitaciones que los diferencian del modelo de escritorio. Dado que la movilidad representa un modelo de uso que está estrechamente alineado con la computación en nube, es importante no descuidar esta gran población de usuarios con necesidades especiales.

No obstante, en la mayoría de los casos una aplicación de navegador es más sencilla de implementar que un cliente nativo y llegará a más dispositivos. El desarrollo de estándares web, especialmente HTML5 y nuevas tecnologías incorporadas dentro del navegador permiten la creación de aplicaciones que provean una mejor experiencia [25].

La capa web en general consiste de un número variable de servidores web sin estado que aceptan solicitudes de usuarios externos. El estado, contexto, o sesión existe para proveer continuidad mientras un usuario utiliza una aplicación. Las técnicas para manejo de estado se describen en detalle en el capítulo 3.1 patrón para escalar horizontalmente.

Uno de los patrones más utilizados para la implementación de esta capa es *Model View Controller* el cual se describen en detalle en el capítulo 3.5.

2.1.2. Capa de negocio

La capa de negocio de la aplicación implementa la lógica del negocio y realiza la mayor parte del procesamiento. Esta capa generalmente es accesible desde la capa web u otras capas de servicio pero no por los usuarios directamente. Los nodos en esta capa no poseen estado [24].

En el capítulo 3.1 se describen en detalle las técnicas empleadas para escalar horizontalmente. En el capítulo 3.2 se describe la comunicación asincrónica de bajo acoplamiento utilizando colas de mensajes. Finalmente en el capítulo 3.3 se describe un patrón para el procesamiento de grandes volúmenes de datos en paralelo.

2.1.3. Capa de datos

La capa de datos almacena la información del negocio, sesiones, etc. en uno o más sistemas de almacenamiento, por ejemplo: bases de datos relacionales, *NoSQL*, y almacenamiento de archivos (*blobs*) [24].

En general las aplicaciones en la nube hacen un uso intensivo de datos, las soluciones son diversas pueden usar datos estructurados, sin estructura, estáticos, volátiles, y aplicar restricciones de integridad. En todos los casos el mayor desafío es el tamaño de los datos almacenados. Grandes conjuntos de datos requieren particionamiento, lo que conduce a un equilibrio entre la disponibilidad y la integridad transaccional [26].

A pesar que SQL (*Structured Query Language*) todavía gobierna la mayoría de las transacciones, existe una creciente necesidad de otras arquitecturas de datos. Estructuras de datos como grafos y árboles se utilizan extensivamente en algunas aplicaciones como las redes sociales. Existen muchas iniciativas para abandonar SQL debido a sus limitaciones de escalabilidad, las alternativas tienen un alto costo: implican una pérdida de muchas características útiles e incompatibilidad con código legado. Sin embargo, en algunos casos parecen ser las únicas opciones viables.

2.1.3.1 SQL

SQL ofrece un conjunto atractivo de funcionalidades y ventajas. Se trata de un lenguaje maduro que se basa en un modelo de datos probado. Las implementaciones comerciales y de código abierto son muy robustas. Fue adoptado como estándar por American National Standards Institute (ANSI) en 1986 y International Organization for Standardization (ISO) en 1987 [27].

Dada su amplia adopción y conjunto de características, SQL es la primera opción para cualquier aplicación basada en un modelo de datos relacional. SQL

proporciona herramientas probadas para realizar consultas complejas de datos transaccionales, normalizados y uniformes. Sin embargo, el trabajo adicional (*overhead*) asociado a asegurar la integridad de datos y esquema hace difícil manejar de forma eficiente grandes volúmenes de información, información no estructurada, que evoluciona rápidamente, o altamente distribuida.

Existen cuellos de botella que hacen difícil escalar horizontalmente las bases de datos SQL. Cuando una única base de datos maneja una gran cantidad de información y consultas, el primer paso para escalar es replicar los datos en varios servidores. Si esta técnica es aplicada correctamente se puede escalar el número de lecturas casi indefinidamente [27].

Sin embargo, es mucho más difícil cuando se trata de escalar una base de datos dinámica con muchas escrituras. Se puede recurrir a la replicación, pero inevitablemente esta se verá limitada por el principio de CAP (*Strong Consistency, High Availability, and Partition-resilience*) - solo se puede escoger dos de las tres dimensiones.

Si se particiona la base de datos en varios servidores entonces inevitablemente existe latencia y la posibilidad de que los enlaces fallen, lo que imposibilita garantizar a la vez disponibilidad y consistencia. Cuando una partición falla se puede escoger mantener la consistencia – bloquear la transacción y perder disponibilidad – o mantener la disponibilidad completando la transacción pero perder consistencia.

Una de las técnicas que permite escalar horizontalmente una base de datos se denomina *Sharding*. La misma se describe en el capítulo 3.4.

2.1.3.2 Pseudo SQL

Existen opciones de almacenamiento en la nube que se asemejan a bases de datos SQL, pero que no se adhieren al estándar completo. En dichas opciones la mayor parte de la información es tabular, pero típicamente no hay esquema y el sistema de gestión no admiten *joins*, claves foráneas, *triggers* o procedimientos almacenados [27].

El hecho de que las entidades pertenecientes a un almacén de datos no poseen esquema, implica que dos entidades de la misma naturaleza no están obligadas a poseer las mismas propiedades, ni tampoco utilizar los mismos tipos de valor. En su lugar, la aplicación es responsable de asegurar que las entidades se ajustan a los esquemas requeridos por la lógica de negocio.

La ausencia de sentencias avanzadas de SQL como JOIN se basa en lo ineficiente que resultan las consultas que abarcan más de una tabla / maquina especialmente si estas se distribuyen geográficamente. Como consecuencia solo es posible consultar tablas individuales y la aplicación es responsable de la agregación y filtrado de datos de múltiples tablas. Como contraparte los proveedores de este servicio ofrecen elasticidad y la habilidad para almacenar datos no estructurados (blobs)

Los proveedores más importantes de infraestructura y plataformas en la nube ofrecen este tipo de almacenamiento: *SimpleDB* (Amazon), *Datastore* (Google App Engine) y *Azure Storage* (Microsoft).

2.1.3.3 NoSQL

NoSQL es un tipo de almacenamiento de datos que pretende eliminar las restricciones de SQL para implementar modelos de datos más flexibles y escalables que son capaces de hacer frente a la demandas de la computación en nube. En general el término describe una base de datos sin esquema, horizontalmente escalable con replicación incorporada. En lugar de utilizar un lenguaje de consulta las aplicaciones interactúan con la base de datos a través de una API sencilla [28].

NoSQL resulta ideal para aplicaciones que requieren procesar inmensas cantidades de información (estructurada o no estructurada) pero que no requieren transacciones o asegurar la consistencia de datos.

2.1.4. Servicios de integración

La computación en la nube provee de un alto nivel de escalabilidad, de manera que los procesos no siempre pueden llevarse a cabo en una sola máquina. La utilización de arquitecturas que promueven un alto grado de descomposición de procesos en servicios granulares, permiten a las aplicaciones alcanzar la máxima eficiencia y flexibilidad. Dado que las aplicaciones no son auto contenido, existe la necesidad de un *framework* que permita la integración, simplificando las tareas de transferencia de información y procesamiento [29]. Para ello se emplean mecanismos de comunicación asincrónicos y de bajo acoplamiento como colas de mensajes las cuales se describen en detalle en el capítulo 3.2.

2.1.5. Balanceadores de carga

Un balanceador de carga fundamentalmente es un dispositivo de hardware o software que se pone al frente de un conjunto de servidores que atienden una aplicación. Su función es el balanceo de las solicitudes que llegan a los servidores usando algún algoritmo [30].

Existen distintos algoritmos para el balanceo de carga. El más simple de ellos se denomina *Round Robin* y asegura una distribución uniforme de las solicitudes entre los nodos disponibles.

2.1.6. Seguridad

A menos que una aplicación proporcione acceso anónimo de forma exclusiva, necesitará distinguir entre usuarios con el fin de optimizar la experiencia, personalizar el contenido y controlar el acceso a información sensible de carácter privado. En general, la información de identidad es almacenada en un directorio, el cual necesita escalar a tamaños considerables para soportar aplicaciones empresariales, o servicios de consumo globales [31].

La escalabilidad no es el único desafío. También es necesario simplificar la experiencia del usuario, reduciendo al mínimo posible la re-autenticación entre servicios, lo cual es difícil en una arquitectura multi-proveedor distribuida. En base a este nuevo requerimiento surgen los mecanismos federados de gestión de identidad.

Finalmente además de proveer seguridad, surge la posibilidad de utilizar las preferencias de usuario, contexto e historial para maximizar la accesibilidad, mejorar la eficiencia, y generar contenido personalizado que se adapte a la ubicación geográfica del usuario.

Con el fin de lograr los puntos descriptos anteriormente, es necesario para los proveedores de servicios estandarizar mecanismos a-demanda para el intercambio entre sí de información de autenticación, autorización y contabilidad (*Authentication, Authorization and Accounting* (AAA) - del término en inglés).

Autenticación.

La autenticación describe el proceso de verificar la identidad de un individuo. Típicamente los usuarios proveen credenciales asociadas a su identidad digital [31]. Estas credenciales se puede clasificar como:

- Lo que se sabe (identificación personal, *passwords*, etc)
- Lo que se posee (un certificado digital, o *token* criptográfico)
- Lo que se es (huella dactilar, iris ocular)

Cada uno de estos tiene fortalezas y debilidades, por ello los sistemas de autenticación seguros combinan al menos dos de estos mecanismos.

Autorización

La autorización refiere al mecanismo para establecer si la entidad autenticada tiene permisos para ejecutar una actividad particular, acceder a un recurso o servicio determinado. Además de la identidad del usuario, los criterios pueden incluir una combinación de la ubicación física, hora del día y del histórico pasado de acceso [31].

2.1.6.1 Soluciones de identificación federadas

No hay falta de estándares relativos a la gestión de identidades. Tecnologías de federación como *OpenID*, *Security Assertion Markup Language (SAML), OAuth* y tarjetas de información suelen ofrecer una combinación de autenticación, autorización y servicios de personalización [31].

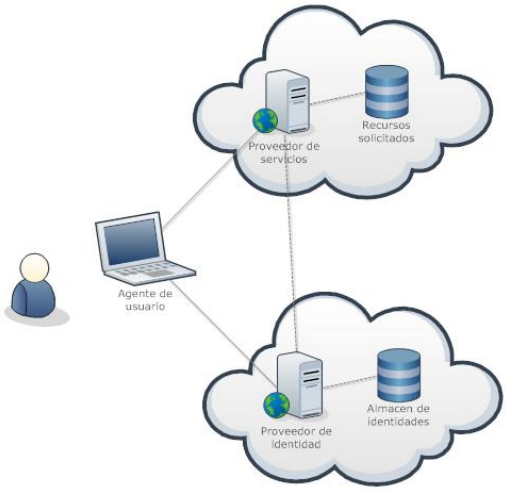

Ilustración 9 Soluciones de identificación federadas

Como se muestra en la figura de arriba, la premisa fundamental de los sistemas de identidad federada es que los servicios solicitados y el almacén de usuarios residen en dominios diferentes. Por lo tanto existe la necesidad de que la entidad que gestiona el servicio valide la identidad del usuario a través del proveedor de identidad.

Esto significa que la aplicación del cliente (agente de usuario) obtiene las credenciales del usuario final y las transfiere al proveedor de identidad para su validación. El proveedor de identidad informa al proveedor de servicio que este se ha autenticado. Si bien es posible la comunicación directa entre el proveedor de identidad y el de servicio, es más común proveer la validación a través del agente de usuario haciendo uso de un *token* de seguridad firmado digitalmente por el proveedor de identidad.

Inicio único de sesión (*Single sign on* (SSO) - del término en inglés) es un mecanismo que se utiliza para facilitar la identificación, la autenticación y autorización del usuario. En una empresa, se basa generalmente en un almacén de identidades, como *Microsoft Active Directory o Sun Identity Manager*. En la nube, se ha vuelto cada vez más frecuente utilizar los almacenes de identidad de grandes aplicaciones como Google Apps, Salesforce.com, Facebook o Twitter.

A pesar de que los estándares de federación son lo suficientemente robustos para soportar múltiples escenarios empresariales incluyendo SaaS, La autorización granular a componentes de una aplicación todavía es relativamente nuevo e inmaduro.

OAuth también está ganando popularidad debido a su simplicidad. Provee de un medio que permite a una aplicación acceder a datos (imágenes, documentos, contactos, etc.), almacenados en aplicaciones de otro proveedor de servicios, sin revelar las credenciales del usuario. En general al solicitar acceso a un recurso una nueva ventana del explorador permite proporcionar las credenciales directamente a la fuente de datos. Por ejemplo, si una aplicación necesita acceso a Google Mail primero redirige al usuario a una URL segura de Google. Luego de una autenticación exitosa, el proveedor de almacenamiento libera los datos solicitados para su consumo.

2.1.6.2 OpenID

OpenID proporciona un *framework* que permite establecer una conexión con un proveedor (por ejemplo Twitter, o Google) y proporcionar capacidades de inicio de sesión en otras aplicaciones que aceptan *OpenID* como mecanismo de autenticación. Formalmente *OpenID* es un sistema de identidad federado usado para autenticar usuarios en distintos tipos de servicios y aplicaciones [31].

Los pasos para completar el proceso son los siguientes:

1. El usuario crea una cuenta con un proveedor de identidad *OpenID* (por ejemplo Google, Yahoo, Verisign, etc).
2. El usuario visita un sitio que utiliza *OpenID* como mecanismo de autenticación (por ejemplo Facebook).
3. El sitio ofrece múltiples opciones para iniciar sesión, entre ellas un link al proveedor de identidad *OpenID*.
4. El sitio redirige al usuario al proveedor de identidad por autenticación.
5. El usuario se identifica contra el proveedor de identidad.
6. El proveedor de identidad confirma los credenciales del usuario al sitio que las solicitó a través de un *token* de seguridad.
7. El usuario puede utilizar el sitio sin proveer una contraseña.

OpenID provee un solo login para múltiples sitios / aplicaciones. Cada vez que el usuario desea iniciar sesión, es redirigido al proveedor de identidad *OpenID* donde inicia sesión para luego ser redirigido nuevamente al sitio original. En algunos casos, la autenticación puede ocurrir transparentemente. Por ejemplo es posible conectar una cuenta de Facebook con una de Google, y luego iniciar sesión automáticamente en Facebook si ya lo habías hecho anteriormente en Google.

2.1.6.3 OAuth

OAuth especifica un protocolo de autorización estándar, que permite a los usuarios conceder permisos de acceso en forma granular a recursos gestionados por otros servicios. En el pasado era necesario proveer credenciales (usuario y contraseña) a cada uno de los sitios que requería acceso a recursos. Este último sistema posee dos desventajas: la contraseña puede ser comprometida y no hay restricciones en lo que se puede hacer con la contraseña [31].

OAuth soluciona estos problemas:

1. Permite compartir recursos entre diferentes sitios sin exponer la contraseña del usuario.
2. Provee acceso de alta granularidad a recursos y servicios. El usuario puede especificar exactamente aquellos recursos que desea permitir acceso.

OAuth provee un mecanismo para autorizar a un sitio (el consumidor) a acceder contenido u información de otro sitio (el proveedor). Por ejemplo se puede permitir que un servicio de redes sociales como Facebook, acceda a imágenes de un repositorio de fotos como *Flickr*. En este caso Facebook es el consumidor y *Flickr* el proveedor.

En estos casos, el usuario inicia sesión en el proveedor, luego éste indica al usuario que el consumidor desea acceder a recursos específicos y solicita autorización. Si el usuario concede la autorización es redirigido al consumidor para que este los utilice.

2.1.7. Escalabilidad

La escalabilidad comprende la asignación de recursos a una aplicación, y su gestión eficiente para minimizar su agotamiento. La experiencia del usuario se ve afectada negativamente cuando una aplicación requiere más recursos de los que se encuentran disponibles.

Existen dos enfoques que permiten escalar se denominan: escalado vertical y escalado horizontal. El primero generalmente es sencillo pero limitado. El segundo es de mayor complejidad pero permite escalar a niveles que exceden ampliamente las posibilidades del primero [32].

2.1.7.1 Definición

La escalabilidad de una aplicación está asociada al número de usuarios que puede soportar efectivamente en un momento dado.

Una definición de mayor rigurosidad considera los siguientes conceptos:

- Usuarios concurrentes: el número de usuarios con actividad en un intervalo de tiempo determinado (por ejemplo cinco segundos).
- Tiempo de respuesta: el tiempo transcurrido entre que un usuario inicia una petición (por ejemplo solicita la página principal) y recibe su respuesta.

Ejemplo: Con 200 usuarios concurrentes, el tiempo de respuesta será inferior a 2 segundos el 70% del tiempo.

El punto donde una aplicación no puede manejar efectivamente usuarios adicionales constituye el límite de escalabilidad. Este límite es alcanzado cuando se agota un recurso crítico de hardware por ejemplo CPU, memoria, disco, conectividad de red, etc.

La escalabilidad de una aplicación puede ser mejorada asignando recursos adicionales de hardware, siempre y cuando la aplicación pueda hacer uso efectivo de los mismos.

La forma en la que se adicionan recursos determina el enfoque que se utiliza para escalar.

- Escalar verticalmente implica incrementar la capacidad de la aplicación a través del aumento de la capacidad de los nodos existentes en los que ejecuta.
- Escalar horizontalmente implica incrementar la capacidad de la aplicación a través de la incorporación de nuevos nodos.

Es importante destacar que estos enfoques no son excluyentes. Una aplicación o sus módulos pueden ser capaces de escalar verticalmente, horizontalmente, ambos, o ninguno.

2.1.7.2 Escalabilidad vertical

La idea principal es incrementar la capacidad de los nodos en donde ejecuta la aplicación mejorando su hardware. Por ejemplo agregar memoria, incrementar el número de procesadores, etc. Debido a que se trata de cambios en el hardware este enfoque implica tiempo fuera de servicio y es limitado [32].

2.1.7.3 Escalabilidad horizontal

La idea principal es incrementar la capacidad agregando nodos nuevos.

Los desafíos arquitectónicos de escalar verticalmente son diferentes a aquellos relacionados con escalar horizontalmente. El foco cambia de la maximización de la capacidad de componentes individuales a la combinación de capacidades de un conjunto de nodos [32].

Escalar horizontalmente suele ser más complejo que escalar verticalmente, y tiene un mayor impacto en la arquitectura de una aplicación.

Las aplicaciones diseñadas para escalar horizontalmente generalmente poseen nodos especializados en funciones concretas. Por ejemplo: servidores web y servidores de agregación de datos. En ese sentido al incrementar la capacidad adicionando nodos, estos son asignados a funciones específicas.

Cuando todos los nodos asignados a una misma función están configurados idénticamente: mismo hardware, mismo sistema operativo, misma función los nodos son homogéneos.

No todos los nodos de una aplicación son homogéneos, solamente lo pueden ser aquellos dentro de la misma función. Por ejemplo todos los servidores web pueden ser homogéneos, pero la configuración de los servidores web no tiene que ser igual a los servidores de agregación de datos.

Cuando se escala horizontalmente se adicionan nodos de múltiples tipos, esto introduce una cantidad predecible de capacidad en la aplicación. Por ejemplo cada 200 usuarios, se necesita un servidor web, un servidor de agregación de datos, y 1TB de disco. Los recursos que deben ser escalados en conjunto se denominan *scaling unit*. Este concepto es frecuentemente utilizado al escalar automáticamente, donde las unidades son agregadas y quitadas en conjunto [33].

2.1.7.4 Performance y escalabilidad

Generalmente la velocidad de una aplicación se confunde con su escalabilidad.

Performance refiere a la experiencia de un usuario individual. Para satisfacer una solicitud (*request*) se debe incurrir en un número de tareas (acceso a datos, generación de página web, envió de imágenes, etc.). El intervalo de tiempo requerido para completar cada una de estas tareas limita la performance [34].

Escalabilidad refiere al número de usuarios que aprecian una experiencia positiva. Si la aplicación provee performance consistente para cada uno de sus usuarios y el número de usuarios concurrentes crece, entonces la aplicación está escalando [35].

Por ejemplo: si el tiempo promedio de respuesta de una aplicación es 2 segundos con 20 usuarios, pero el tiempo promedio aumenta a 10 segundos con 200 usuarios concurrentes, entonces la aplicación no está escalando.

Una aplicación puede escalar correctamente (manejar el aumento de usuarios concurrentes manteniendo la performance consistente), sin poseer buena performance (la performance consistente es muy lenta sin importar el número de usuarios).

3. Patrones

Un patrón es una colección de decisiones de arquitectura que son aplicables a un problema de diseño recurrente y parametrizables para contemplar distintos contextos en el cual el problema aparece.

Este capítulo propone, en cada una de sus secciones, un patrón para el diseño de arquitecturas en la nube. Estos patrones son extraídos empíricamente y su uso favorece a diferentes atributos de calidad de la aplicación.

3.1. Patrón para escalar horizontalmente

Este patrón se enfoca en aumentar la capacidad de una aplicación agregando nodos. Los principales aspectos relevantes son la utilización eficiente de recursos y la eficiencia operativa.

La clave para el uso eficiente de recursos es emplear nodos autónomos sin estado. Utilizar nodos sin estado no implica una aplicación sin estado. El estado relevante para el funcionamiento de la aplicación puede ser almacenado externamente, por ejemplo en un cache distribuido, o servicio de almacenamiento.

3.1.1. Contexto / Aplicabilidad

El patrón para escalar horizontalmente se utiliza para encarar los siguientes desafíos:

- Es necesario escalar de forma eficiente el número de nodos de cómputo asignados.
- Los requerimientos de capacidad de la aplicación exceden la capacidad del nodo de computación más grande disponible.
- Los requerimientos de capacidad de la aplicación varían constantemente: cada día, cada mes, temporada, o pueden tener picos de tráfico variable.

Las plataformas de servicios en la nube están optimizadas para escalar horizontalmente. Instanciar un nodo es tan sencillo como instanciar doscientos, liberar un nodo es igual de sencillo. La plataforma asegura que todos los nodos serán instanciados con la imagen necesaria, ofrece los servicios necesarios para su gestión y provee balanceo de carga.

Atributos de calidad involucrados: disponibilidad, optimización de costos, escalabilidad y performance.

3.1.2. Mecanismo

Para escalar horizontalmente una aplicación (agregar o quitar nodos de cómputo), se puede utilizar la API expuesta por el proveedor de servicios en la nube, su interfaz de usuario, o una herramienta propietaria.

El proceso es sumamente sencillo. Sin embargo, debido a que los nodos son dinámicamente asignados y liberados es necesario prestar especial atención al manejo de estado de sesión, y mantenimiento de la eficiencia operativa. También es importante comprender que contar con recursos elásticos y no fijos tiene asociado un ahorro de dinero [36].

3.1.2.1 Manejo de estado

El estado, contexto, o sesión existe para proveer continuidad mientras un usuario utiliza una aplicación. Por ejemplo en una tienda virtual la información de estado incluye los productos que el usuario ha añadido al carrito, información de inicio de sesión, datos básicos como su nombre, etc.

Cuando una aplicación posee más de un nodo en la capa web, es necesario contar con un mecanismo para dirigir las visitas a uno de los nodos disponibles. Dicha tarea frecuentemente se logra usando un balanceador de carga. Cuando un usuario visita por primera vez un sitio, generalmente el balanceador de carga lo dirige a uno de los nodos disponibles usando un algoritmo sencillo como *round-robin* (el cual distribuye la carga de forma uniforme). El algoritmo o mecanismo empleado para manejar solicitudes posteriores pertenecientes al mismo usuario, está estrictamente relacionado a la forma en que se maneja el estado de sesión [37].

Uso de *sticky sessions* en la capa web.

Algunas aplicaciones web utilizan *sticky sessions*, en esta modalidad un servidor web es asignado al usuario durante su primera visita. Una vez asignado, el nodo procesara y dará respuesta a todas las solicitudes del usuario durante esa sesión. Para lograr este comportamiento se necesitan dos elementos: el balanceador de carga debe asegurar que cada visitante es dirigido al nodo que le fue asignado, y el nodo asignado debe mantener estado de sesión para el usuario.

Un nodo que mantiene la única copia de la información de sesión del usuario constituye un único punto de falla. Si el nodo falla, la información se pierde. Esto puede afectar negativamente la experiencia del usuario, por ejemplo la aplicación requiere que inicie sesión nuevamente [38].

Nodos sin estado en la capa web

Una solución diseñada para la nube, mantiene el estado de sesión del usuario utilizando nodos sin estado. Un nodo se mantiene sin estado al almacenar toda la información de sesión externamente.

Las aplicaciones que poseen estado de sesión muy pequeño, podrán almacenarlo en una *cookie*. Esto elimina completamente la necesidad de almacenar información de estado dentro de los límites de la aplicación. Cada vez que el usuario realiza una solicitud la cookie es enviada.

En la mayoría de los contextos, las *cookies* son demasiado pequeñas o ineficientes para almacenar la información de sesión. En estos casos las *cookies* almacenan un identificador de sesión generado por la aplicación en lugar de los datos en sí mismos. A través del identificador, la aplicación puede recuperar la información de sesión necesaria al inicio de cada solicitud, hacer uso de la misma, y almacenarla nuevamente al final. Para guardar información de sesión se pueden utilizar soluciones provistas por la plataforma en la nube, por ejemplo bases de datos NoSQL o caches distribuidos [37].

3.1.2.2 Administración de múltiples nodos

En una aplicación compleja preparada para la nube, seguramente existan muchos tipos de nodos, y muchas instancias de cada tipo colaborando para su funcionamiento. Los nodos son asignados dinámicamente, y por tanto su número cambia constantemente. En este contexto resulta importante su correcta administración [36].

Definición del tamaño de las máquinas virtuales

Determinar el tamaño adecuado de las máquinas virtuales necesarias para ejecutar cada parte de la aplicación, resulta uno de los aspectos más importantes para escalar horizontalmente. Si se escoge una maquina demasiado pequeña la aplicación no tendrá una performance aceptable y experimentara fallos constantes. Si la maquina es demasiado grande se incurre en costos innecesarios.

Los fallos son parciales

Una capa web que posee varios nodos puede temporalmente perder uno y continuar funcionando correctamente sin degradar la experiencia del usuario. Este comportamiento es diferente al observado cuando se escala verticalmente un solo nodo, el cual constituye un único punto de falla.

Recolección de datos de funcionamiento

Cada uno de los nodos que ejecuta una aplicación genera datos de funcionamiento. *Logear* información directamente en el nodo resulta una forma eficiente de reunir información, pero no es suficiente. La bitácora generada debe ser recolectada, agregada y analizada.

3.2. Flujos de trabajo con colas de mensajes

Este patrón se enfoca en la entrega asincrónica de solicitudes de trabajo creadas en la capa web (donde reside la interfaz de usuario), y enviadas a la capa de servicio para su procesamiento. Además es esencial para lograr bajo acoplamiento en las comunicaciones entre las distintas capas de una aplicación en la nube.

El patrón se usa frecuentemente para permitir que los usuarios realicen actualizaciones a través de la capa web sin demorar o bloquear los servidores web. Esto es especialmente útil para procesar actualizaciones que requieran mucho tiempo, recursos, o dependen de servicios de terceros no siempre disponibles. Por ejemplo, una tienda de compras debe autorizar la transacción con el mecanismo de pago empleado, registrar los cambios de stock, notificar al usuario a través de mail, etc. Si todas las tareas fueran realizadas en la capa web, sus recursos estarían ocupados por mucho más tiempo y no podrían atender nuevas solicitudes [39].

A gran nivel el mecanismo es el siguiente: un usuario emite una solicitud a la aplicación, la solicitud es manejada en primera instancia por la interfaz del usuario (capa web), ella crea un mensaje describiendo el trabajo que se debe completar para cumplir con la solicitud. Este mensaje es agregado a una cola. En algún momento futuro, un proceso en otro nodo (capa de servicios) quita el mensaje de la cola y realiza el trabajo necesario. Los mensajes fluyen en una sola dirección, desde la capa web hacia la cola y luego a la capa de servicios [39].

Se trata de un modelo asincrónico debido a que el emisor de la solicitud no espera una respuesta inmediata.

3.2.1. Contexto / Aplicabilidad

El patrón se utiliza para encarar los siguientes desafíos:

- La aplicación está desacoplada en capas físicas, y se requiere interacción entre ellas.
- La aplicación debe garantizar que los mensajes son procesados al menos una vez.
- Se desea mantener la performance de la interfaz de usuario, y realizar el trabajo dependiente en otras capas.
- Se desea mantener la performance de la interfaz de usuario, aun cuando se utilizan servicios de terceros durante el procesamiento.

Los proveedores de servicio en la nube ofrecen colas de mensajes confiables como servicio. Por esta razón los aspectos de infraestructura asociados al patrón son generalmente sencillos de implementar, fuera de la nube resultan relativamente complejos.

Atributos de calidad involucrados: disponibilidad, confiabilidad, escalabilidad, performance, interoperabilidad, y experiencia de usuario.

3.2.2. Mecanismo

Las aplicaciones que no utilizan este patrón, responden a las solicitudes del usuario directamente desde la capa web. Lo anterior implica que la interfaz de usuario invoque directamente a la capa de servicios. Este enfoque es sencillo, pero tiene sus desventajas en especial para sistemas distribuidos que necesitan escalar. Uno de los problemas es que los llamados a la capa de servicio deben finalizar antes que la capa web pueda responder a la solicitud inicial del usuario. Además es necesario que la escalabilidad y disponibilidad de la capa de servicios iguale o exceda a la de la capa web, de lo contrario se puede arruinar la experiencia del usuario. Conseguir que la capa de servicios iguale a la web en cuanto a disponibilidad y escalabilidad es sumamente difícil considerando que se pueden realizar tareas que utilicen servicios de terceros, o requieren un gran procesamiento [40], [41].

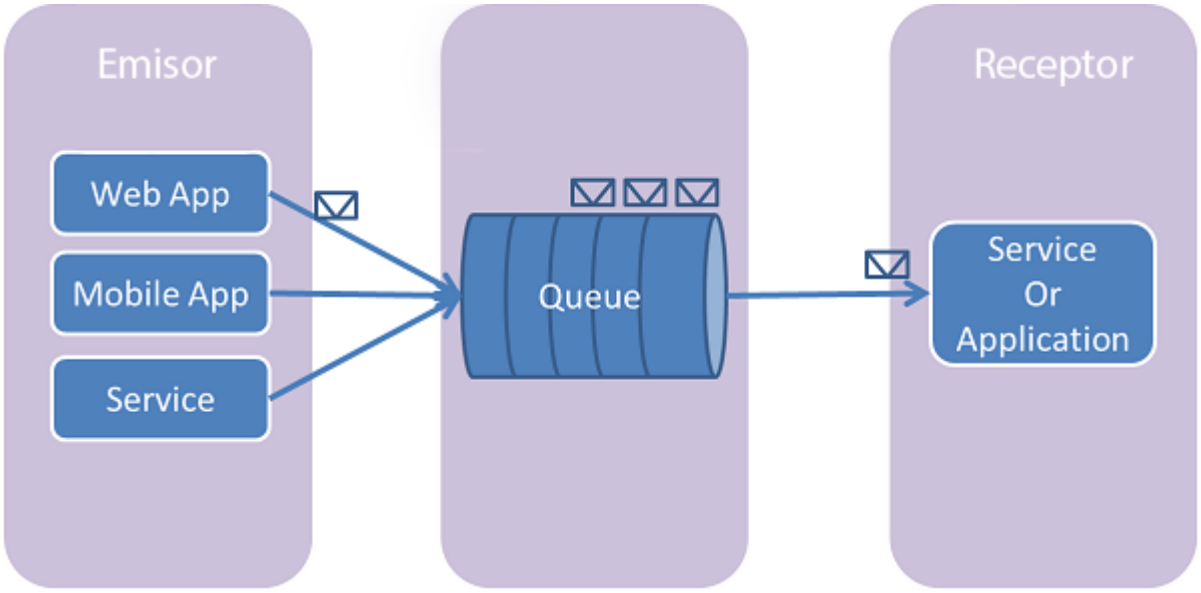

Ilustración 10 Mecanismo de comunicación utilizando colas de mensajes [42]

La solución es la comunicación asincrónica. La capa web envía comandos a la capa de servicios, un comando es una solicitud de trabajo. Por ejemplo: crear una nueva cuenta de usuario, agregar un video, confirmar una compra, etc.

Los comandos son enviados como mensajes utilizando una cola. Una cola es una estructura de datos sencilla que posee dos operaciones fundamentales: agregar y quitar elementos.

En la mayoría de los casos el patrón es sencillo de implementar. Un emisor agrega comandos en forma de mensajes a la cola, luego un receptor procesa y quita los mensajes.

El emisor y receptor están desacoplados. Solo se comunican a través de la cola de mensajes, como consecuencia existe un gran número de importantes beneficios:

- El emisor y receptor pueden operar a diferentes velocidades.
- El emisor puede enviar mensajes aun cuando el receptor no se encuentra operativo. De la misma forma el receptor puede procesar mensajes aun cuando el emisor no está operativo.
- Emisor y receptor no conocen detalles acerca de la implementación del otro. Solo deben acordar que cola de mensajes utilizar y la estructura de cada mensaje.
- Existe flexibilidad para implementar las distintas capas, cada una puede utilizar tecnologías diferentes.

3.2.2.1 Modelo de uso

El modelo para implementar un receptor de mensajes de una cola confiable es el siguiente:

1. Conseguir el próximo mensaje disponible de la cola.
2. Procesar el mensaje.
3. Quitar el mensaje de la cola.

Al menos un procesamiento.

Procesar un comando implica conseguir el próximo mensaje de la cola, entender el trabajo requerido, completarlo y finalmente si todo termina correctamente eliminar el mensaje. Solo en ese punto un comando se considera completo.

Bajo algunas circunstancias, se pueden presentar fallos mientras se completa el trabajo requerido por un comando.

Cuando se obtiene un mensaje de la cola, este no es quitado inmediatamente de la misma, en su lugar el mensaje se oculta. Los mensajes se ocultan por un determinado periodo de tiempo (determinado por las características de la cola). Mientras que un mensaje se encuentra oculto ningún otro nodo lo procesara.

Cuando un mensaje toma más tiempo del permitido en ser procesado, automáticamente será retornado a un estado visible en la cola. De esa forma otro nodo podrá tomarlo y procesarlo, esto es clave para asegurar la resistencia a fallos pero puede causar que un mensaje se procese más de una sola vez. Cualquier mensaje que no haya sido procesado por completo, eventualmente será procesado por otro nodo [43].

Procesamiento de mensajes repetidos.

Cualquier mensaje que se procese más de una vez, puede haber sido parcialmente procesado la vez anterior. Esto puede causar problemas si no se toma en cuenta.

Cuando una operación es ídem potente, no se presentan inconvenientes. Cuando no lo es, hay que considerar que parte del procesamiento pudo haber sido realizado anteriormente y en base a ello tomar decisiones [43].

Mensajes envenenados

Algunos mensajes no pueden ser procesados correctamente debido a su contenido. Estos mensajes se conocen como "envenenados".

Por ejemplo, al procesar un video se encuentra que el archivo tiene un segmento corrupto. Generalmente se tratan de casos límite que la aplicación no considera.

Si la aplicación falla mientras procesa un mensaje, eventualmente el mensaje será visible y otro nodo lo procesara. Es necesario implementar lógica adicional que tome en cuenta este caso y tome medidas adecuadas. Por ejemplo cuando un mensaje reaparece por quinta vez consecutiva debe ser tratado como envenenado.

Luego de que un mensaje de este tipo es identificado, su tratamiento es una decisión de negocio. Desde el punto de vista de diseño, el mensaje debe ser quitado de la cola lo antes posible [43].

Confiabilidad

Las colas de mensajes suministradas por los proveedores de servicio en la nube son confiables. La confiabilidad implica durabilidad de la información (los datos no se pierden) y alto rendimiento (al menos cientos de interacciones por segundo).

Señales para escalar de forma reactiva

El número de mensajes en la cola y el tiempo que cada mensaje permanece en ella antes de ser quitado son señales útiles para escalar de forma reactiva.

3.3. MapReduce

MapReduce es un mecanismo simple de programación que permite procesar grandes volúmenes de datos de forma paralela. Se implementa como un clúster, conformado por varios nodos trabajando de forma conjunta en diferentes subconjuntos de información. Al dividir el trabajo entre los nodos disponibles, se puede completar más rápidamente [44].

Para utilizar esta técnica se deben escribir dos funciones: un trasformador *(mapper)* y un reductor *(reducer)*. Estas funciones aceptan un conjunto de datos como entrada y retornan una transformación de los mismos. La salida del *mappper* es agregada y luego enviada al *reducer,* este proceso se repite con pequeños conjuntos de datos por vez, hasta procesar la totalidad de los datos deseados.

MapReduce está diseñado para procesar por lotes grandes conjuntos de datos. El factor que limita la capacidad de la herramienta es el número de nodos que conforman el clúster. Las funciones (*mapper y reducer*) pueden ser escritas para trabajar con pequeños conjuntos de datos, y luego no es necesario adaptarlas para trabajar con conjuntos de datos inmensos [44].

Ejemplos de los resultados de la información procesada empleando esta técnica incluyen: el robot encargado de indexar páginas de Google, recomendaciones personalizadas en una tienda, sugerencias de conexiones sociales o profesionales.

3.3.1. Contexto / Aplicabilidad

El patrón se utiliza para encarar los siguientes desafíos:

- La aplicación debe procesar grandes volúmenes de datos almacenados en la nube. Los datos pueden poseer una estructura definida o no.
- La aplicación necita generar reportes que los sistemas tradicionales de bases de datos no pueden generar, ya sea debido a que el conjunto de datos es muy grande, o no posee una estructura compatible.
- Los datos son independientes y pueden ser procesados en cualquier orden.

Los principales proveedores de plataformas en la nube provén *MapReduce* como un servicio a demanda. Windows Azure y Amazon Web Services se basan en un estándar de código abierto denominado Apache Hadoop.

Uno de las grandes dificultades que se presentan al utilizar esta técnica, es conseguir que el gran volumen de información se encuentre accesible a los nodos encargados de su procesamiento. Los proveedores de servicios en la nube son muy eficientes al manejar grandes volúmenes de datos, además permiten arrendar cientos de nodos por intervalos de tiempo reducidos. Por estos motivos son un lugar ideal para realizar este tipo de procesamiento.

3.3.2. Mecanismo

Esta técnica tiene su origen en la programación funcional. Una función transformadora (*map)* se aplica a cada elemento de una lista produciendo una nueva lista de elementos (proyección). Luego una función reductora (*reducer*) es aplicada sobre cada elemento de la última lista produciendo un único valor escalar. En el patrón, la lista de elementos puede poseer texto, números, incluso archivos.

Los datos son divididos en pequeñas unidades, y distribuidos entre los nodos que conforman el clúster de procesamiento. El tamaño de un clúster puede ser de varios cientos de nodos. Cada nodo recibe un subconjunto de los datos y aplica las funciones *map y reduce* correspondientes. Finalmente, los resultados de cada nodo pueden convertirse en la entrada de otro par de funciones *map y reduce* que agregan un nuevo resultado hasta alcanzar el final [45].

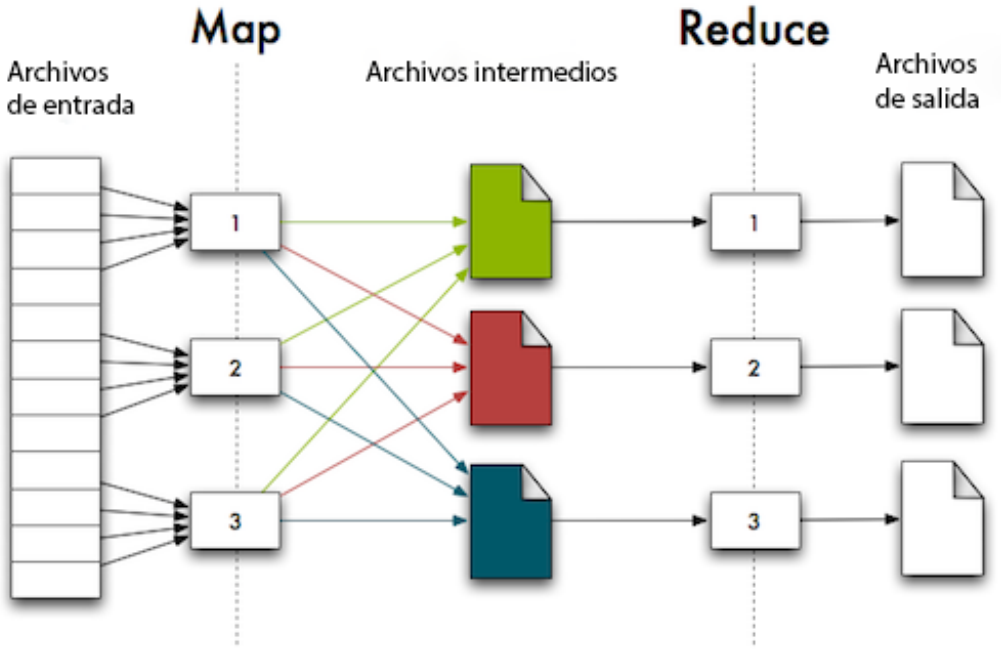

Ilustración 11 Mecanismo de procesamiento de datos con MapReduce [46]

3.4. Sharding

En respuesta a las necesidades de escalar las bases de datos SQL, surge este patrón como mecanismo para escalar horizontalmente la capa de datos.

La idea es comenzar con una sola base de datos, y luego dividir sus datos entre varias (denominadas *shards*) según sea necesario. Cada *shard* posee exactamente el mismo esquema que la base de datos original. La mayor parte de los datos son distribuidos de forma tal que cada tupla pertenezca a un único shard. La unión de todos los datos contenidos en cada *shard* conforma el conjunto completo de datos [47].

3.4.1. Contexto / Aplicabilidad

El patrón se utiliza para encarar los siguientes desafíos:

- El volumen de consultas o actualizaciones a la base de datos de la aplicación excede la capacidad de un solo servidor de bases de datos.
- El tiempo de respuesta es alto por congestionamiento.
- Los requerimientos de conectividad de red para transferencia de datos, o almacenamiento exceden los disponibles en un solo nodo.

Los proveedores de infraestructura en la nube generalmente tienen servicios de bases de datos que ofrecen *sharding* integrado. Sin soporte nativo para esta tecnología, la misma se debe implementar directamente en la aplicación lo cual hace su uso mucho más complejo [48].

3.4.2. Mecanismo

Las bases de datos tradicionales *(sin sharding)* son desplegadas en un solo nodo dedicado a la tarea. Cualquier base de datos ejecutando en un nodo es limitada por la capacidad del mismo.

Existen muchos enfoques para escalar las bases de datos de una aplicación. Por ejemplo: distribuir las consultas entre nodos esclavo, dividir los datos en varios servidores por el tipo de datos, y escalar verticalmente el servidor de bases de datos.

Para manejar alta carga de consultas (solo lectura), se pueden replicar nodos esclavos a partir de una base de datos maestra. Los nodos esclavo son de solo lectura y eventualmente consistentes (pueden demorar en reflejar actualizaciones de la base de datos maestra).

Otra opción es dividir la información entre múltiples bases de datos de acuerdo al tipo de dato. Por ejemplo almacenar productos en una base de datos y usuarios en otra.

El patrón de *sharding* está enfocado en escalabilidad horizontal, permite crecer mucho más allá de las posibilidades de un único nodo mediante la distribución de datos. Cada nodo posee un subconjunto de la totalidad de los datos y es denominado *shard*. En conjunto todos los *shards* representan una única base de datos lógica [48].

Nodos autónomos

Cada *shard* implementa una arquitectura denominada "nada en común", lo que significa que no comparten ningún recurso (incluyendo CPU, disco o memoria) con otros nodos.

En el modelo más sencillo, todos los *shards* comparten el mismo esquema de datos, estructuralmente son idénticos, las tuplas en la base de datos son divididas entre los *shards* disponibles. La forma de dividir las tuplas es fundamental para que esta técnica provea los beneficios esperados [48].

3.4.3. Identificación de Shards

Existe una columna específica en cada base de datos denominada la clave del *shard*. La clave es necesaria para realizar cualquier consulta sobre los datos, y es la que determina que nodo debe satisfacer la solicitud.

Por ejemplo, la clave del *shard* puede ser el primer dígito de la cedula de identidad del usuario. Todas aquellas personas cuya cedula de identidad comience de 0-3 utilizan el primer *shard*, y de 3-6 el segundo. En este caso la clave del *shard* es provista por el usuario y por tanto conocida.

Existen escenarios más complejos donde la clave del *shard* no es inmediatamente conocida, por ejemplo si la clave es el identificador del usuario. Cuando el usuario ingresa a la aplicación no se posee inmediatamente una clave válida para determinar que *shard* consultar. En estos casos se puede mantener mapas, por ejemplo en un cache distribuido, que asocien datos del usuario con la clave. Una vez que se posee la clave del *shard* se puede acceder a su información [47], [48].

3.4.4. Estrategia de distribución de los datos

La estrategia para la distribución de datos entre los *shards* varía según los objetivos que se desean alcanzar:

- Cuando se emplea *sharding* debido a que el volumen de datos no es apto para una sola instancia, generalmente se divide los datos entre shards de tamaño similar.
- Cuando se utiliza *sharding* por motivos de performance, los datos se dividen de forma que todos los nodos reciban una cantidad similar de consultas o actualizaciones.

Es importante que un solo *shard* pueda satisfacer la mayor parte de las operaciones comunes para alcanzar la eficiencia. Por ejemplo si un usuario es almacenado en un *shard,* resulta conveniente que todas sus compras, y direcciones se encuentren en el mismo *shard*. Realizar operaciones sobre datos que se encuentran en *shards* diferentes resulta muy costoso [47], [48].

Finalmente, con el fin de minimizar el número de operaciones entre *shards* diferentes, todas las tablas pequeñas que se utilizan como referencia, por ejemplo una tabla con todos los departamentos del país, son replicadas en todos los *shards*.

3.5. ASP.NET MVC4

ASP.NET MVC4 es un *framework* para construir aplicaciones web escalables, basadas en estándares utilizando patrones de diseño sólidos y las fortalezas de ASP.NET y .NET *framework* [49].

El patrón de diseño MVC es un patrón de interfaz de usuario que separa la aplicación en tres componentes: modelo, vista y controlador.

Modelo. Implementan la lógica del dominio de la aplicación. A menudo incluye acceso a base de datos para almacenar y recuperar el estado de objetos [50].

Vistas. Las vistas son los componentes que despliegan la interfaz usuario. Típicamente, las mismas son sencillas y se crean a partir de los datos del modelo [50].

Controladores. Los controladores son los componentes que manejan y responden a la interacción del usuario, trabajan con el modelo, y finalmente seleccionar una vista para desplegar la UI [50].

En una aplicación MVC, la vista sólo muestra información, el controlador maneja y responde a las entradas e interacción del usuario.

3.5.1. Objetivos del patrón

1) Separación de responsabilidades de los componentes que componen la interfaz de usuario. Una vista no conoce acerca de los orígenes de datos, ya que recibe los datos del controlador. Un controlador no conoce nada acerca de la distribución grafica de los elementos porque es la responsabilidad de la vista.
2) El nivel de aislamiento obtenido debe permitir a la aplicación ser fácil de entender y mantener.
3) MVC es un patrón de diseño para interfaces de usuario, no impone restricciones sobre la lógica de negocios, la cual puede consistir de n capas.
4) Las vistas deben ser simples en comparación con los controladores, solo despliegan el contenido en el lugar adecuado.
5) Utilizar las últimas tecnologías disponibles: JavaScript, CSS, HTML5. No hay abstracciones que intentan ocultar las complejidades de HTTP, y HTML como en WebForms.
6. Extensibilidad, el *framework* provee de varios *hooks* para incorporar el comportamiento deseado.
7. El *framework* incluye muchas características que lo hacen altamente testeable.

3.5.2. Controladores

3.5.2.1 Rutas

Una ruta es un patrón de dirección URL que está asignado a un manejador. El manejador puede ser un archivo físico, como un archivo .aspx en una aplicación de formularios *Web Forms* o una clase que procesa la solicitud, como un controlador en una aplicación de MVC [51].

Para definir una ruta, se crea una instancia de la clase *Route* especificando el patrón de dirección URL, el manejador y, de forma opcional, un nombre para la ruta.

Las rutas se deben agregan durante el inicio de una aplicación MVC al objeto *Routes* de la clase *RouteTable* (método Application_Start en el archivo global.asax). La propiedad *Routes* es un objeto del tipo *RouteCollection* que almacena todas las rutas de la aplicación [51].

En la siguiente tabla se muestran algunos patrones de rutas y ejemplos de URL coincidentes.

Definición de la ruta	**Ejemplo de dirección URL coincidente**
{controller}/{action}/{id}	/Products/show/beverages
{table}/Details.aspx	/Products/Details.aspx
blog/{action}/{entry}	/blog/show/123
{reporttype}/{year}/{month}/{day}	/sales/2008/1/5
{locale}/{action}	/US/show
{language}-{country}/{action}	/en-US/show

Ilustración 12 Patrones de rutas y ejemplos de URL coincidentes

Existen algunas restricciones con respecto a los patrones de rutas:

- Los marcadores de posición deben ser encerrados entre llaves. Ejemplo: {controller}
- Se pueden definir varios marcadores de posición pero estos deben estar separados por un literal, el cual permite marcar su inicio y fin. Por ejemplo {language}{country} es incorrecto mientras que {language}-{country} es válido.

Las rutas permiten al *framework* entregar las distintas solicitudes a los distintos controladores.

Cuando una solicitud con una determinada URL alcanza la aplicación esta es manejada por el objeto URLRoutingModule. Dicho objeto intenta relacionar la URL recibida a alguna de las rutas definidas a través de su patrón. Si no existe ninguna

coincidencia la aplicación retornara un error HTTP 404 no encontrado. En caso de que existan muchas reglas que coincidan se procesara la solicitud utilizando la primera.

A continuación el manejador HTTP MvcHandler determina el tipo del controlador que se va a invocar agregando el sufijo "Controller" al valor provisto en la dirección URL. El valor de acción en la dirección URL determina el método de acción que se va a invocar.

```
public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            "Default",                                  // Nombre de la ruta
            "{controller}/{action}/{id}",               // Patrón para la URL
            new {controller = "Home", action = "Index", id = ""} // Valores por
defecto
        );

    }

protected void Application_Start()
    {
        RegisterRoutes(RouteTable.Routes);
    }
```

https://gist.github.com/jchiarino/c053999ff409d621d8f7

Por ejemplo una solicitud a la siguiente url: http://localhost/home/about coincide con el patrón definido en el ejemplo anterior, será entregada a una clase cuyo nombre es "HomeController", y posee un método *About*.

En el ejemplo además se definen valores por defecto. Son aquellos valores que son completados en caso de que la URL no los defina. Por ejemplo una solicitud a: http://localhost/ será entregada a una clase cuyo nombre es "HomeController", y posee un método *Index*. Ya que estos son los valores por defecto.

De forma predeterminada, las rutas ignoran solicitudes que tienen un archivo físico asociado en el servidor web. Por ejemplo, la siguiente solicitud será ignorada http://server/Home/Index/estilos.css si el archivo Home/Index/estilos.css existe.

3.5.2.2 Action results

La interacción del usuario con la aplicación se organiza en torno a los controladores y métodos de acción. El controlador define los métodos de acción. Los controladores pueden incluir tantos métodos de acción como sea necesario.

Los métodos de acción tienen normalmente una asignación uno a uno con las interacciones del usuario. Ejemplos de acciones son: un usuario solicita una página, un usuario envía un formulario, etc. En cada caso, la dirección URL de la solicitud incluye información que MVC utiliza para invocar un método de acción [52].

```
public class MyController : Controller
{
    public ActionResult HelloWorld()
    {
        return ContentResult ("HELLO WORLD");
    }
}
```

El ejemplo anterior muestra un ejemplo de *Action Result* activado por una solicitud con URL: http://localhost/my/helloworld, asumiendo un servidor instalado en la maquina local y el conjunto de rutas por defecto. La acción imprime en el navegador el texto "hello world"

```
public class MyController : Controller
{
    public ActionResult GetFile()
    {
        return File("document.docx");
    }
}
```

El fragmento anterior muestra un ejemplo de *Action Result* activado por una solicitud con URL: http://localhost/my/getfile, asumiendo un servidor instalado en la maquina local y el conjunto de rutas por defecto. La acción descargar el archivo document.docx.

Existen diferentes métodos de acción que logran diversos resultados. En la tabla siguiente se muestran los tipos de resultado de acción integrados:

Resultado de la acción	Método auxiliar	Descripción
View Result	View	Representa una vista como una página web.
PartialViewResult	PartialView	Representa una vista parcial, que define una sección de una vista.
RedirectResult	Redirect	Redirecciona a otro método de acción utilizando su dirección URL.
RedirectToRouteResult	RedirectToAction RedirectToRoute	Redirecciona a otro método de acción.
ContentResult	Content	Devuelve un tipo de contenido definido por el usuario.
JsonResult	Json	Devuelve un objeto JSON serializado.
JavaScriptResult	JavaScript	Devuelve un script que se puede ejecutar en el cliente.
FileResult	File	Devuelve un stream de bytes asociado a un archivo.
EmptyResult	(Ninguno)	Representa un valor devuelto que se utiliza si el método de acción debe devolver un resultado vacío.

Ilustración 13 Métodos de acción

3.5.2.3 Action filters

Los filtros permiten aplicar pre procesamiento y post procesamiento a la lógica incluida en un método de acción de un controlador. Son muy buenos para aplicar lógica común que se repite en muchos controladores sin repetir código [53].

Mediante su uso, una solicitud puede pasar por una sucesión de procesos antes y luego de ser procesada los cuales pueden transformar la solicitud completamente.

Los filtros más relevantes incluidos con el *framework* son:

Nombre	**Descripción**
OutputCache	Almacena la respuesta a una acción en un cache de memoria. Sucesivos solicitudes tomaran la versión de memoria sin incurrir en el procesamiento completo. Permite mejorar la eficiencia y escalabilidad de una aplicación dramáticamente.
Authorize	Restringe la ejecución de una acción a usuarios autenticados y que posean determinado rol.
ValidateAntiForgeryToken	Permite prevenir ataques de seguridad del tipo *cross site request forgeries*
HandleError	Permite especificar una la vista que se debe mostrar en caso de que ocurra una excepción no controlada.

Ilustración 14 Filtros más relevantes

Además de los filtros incluidos se pueden definir filtros personalizados, los cuales constituyen uno de los puntos de extensibilidad del *framework*.

Para construir un filtro se crea una clase que derive de *ActionFilterAttribute* y luego se sobrescriben algunos de los métodos que se muestran a continuación:

```
public class DemoFilterAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        //ejecuta antes del método de acción
        base.OnActionExecuting(filterContext);
    }

    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        //ejecuta luego del método de acción
        base.OnActionExecuted(filterContext);
    }

    public override void OnResultExecuting(ResultExecutingContext filterContext)
    {
        //antes que la respuesta enviada al cliente sea creada
        base.OnResultExecuting(filterContext);
    }

    public override void OnResultExecuted(ResultExecutedContext filterContext)
    {
        //después que la respuesta enviada al cliente sea creada
        base.OnResultExecuted(filterContext);
    }
}
```

https://gist.github.com/jchiarino/92bb558154d347ad77f7

Cuando se sobrescribe estos métodos se debe llamar al padre en algún momento de la implementación y se puede utilizar el parámetro *filterContext* para acceder a elementos del *request*.

Los filtros se clasifican según su alcance:

- Filtros globales registrados al inicio de la aplicación: estos filtros actúan sobre todos las solicitudes procesadas por la aplicación.
- Filtros aplicados a un controlador entero. Actúan sobre cada método de acción incluido en el controlador.
- Filtros aplicados a un método de acción.

3.5.3. Vistas

ASP.NET MVC utiliza un motor de vistas denominado Razor. Las vistas son archivos con extensión .cshtml. Se trata de plantillas que consisten de código html y expresiones de C# que son evaluadas para producir código HTML [54].

A continuación se muestra un ejemplo y se explican cada uno de los conceptos relevantes:

Controlador

```
namespace MvcApplication1.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Contact()
        {
            ContactModel model = new ContactModel();
            model.StoreName = "Example store";
            model.Address = "1 infinite loop";
            model.Telephone = "1-800 4050";

            ViewBag.Message = "Otra forma de enviar datos a la vista";
            return View(model);
        }
    }

    public class ContactModel
    {
        public string StoreName { get; set; }
        public string Address { get; set; }
        public string Telephone { get; set; }

    }
}
```

https://gist.github.com/jchiarino/c059f85b1d8e07742dcc

El controlador del ejemplo anterior responde a solicitudes de la forma http://localhost/Home/Contact.

Como fue descripto en capítulos anteriores, es responsabilidad del controlador construir las estructuras de datos necesarias para que la vista muestre la información requerida por el usuario.

Existen varias formas en que un controlador puede transmitir información a una vista:

La primera alternativa es utilizar un objeto denominado *ViewBag*. Dicho objeto es un diccionario dinámico que puede ser modificado en el controlador y luego consultado dentro de una vista para extraer información [54]. El siguiente fragmento de código registra la clave *Message* en el diccionario:

```
ViewBag.Message = "Otra forma de enviar datos a la vista";
```

El segundo mecanismo implica utilizar un modelo para la vista (*ViewModel*). En este caso el controlador puede hacer uso de la capa de negocios, bases de datos, servicios, etc. para recopilar la información necesaria. En el ejemplo anterior el *ViewModel* `(ContactModel)` es sumamente sencillo y es construido de forma trivial

pero en un caso real puede requerir varios pasos complejos. Una vez que el objeto se encuentra listo puede ser transferido a la vista utilizando la función: `return View(model);`

`return View(model);` Es una función de una sola línea pero engloba una alta carga semántica:

1. Indica al *framework* que la respuesta al método de acción se trata de una vista, en dicho caso se procederá a buscar la vista correspondiente, la cual será evaluada para producir una salida en código HTML.
2. El nombre del archivo que contiene la vista a utilizar es inferido de acuerdo a la siguiente convención: todas las vistas se encuentran en una carpeta View dentro de la aplicación. Las vistas pertenecientes al controlador "HomeController" deben residir dentro de una carpeta denominada Home. El nombre de la vista es igual al nombre del método de acción. Aplicando las reglas descriptas el nombre completo de la vista debe ser **/Views/Home/Contact.cshtml**.
3. Finalmente el método transfiere a la vista el ViewModel a utilizar y el procesamiento continuo en el motor de vistas *Razor*.

Razor View

```
@model MvcApplication1.Controllers.ContactModel

@{
    ViewBag.Title = "Contact";
}

<hgroup class="title">
    <h1>@ViewBag.Title.</h1>
    <h2>@ViewBag.Message</h2>
</hgroup>

<section class="contact">
    <h1>@Model.StoreName</h1>
    <h3>Address</h3>
    <p>@Model.Address<br />@Model.Telephone</p>
</section>
```

https://gist.github.com/jchiarino/24a6bf75023a7c8f7100

El fragmento anterior demuestra una vista *Razor* muy sencilla. Estos archivos constan de código Html y expresiones en C# que son evaluadas para producir HTML. Las expresiones en C# son precedidas por el carácter '@' el cual indica su inicio. El motor de vistas *Razor* es muy inteligente en diferenciar fragmentos de código C# de HTML.

`@model MvcApplication1.Controllers.ContactModel`
Indica al motor de vistas que el *ViewModel* a utilizar para la vista es del tipo ContactModel. Dicha instrucción se denomina: ***strongly typed views.*** A partir de ese momento la vista conoce los detalles del tipo *ContactModel,* y el compilador podrá asistir en su uso o emitir errores por usos incorrectos de la propiedad *Model*.

`@ViewBag.Message`
Permite acceder al diccionario dinámico *ViewBag* modificado en el controlador. Dichas expresiones retornan un valor en caso de que la clave (en este caso message) se encuentre definida o vacía en caso contrario.

`@Model.StoreName`
Permite acceso al *ViewModel* transferido como parámetro por el controlador. En este caso se accede a la propiedad *StoreName* del *ViewModel*.

Salida HTML

Una vez que el motor de vistas *Razor* termina su procesamiento, el resultado es código HTML listo para ser enviado al cliente que emitió la solicitud. Todas las expresiones de código C# son remplazadas por el código HTML correspondiente.

```
<hgroup class="title">
    <h1>Contact.</h1>
    <h2>Otra forma de enviar datos a la vista</h2>
</hgroup>

<section class="contact">
    <h1>Example store</h1>
    <header>
        <h3>Address</h3>
    </header>
    <p>1 infinite loop<br />
        1-800 4050
    </p>
</section>
```

https://gist.github.com/jchiarino/33819bdd63c5ee5bdf4a

3.5.4. Seguridad

Existen tres mecanismos disponibles:

Ilustración 15 Mecanismos de seguridad

3.5.4.1 Forms Authentication

Es el mecanismo de autenticación preferido para aplicaciones de internet. En esta modalidad el desarrollador es responsable de proveer un formulario donde el usuario puede ingresar su identificador y contraseña. Posteriormente las credenciales son validadas contra el almacén de datos y si son válidas una *Cookie* cifrada es emitida al cliente. El cliente almacena esa *Cookie* la cual permite mantener las credenciales [55].

Todas las solicitudes autenticadas incluyen información acerca del usuario que las emite lo que permite actuar en consecuencia, por ejemplo personalizar áreas, restringir acceso utilizando filtros, etc.

3.5.4.2 Windows Authentication

Se trata del mecanismo de autenticación preferido para aplicaciones de intranet. Se integra perfectamente con servicios e infraestructura ejecutando bajo la plataforma Windows por ejemplo servicios de *Active Directory*. Es frecuentemente denominado autenticación integrada, porque Windows automáticamente autentica al usuario si el sitio lo soporta [56].

3.5.4.3 OpenId / OAuth

Al utilizar este sistema de autenticación los usuarios no deben crear un usuario / contraseña para utilizar la aplicación, y tampoco es necesario almacenar ni validar credenciales. En su lugar se confía en proveedores de identidad populares como Twitter, Microsoft, Facebook, Google.

Cuando es necesario autenticar a un usuario, se lo redirige al proveedor de identidad y el usuario prueba su identidad contra el proveedor. Posteriormente el proveedor de identidad redirige al usuario nuevamente a la aplicación utilizada, incluyendo un mensaje cifrado que prueba que el usuario se ha autenticado.

3.5.5. Caching

Se trata de una táctica para favorecer la performance y eficiencia de las aplicaciones. Consiste en almacenar el resultado de un método de acción en memoria. Posteriormente todas las solicitudes al mismo método de acción retornan el resultado alojado en memoria lo cual es exponencialmente más rápido [57].

Para habilitar el cache se utiliza el *ActionFilter* ***OutputCache*** como se muestra en el ejemplo. La sentencia `Thread.Sleep(3000);` simula una operación costosa que requiere de un tiempo considerable para ser completada, el cache tiene una validez de 60 segundos. El comportamiento resultante será: la primer solicitud a http://localhost/Home/CacheResult será procesada completamente y tomara aproximadamente tres segundos en mostrar el resultado (la hora actual). Solicitudes subsiguientes retornaran de inmediato el resultado almacenado en memoria, idéntico al de la primera solicitud. Transcurridos los 60 segundos desde la solicitud inicial otra solicitud a la misma URL será procesada completamente y almacenada nuevamente en memoria.

```
public class HomeController : Controller
    {
        [OutputCache(Duration=60)]
        public ActionResult CachedResult()
        {
            Thread.Sleep(3000);
            string result = DateTime.Now.ToString("HH:mm:ss");
            return Content(result);
        }
    }
```

Uno de los elementos más importantes para una estrategia de cache exitosa es realizar mediciones para conocer donde se encuentra la mayor carga en la aplicación a optimizar. De esa forma se puede conservar los resultados asociados y entregarlos sin la necesidad de su recalculo.

3.5.5.1 Configuración

Existen parámetros adicionales que pueden ser indicados en el filtro para su configuración.

VaryByParam

```
[OutputCache(Duration=60, VaryByParam="paramName")]
```

Indica que parámetros considerar para generan una versión nueva del cache. Por defecto cualquier permutación de parámetros causa la ejecución del método de acción y su posterior almacenamiento en memoria. Las siguientes solicitudes generan dos copias del cache:

a) http://localhost:1394/Home/CachedResult/1
b) http://localhost:1394/Home/CachedResult/2

Luego cada vez que se solicita (a) se retorna la copia en cache de (a) y cuando se solicita (b) se retorna la copia en cache de (b). Un ejemplo de la utilidad de esta configuración es aplicable a búsquedas, no se desea retornar el mismo juego de resultados para dos búsquedas cuyos términos son diferentes.

Vary by Header

Indica que *headers* considerar para generan una versión nueva del cache. Por defecto se ignoran los *headers*, dos solicitudes con distintos *headers* siempre retornan la misma copia del cache.

```
[OutputCache(Duration=60, VaryByHeader="Accept-Language")]
```

El ejemplo muestra la especificación del *header* "Accept-Language" como parámetro para variar el cache. Dos solicitudes con distinto valor en ese campo generan diferentes versiones del cache.

Un ejemplo de uso es para aplicaciones en distintos idiomas, no se desea entregar una página en ingles a un usuario de habla hispana.

4. Conclusiones

Se espera que el presente documento haya permitido al lector adquirir los conocimientos fundamentales para comprender la arquitectura y diseño de aplicaciones en la nube, sus características, implicancias, y balance de atributos de calidad.

Durante el desarrollo se realizó una introducción al concepto de computación en la nube, detallando cada uno de los modelos de servicio y despliegue disponibles. Para cada modelo se consideró sus ventajas, desventajas y escenarios de uso.

Posteriormente se describió cada uno de los módulos que integran la arquitectura de una aplicación en la nube, cuales son sus responsabilidades y como deben interactuar entre ellos de forma de alcanzar los objetivos buscados.

Finalmente se presentaron varios de los patrones clave, y ampliamente utilizados, que permiten lograr la interacción y atributos de calidad buscados.

Los conceptos tratados fueron abordados en su mayoría desde una perspectiva teórica. Sin embargo, los mismos son aplicables a la construcción de aplicaciones en cualquier plataforma en la nube. En trabajos sucesores se podrá abordar estos conceptos desde una perspectiva práctica, donde se darán ejemplos concretos de uso de los módulos disponibles en varias de las plataformas más importantes.

5. Bibliografía

[1] P. Mell y T. Grance. (2011, Sep) The NIST Definition of Cloud Computing. [Online]. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[2] G. Lewis. (2012, Jan) Architectural implications of Cloud Computing. [Online]. http://www.sei.cmu.edu/library/abstracts/webinars/Architectural-Implications-of-Cloud-Computing.cfm

[3] Wikipedia. (2013, Jul) Virtualization. [Online]. http://en.wikipedia.org/wiki/Virtualization

[4] vmware. (2013, Jan) Virtualization overview. [Online]. http://www.vmware.com/virtualization/what-is-virtualization.html

[5] Fujitsu. (2011) Cloud adoption. [Online]. http://globalsp.ts.fujitsu.com/dmsp/Publications/public/wp-Cloud-Adoption1.pdf

[6] Amazon. (2013) Amazon Web Services. [Online]. http://aws.amazon.com/es/

[7] Amazon. (2012, Aug 15) AWS - Cómo empezar a trabajar con EC2. [Online]. http://docs.aws.amazon.com/es_es/AWSEC2/latest/GettingStartedGuide/Welcome.html

[8] Amazon. (2010, Oct) AWS - What Is AWS Elastic Beanstalk. [Online]. http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html

[9] Amazon. (2013, May 15) AWS - What is Amazon Relational Database Service. [Online]. http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html

[10] Amazon. (2009, Apr 15) AWS - Get Started with Amazon SimpleDB. [Online]. http://docs.aws.amazon.com/AmazonSimpleDB/latest/GettingStartedGuide/Welcome.html

[11] Amazon. (2006, Mar 1) AWS - Amazon Simple Storage Service. [Online]. http://docs.aws.amazon.com/es_es/AmazonS3/latest/gsg/GetStartedW

ithS3.html

[12] Microsoft. (2013) Windows Azure overview. [Online]. http://www.windowsazure.com/es-es/

[13] Microsoft. (2013) Windows Azure - Virtual Machines. [Online]. http://www.windowsazure.com/es-es/services/virtual-machines/

[14] Microsoft. (2013) Windows Azure - Web sites. [Online]. http://www.windowsazure.com/es-es/services/web-sites/

[15] Microsoft. (2013) Windows Azure - Cloud Services. [Online]. http://www.windowsazure.com/es-es/services/cloud-services/

[16] Microsoft. (2013) Windows Azure - Data Management. [Online]. http://www.windowsazure.com/es-es/services/data-management/

[17] Google. (2013, Jul 19) What Is Google App Engine. [Online]. https://developers.google.com/appengine/docs/whatisgoogleappengine

[18] Google. (2013, May 7) App Engine - About Google Cloud SQL. [Online]. https://developers.google.com/cloud-sql/docs/introduction

[19] Google. (2013, Jul 18) App Engine - Storing data. [Online]. https://developers.google.com/appengine/docs/python/storage

[20] Google. (2013, Jun 20) App Engine - Blobstore Python API Overview. [Online]. https://developers.google.com/appengine/docs/python/blobstore/

[21] Risto Haukioja and John Rhoton, "Cloud Definition," in *Cloud Computing Architected*. New york, United States: Recursive Press, 2013, ch. 1, pp. 19-21.

[22] Richard Taylor and Neuad Medvidovic, *Software Architecture: Foundations, Theory, and Practice*. California, United States: Wiley John, 2009.

[23] Risto Haukioja and John Rhoton, "Browser Interface," in *Cloud Computing Architected*. New york, United States: Recursive Press, 2013, ch. 9, pp. 132-140.

[24] Microsoft. (2013, Nov 11) Using a Three-Tier Architecture Model. [Online]. http://msdn.microsoft.com/en-us/library/windows/desktop/ms685068%28v=vs.85%29.aspx

[25] Risto Haukioja and John Rhoton, "Native Clients," in *Cloud Computing Architected*. New york, United States: Recursive Press, 2013, ch. 10, pp. 142-145.

[26] Risto Haukioja and John Rhoton, "Storage," in *Cloud Computing Architected*. New york, United States: Recursive Press, 2013, ch. 17, pp. 248-252.

[27] Risto Haukioja and John Rhoton, "Relational Data," in *Cloud Computing Architected*. New york, United States: Recursive Press, 2013, ch. 18, pp. 262-273.

[28] Risto Haukioja and John Rhoton, "Non relational data," in *Cloud Computing Architected*. New york, United States: Recursive Press, 2013, ch. 19, pp. 276-280.

[29] Risto Haukioja and John Rhoton, "Application Integration," in *Cloud Computing Architected*. New york, United States: Recursive Press, 2013, ch. 16, pp. 223-236.

[30] Wikipedia. (2012, Jul 31) Balanceador de carga. [Online]. http://es.wikipedia.org/wiki/Balanceador_de_carga

[31] Risto Haukioja and John Rhoton, "Authentication," in *Cloud Computing Architected*. New york, United States: Recursive Press, 2013, ch. 13, pp. 183-193.

[32] IBM. (2010) Horizontal and vertical scaling. [Online]. http://pic.dhe.ibm.com/infocenter/brdotnet/v7r1/index.jsp?topic=%2Fcom.ibm.websphere.ilog.brdotnet.doc%2FContent%2FBusiness_Rules%2FDocumentation%2F_pubskel%2FRules_for_DotNET%2Fps_RFDN_Global397.html

[33] Microsoft. (2013, Jul 11) Scaling Up and Scaling Out in Windows Azure Web Sites. [Online]. http://blogs.msdn.com/b/windowsazure/archive/2013/07/11/scaling-up-and-scaling-out-in-windows-azure-web-sites.aspx

[34] Wikipedia. (2013, Jul 31) Computer performance. [Online]. http://en.wikipedia.org/wiki/Computer_performance

[35] Wikipedia. (2013, Aug 1) Scalability. [Online]. http://en.wikipedia.org/wiki/Scalability

[36] Microsoft. (2013) How to Use the Autoscaling Application Block. [Online]. http://www.windowsazure.com/en-us/develop/net/how-to-guides/autoscaling/

[37] Baptiste Assmann. (2012, Mar 29) affinity, persistence, sticky sessions: what you need to know. [Online]. http://blog.exceliance.fr/2012/03/29/load-balancing-affinity-persistence-sticky-sessions-what-you-need-to-know/

[38] Amazon. (2012, Jul 1) Create Sticky Sessions. [Online]. http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/US_StickySessions.html

[39] Martin Fowler. (2011, Jul 14) Command Query Responsibility Segregation. [Online]. http://martinfowler.com/bliki/CQRS.html

[40] Microsoft. (2004, Apr) Messaging Patterns in Service-Oriented Architecture, Part 1. [Online]. http://msdn.microsoft.com/en-us/library/aa480027.aspx

[41] Microsoft. (2004, Jul) Messaging Patterns in Service Oriented Architecture, Part 2. [Online]. http://msdn.microsoft.com/en-us/library/aa480061.aspx

[42] Microsoft. (2013, May) How to Use Service Bus Queues. [Online]. http://www.windowsazure.com/en-us/develop/java/how-to-guides/service-bus-queues/

[43] Valery Mizonov and Seth Manheim. (2013, Apr 18) Windows Azure Queues and Windows Azure Service Bus Queues. [Online]. http://msdn.microsoft.com/en-us/library/windowsazure/hh767287.aspx

[44] Apache. (2013, Aug 04) What Is Apache Hadoop. [Online]. http://hadoop.apache.org/#What+Is+Apache+Hadoop%3F

[45] Highly Scalable. (2012, Feb 1) MapReduce Patterns, Algorithms, and Use Cases. [Online].

http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/

[46] Max Garfinkel. (2010, Feb.) Random. [Online]. http://blog.maxgarfinkel.com/learning-hadoop/

[47] CodeFutures. (2009) Database Sharding. [Online]. http://www.codefutures.com/database-sharding/

[48] Michael Thomassy. (2013, Feb 7) How to Shard with Windows Azure SQL Database. [Online]. http://social.technet.microsoft.com/wiki/contents/articles/1926.how-to-shard-with-windows-azure-sql-database.aspx

[49] ASP.NET Team. (2013, June) The Official Microsoft ASP.NET Site. [Online]. http://www.asp.net/mvc/tutorials/older-versions/overview/asp-net-mvc-overview

[50] Microsoft. ASP.NET MVC Overview. [Online]. http://msdn.microsoft.com/en-us/library/dd381412%28VS.98%29.aspx

[51] Microsoft. (2013, Apr 28) ASP.NET Routing. [Online]. http://msdn.microsoft.com/en-us/library/cc668201.aspx

[52] Microsoft. (2013) Controllers and Action Methods in ASP.NET MVC Applications. [Online]. http://msdn.microsoft.com/en-us/library/dd410269%28VS.98%29.aspx

[53] Microsoft. (2013) Filtering in ASP.NET MVC. [Online]. http://msdn.microsoft.com/en-us/library/gg416513%28VS.98%29.aspx

[54] Scott Guthrie. (2010, Jul 2) Introducing "Razor". [Online]. http://weblogs.asp.net/scottgu/archive/2010/07/02/introducing-razor.aspx

[55] Microsoft. (2013) Using Forms Authentication in ASP.NET MVC. [Online]. http://msdn.microsoft.com/en-us/library/ff398049%28VS.98%29.aspx

[56] Microsoft. (2013) Windows Authentication Provider. [Online]. http://msdn.microsoft.com/en-us/library/907hb5w9%28v=vs.100%29.aspx

[57] Microsoft. (2009, Jan 27) Improving Performance with Output Caching. [Online]. http://www.asp.net/mvc/tutorials/older-versions/controllers-and-routing/improving-performance-with-output-caching-cs