FACULTAD DE INGENIERÍA DE LA UNIVERSIDAD DE BUENOS AIRES

# Aplicaciones de Software como Servicio

## Tesis de Ingeniería en Informática

**Matias Woloski (matias@southworks.net)**

**8/8/2008**

Padrón: 80419

Profesores:

**Lic. Gustavo López y Lic. Ismael Jeder**

*La adopción de software brindado como servicio ha crecido en los últimos años. Según Gartner Inc. para el año 2010 más del 30% de los proveedores de software ofrecerán al menos alguna de sus aplicaciones bajo este modelo. Por otra parte, las empresas que brindan plataforma, desde Microsoft, Google hasta Amazon, están invirtiendo fuertemente y han comenzado a virar lentamente su maquinaria hacia este modelo. Otras empresas como salesforce.com han demostrado resultados exitosos basando su negocio exclusivamente en este modelo. No quedan dudas que el Software como Servicio ganó y seguirá ganando aceptación, sin embargo, ¿está la industria preparada para adoptar este modelo y hacer frente a esta demanda? ¿De qué se trata este modelo exactamente desde la perspectiva del productor de software? ¿Cómo se compone un proyecto basado en software como servicio? Esta tesis trata de responder estas preguntas definiendo un modelo y un body of knowledge para enfrentar este tipo de proyectos.*

# 1 Tabla de Contenidos

# 2 Agradecimientos

*A mi familia y a mi novia (futura esposa) por el apoyo y fuerza que me dieron a lo largo de la carrera de Ingeniería y en especial en este último tiempo que me dediqué a la ardua tarea de escribir la tesis.*

*A Alejandro Jack (Southworks), mentor y amigo, por haberme abierto puertas y haber aportado su visión sobre este trabajo.*

*A Eugenio Pace (Microsoft Corp) por haberme introducido en el tema de Software como Servicio y por el feedback y revisiones sobre la tesis. A Arvindra Sehmi (Microsoft EMEA) por el apoyo que me brindó y su valioso feedback. A Gianpaolo Carraro (Microsoft) y Fred Chong (Microsoft Corp), con quienes tuve el honor de trabajar a partir del año 2006 hasta el día de hoy y por haberme abierto las puertas de su creativo equipo.*

*Por último, no estaría presentando esta tesis si no fuera por el Lic. Gustavo López (FIUBA) quién me brindó su apoyo desde el primer día.*

# 3 Prólogo

*Mi interés por Software como Servicio (SaaS) comenzó a principios del año 2006 en un viaje a Microsoft Corp. En búsqueda de un tema interesante para elaborar mi tesis de grado, comencé a indagar y un colega me comentó sobre un nuevo modelo de distribución de software. Luego de unos meses, Alejandro Jack, mi manager, me puso en contacto con John Devadoss, gerente del grupo de arquitectura estratégica (AST) de Microsoft. Dentro de este grupo trabajan Gianpaolo Carraro, Eugenio Pace y Fred Chong encargados entre otras cosas de investigar el modelo Software como Servicio del punto de vista arquitectónico. Como parte de este desafío, el grupo establece relaciones con clientes interesados en este modelo y con los grupos de producto identificando gaps en la plataforma con el objetivo de proveer feedback. El grupo publica numerosos papers cubriendo esta área también y realiza pruebas de concepto utilizando los productos de Microsoft. Durante estos años colaboré con este grupo en diferentes proyectos. Participé en la construcción y dictado de workshops y vi madurar el concepto de Software como Servicio desde sus etapas iniciales donde se trataba de una incipiente modalidad hasta la actualidad donde está siendo adoptado por la industria.*

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Tel 425 882 8080
Fax 425 936 7329
http://www.microsoft.com/

Microsoft

Gianpaolo Carraro
Architect
Architecture Strategy Team
Microsoft Corp.

To whom it may concern:

This short letter is to attest that my team has successfully collaborated with Matias Woloski, as part of our Software as a Service (SaaS) Architecture Guidance initiative.

Matias introduced very innovative ideas around multi-tenant management of business processes and has provided us with a proof of concept implementation which was used in my TechED US presentation.

Impressed by Matias thinking and diligence in the architecture space, I have suggested he write his findings in a whitepaper that could be published on msdn.microsoft.com architecture. To support our collaboration, I have also arranged for Matias to be able to access our Lab environment for performing performance testing.

If Matias continues like this, he will be a strong contributor to our SaaS Architecture Guidance.

Regards,

Gianpaolo Carraro
Architect
Microsoft Corp.

# 4 Introducción

En los primeros días de la industrialización, las fábricas desarrollaban su propia fuente de energía. La electricidad era un diferenciador para las industrias: aquellas que la tenían eran superiores a aquellas que no. Sin embargo, el costo de instalación y mantenimiento que tenían que afrontar por tener su propia fuente de energía era demasiado alto comparado con el beneficio asociado. Con el tiempo, empujados por la necesidad de una masa crítica de la industria y una tecnología más madura, comenzaron a surgir proveedores especializados. Este proceso culminó con la invención de la electricidad, la forma más versátil de energía conocida por el hombre. El diferenciador dejó de existir como tal y entonces tuvo sentido recurrir a un proveedor externo. Las industrias podrían consumir toda la energía que ellas quisiesen, cuando la necesitasen y pagar solo por lo que usasen. La energía se había convertido en un servicio [Factor, 2001].

Si extrapolamos esta línea de evolución al software, la idea de brindar un *software como un servicio* no es tan ilógica. En lugar de tener que adquirir o desarrollar, instalar y mantener un software, simplemente las empresas lo contratarían y lo utilizarían.

Sin embargo, prestar un servicio en torno a una aplicación de software es algo más complejo que prestar un servicio como el agua, la electricidad o el cable. La electricidad se puede definir fácilmente: 110V/60 Hz en Estados Unidos o 220V/50Hz en Europa o Argentina. El consumidor solo precisa un conector y a lo sumo un transformador. En cambio en la joven industria del software, si bien existen estándares maduros, la interacción entre estos es más compleja y la superficie a resolver es mucho mayor. Esto trae aparejado una serie de dificultades: interoperabilidad, confianza, variabilidad de requerimientos, escalabilidad, seguridad, monitoreo de SLAs, monetización, acceso fuera de línea, entre otros.

No obstante, ciertas funciones de software son más fáciles de ofrecer como servicio que otras: todo aquello que no es parte del núcleo o no es un diferenciador para una empresa es susceptible de ser consumido como servicio. Entre los ejemplos más comunes se encuentra el e-mail.

*Software como Servicio* (*Software as a Service* en inglés) es el término con el cual se da a conocer hoy en día esta nueva tendencia de *software consumido como si fuera un servicio* y cuenta con implementaciones como el caso de Salesforce.com[1] (CRM) o Amazon Web Services[2]. Las grandes

[1] http://www.salesforce.com

empresas de software (por ejemplo Microsoft, Google, IBM) comenzaron a plantearse este modelo de servicios como parte de su estrategia esencial y se encuentran activamente desarrollando productos, frameworks y servicios de distinta granularidad.

## 4.1 Motivación

En los últimos 20 años la tecnología de información comenzó a jugar un papel muy importante en las empresas. Hoy en día una compañía que no destina recursos a IT corre con una desventaja muy grande para desarrollar y expandir su negocio. Es por eso que la mayoría utiliza un software que ayude en la toma de decisiones, manejo de información y registro de transacciones, entre otras cosas. Este software, la mayoría de las veces se instala y se mantiene en la empresa, y se lo denomina "software on-premise"[3]. Esta parece ser una opción natural para él o las personas que tomen las decisiones relacionadas con tecnología dentro de una empresa. Sin embargo, a la hora de evaluar el costo-beneficio, instalar un software "on-premise" implica adquirir una infraestructura de hardware, comprar licencias de software de base, contar con personal calificado para que lo administre y mantenga, y muchos otros costos indirectos. El modelo de software brindado como servicio, en pocas palabras, cambia el método de distribución y monetización tradicional ofreciéndolo como una utilidad.

Durante el desarrollo de esta tesis se enumerarán las razones por las cuales *Software como Servicio* puede ser una alternativa atractiva para las empresas y por qué el modelo puede tener potencial. Estas razones se pueden encontrar en internet en pocos segundos con una simple búsqueda en Google o Wikipedia. Si bien los beneficios que trae este nuevo modelo de distribución son fácilmente visibles, el grueso de la información está dirigido al consumidor y no al proveedor de software. Construir un software y brindarlo como un servicio es un terreno que se encuentra poco explorado aún. Existen implementaciones exitosas pero hay poca información, ya sea en internet, en publicaciones o papers sobre cómo construir este tipo de software y los desafíos técnicos intrínsecos en el modelo. Una de las razones puede ser que el tema es relativamente nuevo y el mercado está enfocado en atraer la demanda y no la oferta. Si la tecnología logra pasar al mainstream, cientos de proveedores independientes de software (ISVs) y empresas se enfrentarán con el desafío de construir y encarar proyectos con estas características. Es por eso que este trabajo trata de brindar un panorama holístico que ayude a empresas a abordar el *Software como Servicio*.

---

[2] http://aws.amazon.com

[3] En español "Premise" significa "lugar físico".

## 4.2 Estructura de la tesis

Este trabajo comienza mencionando los antecedentes del modelo (5.1 Antecedentes), definiendo los roles y el ecosistema dentro del cual se encuadra *Software como Servicio* (5.2Definición de Software as a Service y 5.3 Roles y Ecosistema). Se realiza el análisis de adopción y difusión del modelo basado en la teoría de difusión de innovaciones (5.4 Adopción y Difusión) para luego enunciar el problema que se quiere atacar (6 Definición del Problema). El desarrollo y contribución principal se encuentra en la sección 7 Solución propuesta. Durante la primera parte se realiza un análisis del dominio de *Software como Servicio* (7.1 Análisis de Dominio) identificando y definiendo las características que exhiben las principales soluciones y productos en el mercado. Estas características serán luego encuadradas en un modelo de características (7.2 Modelo de Características) o *feature model* [Kang K., 1990]. El modelo está desarrollado en un nivel conceptual y un nivel de implementación. El primero explora las características de manera agnóstica (7.3 Capacidades) y el segundo propone implementaciones y tecnologías alternativas relacionadas con la capacidad conceptual (7.4 Implementación). Por último, se aplica el modelo en una aplicación de ejemplo (8 Aplicación del modelo), se enumeran las conclusiones (9 Conclusiones) y las líneas de investigación abiertas para futuros trabajos (10 Futuras líneas de investigación).

# 5 Estado de la cuestión

## 5.1 Antecedentes

### 5.1.1 Tercerización

Un siglo atrás, la mayoría de la población mundial trabajaba en el campo. Durante las buenas épocas había mucha comida. Sin embargo, las épocas malas significaban hambre y muerte como ocurrió en la gran hambruna irlandesa entre 1845 y 1849. Hoy en día el empleo relacionado con la agricultura representa un 5% en economías avanzadas. A pesar de este porcentaje tan bajo, no hay problemas masivos de alimentación en este tipo de economías. La diferencia, radica en el aumento de productividad que ocurrió en la industria agrícola y más recientemente en la industria manufacturera. Este aumento de productividad se dio gracias a la lenta migración de los granjeros a industrias especializadas que soportan la producción agrícola: equipamiento, fertilizante, pesticidas, prácticas de manejo de las tierras, semillas superiores, servicios de transporte, servicios de combustible, servicios de soporte del gobierno, y más. En resumen, una gran cantidad de conocimiento embebido en instituciones, industrias, profesiones y tecnologías existen para soportar a un grupo pequeño de granjeros que abastecen a todo el mundo. Con el tiempo, la división del trabajo y la especialización fueron creciendo [A research manifesto for services science, 2006].

La especialización se puede asociar fácilmente a la tercerización. La principal diferencia radica en que el primero es un concepto económico que explica como el mercado fuerza la división del trabajo creando economías más saludables. La tercerización en cambio es un concepto utilizado en el ámbito de negocios y contable. Es un concepto utilitario que se define como "la transferencia de un servicio interno a un proveedor externo"[4].

La tercerización tiende a bajar los costos debido a la especialización, es por eso que es una práctica muy utilizada en los negocios. Existen diversos ejemplos de tercerización en la industria. Las auto partes en la fabricación de un auto, la prefabricación de ladrillos en la construcción de casas y edificios, etc. Para el común de la gente, la tercerización es aún más evidente. De hecho, ninguno cosecha trigo y después hace el pan, simplemente lo compran en la panadería; o no cosechan algodón para hacer sus prendas.

[4] Extraido de Wikipedia, The Free Encyclopedia (http://en.wikipedia.org/wiki/Outsourcing)

En las empresas, históricamente, la tercerización más común es la limpieza, el mantenimiento de las oficinas, la preparación de comida, etc. Otros casos más complejos pueden ser las áreas de soporte como legales, contaduría, auditorías, comunicaciones, publicidad, etc.

¿Por que conviene comprar servicios a otras empresas en lugar de hacerlo dentro la propia? Porque los proveedores de estos servicios se especializaron y aprendieron como brindar el servicio con calidad y a precio competitivo al mismo tiempo. De hecho, muchas veces para ser más eficientes recurren a las redes y computadoras para automatizar muchas de sus funciones y crean o compran aplicaciones que reutilizan para todos sus clientes obteniendo un margen operativo más alto. Por otro lado las áreas que tercerizan las empresas suelen ser áreas de poco o muy bajo aporte al "núcleo del negocio". Son áreas necesarias, pero no suficientes de la actividad de una empresa determinada. Por ejemplo, Microsoft no teceriza el desarrollo de software de sus productos más significativos, porque ese es su "núcleo de negocio". Sin embargo, si terceriza el mantenimiento de sus edificios.

Las empresas de tecnología son uno de los servicios más comunes disponibles hoy en día. Es más, si una empresa de energía puede construir la planta, instalar generadores, desarrollar las redes de distribución y proveer energía a millones de personas, ¿por qué una empresa de servicios de IT no puede construir su propio datacenter, comprar e instalar las computadoras, desplegar una aplicación de software y proveer el software como servicio a sus clientes? Si puede, y de hecho varias empresas lo están haciendo hace tiempo.

### 5.1.2 Contexto histórico

En 1961, John McCarthy[5], fue el primero en sugerir públicamente (en un discurso dado en el marco de celebración del centenario del MIT) que la tecnología de tiempo compartido de la computadora podría conducir a un futuro en el que el poder de cómputo e incluso aplicaciones específicas podrían ser vendidas como un servicio (como el agua o la electricidad). La idea de una computadora o una utilidad informática era muy popular a fines de la década del 60. El mainframe brindaba servicios de cómputo y los usuarios en sus terminales accedían a los programas que este brindaba. Estos programas estaban disponibles "on demand". Era el sueño de cualquier administrador de IT, tener control absoluto y centralizado, sabiendo exactamente qué programas existían y qué usuarios podían acceder a los mismos con una interfaz homogénea. Sin embargo, hacia mediados de los 70

[5] "John McCarty, the computer scientist". Extraido de Wikipedia, The Free Encyclopedia (http://en.wikipedia.org/wiki/John_McCarthy_%28computer_scientist%29)

se volvió claro que el hardware, software y las tecnologías de telecomunicación simplemente no estaban preparados para la potencial demanda [Greschler, y otros, 2002].

A principios de los '80, surgió la PC y los usuarios descubrieron que podían controlar su entorno. Podrían elegir que software correr e incluso modificar el estilo y color de su interfaz. Al funcionar el software de forma local nadie tenía problemas de performance que traía el acceso concurrente al mainframe. Sin embargo surgió otro problema: los usuarios se convirtieron en sus propios administradores. Al no estar entrenados para esta tarea y con el advenimiento de internet y problemas serios como los virus informáticos, no deja de ser llamativo que la principal fuente de ingresos pasó a ser "soporte y mantenimiento".

Finalmente, en los últimos años convergieron una serie de factores que causaron el retorno del concepto del *software como servicio* [Greschler, y otros, 2002]:

- La dificultad que han tenido los usuarios finales para manejar productos de software
- La dificultad que ha tenido IT para manejar productos de software
- La dificultad que han tenido los ISVs (Vendedores de Software Independientes) para vender y distribuir productos de software
- Los cambios en la tecnología
- La llegada de los Application Service Provider (ASP)

Los primeros dos puntos fueron expuestos anteriormente. Los otros tres merecen un tratamiento más detallado.

Los ISVs desean evolucionar a nuevos modelos de negocios basados en subscripción y pay-per-use[6]. Su actual canal de distribución es difícil de mantener y la piratería preocupa cada vez más, sobre todo en Asia y América Latina. El usuario desea poder utilizar una aplicación desde cualquier computadora y dispositivo e incluso las empresas desean "contratar" una aplicación para un proyecto específico y pagar solo por el uso que se le da en ese período.

En cuanto a los cambios en la tecnología, estos abarcan tanto hardware como software de base. Las redes de alta velocidad o banda ancha han proliferado. En Estados Unidos, en el 2005, más del 55% de las conexiones de internet eran de banda ancha. En el 2006 este número creció a 70% y para

[6] Pagar solamente por el uso que se le da a la aplicación.

mayo de 2008 el 90% contaba con conexión de banda ancha [Nielsen Online, 2008]. La web se transformó en un componente casi obligatorio para cualquier empresa. Las herramientas para desarrollar aplicaciones son mucho más completas y los estándares como HTTP y web services están maduros y adoptados por las plataformas más utilizadas (.NET, Java, PHP). La web ha evolucionado de ser una gran biblioteca de solo lectura a ser un medio mucho más natural donde la gente se relaciona, se comunica, interactúa, aprende, juega, compra, vende y socializa [O'Reilly, 2005].

Por último, las empresas comenzaron a confiar en las compañías de web hosting (alojamiento web) para exponer sitios y aplicaciones. Más tarde surgió la idea de los Application Service Providers (ASP) en donde parte o toda la administración de IT se tercerizaría a proveedores especializados. Estos proveedores existieron durante la era de la burbuja ".com" a fines de los '90 y principios del 2000, sin embargo muchos de ellos fracasaron ya que lo único que ofrecían eran datacenters gigantescos para albergar aplicaciones existentes que no fueron diseñadas para escalar. En un estudio realizado en el año 2003, se reveló que de las 424 compañías relevadas, 203 habían fallado, 40 habían sido compradas y 8 se habían asociado. Solo 173 de 424 sobrevivieron [Bhavini y otros, 2003].

En este contexto surge el modelo de Software as a Service cuyas características serán definidas en la siguiente sección.

## 5.2 Definición de Software as a Service

*Software como Servicio* en su definición más ortodoxa es **un modelo de distribución de software mediante el cual una aplicación es ofrecida a múltiples clientes y es accesible a través de la red (ej.: internet)**.

Las principales características del modelo según [IDC, 2008] son:

- El software es accesible, manejado y comercializado vía red
- El mantenimiento y actividades relacionadas con el software se realiza n desde un lugar centralizado en lugar de hacerlo en cada cliente, permitiendo a estos acceder a las aplicaciones vía la red
- La aplicación es distribuida típicamente bajo el modelo de uno-a-muchos, incluyendo su arquitectura, management, precio y partnering
- Generalmente se basa en un modelo de comercialización en el cual no hay un costo inicial, sino que un pago por suscripción o por utilización en el cual no se diferencia la licencia del software del alojamiento del mismo.

La figura a continuación muestra algunas características de *Software como Servicio* en comparación con Application Service Provider y tercerización tradicional de una aplicación [Forrester Research, 2008]:

| | SaaS | | | ASP | Application outsourcing |
|---|---|---|---|---|---|
| **Vendor examples:** | Salesforce.com | RightNow Technologies | Siebel CRM On-Demand | Orange Business Services hosting SAP | PeopleSoft run by Oracle On Demand, Accenture managing SAP |
| **Tenancy** | | | | | |
| All customers run same code version? | Yes | Mostly | Mostly | No | No |
| Code modification possible? | No | No | No | Yes | Yes |
| Multi-tenant architecture? | Yes | Yes | Sometimes (private tenancy opt.) | No | No |
| Originally designed to be SaaS? | Yes | Yes | Yes | No | No |
| **Upgrades** | | | | | |
| Who controls upgrade timing? | Provider | Customer | Customer | Customer | Customer |
| **Payment model** | | | | | |
| How is the software priced? | Subscription | Subscription | Subscription | Subscription | Upfront license + maintenance fee |
| Customer owns license? | No | No | No | No | Yes |
| **Responsibility for managing the application** | | | | | |
| Who develops application? | SaaS provider | SaaS provider | SaaS provider | Software provider (not ASP) | Software provider (not AO provider) |
| Who has resp. for operating & maint. application & infrastructure? | SaaS provider | SaaS provider | SaaS provider | ASP | AO provider |

45600 Source: Forrester Research, Inc.

Figura 1 - *Software como Servicio* vs Application Service Provider vs outsourcing de aplicaciones [Forrester Research, 2008]

## 5.3 Roles y Ecosistema

### 5.3.1 Construir, Ejecutar, Consumir y Comercializar

Dentro del ecosistema del modelo *Software como Servicio* es posible definir un conjunto de roles. Estos roles definen responsabilidades que pueden ser ejecutadas por un mismo actor o diferentes actores. La siguiente figura define los roles: construir, ejecutar, consumir y comercializar.

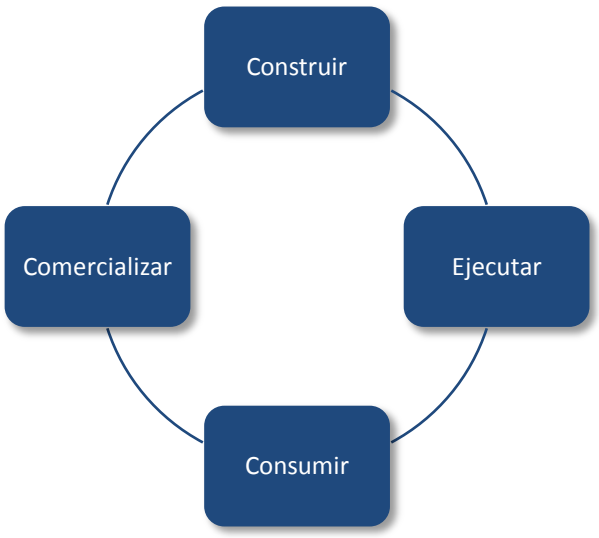



Figura 2 - Roles dentro del modelo de *Software como Servicio* [Carraro y otros, 2007]

#### *5.3.1.1 Construir*

La tarea principal de este rol es construir el software aportando el conocimiento del dominio específico. Por ejemplo, una empresa con conocimientos del dominio financiero construye una aplicación para el manejo de flujo de dinero. Esta empresa tiene el conocimiento necesario para mapear los requerimientos del mundo financiero a una aplicación que automatice las tareas y reportes relacionados con el manejo del flujo de dinero. Sus principales preocupaciones están relacionadas con la arquitectura de la aplicación, la lógica y reglas de negocios, el modelo de datos, etc. Idealmente, esto debería ser ortogonal al modelo de comercialización por suscripción o una licencia perpetua o al hecho de distribuir la aplicación a múltiples clientes por la web o enlatada. Los requerimientos relacionados al negocio en sí, se mantienen inalterables. Sin embargo, tendrá que realizar ciertas optimizaciones relacionadas con el modelo. Un ejemplo de esto es multi tenancy que es la capacidad de utilizar la misma infraestructura (código base, base de datos, etc.) para servir a múltiples clientes. En este rol están los proveedores independientes de software (ISV), empresas como salesforce.com o cualquier empresa que conozca de un negocio específico a modelar y esté interesada en construir el software con las características del modelo.

#### 5.3.1.2 Ejecutar

En el rol de ejecución la tarea principal es proveer plataforma e infraestructura para la ejecución de *Software como Servicio*. Se puede trazar una analogía con los proveedores de alojamiento web que brindan servicios a empresas que quieren exponer y ejecutar sus aplicaciones a través de la web. La diferencia en este tipo de ejecución es que mientras el proveedor de alojamiento solo ofrece el cable de red, el ancho de banda, un servidor y espacio en disco el rol de ejecución en *Software como Servicio* tiene más responsabilidades. Entre ellas se puede nombrar escalabilidad a demanda, manejo de recursos, acuerdos de nivel de servicio y un conjunto de componentes optimizados especialmente para la plataforma. En este rol hay algunos proveedores especializados que ofrecen infraestructura como Amazon Web Services y en un nivel de abstracción más alto proveedores de plataforma como Google App Engine.

#### 5.3.1.3 Consumir

Consumir *Software como Servicio* no debería ser muy diferente a consumir cualquier otra aplicación. Es decir, el consumidor no tiene porque interesarse en la forma en que el proveedor distribuye su aplicación o si es multi tenant o no. En última instancia, el consumidor se interesa en las características del software y si lo ayuda o no a su negocio. Las preocupaciones en este contexto son: cómo se integra el software al conjunto de aplicaciones existentes, cuál es el mecanismo de seguridad y como interactúa con los mecanismos de la empresa, si se está cumpliendo o no con el nivel de servicio acordado, etc. En el rol de consumidor se encuentra cualquier organización que desee automatizar una de sus capacidades de negocio. Las opciones eran construir o comprar, ahora bajo este modelo se agrega una tercera opción: alquilar.

#### 5.3.1.4 Comercializar

La comercialización es la actividad relacionada con sacar rédito económico del software. El rol de comercializador generalmente lo toma el mismo que construye, no obstante la especialización también lleva a que haya empresas dedicadas, exclusivamente o no, a monetizar una aplicación brindada como un servicio. Por ejemplo, una aplicación con un tráfico importante puede no cobrarles a los usuarios pero ganar plata de los avisos publicitarios de Google AdSense. Es decir, Google toma el rol de comercializador. Los tipos de comercialización más comunes son: por suscripción, por utilización y por publicidad. Dentro del modelo de suscripción el más utilizado es "por mes por usuario". También es el menos intrusivo a la aplicación ya que puede manejarse totalmente separado (aunque hay beneficios en la integración). Este modelo es generalmente

utilizado en aplicaciones que brindan cierta funcionalidad de negocios. El modelo por utilización es más granular y quizás más flexible. Se paga por lo que se usa. Es recomendable cuando sea simple medir el uso de los recursos y generar un margen por arriba del recurso. Por último el modelo basado en publicidad utiliza una plataforma para mostrar publicidad a cambio de un monto por cada impresión o click. Este modelo se puede utilizar en sitios con mucho tráfico.

A continuación se ilustran 6 ejemplos en donde se diferencian los roles en cada uno.

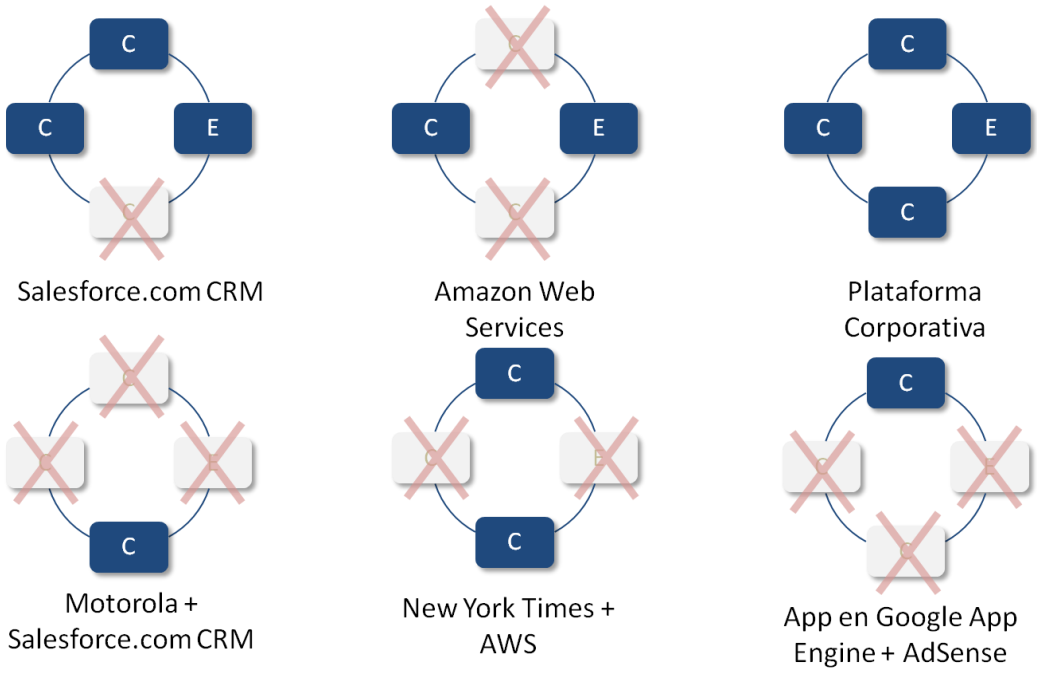



Figura 3 - Ejemplos de empresas tomando diferentes roles dentro del modelo

El primer ejemplo es el CRM de salesforce.com. Esta es una aplicación para manejar las oportunidades y los clientes y está ofrecida como servicio. Salesforce construye, ejecuta y comercializa. Es decir, conoce el dominio de CRM y construyó una aplicación multi tenant que luego desplegó en su datacenter y ejecuta por su cuenta y por último se encarga de la comercialización bajo el modelo de suscripción.

El segundo ejemplo es Motorola, una empresa que "alquila" el software de CRM realizado por salesforce.com. El rol de Motorola en este escenario es solamente consumir la aplicación de CRM, no construye, ni comercializa ni corre la aplicación [Salesforce.com].

El tercer ejemplo es Amazon Web Services. Amazon brinda servicios de infraestructura también llamado *utility computing*, como almacenamiento, poder de procesamiento y hasta una cola de

mensajería accesible por web. Todos estos recursos infinitamente escalables. En este caso Amazon ejecuta, es decir tiene la plataforma para construir sobre y comercializa, cobra por la utilización de los recursos.

El caso de New York Times que utiliza Amazon Web Services, no es únicamente consumidor del servicio sino que además construye sobre su infraestructura. Por ejemplo, construyó una solución sobre Amazon S3 y EC2 para digitalizar todos los diarios desde el año 1851 a 1922 [Gottfrid, 2007].

Por último, una organización puede estar interesada en crear su propia plataforma corporativa basada en un modelo de *Software como Servicio*. Para trazar una analogía, sería similar al concepto de intranet e internet. Es una versión "privada" a una escala menor pero los mismos conceptos. En ese caso, la organización construye, ejecuta, consume y opcionalmente comercializa la plataforma. Esto también es conocido como "IntraSaaS".

## 5.4 Adopción y Difusión

Habiendo definido el modelo de *Software como Servicio*, se ensayará un análisis de adopción y difusión tomando datos hasta Agosto de 2008 con el objetivo de situar está innovación en el ciclo de vida de adopción tecnológica. El análisis estará enmarcado en la teoría de difusión de innovaciones [Rogers, 2003].

El ciclo de vida de adopción de tecnología es un modelo sociológico, originalmente desarrollado por Joe Bohlen y George Beal en 1957. El objetivo era registrar los patrones de compra de semillas de maíz de los granjeros. Seis años más tarde Everret Rogers expandió el uso de este modelo en su libro Difusión de Innovaciones [Rogers, 2003].

El modelo de ciclo de vida de adopción tecnológica describe la aceptación de un nuevo producto o innovación de acuerdo a las características demográficas y psicológicas definidas por los grupos de adopción. El proceso de adopción a lo largo del tiempo se ilustra como una curva de campana, ubicando a los innovadores, tomando el riesgo más grande, luego los que adoptan tempranamente que buscan una ventaja competitiva también tomando riesgo. Cuando se percibe el valor real que aporta la tecnología en cuestión, la mayoría temprana adopta. La mayoría tardía son más cautelosos y los rezagados adoptan cuando es realmente barato y poco riesgo.

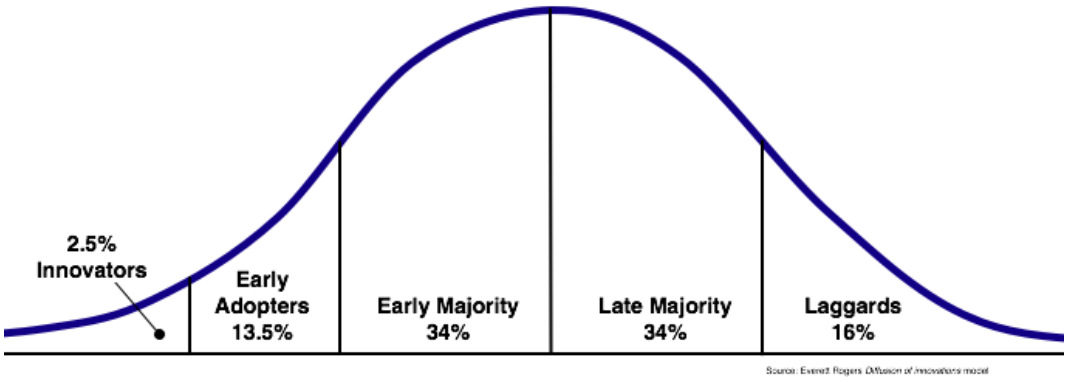



Figura 4 - Grafico de campana del modelo de ciclo de vida de adopción tecnológica [Rogers, 2003]

Con el objetivo de analizar y entender la adopción de *Software como Servicio* y enmarcarlo en un grafico de campana de ciclo de vida adopción tecnológica, se analizarán encuestas y reportes relacionados con *Software como Servicio* de las principales agencias del mundo.

### 5.4.1 Gartner Inc.

Gartner realiza una vez al año un reporte denominado *Hype Cycle*. Este modelo representa la evolución de lo que denomina visibilidad de una tecnología emergente (o expectativas despertadas por esa tecnología) en función del tiempo. Una evolución que progresa desde un entusiasmo exagerado, pasando por un período de desilusión hasta una eventual comprensión de la relevancia de la tecnología y de su papel en un mercado o dominio. El modelo se justifica por la influencia de dos fuerzas principales que conforman las expectativas que se crean alrededor de una nueva tecnología:

- el ruido de comunicación (noticias, publicidad, etc.) sobre dicha tecnología, en una primera instancia
- la madurez de la tecnología (desde un punto de vista técnico y de negocio), en una segunda instancia

El Hype Cycle es más útil desde la perspectiva del comprador de tecnología (no del proveedor), tiene un carácter descriptivo y su utilidad predictiva y de definición de planes de acción es menor. Sin embargo, es una fuente de información relevante para la industria.

El reporte de Hype Cycle *Software como Servicio* del año 2008 de Gartner [Gartner Inc., 2008], define 29 categorías relacionadas con el modelo *Software como Servicio*. Estas categorías son:

| | | | | |
|---|---|---|---|---|
| Application Platform as a Service | BPMS-Enabled SaaS | Business Process Hubs | Communication as a Service | Distributed Order Management |
| E-Commerce on Demand | Email SaaS | Employee Perf Management | E-Recruitment | HRMS SaaS |
| Integration as a Service | Mobile Apps on Demand | MRM on Demand | Multienterprise Business Process Platform | On-Demand Financial Management Application |
| On-Demand Partner Relationship Management | On-Demand Sales Incentive Compensation Management | On-Demand Salesforce Automation | Policy Administration SaaS Options for Life Insurers | Project & Portfolio Management (PPM) SaaS |
| SaaS Business Intelligence | SaaS Data Quality | SaaS Enabled Application Platform | SaaS for Contact Center Customer Service | SaaS Portals |

| | | | | |
|---|---|---|---|---|
| SaaS Procurement Apps | Security SaaS | Supply Chain Planning (SaaS) | Web Analytics | |

El gráfico de Hype Cycle se ve a continuación:

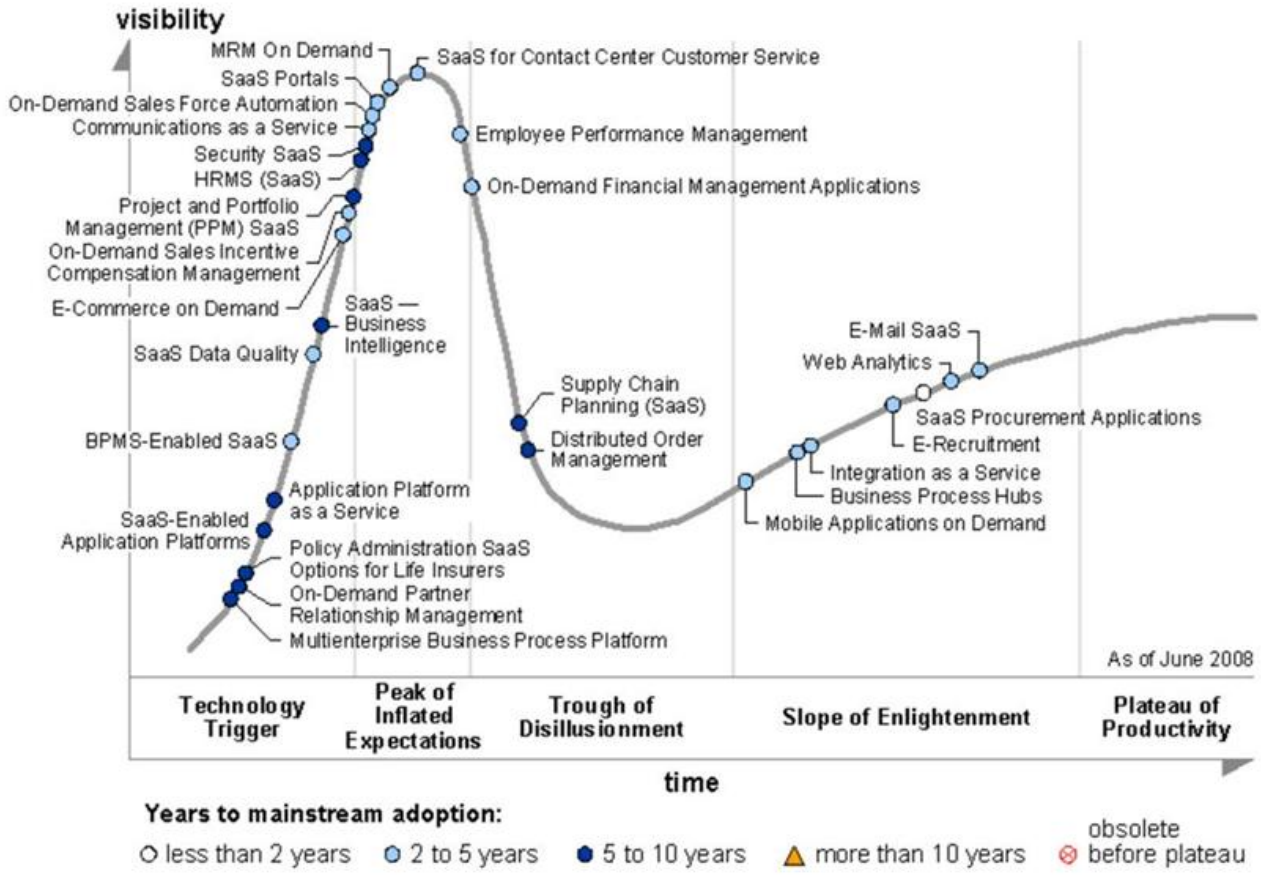



Figura 5 - Hype Cycle de *Software como Servicio* [Gartner Inc., 2008]

Luego de un análisis más detallado del reporte se puede apreciar que dentro del tipo de software que llegó al "slope of enlightment" (cuando la industria comienza a encontrar el valor a la tecnología) se encuentran aquellas aplicaciones que no son del núcleo del negocio pero son necesarias (E-mail, procuración, reclutamiento, integración, etc).

Otra característica de este reporte es que brinda información relacionada con la madurez de la tecnología.

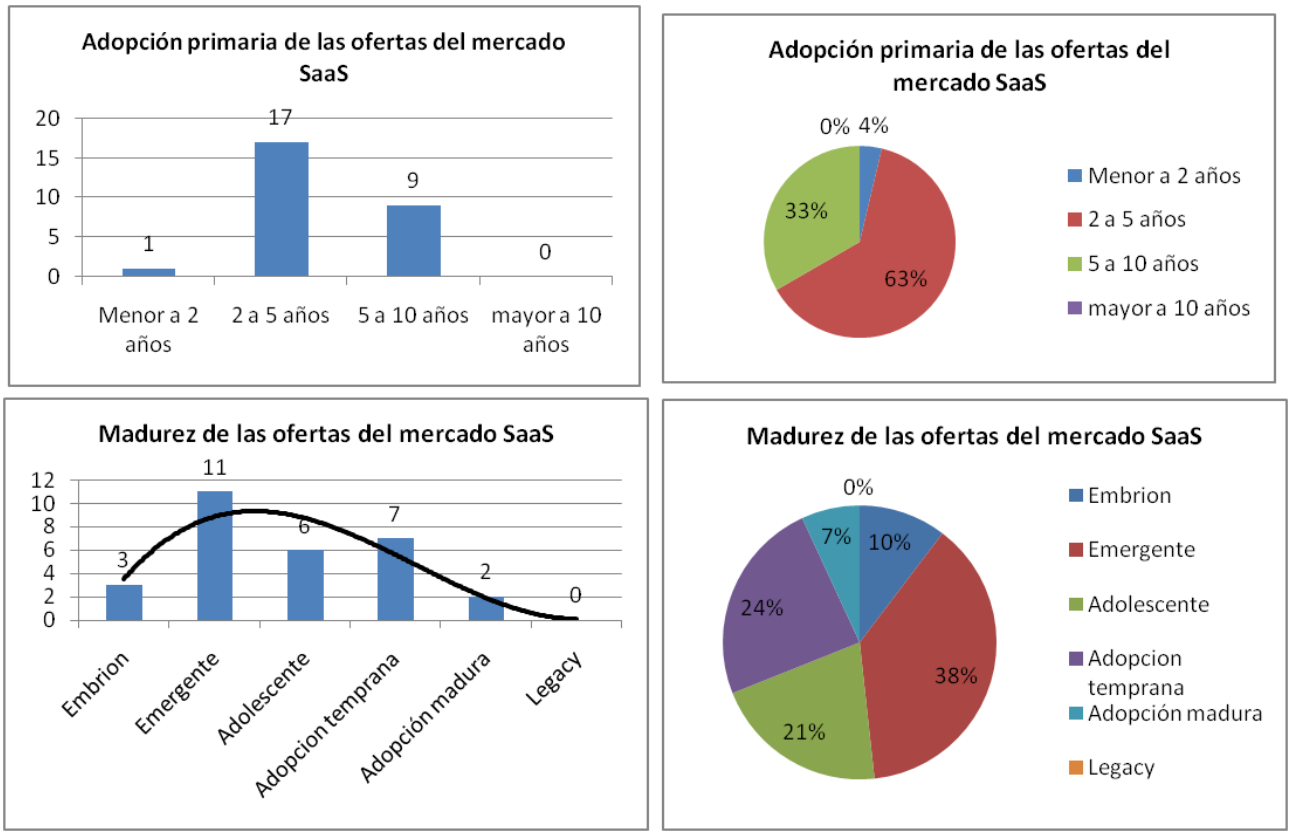



Figura 6 - Gráficos relacionados con el reporte Hype Cycle SaaS de Gartner.

Gartner relaciona la etapa embriónica con actividad en laboratorios. La etapa emergente es la primera generación de producto con pilotos e implementaciones de líderes en la industria. En la etapa adolescente las capacidades de la tecnología están maduras, se trata de una segunda generación, existe un entendimiento y comienzan a aparecer los que adoptan de forma temprana. La tecnología ya está probada y la adopción evoluciona rápidamente en la etapa de adopción temprana. Luego, en la etapa de adopción madura, la tecnología es robusta, no se percibe evolución en los proveedores ni en la tecnología y existen varios proveedores dominantes. Por último, la etapa legacy, la ganancia está dada por el mantenimiento y el costo de migración es lo que impide el reemplazo por otra tecnología.

Los números arrojan que un 50% de las diferentes ramas del mercado relacionado con *Software como Servicio* está en etapa embriónica o emergente, con solo un 10% en etapa embriónica. El otro 50% está en etapa adolescente, adopción temprana o madura. No existe ninguna en modo legacy.

Por otro lado la mayoría de las ramas del mercado van a ser adoptadas por el mainstream dentro de 2 a 5 años. Solo una llegará a ser adoptada en menos de 2 años y el 33% restante dentro de 5 a 10

años. Es decir, según Gartner, recién en los próximos 3 a 6 años habrá una oferta de aplicaciones *Software como Servicio* insertada en el mercado mainstream.

La adopción de software brindado como servicio ha crecido en los últimos años. Según Gartner Inc., para el 2009, 100% de las empresas de servicios de IT más relevantes de la industria (tier 1) van a ofrecer consultoría relacionada con *Software como Servicio*. Para el 2010, el 15% de las grandes corporaciones habrá empezado a reemplazar su ERP con soluciones basadas en Service Oriented Architecture y *Software como Servicio*. Por último, un dato relevante, es que para el 2012, más del 33% de los ISVs van a ofrecer algunas de sus aplicaciones bajo el modelo de *Software como Servicio* [Gartner Inc., 2008].

### 5.4.2 Forrester

Forrester Research evidencia el creciente interés en este modelo comparando encuestas de 2006 y 2007 a grandes, medianas y pequeñas empresas [Forrester Research, 2008]. La pregunta que plantea la encuesta es "¿Cuán interesado está en adoptar *Software como Servicio*?". Del total de encuestados (1025 empresas de Estados Unidos y Europa), un 10% a un 20% están usando actualmente y la tendencia marca que la adopción se da en empresas más grandes. En todos los casos se incrementa el porcentaje en la categoría de "usando actualmente" y "muy interesado pero sin planes de adoptar" de 2006 a 2007.

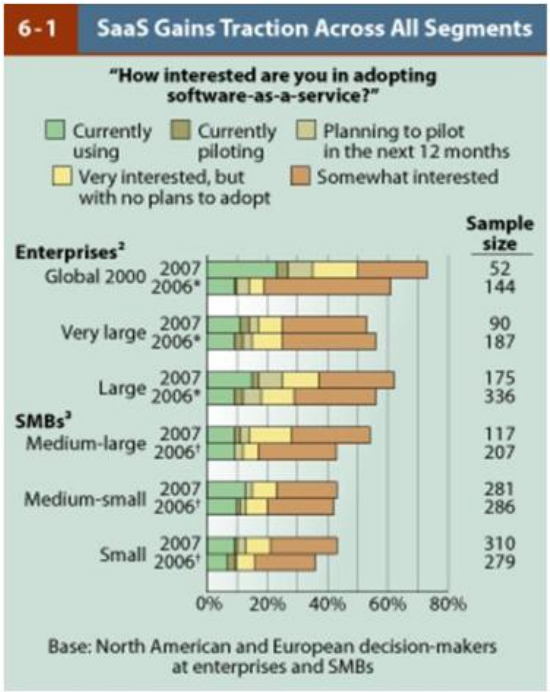



Figura 7 - Forrester muestra el interes en el modelo comparando 2006 y 2007 [Forrester Research, 2008]

### 5.4.3 Saugatuck

De acuerdo a una encuesta realizada por Saugatuck Technology con un muestreo de 418 empresas [Saugatuck Technology, 2008], un 32% de los encuestados está utilizando *Software como Servicio*, un 43% lo está analizando y un 15% no planea utilizar.

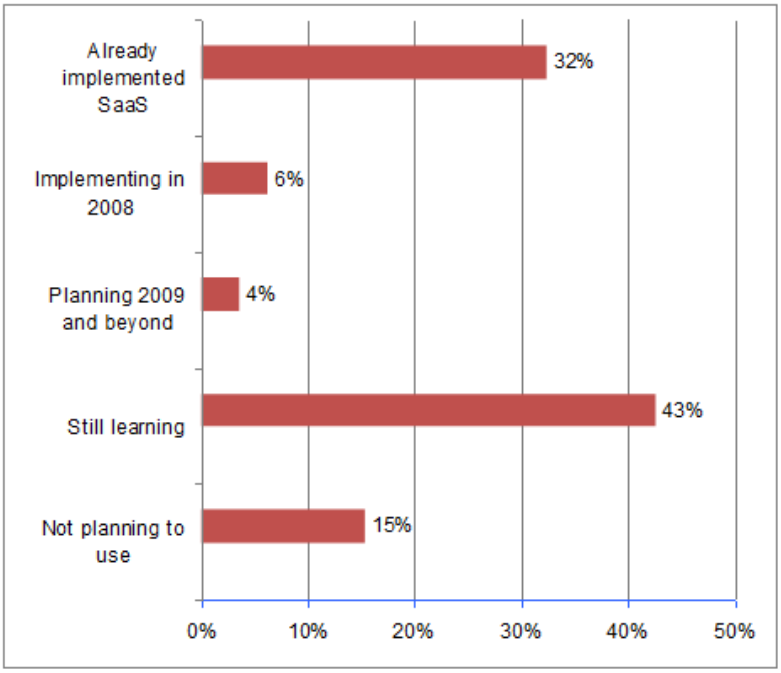



Figura 8 - Adopción de SaaS [Saugatuck Technology, 2008]

Del 15% que no planea usar, solo el 4% corresponde a las empresas con más de 5000 empleados. En empresas medianas la adopción es más cautelosa, más de 20% no planea usar. Los números son similares en empresas más pequeñas, 18%. Por otro lado, hay un porcentaje alto de empresas (más del 40% en todos los segmentos menos en empresas de 100 a 499 empleados) que están analizando el modelo. Estos podrían representar la mayoría temprana que no toma la decisión hasta que no estén seguros del retorno que les traiga.

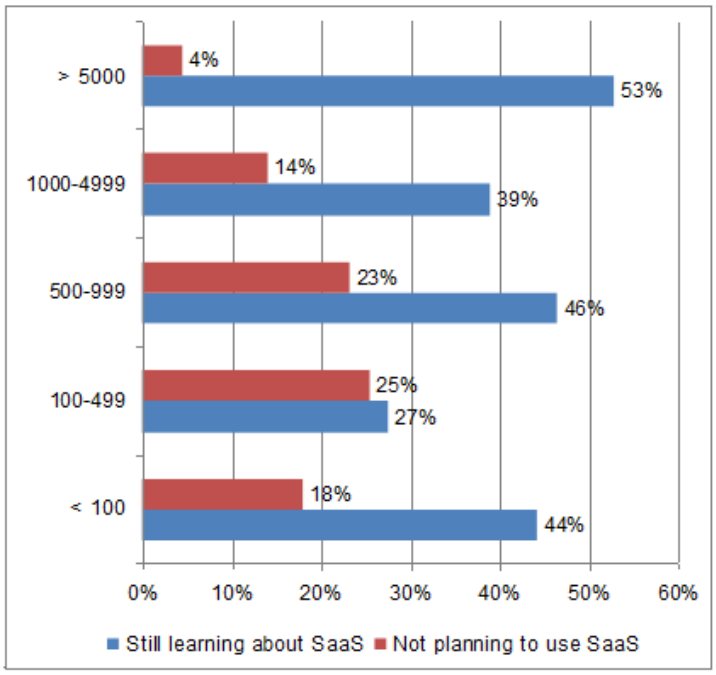



Figura 9 - Encuesta sobre la falta de adopción de SaaS separada por cantidad de empleados [Saugatuck Technology, 2008]

### 5.4.4 Plataforma

Por otra parte, las empresas que brindan plataforma, desde Microsoft, Google hasta SAP, están invirtiendo fuertemente y han comenzado a virar lentamente su maquinaria hacia este modelo. Ray Ozzie, el reemplazo de Bill Gates luego de su retiro, distribuyó un memo en Ocutbre de 2005 llamado "The Internet Services Disruption" [Ozzie, 2005] en donde anuncia el inicio del ciclo de producto más grande de la historia de la compañía, esta vez relacionado con "service-enabled software". Amazon desde el año 2006 que abrió su plataforma al mundo ofreciendo servicios de almacenamiento, procesamiento, mensajería bajo el nombre de Amazon Web Services y el modelo de monetización de *Software como Servicio*, siendo hoy uno de los lideres en el mercado. Google se concentra en el segmento de consumidores y agranda la oferta de software ofrecido como servicio abriendo su plataforma con ofrecimientos como Google App Engine.

En resumen, las grandes empresas de plataforma están apostando al futuro del *Software como Servicio*.

### 5.4.5 Conclusión

A partir del análisis de diferentes reportes y encuestas y de mis propias percepciones nutridas desde los inicios del año 2006 cuando decidí realizar esta tesis, mi opinión es que a la fecha (Agosto de

2008) el mercado de *Software como Servicio* está en una etapa de adopción temprana (*early adopters*) casi llegando a la mayoría temprana (*early majority*), la tecnología está cruzando el abismo (crossing the chasm) [Moore, 1991]. Esta apreciación tiene en cuenta a todos los diferentes tipos de aplicaciones de *Software como Servicio*. Vale aclarar que ciertos tipos de aplicaciones están más maduras que otras y podrían estar en otras etapas de adopción como el caso del e-mail, CRM y procuración.

La primera ola de innovación comenzó en el año 2001. En ese momento el término *Software como Servicio* era nuevo en la industria y venía a reemplazar el viejo Application Service Provider (ASP) que no había tenido la adopción esperada. *Software como Servicio* fue más aceptado por el mercado y algunos proveedores ex ASP y otros nuevos comenzaron a ofrecer sus servicios bajo este modelo. A partir de fines del año 2005 y principios de 2006 el término SaaS se hizo mucho más conocido gracias a implementaciones exitosas que surgieron en la etapa de innovación, como salesforce.com y Amazon Web Services y al avance notable de las redes de banda ancha y la tecnología. En ese momento fue cuando comenzó la etapa de adopción temprana y de a poco comenzaron a aparecer los grandes jugadores de la industria, Microsoft, Google, IBM, etc. Al día de hoy existen más de 3500 soluciones basadas en el modelo, según un directorio independiente de aplicaciones *Software como Servicio* (saas-showplace.com).

En resumen, *Software como Servicio* pasó de ser adoptado solo por los innovadores y los early adopters. Con un promedio del 30% de adopción por parte de las empresas pequeñas, medianas y grandes de acuerdo a los diferentes reportes de las consultoras mencionadas anteriormente y un mercado con un 50% de proveedores en un nivel de madurez medio alto y un 50% en niveles embriónicos y emergentes, la tecnología está atravesando la barrera para llegar a ser adoptado por el mainstream. Las empresas que adoptan elijen aplicaciones que no son de misión crítica para su negocio y que no tienen requerimientos de integración complejos.

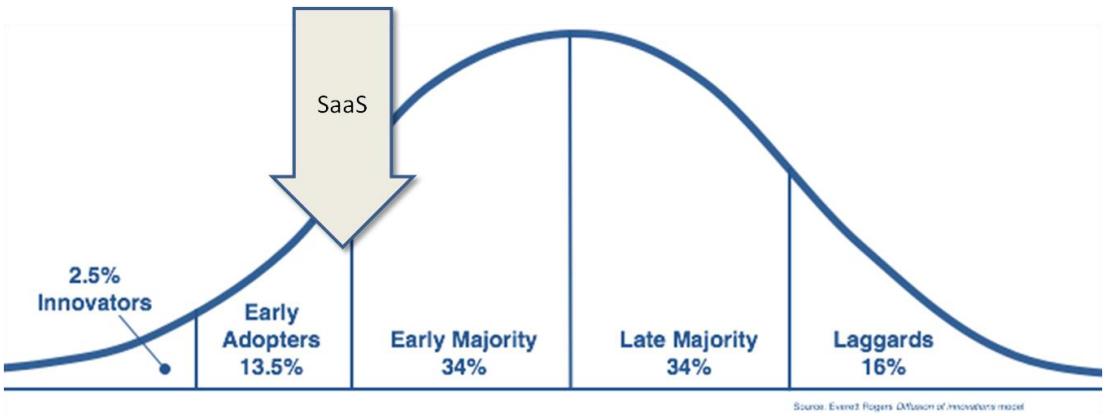



Figura 10 - Ciclo de vida de adopción y donde se encuentra *Software como Servicio* a mitades del año 2008

### 5.4.6 Barreras para la adopción

Las barreras para la adopción hoy en día por parte de los consumidores son:

- **Integración y customización**. según la encuesta de Forrester, la integración es el tema numero uno nombrado por los encuestados de norte América y Europa con un 65% argumentando que la integración es la razón clave por la cual no consideran *Software como Servicio*. Los proveedores avanzaron en los desafíos relacionados con la customización e integración de *Software como Servicio*. Si bien la mayoría de ellos ofrecen APIs basada en web services y exportación de datos, el consumidor todavía percibe que posee más control utilizando las aplicaciones tradicionales.
- **Costo total de propiedad (TCO) del modelo por subscripción versus el modelo tradicional**. Muchos de los consumidores perciben que si bien *Software como Servicio* puede ser económico inicialmente, pero no tanto en el largo plazo. Forrester piensa que esta afirmación por parte de los consumidores es apresurada y no tiene en cuenta factores como costos de mantenimiento, personal y otros beneficios como mejoras de funcionalidad incremental [Forrester Research, 2008].
- **Preocupaciones sobre la seguridad desde seguridad física hasta backup y roles y permisos**. Entre el 40% y el 50% de los encuestados por Forrester citan temas de seguridad. Desde temas específicos como el riesgo de que los usuarios manejen roles y permisos de la aplicación a cosas menos específicas que el consumidor simplemente siente ("nos preocupa la seguridad pero no sabemos exactamente qué")

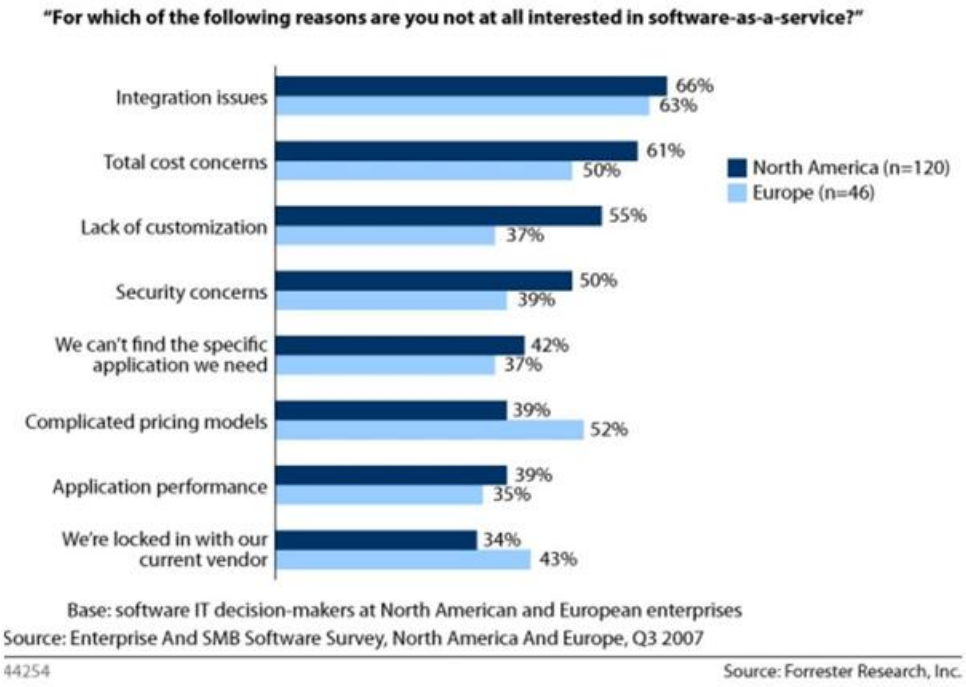



**Figura 11 - Barreras para la adopción de *Software como Servicio* [Forrester Research, 2008]**

Por parte de los proveedores las barreras para la adopción son:

- **Gastos altos en ventas**. El modelo de *Software como Servicio* promete un ciclo de ventas más rápido y fácil por el bajo riesgo que implica para el comprador pagar un bajo costo inicial en comparación con la venta de software tradicional. Sin embargo, a los proveedores de *Software como Servicio* les está costando mucho vender y alcanzar buenos márgenes. Por ejemplo Salesforce.com declaró en la conferencia de resultados financieros del segundo trimestre de 2008 [Seeking Alpha, 2008] tener ingresos por u$s 263 millones y un gasto en ventas y marketing de $130 millones. O sea un 50% de los ingresos. Por otro lado, captaron 4.100 nuevos clientes, eso significa que obtener cada cliente le cuesta a salesforce $32.000. Es un precio alto en comparación con el margen que deja en el corto plazo. La apuesta es que su marca siga siendo más fuerte, atraigan más clientes con menos esfuerzo y retengan los que ya tienen. Este mismo problema lo tienen la mayoría de las empresas que se basan exclusivamente en el modelo de *Software como Servicio* [Lacy, 2008].

# 6 Definición del Problema

Existe una demanda que se evidencia mediante los estudios de mercado citados en la sección 5.4 Adopción y Difusión y a medida que el modelo muestre adopción en la industria y los desafíos no superen las ventajas mencionadas, la demanda debería seguir creciendo. Tanto los productores de software buscando crear una solución *Software como Servicio* en el corto, mediano o largo plazo, como las compañías que quieran construir su propia plataforma basada en este modelo se enfrentarán con los desafíos que este modelo introduce y en poco tiempo, sino lo están haciendo hoy en día, se plantearán:

## ¿Cómo se construye software ofrecido como servicio?

Esta es la pregunta que trata de responder esta tesis y está enfocada a los productores de software que quieren afrontar un proyecto de estas características y a los consumidores que deseen tener un panorama más acabado de los desafíos que esto implica.

El modelo de software brindado como servicio introduce algunos aspectos desconocidos a la arquitectura que son inherentes al modelo y otros que surgen de conocidos modelos, como *utility computing*, o estilos de arquitectura como *SOA* y se manifiestan en otros dominios. Sin embargo es difícil encontrar la definición de estos atributos y su taxonomía en su conjunto. Al ser un modelo relativamente nuevo, el mercado está enfocado más en la demanda que en la oferta. Esto puede resultar en una arquitectura inapropiada o incluso en falta de capacidad de IT para alinearse con los objetivos de negocio.

Por otra parte, no solo el modelo es relativamente nuevo, sino que la infraestructura disponible como el hardware, almacenamiento y conectividad avanzó mucho en los últimos años. Gracias a estos avances proliferan nuevas formas de diseñar, desplegar y mantener soluciones que antes no eran técnicamente posibles.

Si bien existen implementaciones de software brindado como servicio en diferentes plataformas, todas estas son de código cerrado ya que brindan un servicio pago y los diseños, algoritmos, patrones y arquitecturas permanecen bajo su dominio.

En resumen, el problema se define de la siguiente manera:

- No existen herramientas que permitan hacer una evaluación y un plan de ejecución para el ciclo de vida de aplicaciones de software ofrecido como servicio
- No existe un Cuerpo de Conocimiento (BOK) que identifique y defina los atributos de software ofrecido como servicio
- No existe un Cuerpo de Conocimiento (BOK) que identifique y defina las arquitecturas, diseños, tecnologías y patrones asociados a esos atributos
- Si existen soluciones que resuelven estas problemáticas pero son propietarias

Por lo tanto, el objetivo es:

- **Identificar, definir y categorizar** los **atributos**, **capacidades** y **desafíos técnicos** del software brindado como servicio y a partir de estos **plantear alternativas de solución**, **patrones de diseño**, **estilos de arquitecturas** y **tecnologías** con el fin de **proveer un marco** para **evaluar**, **planificar** y **diseñar proyectos de software brindado como servicio** basados en un lenguaje común

Esta tesis **no** pretende:

- Brindar una única solución que se adapte a todas las necesidades y objetivos de negocios relacionados con *Software como Servicio*
- Crear nuevos lenguajes de programación o plataformas. Por el contrario, el trabajo estará basado sobre plataformas, especificaciones, estándares y lenguajes existentes
- Abordar las problemáticas de negocios asociadas a las diferentes alternativas planteadas
- Incluir en el análisis aspectos relacionados con el proceso de desarrollo de software ni sus correspondientes metodologías

# 7 Solución propuesta

La solución incluye definir las características de una aplicación de *Software como Servicio*, definir una taxonomía y por último darle un tratamiento a cada una de ellas. Este proceso se realizará bajo un método llamado *Feature Modeling* , muy utilizado en Software Product Line Engineering (SPLE) [Kang K., 1990]. El objetivo es analizar el dominio de aplicaciones brindadas como servicio (ver 7.1 Análisis de Dominio) que se puede ver como una familia de productos y ensayar un modelo de características (ver 7.2 Modelo de Características) que cubra el dominio holísticamente.

## *Desarrollo de software generativo*

Un concepto clave en el desarrollo de software generativo [Overview of Generative Software Development, 2005] es el mapeo entre el espacio del problema (*problem space*) y el espacio de la solución (*solution space*). El espacio del problema es un conjunto de abstracciones específicas del dominio que pueden ser usadas para expresar las necesidades para ese dominio. Por ejemplo, en un sistema de e-commerce se podría especificar los diferentes métodos de pago. El espacio de solución por otro lado consiste en abstracciones orientadas a la implementación que pueden ser instanciadas para crear soluciones a las especificaciones de dominio expresadas en el espacio del problema. Por ejemplo, los métodos de pago se podrían implementar como llamadas a servicios web como PayPal o a otro Gateway de pagos en caso de utilizar tarjeta de crédito. Si el método de pago es con cupones de descuento o puntos se utiliza una base de datos con un mecanismo que maneja los cupones. El mapeo entre estos dos espacios se traduce en la realización de una instancia específica de la familia de productos potenciales y una técnica para definir ese mapeo es utilizando un modelo de características.

**Figura 12 – Mapeo del espacio del problema al espacio de solución [Overview of Generative Software Development, 2005]**

Durante el desarrollo de esta tesis se utilizó esta separación con el objetivo de definir el problema y desacoplarlo de la o las posibles soluciones.

## *Modelado de características*

El modelado de características es un método y una notación para representar características comunes y variables de los sistemas dentro de una familia de sistemas [Overview of Generative Software Development, 2005]. Este modelo, permite en las etapas tempranas del desarrollo de familias de productos definir el alcance del dominio evaluando e identificando las características importantes del dominio. Utilizando este método se puede priorizar y diseñar una arquitectura tomando las características de a una por vez, dejando registradas las decisiones tomadas en las etapas posteriores.

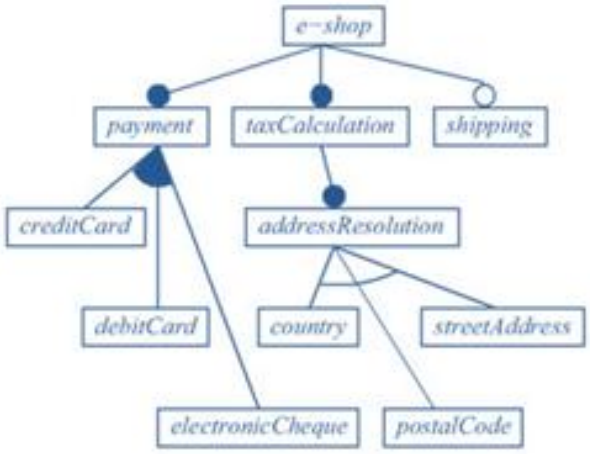



**Figura 13 - Ejemplo de un diagrama de características [Overview of Generative Software Development, 2005]**

El primer paso hacia el modelado de características es el análisis del dominio [Concepts and Guidelines of Feature Modeling for Product Line Software Engineering, 2002]. Sin un entendimiento total del dominio este no puede ser caracterizado y modelado.

## 7.1 Análisis de Dominio

*Software como Servicio* es un dominio emergente e inmaduro. En este contexto diferentes terminologías con el mismo significado pueden ser utilizadas en diferentes contextos. Estandarizar los conceptos del dominio es una actividad esencial para el modelado de características. Si esto no se realiza, las diferentes concepciones del dominio podrían causar confusión en etapas posteriores [Concepts and Guidelines of Feature Modeling for Product Line Software Engineering, 2002]. La caracterización de los conceptos se basó en las siguientes actividades:

- Análisis de la industria
- Participación en proyectos de *Software como Servicio*
- Exploración de papers de la industria, de investigación académica e investigación de mercado
- Interacción con expertos

El siguiente diagrama sintetiza el análisis realizado de las características relacionadas con software brindado como servicio.

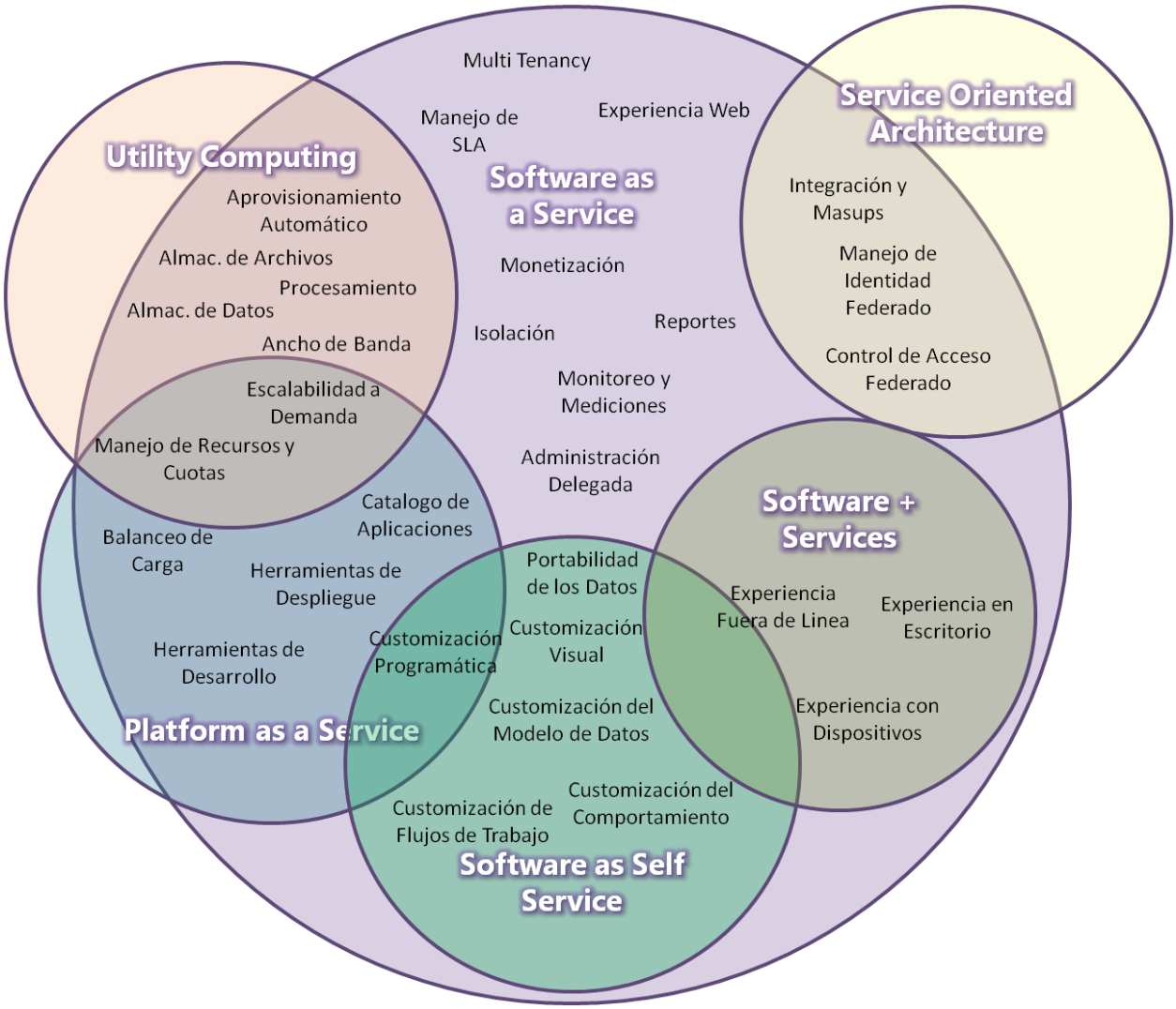



Figura 14 - Análisis del dominio de *Software como Servicio*

*Software como Servicio* se forma a partir de diferentes orígenes como muestra la figura anterior. En el último tiempo el término fue mutando y hoy también se conoce como parte de un concepto más extenso "Cloud Computing". Sin embargo, a la fecha, el término Software como Servicio sigue siendo significativo e identifica al modelo unívocamente. En las secciones subsiguientes se definirá cada una de las ramas del diagrama y las características enunciadas.

### 7.1.1 SaaS y Utility Computing

La primera rama a analizar es *utility computing*. Según Wikipedia la siguiente es la definición de utility computing:

*"Utility computing se define como el suministro de recursos computacionales, como puede ser el procesamiento y almacenamiento, como un servicio medido similar a las utilidades públicas*

*tradicionales (como la electricidad, el agua, el gas natural o el teléfono). Este sistema tiene la ventaja de tener un costo nulo o muy bajo para adquirir hardware; en cambio, los recursos computacionales son esencialmente alquilados. Los clientes que realizan procesamiento de datos a gran escala o que están frente a un pico de demanda, también pueden evitar los atrasos que resultarían de adquirir y ensamblar físicamente una gran cantidad de computadoras."*[7]

Este concepto no es nuevo, sino que data de la época en que IBM y otros proveedores de mainframe ofrecían procesamiento y almacenamiento, también conocido como *time-sharing*, a bancos y grandes corporaciones desde sus datacenters. Los sistemas operativos del mainframe evolucionaron de acuerdo al modelo de negocios que se había creado introduciendo capacidades de seguridad y medición y quota por usuario. Años más tarde, con la llegada de la computadora personal el modelo de negocios había cambiado y las compañías pudieron afrontar el gasto de tener sus propios datacenters.

Volvió a resurgir este modelo, esta vez empujado por la ciencia y la necesidad de mercados específicos como el financiero. La idea de utilizar todas las PC del mundo para resolver problemas que requerían un gran poder de procesamiento se popularizó a finales de los 90 con programas como *SETI@home*. El termino *grid computing* nació como una metáfora relacionada con lograr que el poder de procesamiento de las computadoras sea tan accesible como una red eléctrica (*power grid*).

En el año 2006 la compañía 3tera anunció su servicio AppLogic mediante el cual brinda pool de recursos virtuailizados y almacenamiento. Ese mismo año Amazon lanzó Amazon EC2 (*Elastic Computing Cloud*). Ambos servicios están construidos sobre el software de virtualización *Xen* y corren sobre distribuciones *Linux*. El mercado bautizó este tipo de servicios como *utility computing*.

Las empresas que desean construir una aplicación SaaS ven a los proveedores de *utility computing* como una excelente alternativa ya que reduce costos operativos, de hardware y hasta infraestructura edilicia. Asimismo proveen recursos infinitos, fácilmente aprovisionados, brindando una alta disponibilidad y escalabilidad.

Basado en la definición mencionada anteriormente, *utility computing* lleva el concepto de servicio o utilidad al campo de recursos computacionales, más específicamente ligados al hardware

[7] "Utility Computing". Extraído de Wikipedia, The Free Encyclopedia (http://es.wikipedia.org/w/index.php?title=Utility_computing&oldid=19651049)

(almacenamiento, poder de cómputo y red). Incluye en la definición la palabra "medido", "alquilado" y "costo bajo". Por lo tanto debe existir una arquitectura para optimizar los recursos y albergar a múltiples clientes (*multi tenancy*) logrando un costo bajo (*monetización*) y una economía de escala. El servicio debe ser medido para ser cobrado por lo tanto requiere una actividad de *monitoreo y medición* y el establecimiento de un acuerdo de nivel de servicio *(manejo de SLA)*. Por último, enuncia picos de demanda y atrasos en adquirir y ensamblar computadoras, lo que está relacionado con *escalabilidad a demanda*, *manejo de recursos y cuotas* y *aprovisionamiento automático* para minimizar los tiempos de alta y mantenimiento del servicio.

Entre las compañías más conocidas que trabajan con este modelo se encuentran Amazon (Amazon Web Services), GoGrid, Mosso y otras.

### *Características resumidas*

- Recursos computacionales provistos a demanda
  - Disco
  - CPU
  - Red
- Medición del uso y cuotas
- Acuerdo de nivel de servicio
- Automatización del proceso de aprovisionamiento de los recursos
- Multi tenancy y monetización

### 7.1.2 SaaS y Platform as a Service

Continuando con el análisis, la siguiente es la definición de *platform as a service*:

"*El modelo de platform as a service integra todas las facilidades necesarias para soportar el ciclo de vida de punta a punta para la construcción y la distribución de aplicaciones web y servicios enteramente disponibles via internet… PaaS ofrece facilidades para el diseño de aplicaciones, desarrollo de aplicaciones, testing, despliegue y alojamiento.* "[8]

[8] "Platform as a Service". Extraído de Wikipedia, The Free Encyclopedia (http://en.wikipedia.org/w/index.php?title=Platform_as_a_service&oldid=232984440)

Como indica la definición, *Platform as a Service* engloba al conjunto de soluciones que brinand una plataforma para desarrollar. A diferencia del modelo tradicional donde el desarrollo y el despliegue se realizan en ambientes controlados y en infraestructura propia, tanto el desarrollo como el despliegue se trasladan a internet. Para eso brindan *herramientas de desarrollo* y *herramientas de despliegue* que preparan el código base para ser instalado en la plataforma y luego dependiendo del tipo de plataforma, la posibilidad de compartir la aplicación en un *catalogo de aplicaciones* creando un espacio de comercialización (*marketplace*). Esta rama comparte algunas de las características de *utility computing* como el *manejo de recursos y cuotas* la *escalabilidad a demanda* y agrega otras como el *balanceo de carga* y *customización programática* debido a que el grado de abstracción en el cual se trabaja es a nivel de aplicación y no a nivel de infraestructura.

Dentro de las compañías que construyeron este tipo de plataformas se encuentra Google Inc. (Google App Engine), Bungee Labs y otras.

#### *Características resumidas*

- Desarrollo de aplicaciones bajo una plataforma
    - Escalable a demanda
    - Proporciona balanceo de carga
    - Maneja recursos
- La plataforma nuclea una comunidad de aplicaciones
- El desarrollo es soportado por herramientas de desarrollo, empaquetado y despliegue
    - Estandarización del modelo de aplicación
    - Conociendo el modelo se puede tratar a los componentes a discreción
- Se comparten características con el modelo base de *Software como Servicio* y Utility Computing

### 7.1.3 SaaS y Software as Self Service

En tercer lugar aparece una categoría denominada *software as self service*. Esta rama no se encuentra definida en ningún lado sino que es mi propia aproximación a un conjunto de ofrecimientos relacionados con *Software como Servicio* con un fuerte ingrediente de auto servicio.

En un nivel de abstracción más alto que *utility computing* y *platform as a service*, *software as self service* nuclea aquellas aplicaciones que brindan cierta funcionalidad (CRM, ERP, procuración,

recursos humanos, etc.) apuntando a empresas que buscan customizar la aplicación a sus propias necesidades. El tipo de funcionalidad abarca desde la *customización del modelo de datos*, la *customización del comportamiento* de la aplicación hasta la *customización de flujos de trabajo*. Relacionado con el concepto de self service, el consumidor podría realizar todo esto por sí mismo con alguna aplicación de administración.

#### *Características resumidas*

- Customización de las aplicaciones
  - Modelo de datos
  - Flujos de trabajo
  - Interfaz de usuario
- Herramientas para facilitarle el proceso de configuración al cliente (auto servicio)
- Extensión del modelo base de *Software como Servicio*

### 7.1.4 SaaS y Software + Services

El término "Software + Services" fue concebido por Microsoft con el objetivo de ponerle un nombre a la estrategia relacionada con *Software como Servicio*. La categoría software + services aporta otras características que no son parte del modelo base, pero que agregan valor al modelo en general. La siguiente definición, extraída del sitio de Microsoft, resume el concepto:

"*Software-plus-Services is an industry shift - combining Internet services with client and server software to deliver more compelling opportunities and solutions… Microsoft helps its customers and partners capitalize on the opportunities presented by the evolution to Software-plus-Services. Microsoft's platform uniquely supports all classes of the services manifestation — services delivery (Saas), services composition (SOA) or the services experience (Web 2.0) — across a range of devices in addition to the PC, the various types of browser clients, servers in data-centers and Internet services.*"
[Microsoft Corp., 2008]

La combinación de servicios de internet con software cliente y de servidor como indica la definición y el acceso a través de diferentes dispositivos, no solamente la PC es la visión de Microsoft. En resumen, no solamente la experiencia web es importante, sino también la *experiencia con dispositivos*, la *experiencia del escritorio* y la *experiencia sin conexión*.

#### *Características resumidas*

- Visión de Microsoft
    - Flexibilidad
    - Mezcla de los dos mundos (internet y PC)
- Extiende la experiencia del usuario
    - Web, Escritorio, Dispositivos
- Experiencia Offline
- Extensión del modelo base de SaaS

### 7.1.5 SaaS y Service Oriented Architecture

"*Service-oriented architecture (SOA) is a methodology for systems development and integration where functionality is grouped around business processes and packaged as interoperable services.*"[9]

Según la definición, SOA permite exponer funcionalidad empaquetado como servicios interoperables que permiten la integración. Es decir SOA agrega un ingrediente importante al modelo que está relacionado con la *integración* y la interoperabilidad con aplicaciones de terceras parte o propias. SOA entonces es un sub conjunto que agrega valor a una solución SaaS no solo en la integración a nivel funcional sino también a nivel *seguridad* con estándares como WS-Trust y WS-Federation.

#### *Características resumidas*

- SOA agrega valor a SaaS
    - Permitiendo que la aplicación SaaS no se convierta en un silo
    - Habilitando la composición de nuevas aplicaciones
- Permite pensar en un modelo de seguridad federado
- Sub conjunto del modelo base de SaaS

### 7.1.6 Análisis del Mercado

A la fecha de realización de esta tesis la siguiente es una fotografía del mercado. Se relevaron las soluciones y productos más significativas y a partir de un análisis se las categorizó en su respectiva esfera. Las categorizaciones no son determinísticas y en ciertos casos existe un solapamiento.

[9] "Service Oriented Architecture". Extraído de Wikipedia, The Free Encyclopedia (http://en.wikipedia.org/w/index.php?title=Service-oriented_architecture&oldid=234288942)


Figura 15 - Taxonomía del mercado

Luego de haber realizado un primer análisis de las diferentes ramas que influencian el *Software como Servicio*, en las siguientes secciones se estudiará con más detalle cada una de estas características.

### 7.1.7 Capacidades intrínsecas del modelo

Existen un conjunto de características que pueden clasificarse como intrínsecas o base del modelo de *Software como Servicio*. El concepto de multi tenancy y la posible aislación de cada cliente, la experiencia del usuario, la monetización de la aplicación, el monitoreo y medición, el manejo de acuerdos de niveles de servicio, la disponibilidad de los datos y la administración delegada son algunas de las características que fueron mencionadas en la sección anterior.

#### 7.1.7.1 Multi tenancy y aislación

Una de las características arquitectónicas más asociadas a este modelo es *multi tenancy*. Este término se podría traducir como *multiples inquilinos*. Es decir, un mismo código base y/o infraestructura que sirva a múltiples clientes (*tenants*). Por ejemplo, en un contexto en el cual una aplicación bancaria es ofrecida como servicio por una empresa de la industria financiera, *multi tenancy* se refiere a la posibilidad de ofrecer el servicio a múltiples bancos compartiendo entre todos una única instancia de la aplicación bancaria.

Una aplicación puede ser diseñada para un único cliente, para múltiples clientes o para múltiples clientes y adicionalmente proveer la capacidad de customizar la instancia. El nivel de complejidad aumenta y el tiempo y costo posiblemente también. La siguiente tabla ilustra las diferencias en las diferentes etapas de un proyecto.

| | Etapa de Diseño y Desarrollo | Etapa de Ejecución |
|---|---|---|
| **Multi tenant** | Implementar un feature cuesta más (costo)<br>Salir al mercado lleva más tiempo (tiempo) | Agregar un tenant cuesta menos (costo) |
| **Single tenant** | Implementar un feature cuesta menos (costo)<br>Salir al mercado lleva menos tiempo (tiempo) | Agregar un tenant cuesta más (costo) |

Durante el diseño y la construcción sería más costoso implementar la aplicación *multi tenant* pero en el largo plazo esto debería tener un retorno de la inversión (ROI) más alto a medida que se agreguen nuevos tenant durante la operación del sistema [Carraro, 2007].

Por otro lado, al compartir la misma instancia aparecen nuevos desafíos. Por ejemplo, si cierta empresa hace un uso intensivo del servicio, el proveedor puede querer, en primer lugar detectar esto y en segundo lugar tomar acciones. Estas acciones pueden ser aislarlo o moverlo a un cluster junto con otros tenants que no utilicen tantos recursos. Esta característica se denomina aislación. Otro desafío se manifiesta cuando el servicio ofrece capacidades relacionadas con customización. En este escenario, si la plataforma no tiene los mecanismos de protección necesarios el tenant podría realizar una customización que consuma considerables recursos, perjudicando a otros tenants.

#### 7.1.7.2 Experiencia del usuario

Otra característica de las aplicaciones ofrecidas como servicio es la usabilidad y la experiencia del usuario final. La web y el browser es un medio ubicuo hoy en día y el modelo *Software como Servicio* lo adopta como canal de distribución principal. La mayoría de los proveedores exponen su software al menos vía web.

En etapas iníciales del modelo, los proveedores ofrecían únicamente acceso por internet y via web, pero de a poco fueron incorporando capacidades offline y accesos via dispositivos móviles por la demanda de los consumidores.

En particular, Microsoft bajo su programa *software más servicios* pregona la importancia de combinar lo mejor de los dos mundos. Es decir, la composición de software local con servicios de internet interactuando entre sí.

#### 7.1.7.3 Esquemas de monetización, monitoreo y medición, manejo de SLA

Este grupo de características se define de la siguiente manera

- Cobrar por el uso de la aplicación
- Medir y monitorear el uso del servicio
- Manejar un acuerdo de nivel de servicio

Los esquemas de monetización más utilizados son: por subscripción, por utilización o basado en publicidad. Por ejemplo, *Salesforce* monetiza su aplicación de CRM cobrando una subscripción por usuario. Los servicios S3 o EC2 de Amazon se pagan por utilización, es decir solo se paga lo que se utiliza. Además, los servicios de Amazon varían su precio según si el cliente desea utilizar los datacenters europeos o estadounidenses. Por último, una manera menos ortodoxa de monetizar especialmente usada en servicios con importante tráfico, es la utilización de publicidad, en donde por cada impresión y/o click el proveedor cobra una comisión.

Cuando un proveedor elije cobrar por el servicio a sus clientes, estos esperan una determinada calidad de servicio. El proveedor comienza a manejar algún tipo de acuerdo de nivel de servicio, a monitorearlo, medirlo y reportarlo. Para esto se definen *Service Level Objectives (SLO)*, que son objetivos de un determinado nivel de servicio para un recurso o un conjunto de recursos. Este puede ser expresado en unidades como tiempo de respuesta promedio o en función de la disponibilidad

mensual de un cierto servicio. Un SLA debería estipular el pago y/o las penalidades asociadas al incumplimiento de un criterio acordado.

#### *7.1.7.4 Administración delegada*

Otra de las características comunes del modelo es la delegación de la administración. El concepto de administración delegada proviene del uso de servicios de directorios para administrar usuarios y roles que es muy utilizado en grandes empresas multi departamentales.

Este concepto extrapolado a otras tareas administrativas se puede encontrar en aplicaciones de *Software como Servicio*. El cliente o *tenant* podría realizar tareas administrativas relacionadas a la aplicación utilizando alguna herramienta evitando al proveedor incurrir en costos de administración y agilizando las operaciones. Por ejemplo, si el tenant desea habilitar un nuevo usuario para que acceda a la aplicación, lo podría hacer desde un panel de control sin intervención del proveedor.

### 7.1.8 Capacidades relacionadas con Service Oriented Architecture (SOA)

El software brindado como servicio es un modelo de negocios que puede o no incluir un estilo de arquitectura basado en servicios. Adoptar SOA puede ser una ventaja competitiva para el proveedor porque habilita escenarios de integración, mashups[10] y automatización.

#### *7.1.8.1 Integración, mashups y automatización*

La integración de aplicaciones representa la mayor preocupación en una encuesta realizada por Forrester. Más de un 60% alega que no está interesado en adoptar *Software como Servicio* porque no ve una clara estrategia de integración con su portfolio de aplicaciones [Forrester Research, 2008]. Por ejemplo, una empresa que desea crear un panel de control integrado de recursos humanos y mantiene su nómina de empleados en un sistema in-house y por otro lado contrata una aplicación de reclutamiento bajo el modelo SaaS se encuentra con el desafío de interactuar y agregar los datos de ambas aplicaciones. Si el proveedor que ofrece el servicio expusiese los servicios de la aplicación, el cliente podría consumirlos y crear el panel de control sin intervención del proveedor.

[10] Una aplicación "Mashup" es un sitio web o aplicación web que usa contenido de otras aplicaciones Web para crear un nuevo contenido completo, consumiendo servicios directamente siempre a través de protocolo http://es.wikipedia.org/wiki/Mashup_(aplicaci%C3%B3n_web_h%C3%ADbrida)

En una encuesta realizada en julio de 2008, se reveló que salesforce.com procesa por día 150 millones de transacciones y más de la mitad provienen de otras aplicaciones que no son nativas de Salesforce [ProgrammableWeb, 2006]. Esto demuestra la relevancia de proveer mecanismos que permitan al cliente integrar sus aplicaciones tradicionales con aquellas en "la nube".

No obstante, la exposición de los servicios puede ser una brecha de seguridad que violaría la confidencialidad de los datos del consumidor. Por tal razón, el proveedor debe pensar una estrategia para asegurar los servicios.

#### *7.1.8.2 Manejo de identidad y control de acceso federado*

El manejo de identidad y la seguridad es otra de las preocupaciones de los consumidores, un 50% reveló esta preocupación según encuestas [Forrester Research, 2008]. En un análisis más detallado, las empresas ven un riesgo en tener usuarios con un perfil de negocio con poco entrenamiento, manejando los usuarios, roles y controles de acceso de la aplicación, caso que se da seguido en aplicaciones SaaS con *administración delegada*.

Una característica que hoy no está del todo explotada en *Software como Servicio* es el manejo federado de la identidad y los controles de acceso. Mediante soluciones de Identidad Federada[11] los usuarios pueden emplear la misma identificación personal (típicamente usuario y contraseña) para identificarse en redes de diferentes departamentos o incluso empresas. De este modo las empresas comparten información sin compartir tecnologías de directorio, seguridad y autenticación, como requieren otras soluciones. Para su funcionamiento es necesaria la utilización de estándares que definan mecanismos que permiten a las empresas compartir información entre dominios. El modelo es aplicable a un grupo de empresas o a una gran empresa con numerosas delegaciones y se basa en el "círculo de confianza" de estas, un concepto que identifica que un determinado usuario es conocido en una comunidad determinada y tiene acceso a servicios específicos.

En el contexto de *Software como Servicio*, las empresas que formarían la federación son el consumidor y el proveedor. De esta manera, el personal de IT de la empresa que consume el servicio manejaría un único repositorio de usuarios y roles.

[11] "Identidad Federada". Definición extraída de Wikipedia, The Free Encyclopedia (http://es.wikipedia.org/wiki/Identidad_federada)

### 7.1.9 Capacidades relacionadas con utility computing y platform as a service

En el año 2006, la aplicación SmugMug que ofrece un servicio de almacenamiento de fotos, publicó el ahorro logrado al utilizar Amazon S3 para almacenar y servir los 193TB de fotos de sus usuarios. Por año ahorrarían 692.000 dólares entre costos de discos y personal y aumentarían la disponibilidad de su aplicación al confiar en el servicio de Amazon [Amazon.com, 2006].

Los proveedores que se aventuran en la construcción de una aplicación de *Software como Servicio* pueden decidir utilizar recursos ubicuos no sólo como una alternativa que reduce costos operativos, de hardware y hasta infraestructura edilicia sino que también como parte del núcleo de su arquitectura y estrategia de escalabilidad y disponibilidad.

Como toda innovación, requiere de tiempo para que sea adoptada por compañías que tienen un departamento de IT establecido. Sin embargo, podrían encontrar usos a estos servicios en casos donde se requiera de poder computacional y almacenamiento a escala. Un ejemplo de esto es el diario New York Times que utilizó Amazon EC2 y S3 para procesar 11 millones de artículos de diarios desde 1851 a 1922 convirtiéndolos a PDF en sólo un día [Gottfrid, 2007].

A continuación se detallan las características más significativas de este tipo de servicios.

#### *7.1.9.1 Recursos computacionales*

Los recursos computacionales a demanda incluyen capacidades de almacenamiento de datos, de archivos, poder de procesamiento y ancho de banda.

En el almacenamiento de datos, Google BigTable es un ejemplo de una base de datos (basada en tablas de hash distribuidas) y Microsoft SQL Server Data Services es una base de datos más orientada al modelo relacional. Ambas son manejadas y provistas como servicios por sus correspondientes proveedores y permiten al consumidor guardar y extraer datos utilizando un lenguaje de consulta.

En segundo lugar se encuentra el almacenamiento de archivos y el ancho de banda. El proveedor ofrece capacidad de almacenamiento virtualmente infinito, cobrando por el espacio utilizado y por el ancho de banda. El ejemplo más conocido es Amazon S3 que ofrece espacio de almacenamiento que es manejado mediante una API de webservices. Las content delivery networks (CDN) también podrían entrar en esta categoría, con la salvación de que este servicio está orientado a imágenes solamente.

Otro de los recursos disponibles es el poder de procesamiento. Los proveedores que ofrecen este servicio utilizan la virtualización y permiten crear un número arbitrario de máquinas virtuales en donde el usuario tiene la libertad de manejarlas.

#### *7.1.9.2 Platform as a service*

*Platform as a service* puede verse como una especialización de *utility computing* en donde se añaden servicios de valor agregado sobre la infraestructura base. Es decir, son servicios ubicados en una capa de abstracción más alta que permite no solamente utilizar recursos computacionales sino también desarrollar y desplegar aplicaciones sobre esos recursos y entregar recursos de plataforma. Similar a un *application server* pero disponible en la red y con capacidades pensadas para el desarrollo de aplicaciones de *Software como Servicio*. Este tipo de plataformas cuentan con herramientas de empaquetado de aplicaciones, scripts, SDK y hasta un ambiente de desarrollo offline. Google App Engine es un ejemplo de un proveedor de plataforma de aplicaciones que expone la masividad de los recursos de Google para desarrollar utilizando Python como lenguaje de programación y expone una API para brindar almacenamiento en BigTable.

Las plataformas pueden ofrecer la publicación de la aplicación en un catálogo y así exponer la aplicación aprovechando la base de usuarios de la plataforma. Esto es lo que hacen plataformas como Facebook o Salesforce AppExchange.

Otra característica que puede ofrecer este tipo de plataformas es el balanceo de carga, que se define como la capacidad de distribuir la carga entre instancias de servidores de aplicación. Relacionado con el *aprovisionamiento automático* de instancias y la *escalabilidad a demanda*, la característica de balancear la carga automáticamente entre los servidores permite hacer un mejor uso de los recursos en una aplicación expuesta públicamente en internet.

Las compañías con cierto volumen de aplicaciones pueden explorar el concepto de plataforma de aplicaciones en el contexto de una estrategia corporativa, automatizando y estandarizando su portfolio de aplicaciones y aprovechando la cantidad de recursos computacionales disponibles. Este concepto se conoce también como *private cloud* o *IntraSaaS*.

#### *7.1.9.3 Escalabilidad a demanda*

La escalabilidad a demanda se define como la capacidad de aumentar o disminuir los recursos a medida que el consumidor lo requiera. Esto aplica a todo tipo de servicio que pueda escalar

horizontalmente como almacenamiento, poder de procesamiento e instancias de servidores web y representa una típica característica ofrecida en este tipo de servicios.

#### *7.1.9.4 Aprovisionamiento automático y Manejo de quota y recursos*

Una de las características fundamentales de *utility computing* es la automatización del proceso de aprovisionamiento del servicio. El servicio no escalaría si habría una intervención humana por cada nueva instancia a crear. El proceso de aprovisionamiento será un conjunto de pasos estipulados o un workflow en el cual se crearán, según el Acuerdo de Nivel de Servicio (SLA) establecido por el servicio seleccionado y los recursos disponibles, las instancias necesarias.

El aprovisionamiento está relacionado con el manejo de recursos y quotas. Es decir, la arquitectura de este servicio debe contemplar el manejo de recursos y sus respectivos límites.

Por ejemplo, un tenant contrata una aplicación para manejo de proyectos bajo el modelo *Software como Servicio*. El proveedor aprueba la orden y lanza el aprovisionamiento del cliente. El servicio de aprovisionamiento puede decidir ubicar al tenant en un servidor web "Y" porque el servidor "X" no tiene recursos disponibles (la cantidad de recursos llegó al límite).

Otro caso sería un tenant utilizando una *plataforma de aplicaciones* que según el SLA soporta una cantidad X de requests por día (*quota*). El tenant podría pedir un aumento de la *quota* y el proveedor aprovisionar un nuevo servidor web y un balanceador de carga que aumente a 2X la cantidad de requests por día.

En resumen, las características básicas de *utility computing* son el manejo de quotas y recursos, el aprovisionamiento automático y la escalabilidad a demanda. Combinando estas capacidades se logra el manejo de recursos computacionales (*Infrastructure as a Service*) o recursos de plataforma (*Platform as a Service*) como ilustra la siguiente figura.

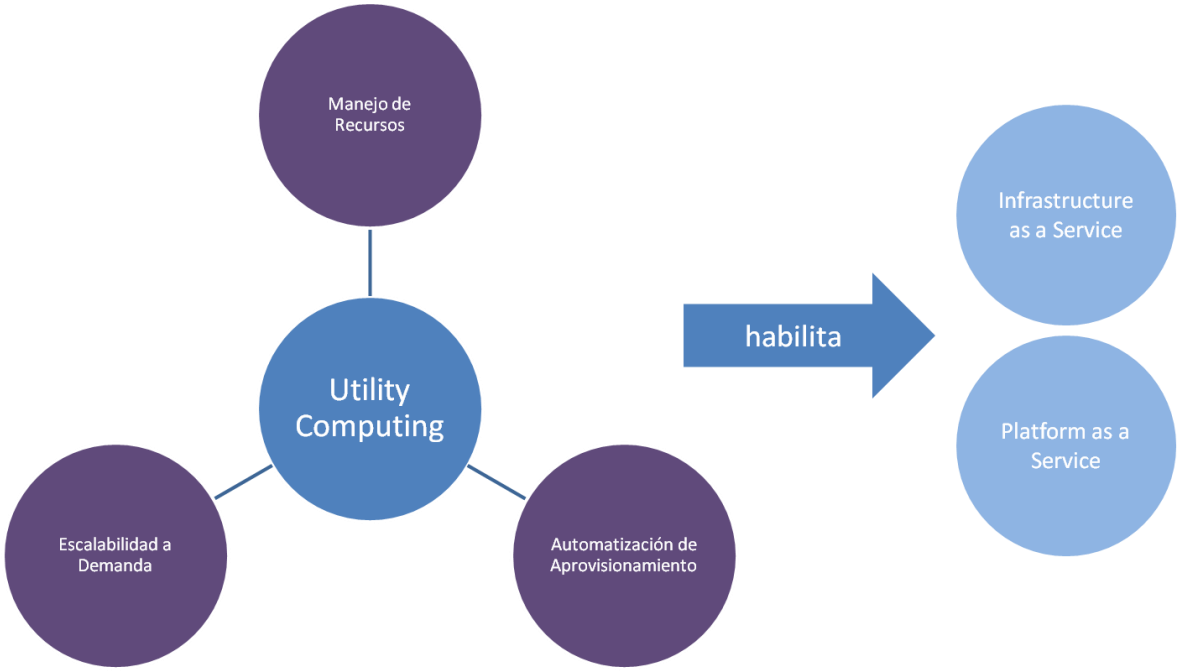



Figura 16 - Características de *utility computing*

### 7.1.10 Capacidades relacionadas con customización y configuración

Diferentes tipos de aplicaciones requieren diferentes niveles de configuración y customización. Un ejemplo cotidiano son las aplicaciones de escritorio, como ser hojas de cálculo o procesadores de texto, que son configuradas por el usuario final generalmente utilizando menús y wizards. Por otro lado, aplicaciones más complejas como Enterprise Resource Planning (ERP) o Customer Relationship Management (CRM) requieren compañías integradoras o desarrolladores que escriban el código específico para las necesidades de la empresa.

La posibilidad de configurar y customizar una aplicación de software está relacionada con el concepto de *Software Product Lines* o *Líneas de Producto* [Kang K., 1990]. El objetivo de *Líneas de Productos* de software es identificar oportunidades de reusabilidad en una línea de producto. Para eso identifica las responsabilidades del sistema que resuelven los requerimientos funcionales comunes y a su vez la variabilidad del dominio en cuestión. La diferencia radica en que en el caso de SPL esto ocurre en las etapas de diseño y construcción y en *Software como Servicio* ocurre en tiempo de ejecución. Es decir el usuario del sistema es el que decide la variabilidad utilizando herramientas de configuración y el proveedor es el que brinda la plataforma para que eso suceda.

En este escenario es fundamental acotar el espectro de *configurabilidad* a un conjunto que permita mantener los acuerdos de nivel de servicio sin poner en riesgo a otros *tenants*. Ligado a esto se definen capacidades como *nivel de aislación del tenant*.

El primer paso para lograr un sistema configurable es que el proveedor tenga herramientas para manejar la variabilidad de cada *tenant*. Una vez que detectadas las bifurcaciones del sistema original con cada instancia se puede comenzar a explorar la exposición de herramientas de *auto servicio* para transferirle la autoridad al *tenant*.

Los cuatro grupos de capacidades principales son *customización visual*, *customización del comportamiento*, *customización del modelo de datos* y *configuración autoservicio*.

#### *7.1.10.1 Customización del modelo de datos*

La capacidad de extender el modelo de datos se compone de dos características:

- Poder agregar nuevas columnas a una entidad del modelo
- Poder agregar nuevas entidades

En la primera se trata de extender una o varias entidades de la definición base del modelo de datos con nuevos campos. Por ejemplo, un catálogo de productos que posee la entidad *Producto* posee los atributos base: nombre y precio. Sin embargo, cierta empresa que comercializa fertilizantes está interesada en incluir en la definición de producto un campo que indique si se trata de un fertilizante orgánico o no y además quiere incluir el costo de producción.

La segunda característica puede ser más compleja ya que la variabilidad es más importante. Siguiendo el ejemplo del fertilizante, a la empresa podría interesarle almacenar el químico principal que compone a cada fertilizante. Para eso deben introducir una nueva entidad Componente y extender la entidad Producto con un campo de referencia a Componente.

Salesforce.com provee esta característica a sus clientes y de hecho han patentado el mecanismo de extensibilidad que utilizaron [Weissman y otros, 2005].

#### *7.1.10.2 Customización del comportamiento*

La capacidad de extender el comportamiento de una aplicación se compone de tres características:

- Poder reemplazar funcionalidad por implementaciones específicas para un tenant
- Poder modificar los flujos de trabajo

La modificación de ciertos servicios está relacionada con estrategias, reglas de negocio o algoritmos específicos del dominio de la aplicación. Por ejemplo, en un sistema de cálculo de salarios el algoritmo de cálculo podría diferirse según el país donde opera el cliente. El objetivo es que el proveedor en primera instancia pueda modificar estas reglas sin tener que modificar gran parte de la aplicación y en segunda instancia exponer al tenant esta capacidad promoviendo el *self-service*.

La modificación de flujos de trabajo apunta primero a poder definir los flujos de trabajo de la aplicación en cuestión y luego a identificar la variabilidad y habilitar la extensibilidad necesaria para poder realizar las modificaciones a la instancia del *tenant*. Dentro de esta característica se encuentra la posibilidad de modificar expresiones que definan una decisión dentro del workflow. Un ejemplo podría ser la expresión "si el monto de la compra es mayor a X", donde X y monto de compra son configurables (if compra.monto > X).

Un segundo nivel de variabilidad se manifiesta en la posibilidad de agregar actividades en medio de un flujo de trabajo en lugares preestablecidos por el proveedor. Como ejemplo de este caso se podría pensar en el deseo de un tenant de enviar un email a todos los participantes de la compra una vez que se decidió el monto de compra y esta fue ejecutada. Otro ejemplo más complejo podría incluir la llamada a un webservice y así permitir al tenant realizar lo que desee mientras cumpla con el contrato del servicio.

La modificación de flujos de trabajo podría ser expuesta al tenant y así aplicar a esta característica la capacidad de self-service. Sin embargo si el proveedor desea que esto ocurra, debe manejarse con extremo cuidado y proteger a los demás tenants de las modificaciones que puedan perjudicarlos.

#### *7.1.10.3 Customización visual*

La capacidad de modificar la apariencia o la presentación de la aplicación se basa en las siguientes características:

- Poder modificar la manera en que se visualiza una entidad
- Poder customizar el estilo de la aplicación
- Poder customizar el layout de la aplicación
- Poder customizar los mensajes e imágenes

El concepto de *customización visual* está relacionado con *white labeling*[12] que se utiliza en marketing haciendo alusión a un producto o servicio producido por una compañía (productora) que es *remaracado* por otra compañía (comercializadora) simulando que fue de su propia producción.

La customización del estilo y layout permite dar una identidad visual a la aplicación que se alinee a los estándares de la empresa que utiliza el servicio. Como ejemplo se puede citar una plataforma de *blogging*, donde la empresa no quiere utilizar el formato default sino que quiere modificarlo con el objetivo de difundir su propia identidad no solo mediante el contenido sino también la forma en que se ve.

Tanto el estilo como el layout son customizaciones ubicadas en un primer nivel de dificultad y aplicable a un espectro mayor de aplicaciones. Más ligado a las aplicaciones de negocio y a la característica de *extensión del modelo* está la customización visual de una entidad y de tipo de datos. Por ejemplo, un tenant desea ordenar y agrupar los campos de cierta entidad con el objetivo de mantener una disposición similar a la que los operadores estaban acostumbrados en el sistema anterior. Otro tipo de customización más avanzada podría ser visualizar un campo de texto que guarda una georeferencia con un control de mapa.

Otra característica relacionada con la visualización es la capacidad de modificar los mensajes o imágenes de acuerdo a ciertos parámetros. Un claro ejemplo es poner el logo de la empresa o definir mensajes según el segmento al cual pertenece el cliente.

Resumiendo esta sección, la siguiente figura muestra los atributos principales relacionados con configuración y customización. En el medio aparece el concepto de *self service* que es una característica opcional aplicable a estos tres atributos que habilita al tenant a que customize por sus propios medios.

[12] "White Labeling". Definición extraida de Wikipedia, The Free Encyclopedia (http://en.wikipedia.org/w/index.php?title=White_label_product&oldid=234559087)

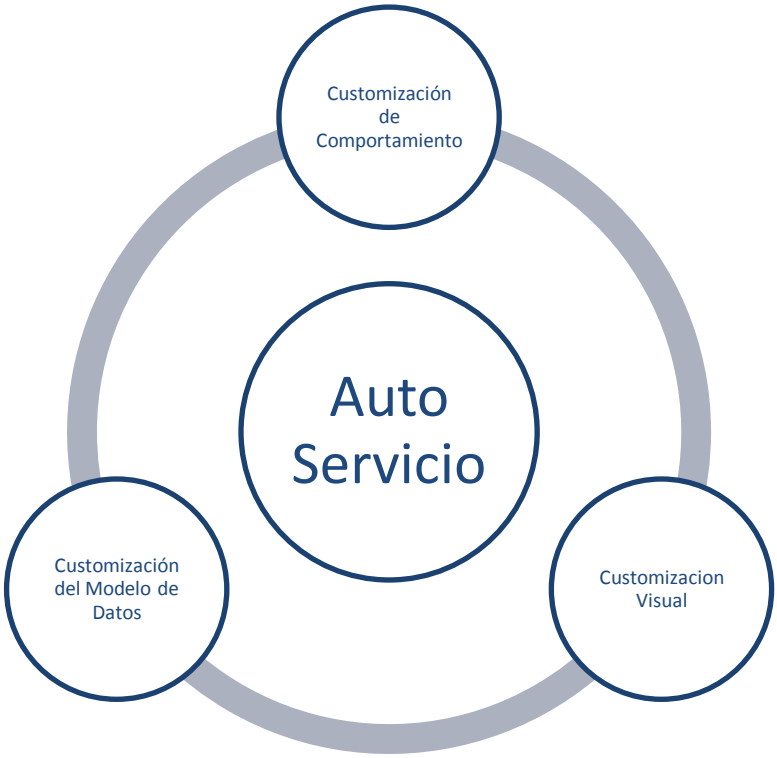



**Figura 17 - Las esferas de customización y el auto servicio. El auto servicio no es una condición obligatoria para lograr customización y debe planearse con extrema precaución.**

## 7.2 Modelo de Características

Luego de haber realizado el análisis de dominio, las características de alto nivel han sido definidas. El siguiente paso es organizarlas de modo tal que los aspectos comunes y la variabilidad se destaquen. El objetivo final es crear el modelo de características (*feature model*) que permita identificar y seleccionar las características que definan un proyecto de *Software como Servicio*.

El siguiente diagrama proporciona una vista de capas de las categorías principales relacionadas con características de *Software como Servicio*.

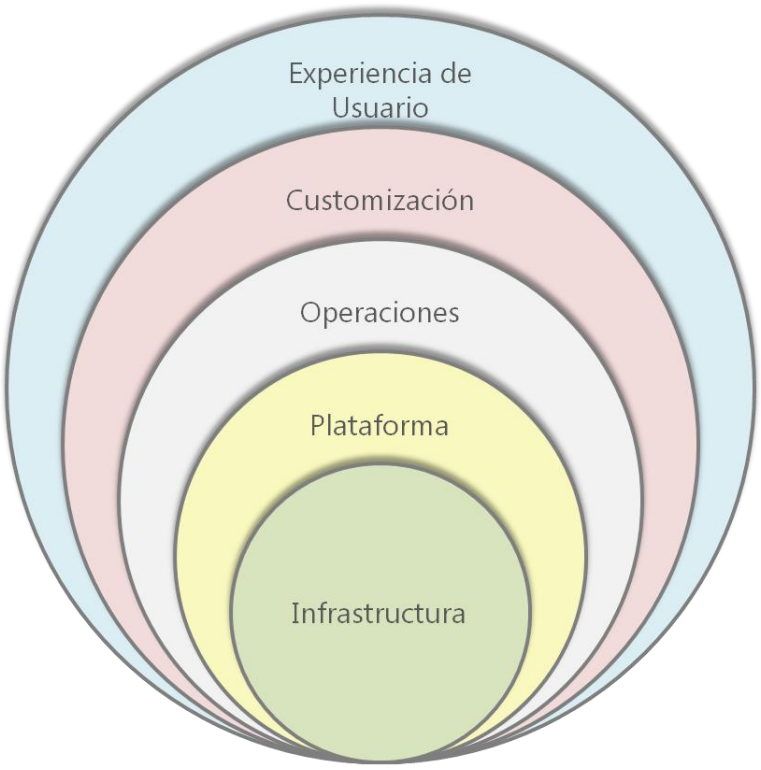



**Figura 18 - Modelo de capas para categorizar las características de Software como Servicio**

El formato de aros concéntricos (capas de cebolla) se asemeja a la forma en que está diseñado el kernel de UNIX, múltiples capas que agregan funcionalidad a la capa anterior.

En el centro, se encuentran las características de infraestructura que cualquier solución de software debe incluir. Un nivel más arriba se encuentra las características de plataforma, intrínsecas al modelo *Software como Servicio*, como multi tenancy. Luego la capa de operaciones agrega un conjunto de características relacionadas con la operación de la aplicación, como el monitoreo, la medición, manejo de SLA, etc. La capa de customización adiciona las características de configuración y customización de la aplicación. Por último, se encuentran las características

relacionadas con la experiencia del usuario, como ser la interfaz web, el acceso con dispositivos móviles y la experiencia fuera de línea.

La disciplina o el método de *feature modeling*, recomienda separar en partes la clasificación de las características. La *capa de capacidades (capability layer)* organiza las características en términos entendibles por el consumidor del modelo y hasta se podría pensar en términos de unidades de incremento de valor en una línea de producto. Más adelante se definirá la *capa de implementación (implementation technique layer)*. Las características en esta capa se constituyen a partir de decisiones claves de diseño e implementación que serán "mapeadas" por características que se encuentran en la *capa de capacidades* [FORM: A Feature-Oriented Reuse Method with Domain Specific Reference Architectures, 1998]. Por ejemplo, la *customización del modelo de datos* es una característica que se encuentra en la *capa de capacidades*. En la *capa de implementación* se ofrecerán un conjunto de alternativas como la utilización de una base de datos sin esquema fijo (tabla de hash distribuidas) o en el caso de utilizar base de datos relacional, los diferentes patrones para implementar la capacidad (tabla de clave valor, columnas extra, columna Xml).

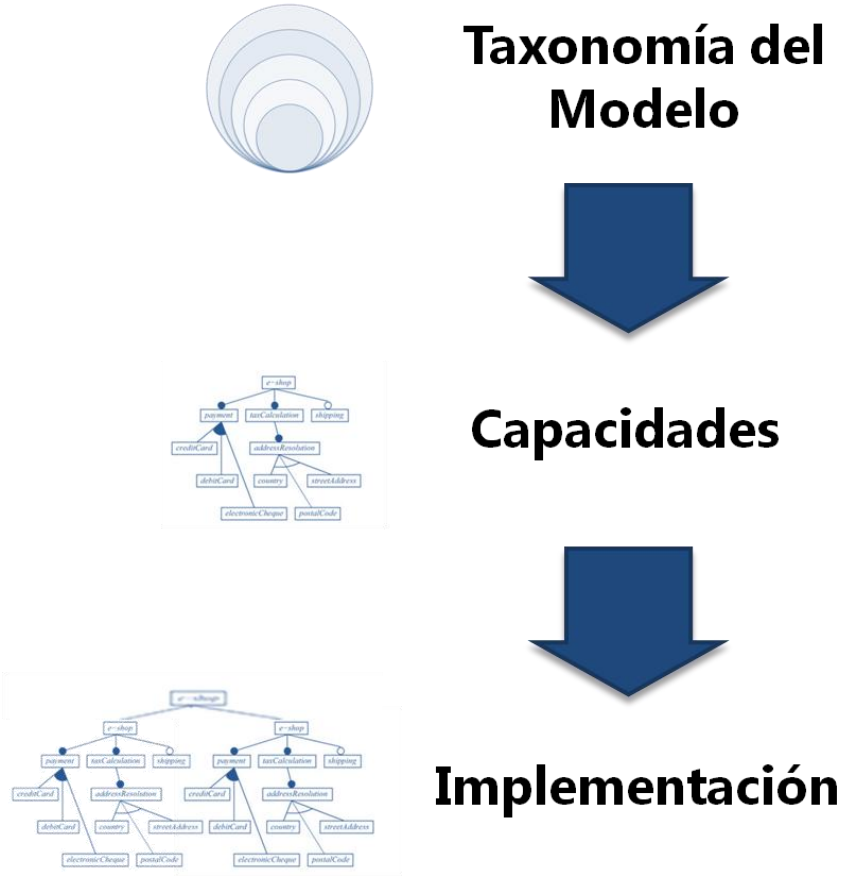



**Figura 19 – Refinamiento del modelo desde la taxonomía a la capa de capacidades y luego a la capa de implementación**

## 7.3 Capacidades

A continuación se ilustra la capa de capacidades (*capability layer*) del modelo de características diseñado a partir del análisis de dominio.

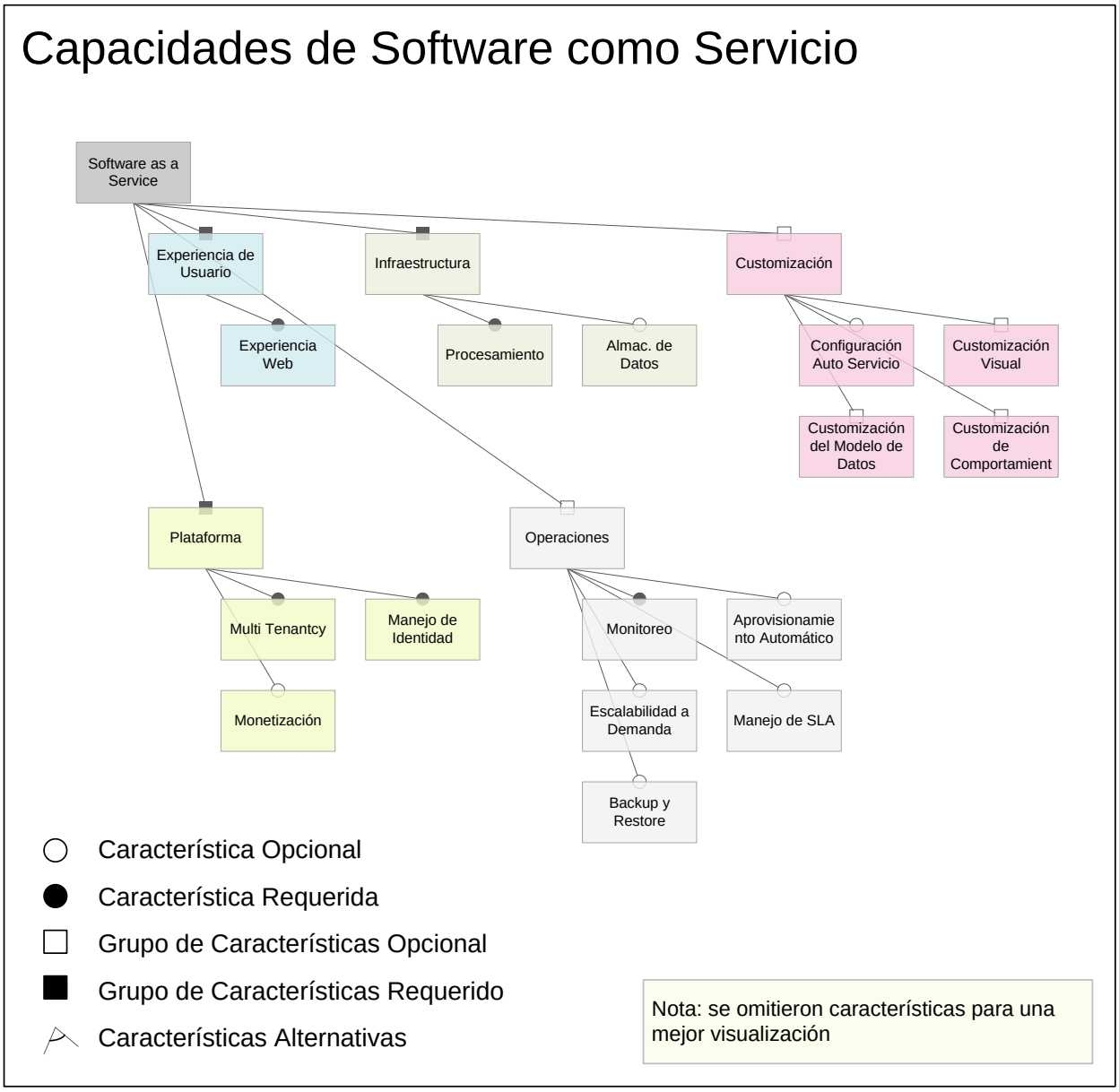



Figura 20 - Capacidades del modelo de características de SaaS

En las siguientes secciones se pueden ver cada una de las categorías del diagrama en detalle.

### 7.3.1 Infraestructura

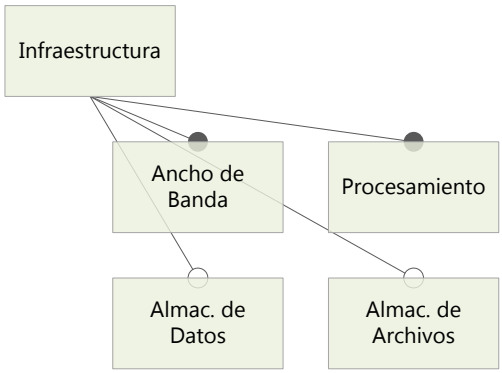



Figura 21 - Características ligadas a la infraestructura

En este grupo de características encontramos elementos obligatorios de cualquier solución de software. Se requiere poder de procesamiento y ancho de banda y dependiendo de la solución opcionalmente almacenamiento de datos y/o de archivos. Más adelante en la capa de implementación se plantearán alternativas más relacionadas con *Software como Servicio*.

### 7.3.2 Plataforma

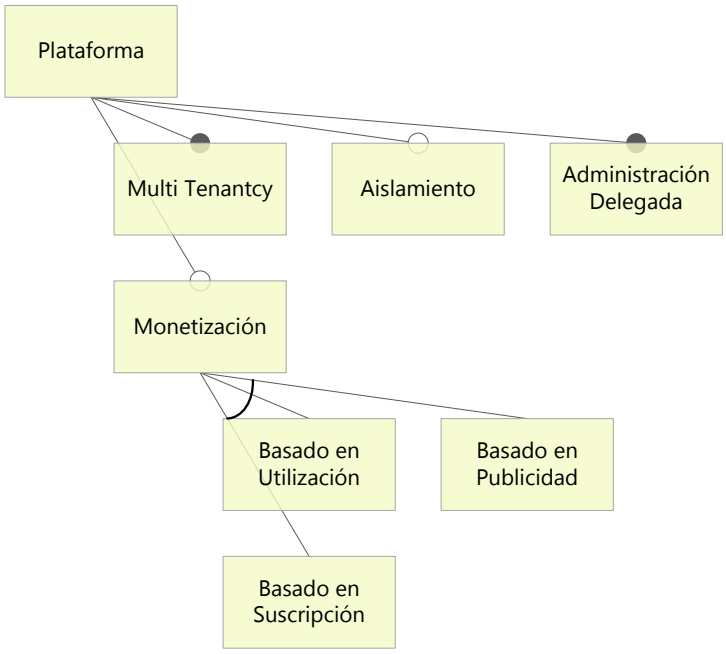



Figura 22 - Características ligadas a la plataforma

Dentro de esta categoría, multi tenancy y administración delegada son características necesarias para el modelo. Cualquiera sea el tipo de aplicación, se parte de la premisa que va a ser brindada a múltiples clientes y estos clientes tendrán la posibilidad de administrarla sin intervención del

proveedor. Se analizará en la capa de implementación los patrones y principios de arquitectura asociados con multi tenancy. Tanto monetización como aislación son características opcionales en la plataforma. Es decir puede haber casos que se decida no cobrar por el uso de la aplicación, como podría pasar en el contexto de una plataforma corporativa. También puede suceder que no traiga ningún beneficio aislar a cada cliente, eso dependerá de las características de la capa de customización.

### 7.3.3 Operaciones

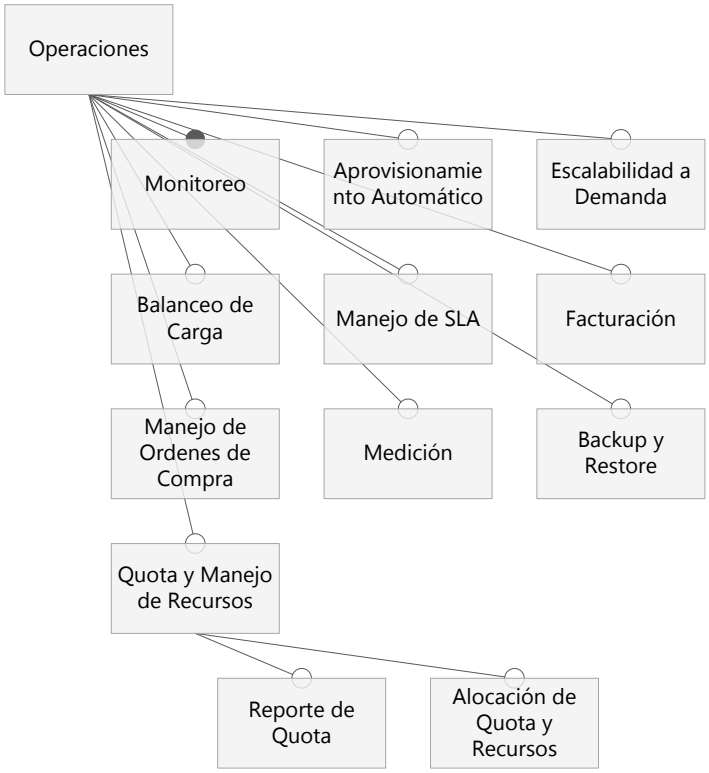



Figura 23 - Características relacionadas con la operación de una aplicación *Software como Servicio*

Este conjunto de características se relaciona con la operación post implementación de la solución. Una aplicación de *Software como Servicio* es posible que esté diseñada para minimizar los errores y manejar las excepciones. Sin embargo, los errores ocurren y es necesario tener un ambiente operacional con las herramientas necesarias para detectar, diagnosticar y resolver los potenciales errores. Luego, si el producto va a ser comercializado, deben manejarse las órdenes de compra de los clientes, medir el uso del servicio y facturar de acuerdo a eso. Durante la operación también puede requerirse que los recursos estén manejados y si fuese necesario, escalarlos a demanda mediante un aprovisionamiento automático. Por último, el ambiente de operaciones puede ofrecer copias de resguardo y restauración de los datos del cliente.

### 7.3.4 Customización

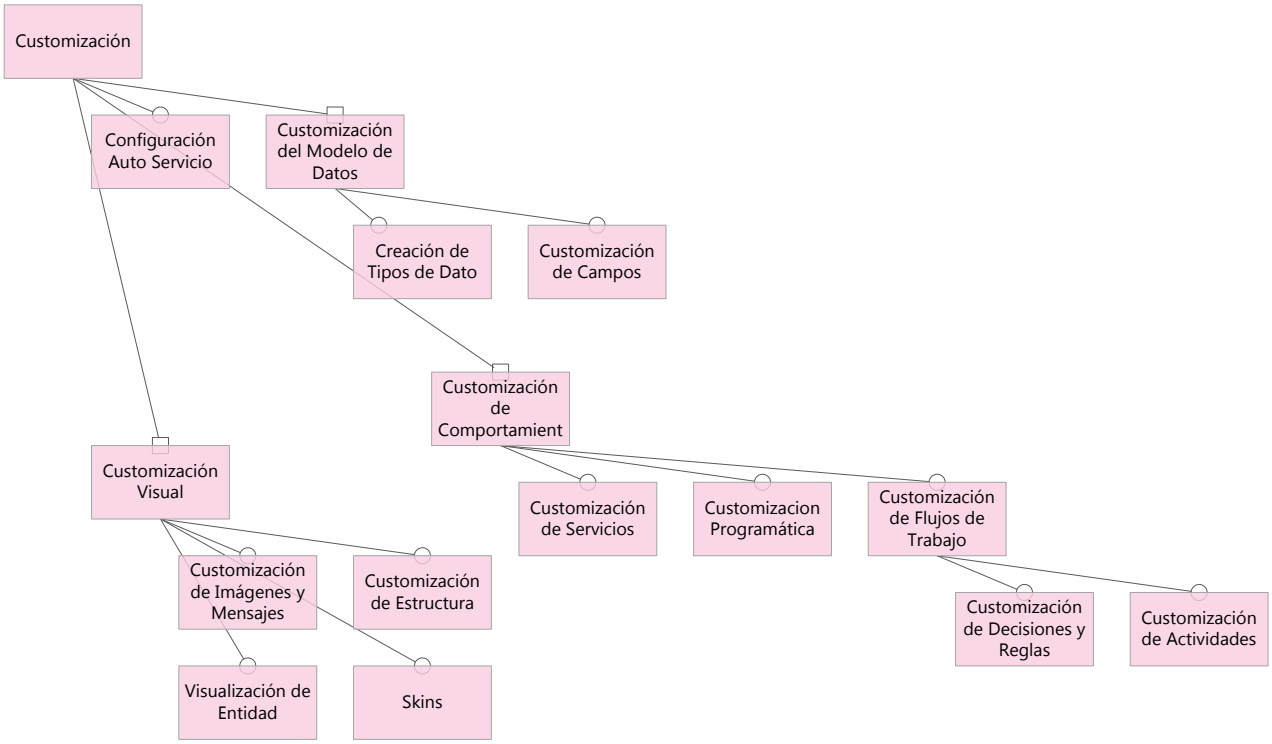



Figura 24 - Características de customización

No existiría la customización o la variabilidad de una aplicación, si esta no fuese brindada a múltiples clientes. En este grupo de características encontramos diferentes tipos de customización de una solución multi tenant. Modificaciones en el modelo de datos, en el comportamiento y en la visualización. Opcionalmente, como especialización de la *administración delegada*, el proveedor puede brindar una herramienta para manejar las customizaciones y configuraciones que el cliente puede utilizar por su cuenta.

### 7.3.5 Experiencia de Usuario

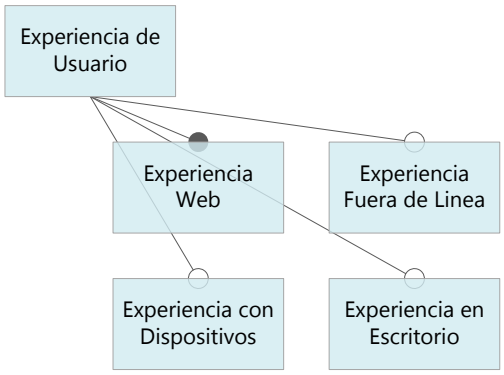



Figura 25 - Características relacionadas con la experiencia de usuario

La última categoría introduce el concepto de experiencia de usuario. Si bien la experiencia del usuario no es una característica exclusiva del modelo *Software como Servicio*, es un ingrediente importante. Para construir una aplicación *Software como Servicio* exitosa, es recomendable proveer una experiencia de usuario atractiva que apunte hacia la fidelidad del cliente [Giurata, 2008]. La interfaz web es una característica intrínseca del modelo, sin embargo en ciertos escenarios llevar la aplicación al escritorio con soporte fuera de línea o a un dispositivo puede atraer a más clientes.

## 7.4 Implementación

Antes de seguir con la capa de implementación, es necesario aclarar que lo expuesto a continuación son simplemente guías que deben ser tomadas como recomendaciones y no como soluciones absolutas. La intención es brindar una vista a alto nivel y crear disparadores para cada característica sin entrar en detalle en cada uno de los patrones, tecnologías o estilos de arquitectura.

Al igual que la capa de capacidades, aquí se ilustra la capa de implementación. El diagrama es vectorial por lo tanto la única manera de verlo es en formato digital haciendo zoom en cada sección. De cualquier manera, a lo largo de esta sección, se mostrarán las diferentes partes y se explicará de acuerdo a la categorización del diagrama de la Figura 18.

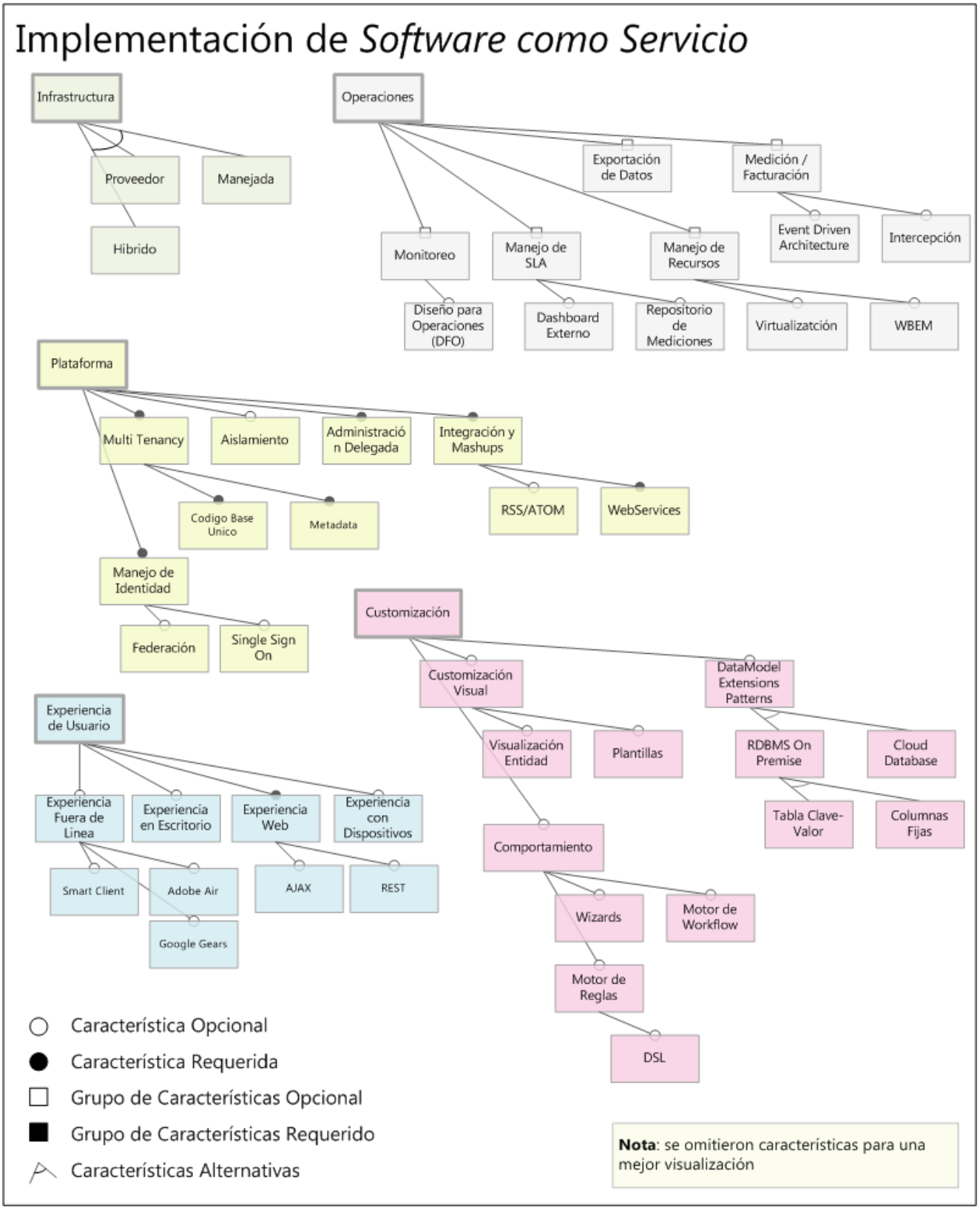



Figura 26 - Características de implementación de aplicaciones brindads como servicio

Las siguientes técnicas, principios, patrones y productos son recomendaciones basadas en la investigación de diferentes productos del mercado y la experiencia propia. No pretenden ser

definiciones absolutas, por lo tanto el usuario debe realizar pruebas de concepto, prototipos de las diferentes soluciones y evaluar de acuerdo a sus objetivos de negocio.

### 7.4.1 Infraestructura

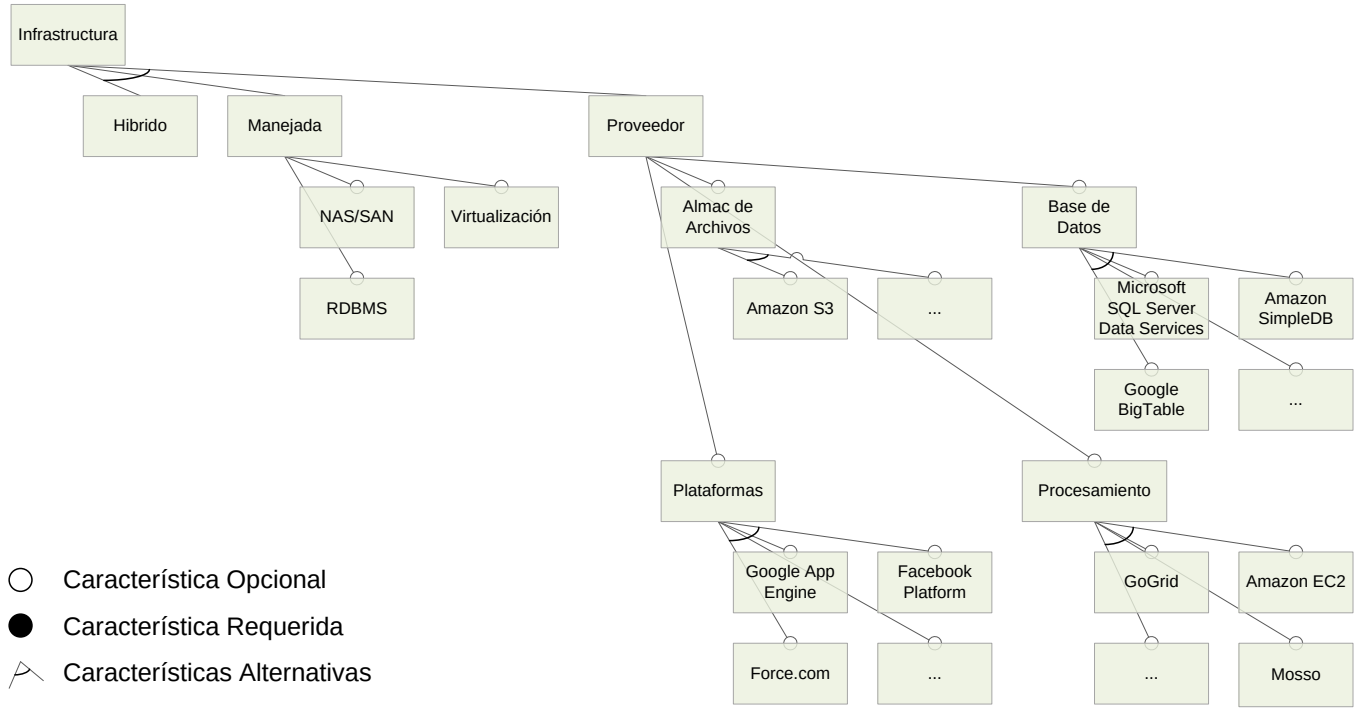



Figura 27 - Implementación de características ligadas a la infraestructura

La infraestructura se puede tercerizar a un proveedor, alojar en datacenters propios o una opción híbrida. Esta decisión estará influenciada por la capacidad de operar un datacenter y mantener en línea una aplicación de las características de *Software como Servicio*. En el caso de una empresa que recién comienza, que posiblemente no tenga un datacenter y que está más interesada en detectar si el modelo de negocios es rentable, la alternativa de utilizar infraestructura como un servicio (*infrastructure as a service*) puede ser llamativa. En este esquema, se paga solamente por los recursos utilizados y se puede ir escalando a medida que sea necesario.

#### *7.4.1.1 Infraestructura propia*

Si se optase por una infraestructura propia, se pueden mencionar un conjunto de buenas prácticas para alojar este tipo de aplicaciones. No necesariamente estas prácticas son exclusivas del modelo, sino que son aplicables a diferentes tipos de software. En primer lugar, la práctica de virtualizar el ambiente tiene la ventaja de abstraerse de los recursos de hardware brindando agilidad en las operaciones, manteniendo un ambiente estandarizado y haciendo un mejor uso de los recursos de

un servidor físico [Agility in Virtualized Utility Computing, 2007]. Hoy en día, las máquinas virtuales están llegando a niveles similares al hardware con técnicas como *paravirtualization*, *hypervisor* y virtualización a nivel de sistema operativo.

Las mismas prácticas se pueden aplicar al almacenamiento, utilizando Network Access Storage y Storage Area Network logrando una administración más simple y flexible.

El objetivo que se busca en el contexto de operar *Software como Servicio*, es poder manejar los recursos de hardware de manera más ágil. Esto permite alocar recursos con más dinamismo y en última instancia manejar la carga de los tenants, aumentar la densidad de tenants por máquina física y hasta promover el ahorro de energía por la optimización de los recursos.

Por último, optando por una infraestructura propia, se debe elegir un motor de base de datos si la aplicación lo requiriese. En este segmento las opciones son las mismas que se manejan en cualquier tipo de aplicación web, al momento de realizar esta tesis. En este caso, puede ser interesante explorar un modelo híbrido y utilizar una base de datos como Microsoft SQL Server Data Services, que extiende el modelo relacional pero con características para aplicaciones SaaS.

#### *7.4.1.2 Infraestructura tercerizada*

En este escenario se abre un abanico de nuevas posibilidades. Seguirán existiendo los servicios de alojamiento tradicionales donde el cliente se hace cargo de la operación y el proveedor solo se ocupa de brindar acceso a la red, electricidad y servidores. Esto es equivalente a manejar infraestructura propia. En cambio, este segmento, también llamado *infrastructure as a service (IaaS)* y *platform as a service (Paas)*, ofrece un servicio más específico, exponiendo recursos como servicio. La diferencia entre IaaS y PaaS es que el primero expone recursos más básicos como disco y procesador y el segundo provee servicios de un nivel de abstracción más alto (nivel de aplicación).

Es necesario realizar un estudio de los diferentes proveedores y un análisis de fortalezas y debilidades que presenta cada uno. En términos generales, estos son los parámetros a tener en cuenta:

| Costo | SLA | Herramientas de Administración | Seguridad |
|---|---|---|---|
| •Análisis de corto plazo<br>•Análisis de largo plazo | •Disponibilidad<br>•Performance<br>•Manejo de interrupciones en el servicio<br>•Soporte | •Usabilidad | •Confidencialidad |

Figura 28 - Análisis previo a contratar infraestructura o plataforma as a service

### 7.4.2 Plataforma

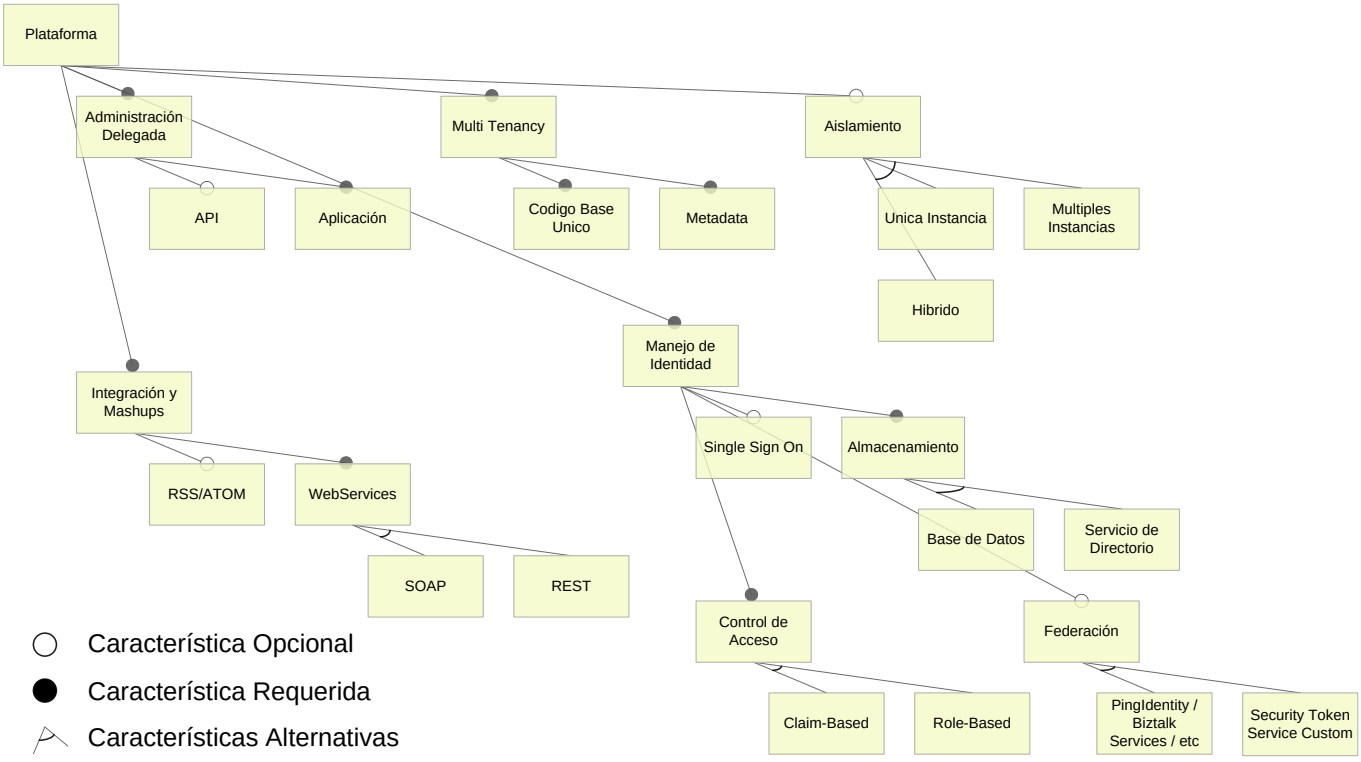



Figura 29 – Implementación de características ligadas a la plataforma

#### 7.4.2.1 *Multi tenancy*

Multi tenancy se puede ver como una optimización del modelo. Proveer un servicio a 10.000 clientes se podría lograr instalando un código base modificado para cada cliente, cada instancia en una máquina diferente. Sin embargo, esto podría ser costoso para el proveedor, por lo tanto se tiende a tratar de modelar, diseñar e implementar el servicio para que pueda servir la mayor cantidad de clientes optimizando los recursos utilizados.

El presente trabajo separa, por un lado el concepto multi tenant que se refiere a servir a múltiples clientes con un mismo código base y por otro lado la forma en que este código está escrito para soportar la customización por tenant. Esta separación se realiza debido a que son características excluyentes. Un ejemplo claro de esto es un motor de blogging. El motor de blogging es multi tenant pero el comportamiento y el modelo de datos es el mismo para todos los tenants. La variabilidad es mínima y se puede manejar con algunas opciones de configuración. Sin embargo, un sistema de facturación puede llegar a tener una variabilidad mucho mayor que requiere no solamente de una arquitectura multi tenant sino también de elementos customizables.

#### *7.4.2.2 Metadata*

La metadata juega un papel importante como elemento de diseño de una arquitectura multi tenant. Cada tenant está caracterizado por la instancia de la aplicación más la metadata.


Algunas de las prácticas enunciadas en esta tesis con respecto a la metadata son:

- Separar la metadata de la data en diferentes bases de datos con el objetivo de poder mover tenants entre máquinas manteniendo centralizada la información de la metadata
- La metadata es información de actividad que se caracteriza por una mayor cantidad de operaciones de lectura que de escritura, por lo tanto puede ser beneficioso mantenerla separada de la información transaccional para optimizar el acceso de disco
- El manejo de la metadata en la aplicación debería ser lo menos intrusiva posible. Para eso se puede mantener tanto en la capa de presentación como la capa de servicios un contexto que permita acceder en fácilmente a la metadata
- El parámetro más importante que permitirá consultar la metadata será el identificador único del tenant.
    - En la capa de presentación puede extraerse el tenant a partir de la URL o de información de sesión del usuario perteneciente a un tenant.

- En la capa de servicios se puede enviar el identificador del tenant por medio de un encabezado de SOAP o HTTP e implementar un mecanismo de intercepción que agregue al mensaje saliente el encabezado.

#### 7.4.2.3 Aislamiento

El principio de *multi tenancy* está fuertemente ligado al aislamiento. La presente investigación enuncia dos formas de lograr que una aplicación sea multi tenant: múltiples instancias o única instancia. La primera soporta a cada tenant con una instancia dedicada, compartiendo únicamente el sistema operativo base y los recursos de hardware. La segunda puede soportar a múltiples tenants en una única instancia de aplicación compartiendo todos los recursos. Entre estos extremos existe un conjunto de alternativas que incrementan la densidad de tenants por máquina física como muestra la siguiente figura.

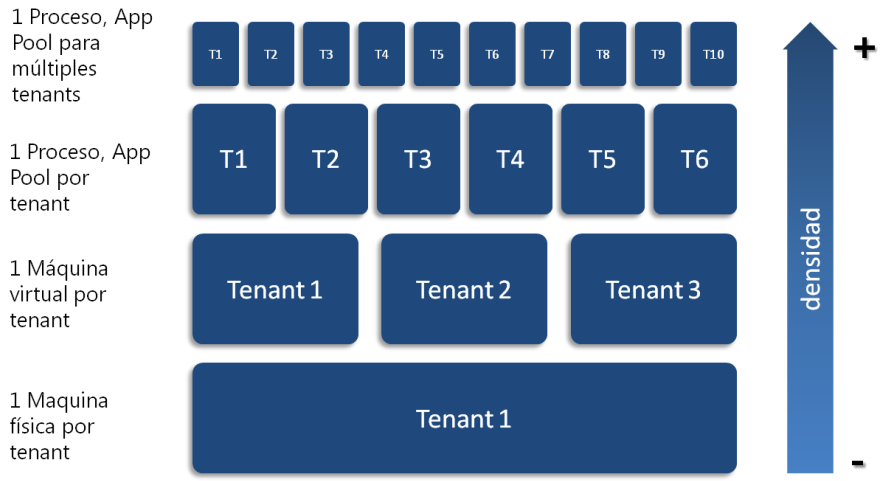



Figura 30 - Densidad de tenants con diferentes niveles de aislamiento (la cantidad es figurativa)

El precio del aislamiento se paga con escalabilidad. Al tener una única instancia hay que poner más énfasis en el nivel de servicio (QoS) previniendo que un tenant no se vea afectado por otro cuando comparten un mismo ambiente. La selección del tipo de aislamiento dependerá de los escenarios y requerimientos del cliente final. Por ejemplo, una empresa puede preferir pagar una tarifa "Premium" y acceder a su instancia propia para mitigar los riesgos de compartir los recursos y poner en juego la privacidad de sus datos. En cambio una empresa de menor tamaño estaría dispuesta a pagar por un servicio más económico y por el valor agregado que le brinda la funcionalidad de la aplicación sin preocuparle el hecho de compartir recursos.

El modelo híbrido es una alternativa permitiendo al proveedor mantener una infraestructura compartida y al mismo tiempo brindar la posibilidad de incrementar la performance, la privacidad y el nivel de servicio en general moviendo a un tenant a su propia instancia.

En términos de mantenimiento, se recomienda que exista un mecanismo de despliegue robusto para hacer las actualizaciones correspondientes a todos los tenants a partir del mismo código base sin importar si se están compartiendo recursos o se trata de instancias separadas.

#### *7.4.2.4 Integración y mashups*

La integración de aplicaciones es una de las preocupaciones más grandes de las empresas como se había adelantado en la sección 7.1.8.1 Integración, mashups y automatización. Esta tesis propone la adopción de una arquitectura orientada a servicios basada en estándares utilizados en la industria. En esta sección no se tratará la manera de plantear una arquitectura SOA ya que hay una gran cantidad de literatura que trata el tema. En cambio se discutirán ciertos argumentos a tener en cuenta a la hora de resolver los desafíos de integrar una aplicación *Software como Servicio* con aplicaciones existentes:

- Buscar un transporte común para el intercambio de datos
- Buscar un lenguaje común para el intercambio de datos
- Asegurarse que los datos sensibles permanezcan seguros
- Asegurar la integridad de la transacción

En cuanto el transporte, el más ubicuo hoy en día es HTTP. Cualquier lenguaje y plataforma hoy en día soportan este protocolo. En segundo lugar, por sobre el transporte está el lenguaje en común. A lo largo del tiempo se ha avanzado en este tema introduciendo SOAP y WS-* como una solución que apunta a resolver el problema de estandarización en el ámbito corporativo. Sin embargo con la llegada de la Web 2.0, queriendo resolver el mismo problema de estandarización e integración pero a escala de internet, surgió REST [Fielding, 2000]. A continuación se enumeran algunas de las características de estas dos alternativas.

#### 7.4.2.4.1 REST

| Fortalezas | Debilidades |
|---|---|
| • Simplicidad<br>• HTTP/POX (Plain Old Xml) es ubicuo<br>• Stateless y sincrónico<br>• Idempotencia y escalabilidad (GET HTTP cache) | • No es asincrónico a nivel de protocolo<br>• Seguridad (solo con HTTP/SSL)<br>• No hay una meta descripción del servicio más alla de la URI<br>• Ruteo de mensajes<br>• Garantía de entrega (el mensaje puede llegar dos veces) |

#### 7.4.2.4.2 SOAP

| Fortalezas | Debilidades |
|---|---|
| • Sincrónico y asincrónico<br>• Independiente de transporte (aunque HTTP es el más usado)<br>• Tipado fuerte (XML Schema)<br>• Conjunto de protocolos para seguridad, transacciones, ruteo, etc (WS-*)<br>• Descripción de servicios (WSDL)<br>• Adopción y herramientas en diferentes plataformas | • Especificaciones complejas<br>• Problemas de interoperabilidad entre diferentes implementaciones de las especificaciones más avanzadas |

En términos generales SOAP es utilizado en el ámbito corporativo para integrar procesos de negocios, en cambio REST cuenta con una mayor adopción en el contexto de la Web 2.0 donde la simplicidad y flexibilidad es priorizada por sobre la seguridad. En aplicaciones brindadas como servicio donde no se tenga control sobre las plataformas y clientes que quieran integrarse, esta tesis propone utilizar REST o SOAP bajo el perfil básico (*basic profile*) como alternativas que harán la adopción más sencilla.

Estas son las opciones que brindan los proveedores más populares:

- Amazon Web Services ofrece la opción SOAP y la opción REST. El 85% del uso proviene de la interfaz REST [O'Reilly, 2006].
- Salesforce.com ofrece solamente la interfaz SOAP con basic profile y el 40% de las transacciones son realizadas via SOAP.
- Google ofrece solamente la API REST y retiró el soporte para la interfaz SOAP

#### 7.4.2.4.3 Syndication con RSS/ATOM

RSS y ATOM son protocolos utilizados para exponer colecciones de datos con un formato XML simple (RSS significa Really Simple Syndication). Combinado con REST, la consumición de datos desde cualquier plataforma se reduce a una llamada HTTP y un simple parser de XML. Una gran cantidad de tipos de datos se puede representar con RSS o ATOM y en caso de que se necesite, el protocolo soporta extensiones.

### *7.4.2.5 Administración delegada*

La administración delegada surge de la necesidad de disminuir la interacción entre el cliente y el proveedor logrando que este pueda realizar funciones básicas de administración de su instancia por sí mismo.

Reducir la intervención humana al mínimo posible va a reducir los errores, los costos y tiempos. La estrategia para que esto ocurra es automatizar y delegar. Automatizar es lograr que la tecnología haga las tareas que requiera intervención humana. Delegar significa darle herramientas al cliente para que este realice las tareas que de otra forma las haría el proveedor.

Esta tesis propone diferentes alternativas para brindar administración delegada:

- Proveer APIs para realizar operaciones de administración dejando que el tenant se encargue de crear una interfaz o una aplicación por sobre la API
    - Más flexibilidad, menos usabilidad
    - Ideal para servicios más de bajo nivel (ej: Amazon S3)
- Proveer una aplicación de acceso privado al tenant que permita hacer las operaciones
    - Posiblemente menos flexibilidad pero más usabilidad
    - Ideal para servicios de más alto nivel, como aplicaciones de línea de negocio (ej: Salesforce)

En el caso de optar por la segunda opción esta tesis propone la separación del sitio público del privado. El sitio privado realiza mayormente escrituras al repositorio de metadata y el sitio público lecturas.

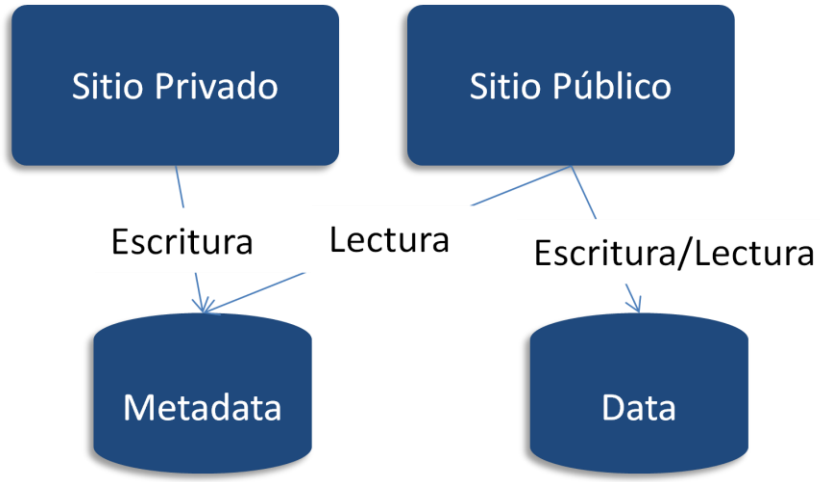



Figura 31 - Administración separada de funcionalidad

#### *7.4.2.6 Manejo de Identidad*

El manejo de identidad es uno de los desafíos más grandes de una aplicación *Software como Servicio* (ver 7.1.8.2 Manejo de identidad y control de acceso federado). Desde la perspectiva del consumidor, la aplicación se integraría a su portfolio de aplicaciones existentes. En una empresa de tamaño considerable eso significaría:

- que los usuarios tienen que recordar un password más
- los roles y permisos se duplican
- más llamadas al helpdesk de la empresa

Desde la perspectiva del proveedor, la aplicación tiene que manejar seguridad a nivel de la interfaz de usuario y a nivel de la API que expone como servicios. Posiblemente la aplicación forme parte de un ecosistema de aplicaciones y servicios donde idealmente el usuario pueda realizar *single sign-on* con un único set de credenciales.

El manejo de identidad puede atacarse teniendo un sub sistema propio de seguridad con usuario y password creando otro *silo de identidad* para el consumidor. Crear este sub sistema es algo que se viene haciendo en la mayoría de las aplicaciones y no representa un desafío. No obstante, como

parte de este trabajo de investigación, los problemas mencionados anteriormente se podrían resolver proponiendo un manejo de identidad federado basado en *claims*.

#### 7.4.2.6.1 Federación y seguridad basada en *claims*

En un escenario de federación, el proveedor establece una relación de confianza con el consumidor y al momento de utilizar la aplicación se hace un intercambio de mensajes entre el proveedor y el consumidor que finalmente genera un *token* que probará la autenticidad del usuario y habilitará al mismo a hacer ciertas operaciones. El intercambio de mensajes está basado en estándares como *WS-Trust* y eso implica que funcionará sin importar la plataforma sobre la que corre el cliente.

Antes de analizar el escenario, se aclararán algunos conceptos. Un *token* es un fragmento de XML firmado por la autoridad que lo emite que contiene un conjunto de *claims*. Un *claim* es una afirmación sobre cierta entidad ("el sujeto") declarada por otra entidad ("la autoridad"). Un ejemplo de esto sería "José es mayor de 21 años", "José pertenece al grupo de gerentes del dominio abc.com". Un *Security Token Service* (STS) es un web service que emite tokens de seguridad según el estándar WS-Trust.

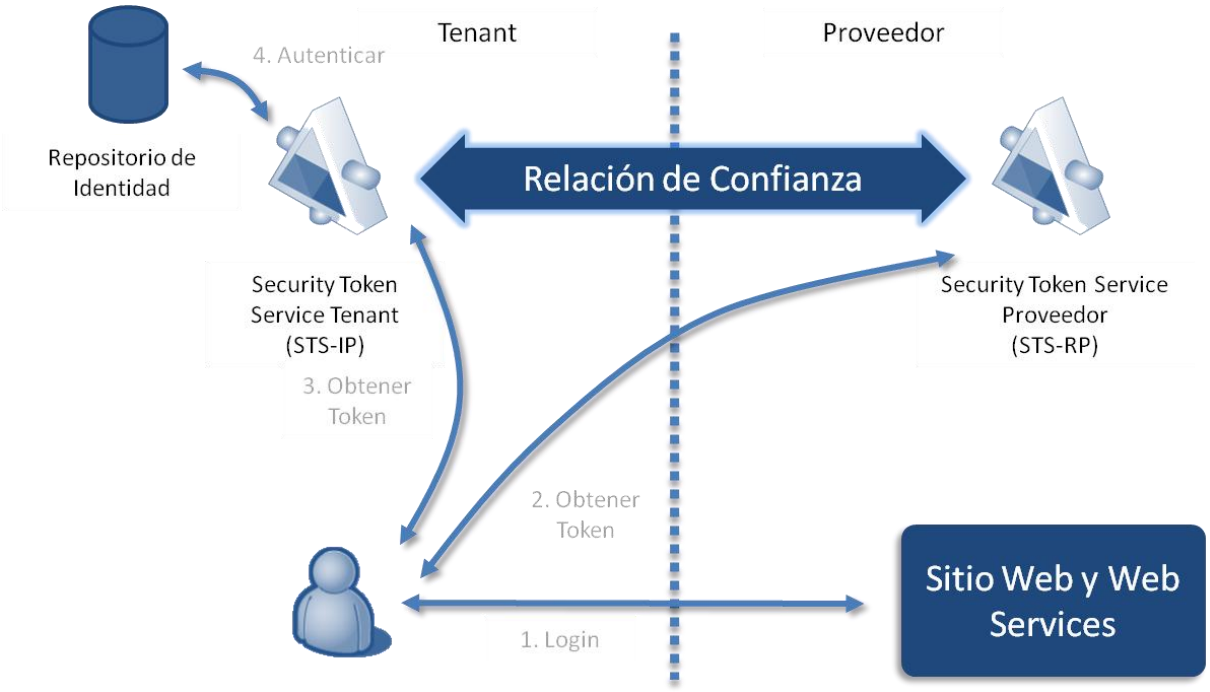



**Figura 32 - Identidad federada**

El escenario de alto nivel de identidad federada se detalla a continuación:

1. Un usuario del tenant quiere establecer una sesión con la aplicación SaaS. La política del servicio es que necesita presentar un *token* firmado por el STS-RP para poder ingresar. Ese *token* contendrá ciertos claims que entenderá el servicio (como por ejemplo "usuario X puede acceder al servicio PagarFactura"). El *token* se presentará como una cookie HTTP en el caso de acceder al sitio web o como parte de un encabezado del mensaje SOAP en el caso de acceder a un webservice.
2. El cliente se dirige al STS-RP y la política de este STS indica que por ser usuario del tenant "X" tiene que ir a buscar el *token* a su proveedor de identidad, el STS-IP. El tenant deberá tener su *Security Token Service* expuesto a la web. Existe una relación de confianza entre estos dos STS. Esto significa que cuando STS-IP le devuelva un *token* al cliente y este se lo presente al STS-RP, este último va a chequear si el *token* está firmado (generalmente usando certificados X509) por el STS-IP.
3. El cliente se dirige al STS-IP y la política indica que puede obtener un *token* siendo del tenant "X" y teniendo como credenciales usuario y password (u otro tipo de credenciales).
4. El STS-IP va a verificar las credenciales (puede ser un usuario y password, un certificado de una *smartcard*, un ticket de kerberos, etc.) contra un repositorio de identidad que es de su propiedad (base de datos, Active Directory, etc.) y en caso exitoso va a devolver un *token* firmado con la clave privada de un certificado X509. El *token* posiblemente contenga *claims* sobre el sujeto (a que grupos pertenece, o alguna particularidad).
5. El *token* firmado es recibido por el STS-RP y este genera un nuevo *token* transformando los *claims* del *token* del STS-IP en *claims* que entiende el proveedor (este proceso se llama transformación o mapeo de *claims*). El *token* es firmado por el STS-RP y encriptado con la clave pública del proveedor. Solo podrá ser desencriptado por el que posea la clave privada, es decir el proveedor.
6. Se realiza la llamada finalmente y se intercepta para verificar que efectivamente el mensaje contiene un *token*, que ese *token* está firmado por el STS-RP y que los *claims* que contiene lo habilitan al usuario a acceder (control de acceso).

En resumen, utilizando identidad federada, el tenant puede seguir manejando los usuarios y roles de la organización sin tener que preocuparse por otro repositorio de identidad del proveedor. Para el proveedor, el escenario de identidad federada lo habilita a tener *single sign on* entre aplicaciones y a ser integrable con otras aplicaciones. Para esto se establece una relación de confianza entre el tenant y el proveedor y se utilizan estándares como WS-Trust, SOAP y HTTP para lograr una solución que funcione en múltiples plataformas.

El componente que debe desarrollar el proveedor es el STS. Las plataformas, no todas, tienen soporte para construir un STS y algunas trabajan en un nivel de abstracción que permite implementarlo en poco tiempo. Algunas de las plataformas más conocidas para desarrollar bajo el paradigma propuesto en esta tesis son: Microsoft con Windows Communication Foundation o Identity Framewrok (Zermatt) y Sun con Metro Web Services. También existen empresas como PingIdentity que ofrecen la implementación de la identidad federada. Microsoft ofrece, con el Windows Server 2008, un producto llamado ADFS (Active Directory Federation Server) que expone un STS por sobre el Active Directory. Por último, Microsoft Biztalk Services ofrecerá un STS que estará disponible como un servicio con características genéricas como transformación de *claims* y manejo de relaciones de confianza. Otras alternativas a investigar incluyen a *OpenID* que provee un mecanismo de autenticación utilizado, por ejemplo, para dejar comentarios en motor de blogs en donde la autenticación es más importante que la autorización. Por último un estándar relativamente nuevo *OAuth* está siendo adoptado por su simplicidad. Basado en REST trata de tanto la autenticación como la autorización.

### 7.4.3 Operaciones

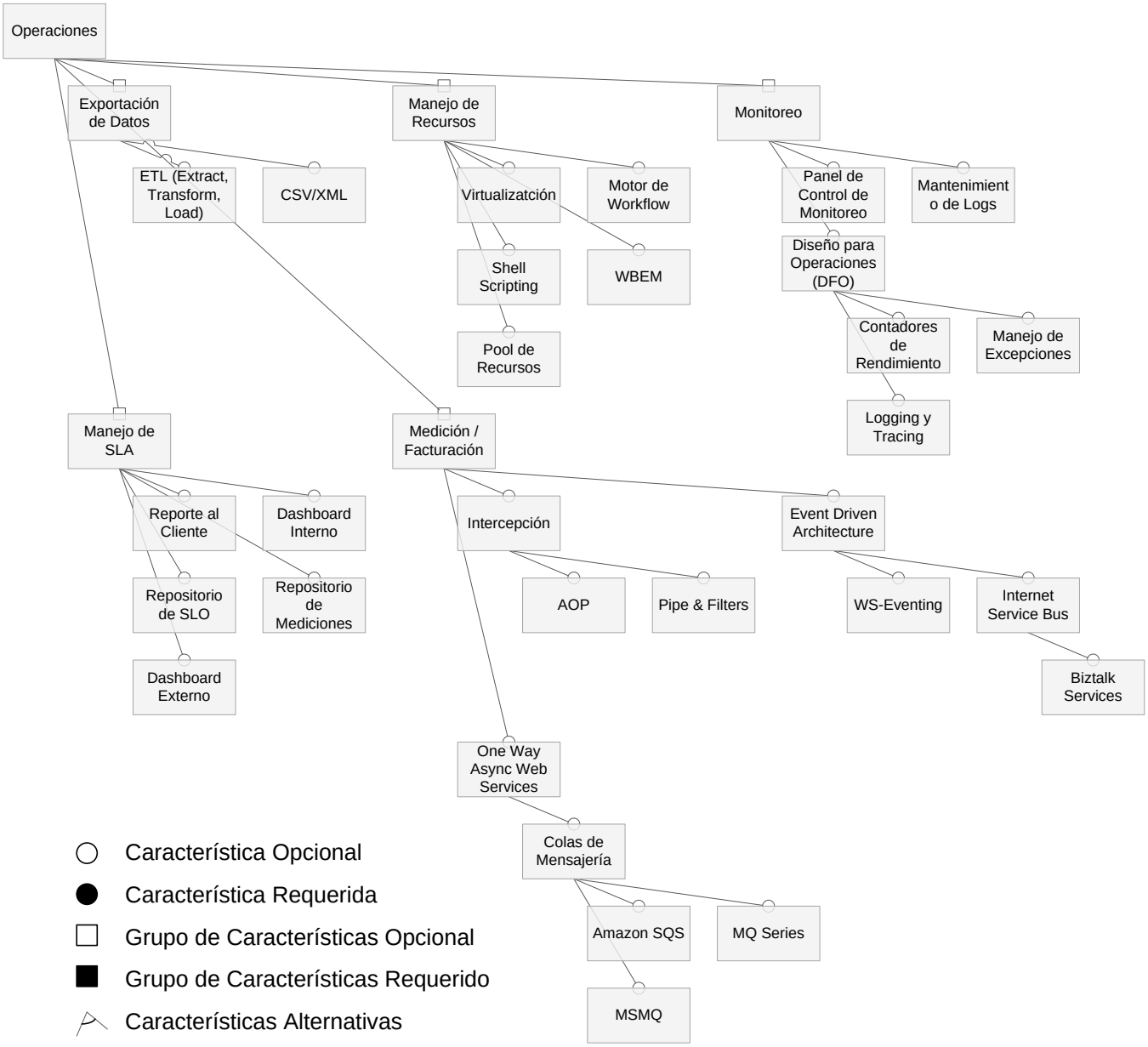



Figura 33 - Características de implementación relacionadas con la operación de *Software como Servicio*

Si bien las prácticas mencionadas a continuación aplican a cualquier sistema en producción, esta tesis plantea que en una aplicación *Software como Servicio* estas prácticas son fundamentales por las características del modelo y deberían incluirse desde la versión inicial de la aplicación.

#### *7.4.3.1 Diseño para Operaciones (Design for Operations)*

En la operación de una aplicación se plantean las siguientes preguntas cuando esta falla:

- ¿Cuál es el origen del problema?

- ¿Cómo obtengo información detallada del problema?
- ¿Cuáles son los síntomas que llevan a este tipo de problemas?

Por otro lado cuando una aplicación falla los usuarios finales se preguntan ¿Por qué no funciona? ¿Por qué funciona lento? ¿Por qué no puedo realizar esta operación? Los operadores se hacen otro tipo de preguntas como: ¿Por qué falló y no se generó ninguna alerta previa? ¿Qué debería hacer si encuentro una alerta? ¿Cómo configuro la aplicación para que genere eventos y datos operacionales? Por último aquellos que escribieron el código tienen otras inquietudes ¿Qué componentes tiran excepciones? ¿Qué cosas debería controlar? ¿Cuáles son los eventos de negocio? El diseño para operaciones (DFO) es una filosofía que tiene en cuenta el diseño de una aplicación teniendo en mente que se va a poner en producción, va a fallar y va a ser operada por alguien.

Es importante reconocer que no alcanza con mirar un sólo lugar para detectar, debuggear y resolver un problema ya que los datos provienen de diferentes fuentes. Los problemas pueden surgir en cualquier parte, sobre todo en una arquitectura distribuida.

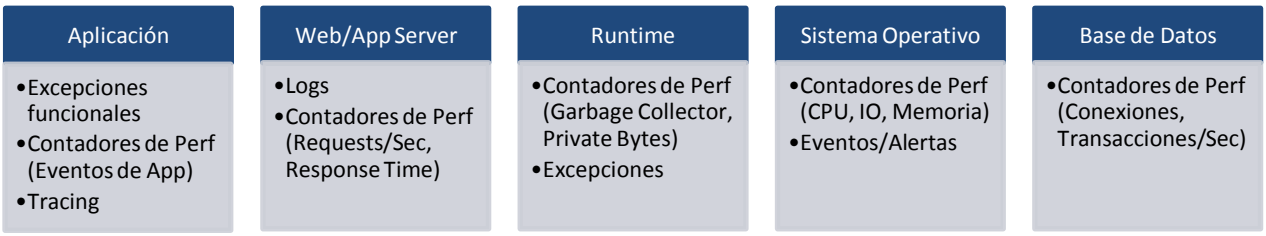



Figura 34 - Fuentes de datos y eventos operacionales

El siguiente ciclo muestra las diferentes etapas en la generación de datos operacionales. Los eventos y datos operacionales se generan durante la ejecución del servicio. Si hay alguna anormalidad, algún límite sobrepasado o alguna excepción, esto es notificado. Una vez notificado se trata el problema, se resuelve y se analiza la causa raíz. Por último, los *logs* comienzan a crecer en un sistema en producción rápidamente, por lo tanto es necesario tener una estrategia para purgarlos cada un período determinado.

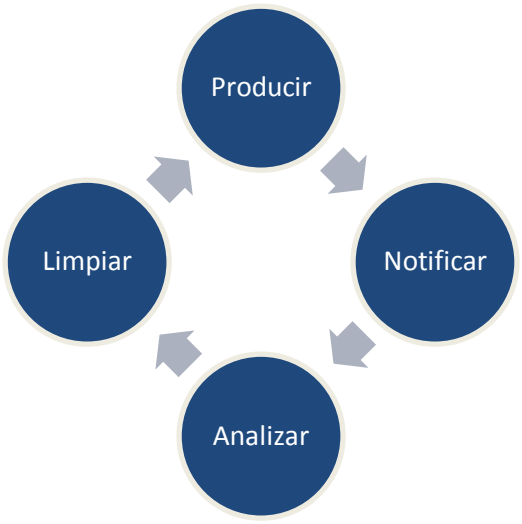



Figura 35 - Ciclo de monitoreo de una aplicación

Deberían considerarse los siguientes puntos a la hora de instrumentar una aplicación [Microsoft Patterns & Practices, 2008]:

- Usar las capacidades de la plataforma
- Separar los diferentes tipos de instrumentación para cada propósito (salud, performance, debugging, planeamiento de capacidad)
- Crear una arquitectura de instrumentación extensible (posibilitar cambios en el motor de instrumentación sin afectar el código que lo utiliza)
- Definir los nombres de los eventos y los identificadores consistentemente
- Soportar logging remoto
- Considerar el uso de correlación en arquitecturas distribuidas

Por el lado de la aplicación se puede contar con una API para hacer logging que sea simple de usar para el desarrollador pero que por debajo escriba al log, no solo el mensaje, sino también información de contexto como el usuario, la máquina donde se está ejecutando, el *stack* de llamadas, etc. Debe permitir prender y apagar el logging y poder seleccionar el destino físico de las entradas del log (un archivo, el log de eventos, una base de datos, etc.)

```
Logger.LogWarning( “No se encontró el usuario” );
Logger.LogInfo( “El método se ejecutó correctamente” );
Logger.LogError( “Ha ocurrido una excepción, etc” );
```

Existen varias librerías que proveen las características mencionadas: Log4Java, Log4Net, Enterprise Library, etc.

Otra fuente de datos son los contadores de performance. Además de monitorear los contadores de la plataforma y los productos, una aplicación puede crear sus propios contadores de performance y categorías con un código como el que se ve a continuación:

```
CounterCreationData totalOps = new CounterCreationData();
totalOps.CounterName = "# operaciones ejecutadas";
totalOps.CounterHelp = "Total de operaciones ejecutadas";
totalOps.CounterType = PerformanceCounterType.NumberOfItems32;
counters.Add(totalOps);
PerformanceCounterCategory.Create("ApplicacionCRM", "Aplicación de CRM",
counters);
```

En el sistema operativo Windows se puede monitorear la performance leyendo estos contadores con el Performance Monitor.

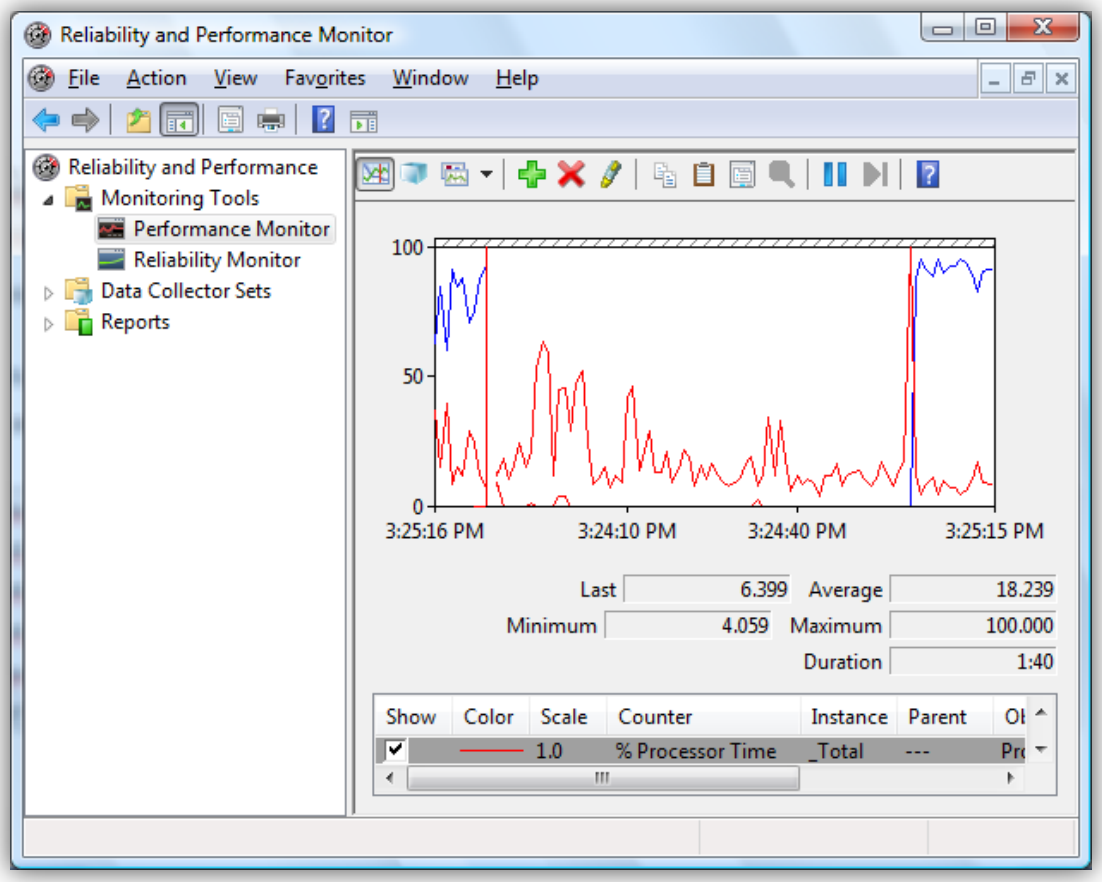



Figura 36 - Herramientas de monitoreo

Es posible utilizar técnicas como AOP (Aspect Oriented Programming) o mecanismos de intercepción para generar logs. Utilizando AOP se puede interceptar la llamada a un método y

escribir a un log cuando entró, cuando salió y si se produjo alguna excepción. En algunas plataformas estos mecanismos de intercepción son intrinsecos y se pueden crear filtros que intercepten las llamada. El siguiente es un pseudo código figurativo que muestra este concepto:

```
public void BeforeActionExecuted( ... ) {
      Logger.LogInfo( “Entrando a acción ” + context.Action );
}
```

#### *7.4.3.2 Medición y facturación*

La medición y la facturación son dos características que están relacionadas. La primera trata de registrar los eventos relacionados con el uso del servicio y la segunda toma estos eventos y a partir de las políticas de monetización genera las facturas a los clientes.

El escenario más simple es la monetización por subscripción en donde se registra la cantidad de usuarios que fueron aprovisionados en un mes. Para determinar el monto mensual de la factura, el servicio de facturación simplemente multiplica la cantidad de usuarios por el costo mensual deduciendo descuentos especiales si los hay y el prorrateo en caso de activaciones posteriores al inicio de mes. Un caso más complejo se presenta cuando el costo se calcula a partir de los recursos utilizados. Los recursos deberán ser medidos y de acuerdo a la valorización que se le da a cada uno, se generará el monto a pagar [Chong y otros, 2007]

Esta tesis propone contar con un mecanismo asincrónico de eventos. La cantidad de eventos que se pueden llegar a generar es muy grande y si se hace de manera sincrónica por cada llamada se puede generar un cuello de botella.

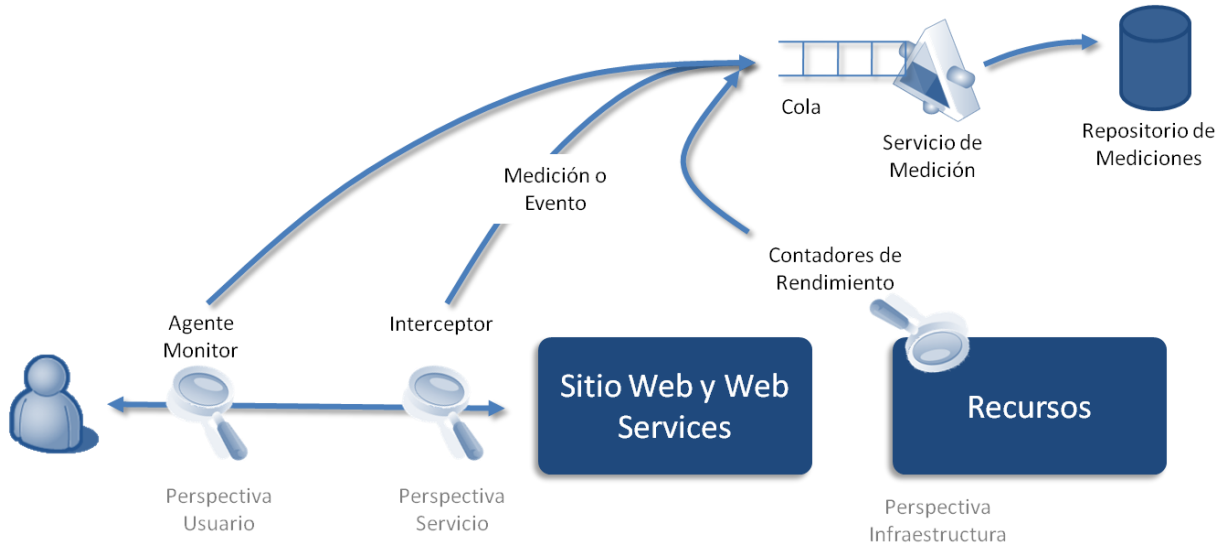



Figura 37 - Arquitectura de intercepción y medición

La ilustración plantea una arquitectura para medir el uso en una aplicación *Software como Servicio*.

Se pueden definir tres perspectivas: la perspectiva del usuario, del servicio y la infraestructura. Según la perspectiva, el método de monitoreo y medición son diferentes. Mediciones como tiempo de respuesta y disponibilidad se toman desde la perspectiva del usuario. El mecanismo puede ser un agente monitor que cada un lapso de tiempo determinado realiza una llamada (HTTP) al servicio y realiza mediciones como tiempo de respuesta promedio y disponibilidad.

Dentro del datacenter, conociendo su estructura, se puede contar con otras mediciones. La perspectiva de infraestructura mide procesamiento, memoria, actividad de input/output en discos, indicadores de base de datos y web servers, etc. Esta información se extrae de contadores de rendimiento utilizando WBEM por ejemplo.

Desde la perspectiva de servicio, se pueden medir los tiempos de las operaciones más significativas ligados a la lógica y funcionalidad del servicio. El mecanismo de intercepción se puede implementar con AOP o el patrón *pipe & filters* implementado en varias plataformas. Las llamadas son interceptadas y medidas y la medición se envía asincrónicamente a un servicio expuesto a través de una cola de mensajería.

El servicio utiliza la cola de mensajería para leer los eventos uno a uno y los graba en un repositorio de mediciones que será consultado a posteriori para generar la factura. El uso de una cola de mensajería durable habilita el transporte one-way asincrónico y la garantía de que ninguna medición se va a perde.

Por otro lado, esta tesis plantea un posible sub sistema de facturación. Este tiene como principal componente orquestador el workflow de facturación que se encargará de obtener las cuentas a cobrar del repositorio de facturación y las métricas del repositorio de mediciones (alimentado por el sub sistema de medición). Según las reglas definidas en este workflow, que pueden llegar a ser muy complejas, se generará una llamada asincrónica al servicio de facturación que finalmente utilizará un Gateway de pagos para finalizar la transacción. En caso exitoso se le enviara una factura al cliente y en caso de haber algún problema en el proceso o en el Gateway de pagos se le enviará un mensaje al cliente.

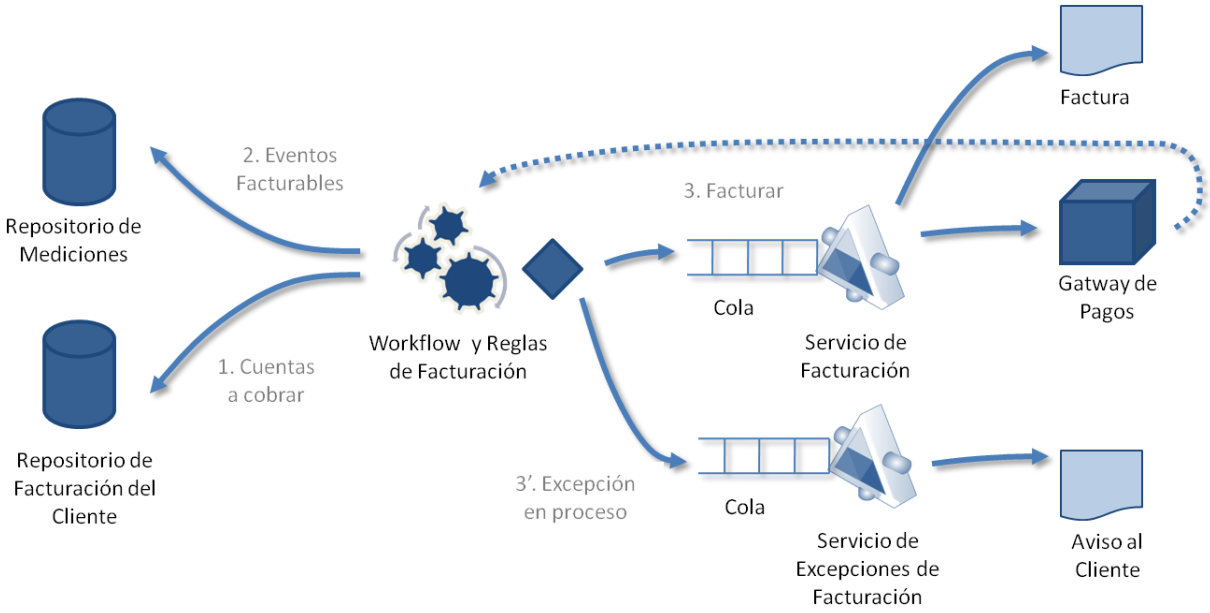



Figura 38 - Arquitectura del sub sistema de facturación

Una alternativa más sofisticada es la utilización de un *service bus*. Este componente permite publicar eventos a una cañería compartida por todos los sistemas en donde cualquiera de estos puede subscribirse a eventos publicados. Un estándar de web services utilizado para el mecanismo publicación subscripción de eventos es WS-Eventing.

#### *7.4.3.3 Aprovisionamiento automático y manejo de órdenes de compra*

Otra característica de la capa operacional es el aprovisionamiento automático de un nuevo cliente. El aprovisionamiento está relacionado con el manejo de órdenes de compra y recursos. El proveedor puede definir un workflow de aprovisionamiento que se dispare a intervalos determinados y en su ejecución llame al servicio de manejo de órdenes de compra trayendo todas las órdenes correspondientes que no fueron tomadas aún. El workflow de aprovisionamiento tendrá ciertas reglas y políticas definidas (por ejemplo el manejo de los diferentes planes, tipos de recursos

por tipo de aprovisionamiento, aprobaciones, etc.). Luego que el workflow identifique los recursos necesarios para cumplir con la orden del cliente, inicia una conversación con el servicio de alocación de recursos. El servicio se encargará de la creación física de los recursos (archivos, maquina virtual, pool de aplicación, base de datos, etc.). Una vez creados registrará los recursos virtualmente, posiblemente en una base de datos. El proceso finaliza con la llamada al servicio de facturación que registrará al cliente en el repositorio correspondiente para futuros cobros. Por otro lado se le envía un mensaje al cliente si el aprovisionamiento fue exitoso.

Debe tenerse en cuenta los casos excepcionales ya que la orquestación maneja recursos no transaccionales. El workflow se encargaría de la lógica de compensación en caso que la alocación de alguno de los recursos falle.

La siguiente figura ilustra una posible arquitectura para este sub sistema.

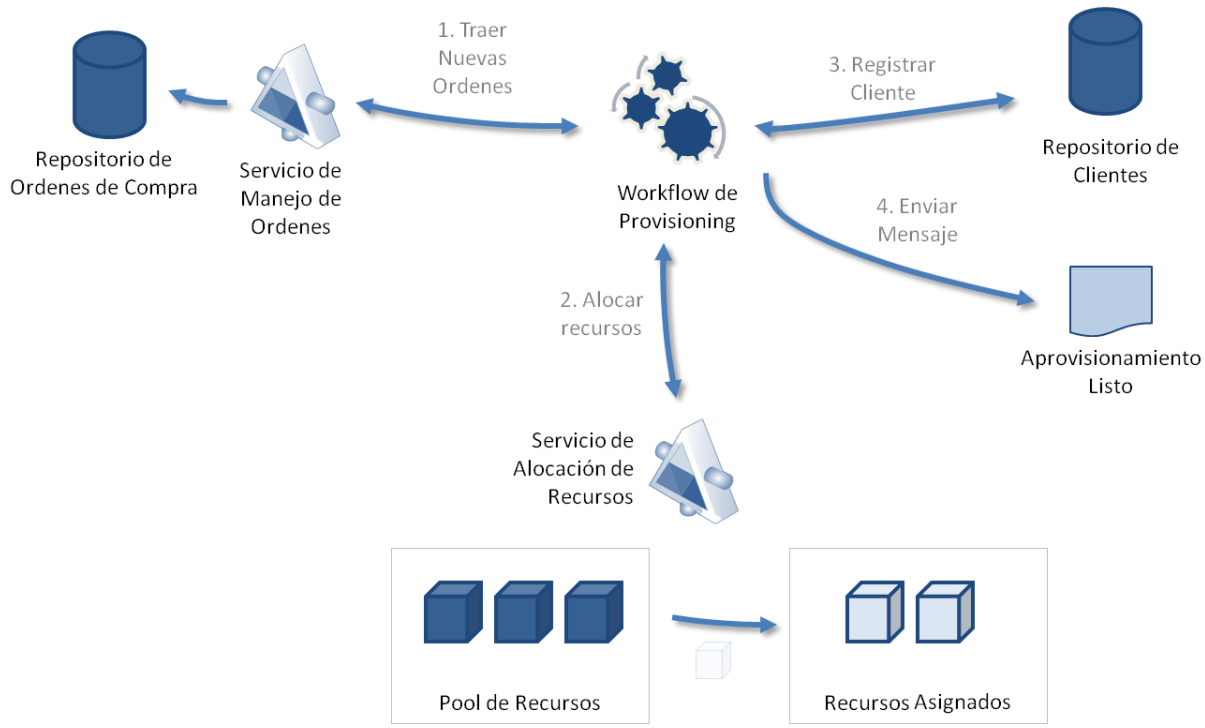



Figura 39 – Mecanismo de Aprovisionamiento

El sub sistema de manejo de recursos mostrado en la figura anterior es una versión simplificada. En la sección siguiente se realiza un análisis más exhaustivo.

#### *7.4.3.4 Manejo de Recursos y Cuotas, Escalabilidad a demanda y Balanceo de Carga*

Dependiendo de los requerimientos y la complejidad de la aplicación el manejo de recursos puede ser complejo, involucrando tecnologías de virtualización, un inventario de recursos, etc. O tan simple como crear un directorio virtual y una nueva base de datos. La Figura 40 ilustra un caso complejo en donde existe un pool de recursos que podría incluir

- Máquinas virtuales: plantillas con diferentes tipos de máquinas virtuales
- Hardware: CPU; memoria; almacenamiento
- Web Server: directorios virtuales; pool de aplicación; web site
- Servicios: DNS; servicios de directorio; sub sistema de seguridad; load balancers
- Recursos *on the cloud*: almacenamiento; procesamiento
- Otros recursos

El servicio de alocación de recursos interactúa con un repositorio de recursos que posee un inventario de todos los recursos disponibles y asignados. Con la llegada de un pedido de alocación (alta, baja o modificación) el servicio interactúa con el pool de recursos mediante una API que abstrae los detalles de implementación de los recursos. Esta API posiblemente utilice Web-Based Enterprise Management[13] (WBEM) para el manejo remoto de los recursos. WBEM es un conjunto de tecnologías desarrollado para unificar el manejo de sistemas distribuidos. Está basado en estándares abiertos de Distributed Management Task Force (DMTF): Common Information Model (CIM) infrastructure y schema, CIM-XML y WS-Management. Otras tecnologías posibles para el manejo de recursos podría ser Shell scripting (Bash en Unix, Powershell en Windows) y SNMP.

[13] "Web Based Enterprise Management". Definición extraída de Wikipedia, The Free Encyclopedia (http://en.wikipedia.org/wiki/Web-Based_Enterprise_Management)

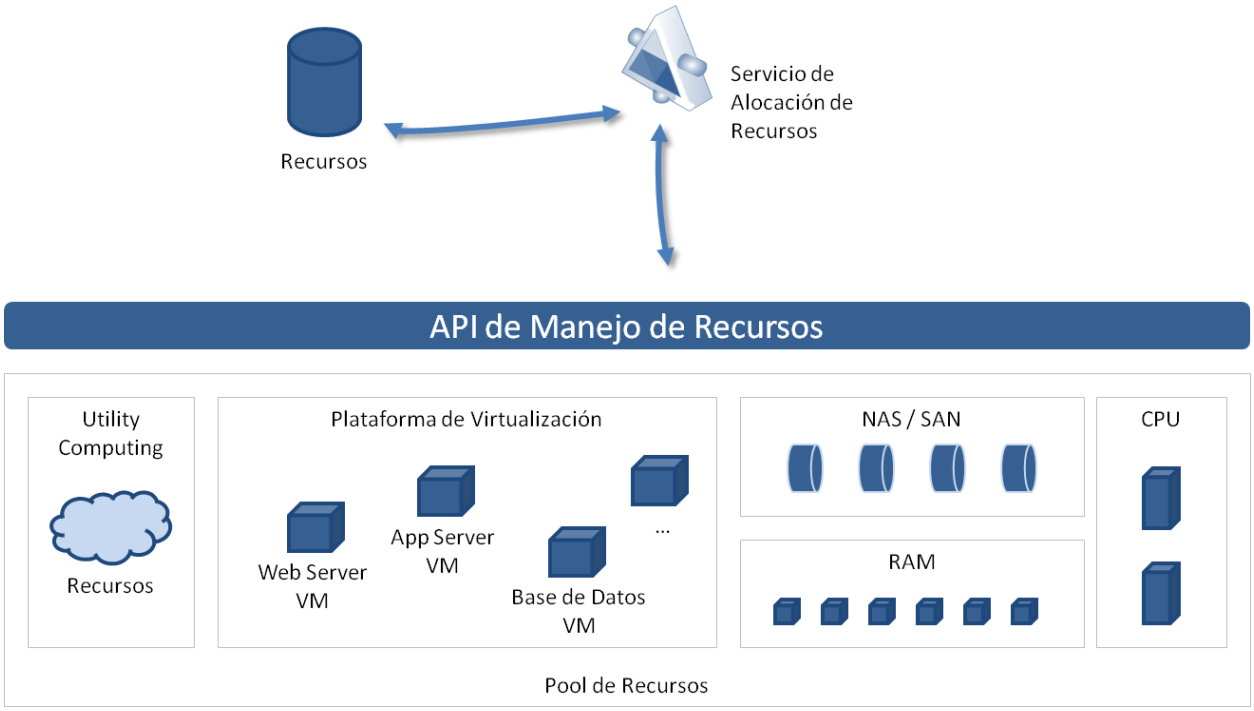



Figura 40 - Sub sistema de manejo de recursos

Utilizando estos servicios se puede implementar una solución de escalabilidad a demanda y balanceo de carga. Si el cliente realiza un pedido de aumento de capacidad para soportar una mayor cantidad de usuarios concurrentes, el servicio de alocación de recursos podría realizar las siguientes operaciones:

1. Chequear la disponibilidad de espacio en máquinas físicas para alocar una nueva máquina virtual
2. Tomar la plantilla de máquina virtual de web server y crear una nueva instancia tomando recursos de la máquina física (memoria y red)
3. Crear un host virtual en el web server apuntando al NAS/SAN donde se encuentra el código base
4. Configurar el load balancer entre los web servers
5. Registrar los recursos en el repositorio de recursos

#### *7.4.3.5 Manejo de SLA*

Los acuerdos de nivel de servicio se generan a partir de los objetivos de nivel de servicio negociados con cada cliente en la etapa de aprovisionamiento. Estos objetivos pueden ser genéricos o particulares dependiendo del cliente.

El proceso comienza con la extracción de los objetivos de nivel de servicio (SLO) y las mediciones relacionadas con los SLO. Un SLO puede tener asociado uno o más indicadores y cada indicador abarca múltiples mediciones en un periodo de tiempo. El cálculo se realiza sobre un período determinado a través del servicio. El resultado puede ser un reporte mensual al cliente enviado vía email con los SLO correspondientes y un dashboard de acceso público que exhiba la disponibilidad del servicio alimentando la transparencia y confianza con el cliente actual y los prospectos. Por otro lado, un dashboard interno servirá para monitoreo, diagnóstico y detección de desviaciones potenciales.

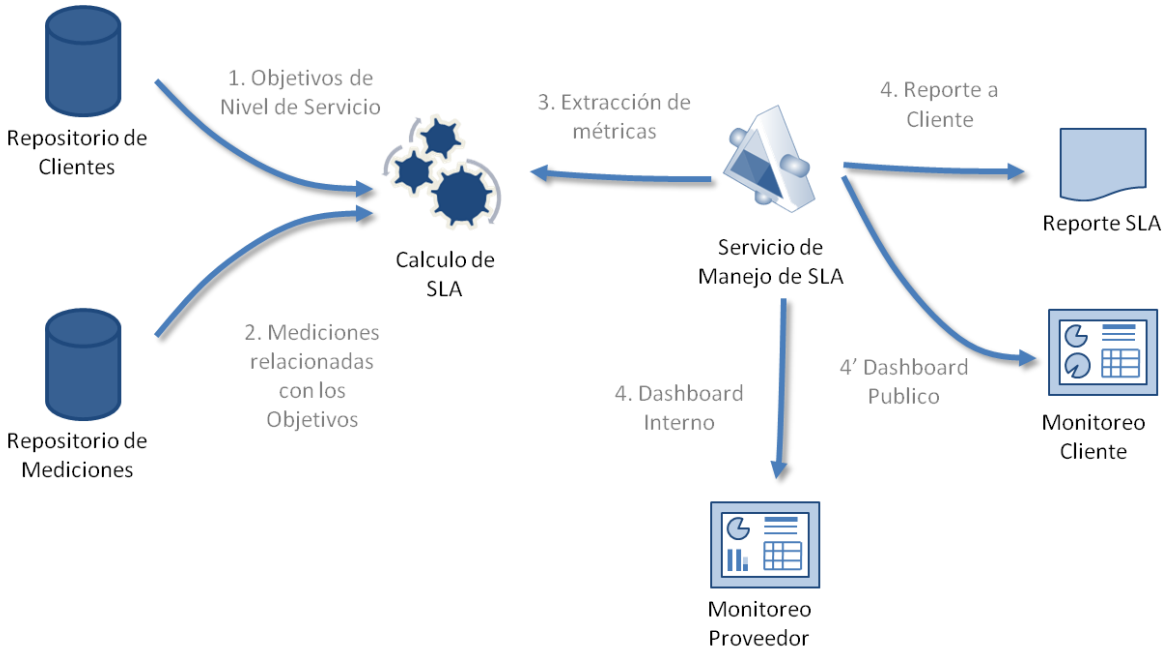



Figura 41 - Mecanismos para el manejo del SLA

La siguiente ilustración muestra un ejemplo de un dashboard de salud del servicio que pertenece a Amazon Web Services. A lo largo del tiempo se muestra la disponibilidad de cada servicio y en ciertos casos se muestra información relacionada con problemas en el servicio y su resolución.

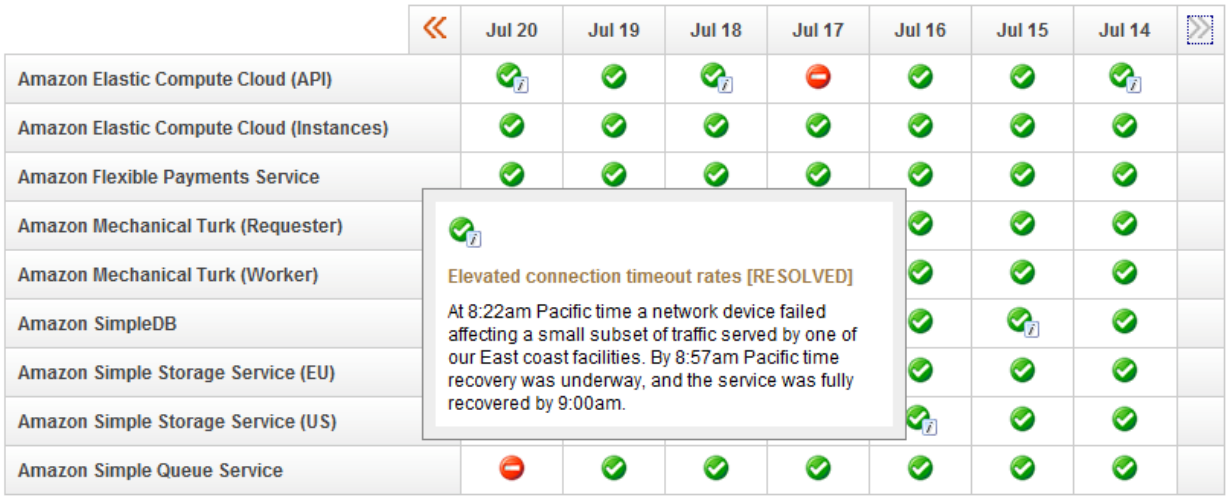



Figura 42 - Dashboard de salud del servicio de Amazon Web Services

Esto puede influir en el cliente positivamente percibiendo la transparencia del servicio.

#### *7.4.3.6 Backup y Restore*

En una aplicación de estas características, el cliente no maneja el almacenamiento de sus datos. La confidencialidad y la privacidad son temas importantes, pero sería más grave si se perdieran los datos y no se pudieran recuperar o si el cliente deseara rescindir el contrato con el proveedor y llevarse los datos y no pudiese hacerlo. Si se optase por un nivel de aislamiento en donde cada cliente tiene su propia instancia de base de datos, el backup sería simplemente a la base de datos entera. Sin embargo en una configuración compartida debería pensarse en una estrategia de exportación de datos propietaria.

La estrategia que utilizan la mayoría de los proveedores de cara al cliente es una exportación semanal de los datos a XML, CSV o texto separado por comas. Algunos también ofrecen grabar un DVD y enviarlo por correo.

Una posible solución planteada en este trabajo se compone de un proceso que corre cada un período determinado (posiblemente en horarios de uso no frecuente), que extrae los datos específicos del tenant, aplica alguna transformación si es necesario y escribe en el medio de soporte de almacenamiento. El proceso podría implementarse con alguna herramienta de ETL (Extract Transformation Load). Luego un servicio expuesto al cliente expondría los últimos X backups (donde X debería ser calculado de acuerdo a la capacidad del proveedor).

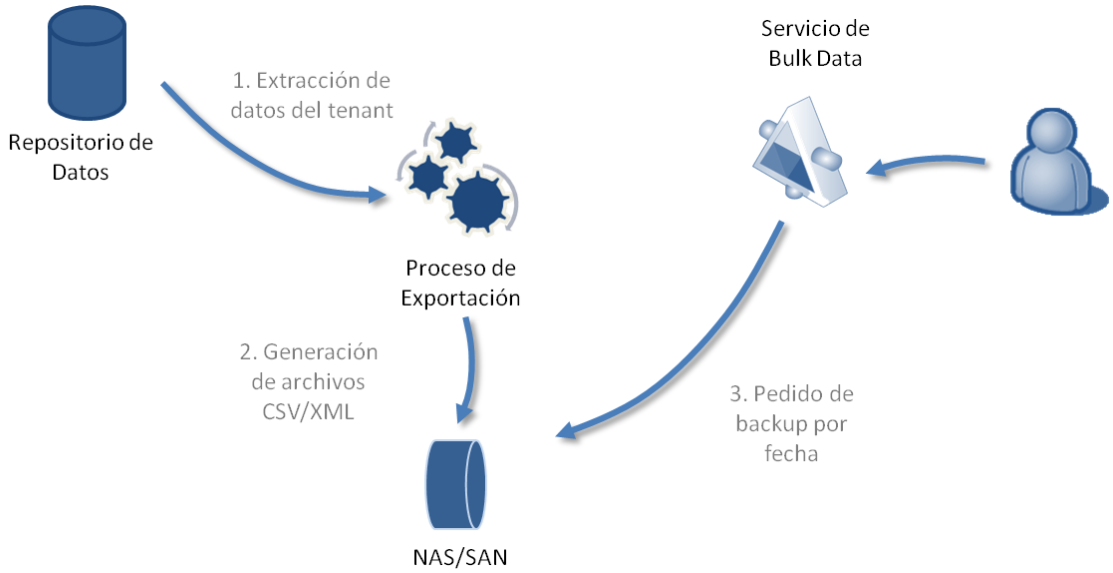



Figura 43 - Mecanismo de backup y extracción de los datos del tenant

### 7.4.4 Customización

Las siguientes características, patrones y tecnologías están relacionadas con la customización de aplicaciones multi tenant.

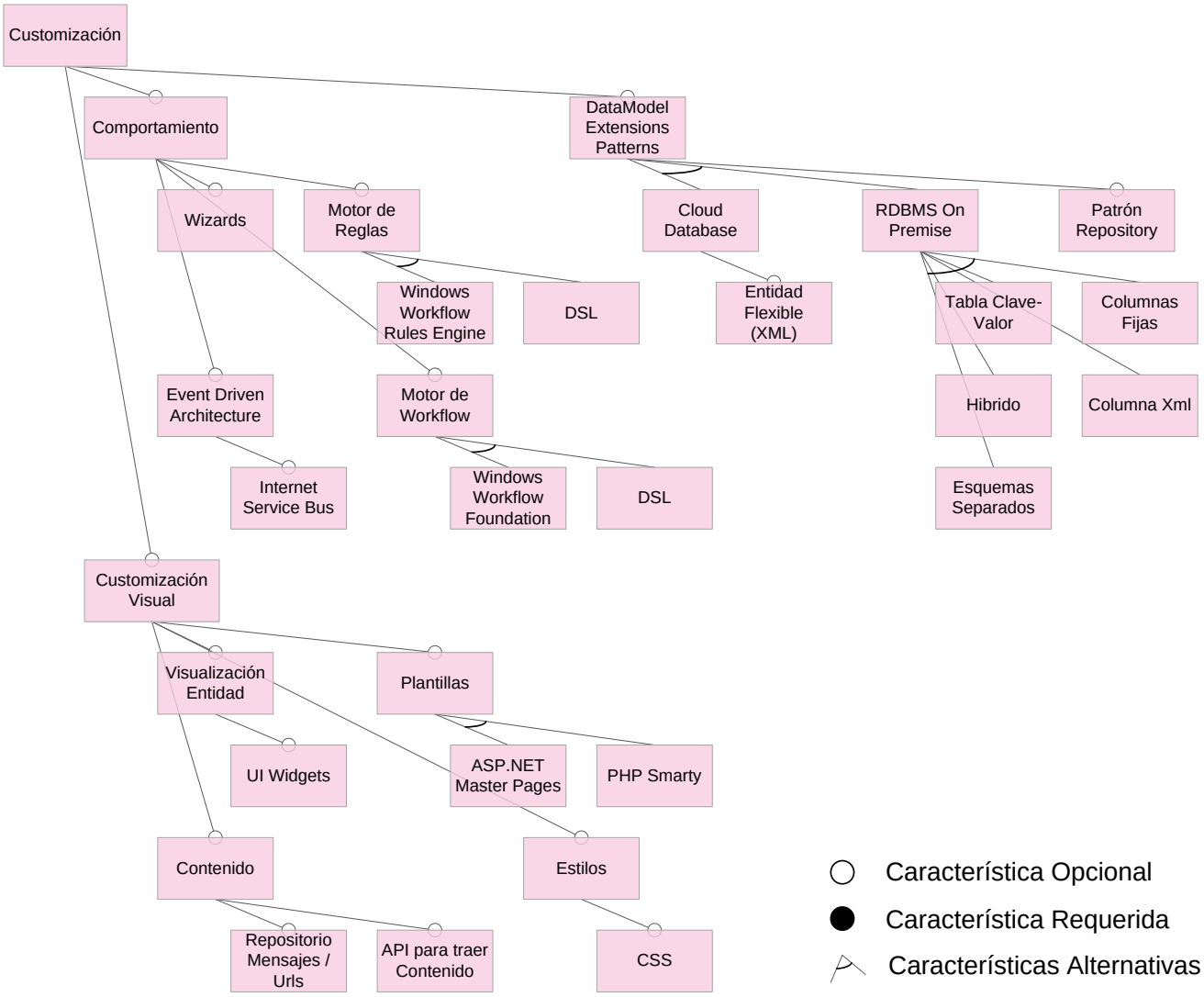



Figura 44 - Características relacionadas con la customización de aplicaciones multi tenant

### *7.4.4.1 Modelo de Datos*

Uno de los principales campos de aplicación de multi tenancy es en la base de datos. El desafío es crear e implementar un modelo de base de datos multi tenant que sea seguro, robusto, escalable y que goce de buena performance. Customizar el modelo de datos significa extender el esquema base de la aplicación de dos formas:

- Agregar un campo a cierta entidad para un tenant especifico
- Agregar un nuevo tipo de entidad

Existen diferentes implementaciones y patrones de customización que investiga esta tesis enumeradas en la siguiente figura.

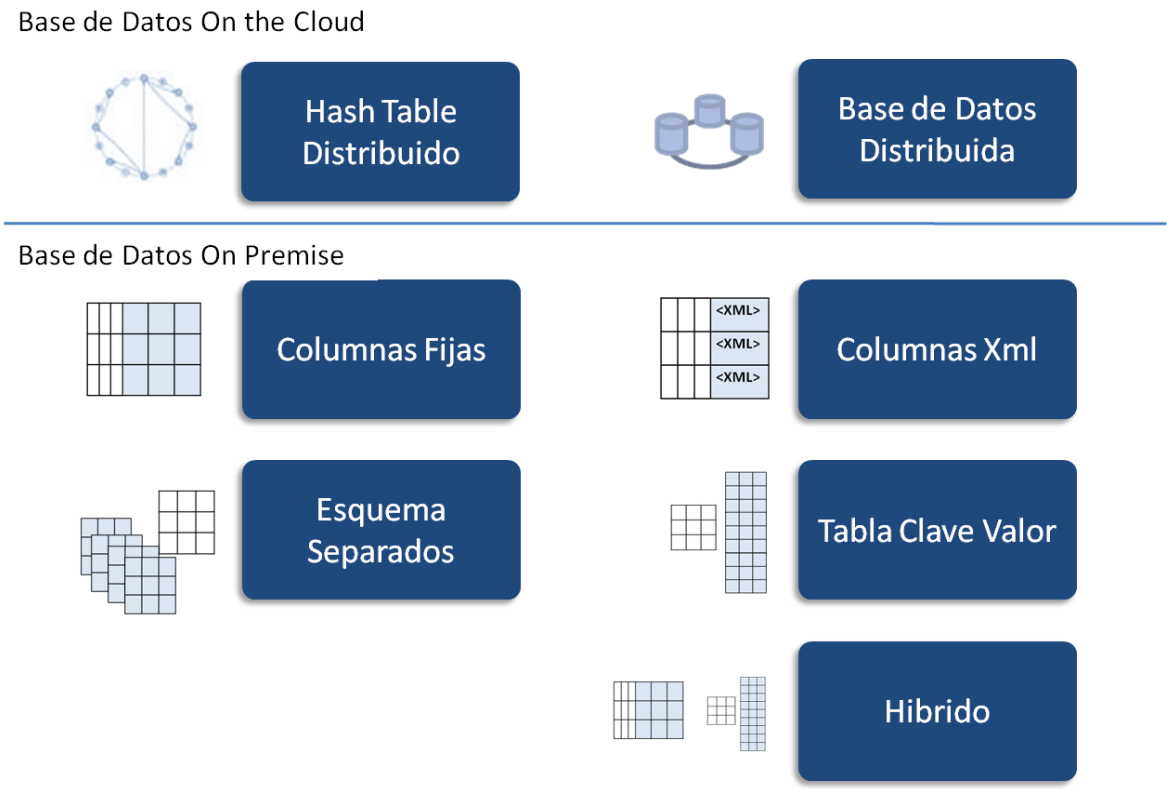



Figura 45 - Patrones de customización de bases de datos

Existe una primera separación entre las bases de datos relacionales *on premise* (que se instalan en un servidor localmente) y las que son brindadas como servicio. En el primer grupo se encuentran las bases de datos comerciales y de código abierto como MySql, Oracle, Microsoft SQL Server, PostgreSQL, etc. En el segundo grupo, relativamente nuevo en la industria se encuentran implementaciones como Google BigTable, Amazon SimpleDB y Microsoft SQL Server Data Services. Este segundo grupo se caracteriza por ser un tipo de almacenamiento de datos muy flexible y cuyo modelo de negocios se basa en *utility computing* (pago por utilización).

#### 7.4.4.1.1 Hash Table Distrubuido

Este tipo de almacenamiento puede crecer en tamaño en el orden de los petabytes. Está implementado a lo largo de miles de servidores pequeños y su diseño está optimizado para accesos de lectura. Las implementaciones más conocidas es BigTable [Google Inc., 2006] y Amazon SimpleDB.

La diferencia más grande es que en este tipo de almacenamiento no existen las relaciones, por lo tanto no existe el JOIN. Por ejemplo, en Amazon SimpleDB los datos son organizados en dominios (*domains*). Los dominios son colecciones de ítems descriptos por pares de clave-valor.

| itemID | description | color | Material |
|---|---|---|---|
| 123 | sweater | Blue, red | |
| 456 | dress shirt | white, blue | |
| 789 | shoes | black | Leather |

Figura 46 - Ejemplo de una tabla (dominio) de Amazon SimpleDB

Las APIs de este tipo de almacenamiento por lo general están basadas en HTTP y REST. Para crear la tabla de la figura anterior, se haría algo como:

PUT (item, 123), (description, sweater), (color, blue), (color, red)
PUT (item, 456), (description, dress shirt), (color, white), (color, blue)
PUT (item, 789), (description, shoes), (color, black), **(material, leather)**

Este tipo de bases de dato ofrecen la posibilidad de agregar atributos a la entidad en cualquier momento. Notar que en la tercera llamada se agrega un atributo "material" que no estaba en los ítems anteriores. Esta capacidad permitiría generar nuevos campos por tenant sin depender del esquema fijo de las base de datos relacionales.

#### 7.4.4.1.2 Base de Datos Distribuida

Otra alternativa son las bases de datos distribuidas. Un ejemplo de este tipo es Microsoft SQL Server Data Services que se paga por utilización y posee servidores distribuidos en localidades estratégicas para obtener buena performance dependiendo de la ubicación del cliente. A diferencia del almacenamiento con hash table distribuidas (DHT), este tipo de base de datos está basado en un modelo relacional. Para acceder y modificar datos provee una API REST o SOAP y al igual que las DHT permiten agregar nuevos campos a una entidad en cualquier momento.

El modelo de Microsoft SQL Server Data Service se separa jerárquicamente en *authorities, containers y entidades*. Un usuario puede contener 1 a N authorities, cada uno de ellos puede contener 1 a N containers y dentro de cada conteiner se pueden almacenar 1 a N entities.

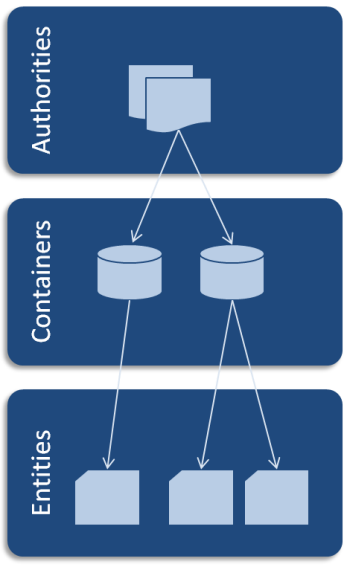



Figura 47 - Modelo jerárquico de Microsoft SQL Server Data Services

Las entidades son flexibles, es decir aceptan cualquier cantidad de campos y se crean mediante llamadas HTTP via REST o SOAP. Por ejemplo, para crear una entidad "Book" utilizando la API REST se enviará el siguiente mensaje HTTP

```
POST https://authority.microsoft.com/container

<Book xmlns:s='http://schemas.microsoft.com/sitka/2008/03/'
      xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
      xmlns:x='http://www.w3.org/2001/XMLSchema'>
      <s:Id>MyBookId</s:Id>
      <title xsi:type='x:string'>Some Title</title>
      <summary xsi:type='x:string'>Some Summary</summary>
      <isbn xsi:type='x:string'>11-1111-11-1</isbn>
      <author xsi:type='x:string'>Mr. Author</author>
      <publisher xsi:type='x:string'>Mr. Publisher</publisher>
 </Book>
```

Utilizando este esquema sería posible implementar diferentes esquemas para cada tenant. Cuando se quiere extender el esquema simplemente al grabar la entidad se envían el xml definido en la metadata del tenant para esa entidad.

La ventaja más grande de esta tecnología sería además de las entidades flexibles, la posibilidad de realizar relaciones entre entidades y hacer consultas con JOIN.

#### 7.4.4.1.3 Columnas Fijas

En el ámbito de las bases de datos relacionales, el primer patrón a que investiga esta tesis es el de columnas fijas. Este patrón se implementa agregando a la tabla que se quiera customizar una serie de columnas fijas de tipo texto (varchar). Por otro lado se crea una tabla de metadata que guardará la definición de los campos tanto compartidos como extendidos para cada entidad, especificando el tipo de datos. Cuando el tenant quiere extender el esquema, se modifica la tabla de metadata asignando una de las columnas (Col1 a ColN) a la nueva columna tratando de elegir de forma consecutiva para mejorar la compresión de datos.

| TenantId | Id | Nombre | Edad | Col1 | Col2 | … | ColN |
|---|---|---|---|---|---|---|---|
| 1 | 1 | Jose | 32 | 2007-02-30 | Peru 375 | | <null> |
| 2 | 2 | Juan | 20 | Argentino | <null> | | <null> |

Figura 48 - Esquema de tabla del patrón columnas fijas

| TenantId | Entidad | Campo | Tipo | NombreCol | Extension |
|---|---|---|---|---|---|
| 0 | Cliente | Nombre | Text | Nombre | No |
| 0 | Cliente | Edad | Int | Edad | No |
| 1 | Cliente | FechaNac | Date | Col1 | Si |
| 1 | Cliente | Direccion | Text | Col2 | Si |
| 2 | Cliente | Nacionalidad | Text | Col1 | Si |

Figura 49 - Tabla de metadata para el patron columnas fijas

En este esquema, las operaciones (consultas, inserciones o actualizaciones) deben ser armadas en tiempo de ejecución ya que las columnas varían entre tenant y tenant. Además se pueden ejecutar las conversiones de tipo necesarias. La siguiente consulta trae todos los registros de clientes con sus campos extendidos del tenant 1.

```
SELECT Id, Nombre, Edad, CONVERT(Col1, DATETIME) as FechaNac, Col2 as Direccion
FROM Cliente
WHERE TenantId = 1
```

Una variante en este patrón es utilizar una serie de columnas de cada tipo. Por ejemplo de Col1 a Col10 son VARCHAR, Col10 a Col15 enteros y así sucesivamente. Esto evitaría hacer conversiones

pero traería otro problema, la dispersión de datos. Si solo se utilizan dos campos de texto y uno entero, hay 8 columnas que quedan sin datos hasta la primera columna de enteros. El efecto de esto es mayor actividad de disco para traer una página de datos. Para resolver esto se puede separar en diferentes tablas cada tipo. *Cliente_ExtText* tendría 10 columnas de texto, *Cliente_ExtInt* tendría 10 columnas de enteros y así sucesivamente.

Otro desafío es la indexación de columnas para realizar búsquedas con filtrado sin incurrir en problemas de performance. Una manera de resolver esto es crear columnas repetidas que guarden los mismos valores que las columnas originales. Estas columnas tendrán un tipo fuerte y se creará un índice sobre ellas. Del lado de la aplicación habrá que manejar la sincronización entre estas la columna original y la indexada.

#### 7.4.4.1.4 Columnas XML

Una alternativa utilizada en bases de datos con soporte XML, es utilizar una columna de este tipo y guardar allí las extensiones de cada tenant. La tabla de metadata se mantendría similar al patrón columnas fijas sin la necesidad de guardar el nombre de columna (Col1, Col2, etc) ya que existe una única columna: *ColExt*. Cuando el tenant quiere extender el esquema, simplemente se agrega un registro en la tabla de metadata.

| TenantId | Id | Nombre | Edad | ColExt |
|---|---|---|---|---|
| 1 | 1 | Jose | 32 | <Extensiones><br><FechaNac>2007-02-30</FechaNac><br><Direccion>Peru 375</Direccion><br><Extensiones> |
| 2 | 2 | Juan | 20 | <Extensiones><br><Nacionalidad>Argentino</Nac…><br><Extensiones> |

Figura 50 - Customización utilizando columnas XML

Las consultas se realizarían de la siguiente forma (T-SQL de Microsoft SQL Server):

```
SELECT Id, Nombre,
       ColExt.value('(/Extensiones/FechaNac)[1]', 'DATETIME') as FechaNac,
       ColExt.value('(/Extensiones/Direccion)[1]', 'VARCHAR(MAX)') as Direccion,
FROM Cliente
WHERE TenantId = 1
```

#### 7.4.4.1.5 Esquemas separados

El patrón *esquemas separados* se implementa manteniendo una tabla por tenant por entidad. En este caso cada customización que haga el tenant va a alterar el esquema de la tabla agregando una nueva columna con el nombre y el tipo asignado por el tenant. Se utilizan los tipos de la base de datos directamente.

| Id | Nombre | Edad | FechaNac | Direccion |
|---|---|---|---|---|
| 1 | Jose | 32 | 2007-02-30 | Peru 375 |

Cliente_Tenant1

| Id | Nombre | Edad | Nacionalidad |
|---|---|---|---|
| 2 | Juan | 20 | Argentino |

Cliente_Tenant2

**Figura 51 - Patrón de esquemas separados. La entidad cliente separada en dos tablas, una para cada tenant con sus propios campos**

| TenantId | Entidad | Campo | Tipo | Extension |
|---|---|---|---|---|
| 0 | Cliente | Nombre | Text | No |
| 0 | Cliente | Edad | Int | No |
| 1 | Cliente_Tenant1 | FechaNac | Date | Si |
| 1 | Cliente_Tenant1 | Direccion | Text | Si |
| 2 | Cliente_Tenant2 | Nacionalidad | Text | Si |

**Figura 52 - Metadata para el patrón de esquemas separados**

Las consultas en este escenario se realizan de la siguiente forma:

```
SELECT Id, Nombre, Edad, FechaNac, Direccion
FROM Cliente_Tenant1
```

En esta alternativa hay que estudiar la cantidad de tenants potenciales porque se podría llegar a generar una cantidad de objetos en la base de datos inmanejables (10 entidades x 1000 tenants = 10.000 tablas).

#### 7.4.4.1.6 Tabla Clave-Valor

La alternativa tabla clave-valor se implementa creando una tabla por entidad que guarde los valores de los campos extendidos. La tabla de metadata se mantiene igual que en el patrón columnas fijas

(sin el nombre de columna). Cuando el tenant quiera extender la entidad, agrega un nuevo campo en la tabla de metadata. Las consultas, las inserciones y actualizaciones se generarán a partir de la metadata.

Cliente_Base

| TenantId | Id | Nombre | Edad |
|---|---|---|---|
| 1 | 1 | Jose | 32 |
| 2 | 2 | Juan | 20 |

Cliente_Extension

| Id | Clave | Valor |
|---|---|---|
| 1 | FechaNac | 2007-02-30 |
| 1 | Direccion | Peru 375 |
| 2 | Nacionalidad | Argentino |

Figura 53 - Tabla base y tabla extensión para el patron Tabla Clave-Valor

Las consultas se harían de la siguiente forma:

```
SELECT b.Id, b.Nombre, b.Edad,
    MIN(CASE WHEN e.Clave = 'FechaNac' THEN e.Valor END) as FechaNac,
    MIN(CASE WHEN e.Clave = 'Direccion' THEN e.Valor END) as Direccion
FROM Cliente_Base as b
LEFT OUTER JOIN
    Cliente_Extension AS e ON b.Id = e.Id
WHERE
    b.TenantId = 1
GROUP BY
    b.Id, b.Nombre, b.Edad
```

Tener en cuenta que se está realizando un pivot de los datos y esta operación puede ser costosa para el motor de base de datos.

#### 7.4.4.1.7 Híbrido

El modelo híbrido propone mezclar algunos de estos patrones. Por ejemplo, se podría mezclar el patrón de columnas fijas con el de esquemas separados. Dependiendo de la cantidad de columnas extendidas se utilizaría uno u otro. Todas las entidades tendrían 5 columnas para extender, los que quieren más de esa cantidad se les crearía una tabla separada con los campos extra.

### *7.4.4.2 Comparación de los diferentes patrones*

Habiendo definido los diferentes patrones se analizará la performance de alguno de ellos. Este trabajo fue realizado en conjunto con el grupo de Arquitectura Estratégica de Microsoft

Corporation. Los documentos están disponibles en la web bajo el nombre *Multitenant Database Performance Guide* [Pace y otros, 2007].

El siguiente laboratorio fue montado con 6 agentes generadores de carga, un controlador con la lógica de cada test y el sistema bajo testeo, la base de datos.

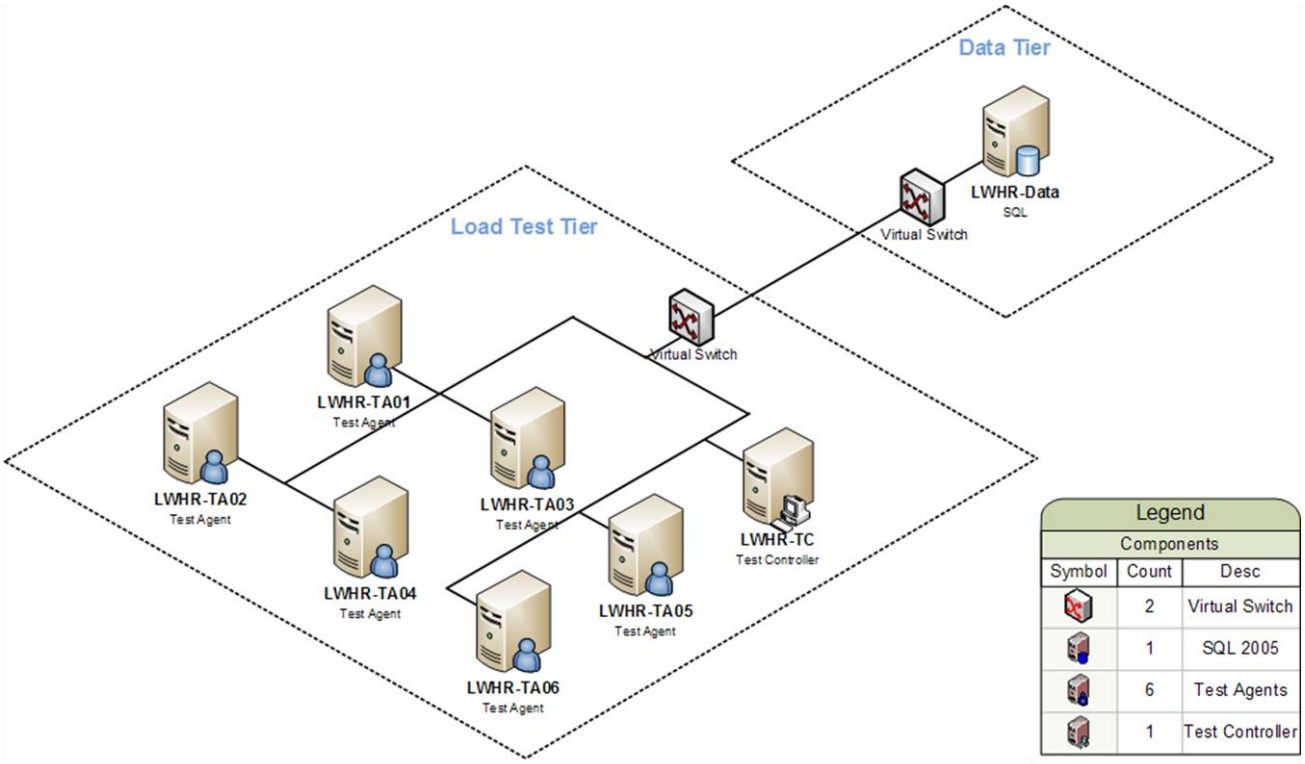



Figura 54 - Lab de test de performance y carga

La base de datos contaba con la configuración siguiente:

- SQL Server 2005
- HP Proliant DL 385 G1
- 2 32 Bits Processors (1.8 Ghz)
- 4 GB Memory
- 4 spindles SCSI 10000 RPM

Se corrieron los tests para traer 50 registros de una tabla que implementa el patrón Tabla Clave-Valor, Columna Xml y Columnas Fijas. Los resultados se describen a continuación.

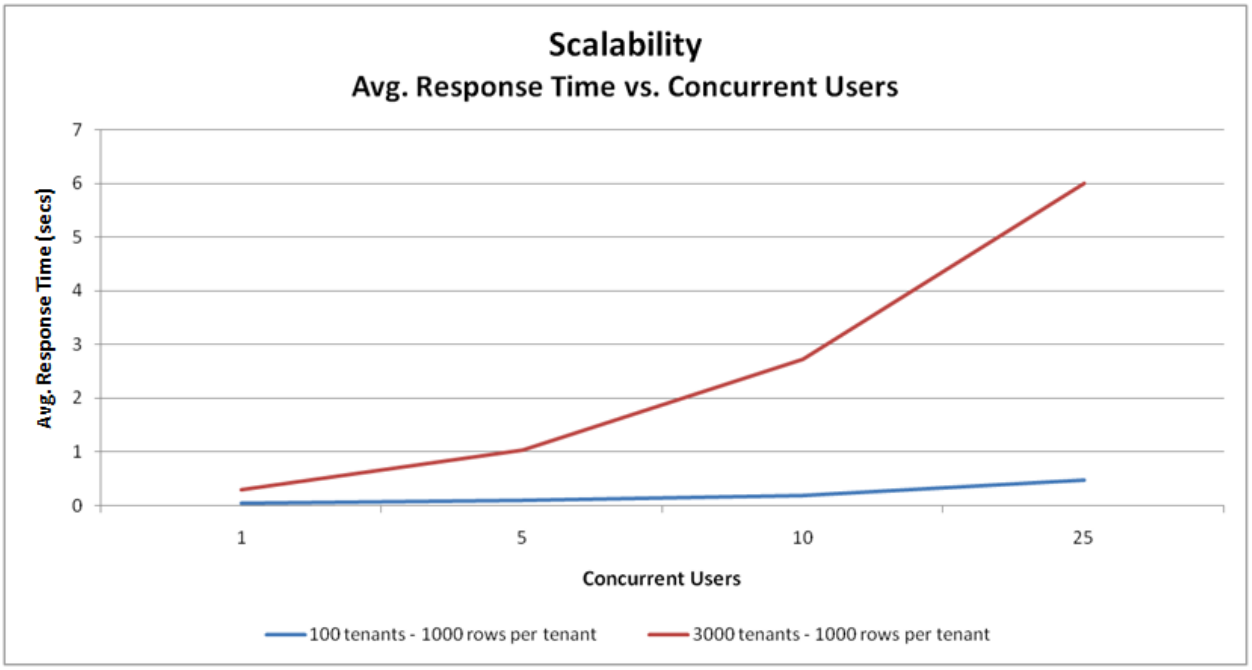



Figura 55 - Escalabilidad del patrón Tabla Clave-Valor [Pace, et al., 2007]

La línea azul muestra el tiempo de respuesta promedio a medida que la cantidad de usuarios concurrentes aumenta de 1 a 25. La base de datos esta pre-cargada con una entidad que posee 4 campos base y 9 campos extendidos, 100 tenants y 1.000 filas por tenant, es decir 100.000 registros para la tabla base y 900.000 registros para la tabla extendida. La escalabilidad con esta cantidad de registros es buena.

La línea roja en cambio muestra que con una mayor cantidad de tenants (30 veces más que el test anterior) el patrón Tabla-Valor comienza a mostrar tiempos de respuesta cada vez más altos. La tabla base tendría 3.000.000 de registros y la tabla extendida tendría 27.000.000 de registros.

Para validar esto se puede analizar el siguiente gráfico tomado con 25 usuarios concurrentes.

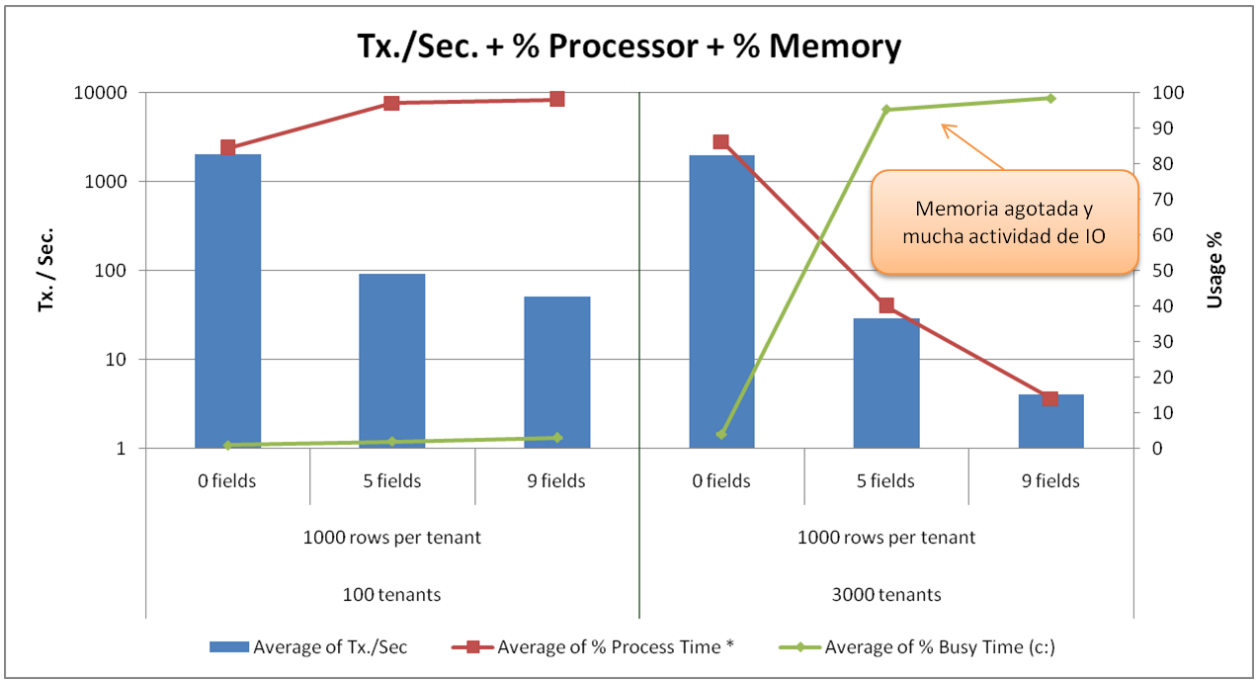



Figura 56 - Análisis de contadores de rendimiento para el test del patrón Tabla Clave-Valor con 25 usuarios concurrentes

Cuanto más tenants y filas haya y el número de usuarios concurrentes se incremente, la consulta de registros random de las tablas base y extendida genera un agotamiento de memoria ya que una gran cantidad de páginas de registros son bajados a memoria. Al agotarse la memoria, se bajan las páginas a disco y eso crea un cuello de botella que genera un tiempo de respuesta cada vez más alto.

El comportamiento es similar cuando se selecciona un único registro aplicando la clausula WHERE. De este modo se genera una carga más fuerte en el procesador debido al filtrado.

La inserción no representa mayores problemas. El tiempo de respuesta es saludable y la actividad de los discos de log y datos se mantiene en los valores normales.

El patrón Columnas Fijas muestra un tiempo de respuesta mucho más saludable. Al subir la cantidad de usuarios concurrentes, el tiempo de respuesta se mantiene en los mismo valores, lo cual significa que es este patrón es más escalable.

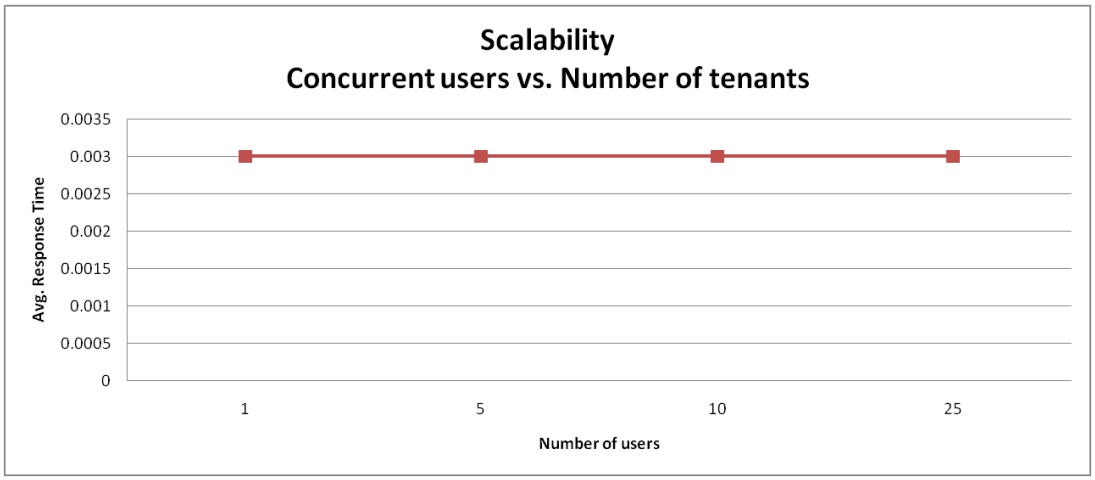



Figura 57 - Tiempo de respuesta para el patrón Columnas Fijas

Los valores de CPU y disco se mantienen por debajo del 40% y 5% respectivamente, lo cual es totalmente saludable.

Por último, el patrón Columnas XML muestra un comportamiento similar al de Tabla Clave-Valor. De hecho los tiempos de respuesta son peores, llegando a más de 20 segundos por operación con 25 usuarios concurrentes, 3000 tenants y 1000 registros por cada uno.

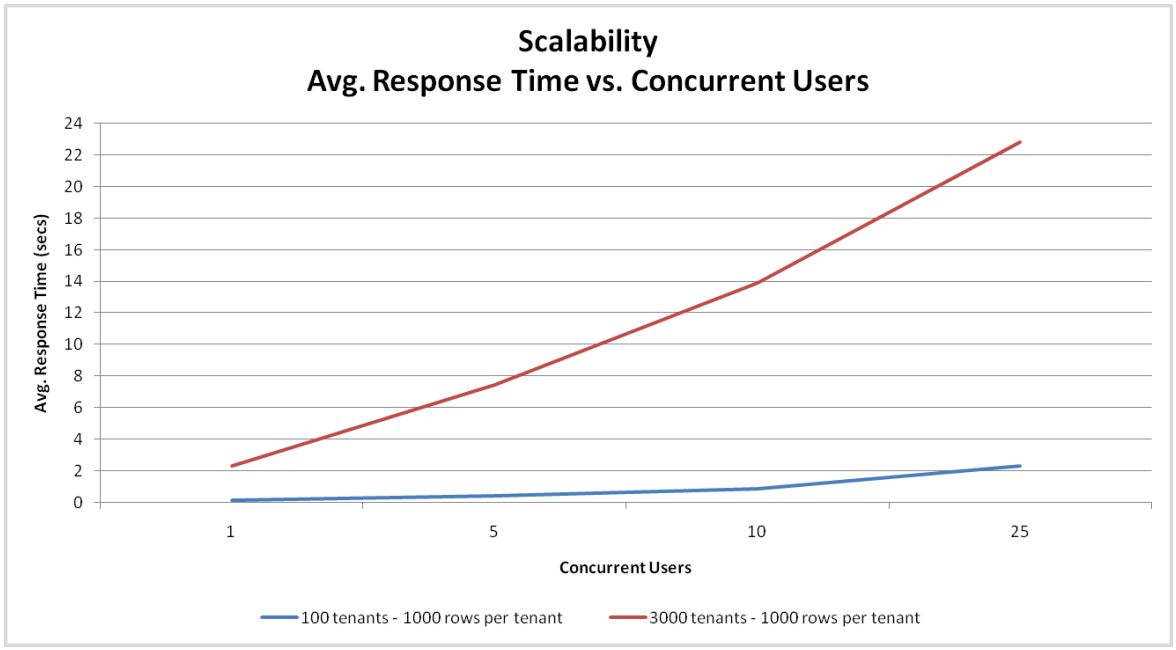



Figura 58 - Tiempo de respuesta para el patrón Columna Xml

Los contadores de rendimiento de disco y CPU en el caso de 100 tenants se mantienen por debajo de 5% y por arriba de 70% para la CPU. Eso significa que el uso de columnas Xml produce un trabajo

importante para la CPU en las consultas. El gráfico cambia cuando son 3000 tenants. En este caso el disco vuelve a ser el cuello de botella con 100% del tiempo ocupado.

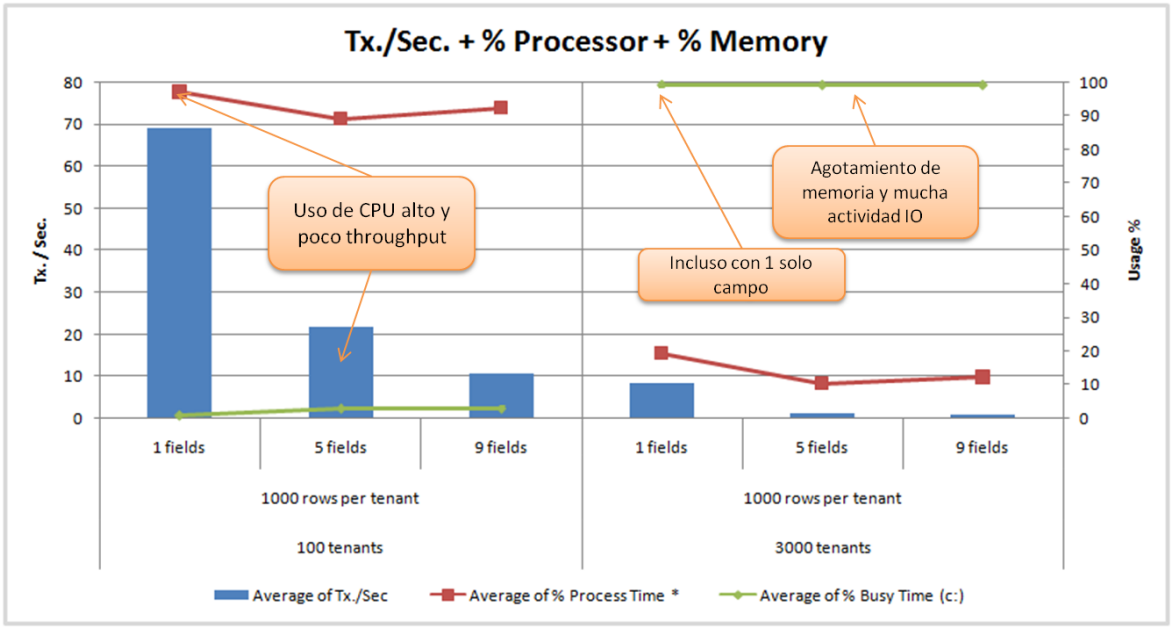



Figura 59 - Contadores de rendimiento de disco y CPU para el patrón Columna XML

#### 7.4.4.2.1 Conclusión

En resumen, este trabajo de investigación demuestra que los patrones más convenientes son los de columnas fijas o esquemas separados. De cualquier manera, previo a la selección del patrón es recomendable hacer una evaluación tomando en cuenta la carga y performance de el o los patrones elegidos utilizando los parámetros (como cantidad de tenants y registros por tenant) que tengan sentido para la aplicación en cuestión.

### *7.4.4.3 Acceso a Datos*

Una vez definido el modelo, es necesario tener una estrategia para acceder a los datos programáticamente. El hecho de tener diferentes campos extendidos dependiendo el cliente, hace que el acceso programático sea más complejo comparando con el caso en donde las entidades son conocidas previamente en tiempo de diseño.

Las entidades se pueden manejar como diccionarios de clave valor

```
public class ExtensibleEntity
{
    public Guid Id { get; set; }
```

```
    public IDictionary<string, object> Fields { get; private set; }
}
```

Luego, se puede utilizar el patrón *Repository* que abstrae el código de acceso a los datos en una clase abstracta.

```
public abstract class Repository {
    void Insert( ExtensibleEntity e );
    void Update( ExtensibleEntity e );
    void Delete( Guid id );
    IEnumerable<ExtensibleEntity> GetAll();
    IEnumerable<ExtensibleEntity> Search(string criteria);
    ExtensibleEntity GetById( Guid id );
}
```

A partir de esto se pueden crear implementaciones para los diferentes patrones mencionados en la sección anterior.

#### *7.4.4.4 Customización Visual*

Esta tesis define la customización visual como la capacidad de cambiar el diseño estructural y visual de una aplicación *Software como Servicio*.

##### 7.4.4.4.1 Skins y Customización de estructura

La implementación de *skins* es usada en diferentes dominios. Por ejemplo, varios motores de blog proveen la capacidad de cambiar el estilo de un blog.

Una de las maneras más simples de lograr esto es utilizando Cascading Style Sheets (CSS) u hojas de estilo. Un tenant podría customizar el estilo de la aplicación simplemente creando un nuevo CSS siguiendo las reglas predeterminadas por el proveedor. Una particularidad de este mecanismo es que el HTML debe ser cuidadosamente realizado para realmente separar el contenido de la forma.

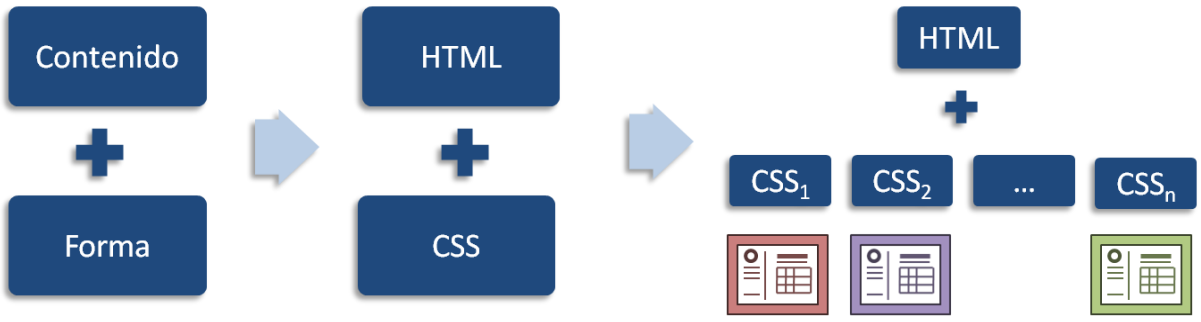



Figura 60 - Separación del contenido de la forma con HTML y CSS

El administrador de la aplicación crea la hoja de estilos y la sube al repositorio de metadata. La aplicación web utiliza el servicio de visualización para obtener la hoja de estilos específica para ese tenant.

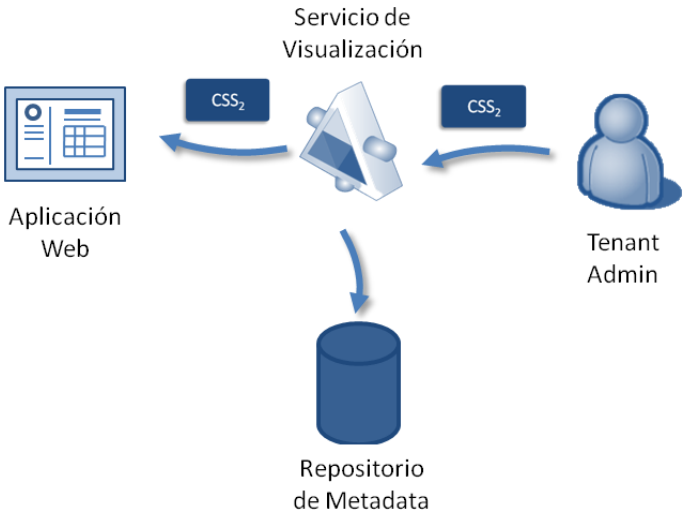



Figura 61 - Componentes para implementar skins

En la aplicación web, por cada página se declara la hoja de estilos programáticamente.

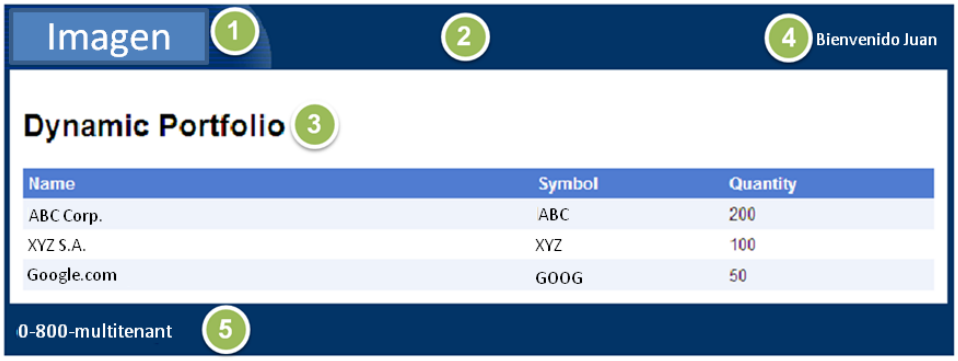
```
<link rel="stylesheet" type="text/css" href="<%=css_tenant%>" />
```

La customización de la estructura de la página es similar. Lo que cambia es la estructura del HTML. Es decir donde se ubican el encabezado, pie de página, contenido y menúes de navegación. Esta característica se puede implementar con plantillas (*templates*). Previo al proceso de rendering de la página, se detectará qué tipo de estructura eligió el usuario y se cambiará la plantilla en tiempo de ejecución. A diferencia de CSS, no existe un lenguaje estándar para plantillas estructurales (HTML 5 comienza a proponer algunos tags para dar estructura), por lo tanto una alternativa utilizada es que el proveedor proponga algunas alternativas fijas de estructura diferente y el tenant elija mediante una herramienta de configuración.

Dependiendo de la tecnología a utilizar, en ASP.NET la implementación se llama Master Page, en PHP hay diferentes implementaciones (smarty, FastTemplate, etc.), en Java se llama Velocity y hay muchas más.

#### 7.4.4.4.2 Contenido customizable

La siguiente figura muestra los diferentes escenarios en los cuales se puede aplicar la customización de contenido.




| Escenarios | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Traer un archivo en base a un contexto determinado | X | X | | | |
| Traer una cadena de texto en base a un contexto determinado | | | X | | |
| Traer una cadena de texto a procesar y concatenar | | | | X | X |

Figura 62 - Escenarios de customización de contenido

La implementación de este tipo de mecanismos requiere de una API que permite traer contenido a partir de cierto contexto. Ese contexto se puede definir jerárquicamente. Es decir el contenido existe a nivel de aplicación, pero un tenant puede decidir sobrescribir cierto contenido (mensaje, imagen, etc.) y hasta puede haber un tercer nivel, llamado arbitrariamente segmento, que sobrescriba al tenant o a la aplicación.

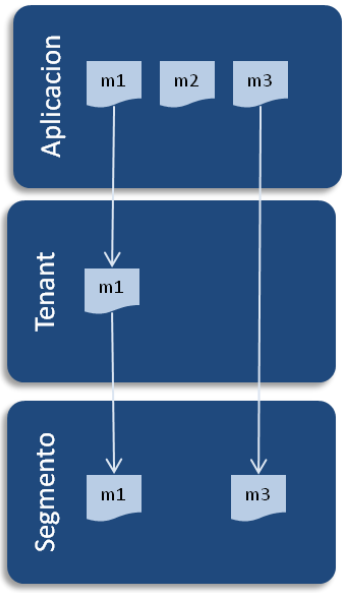



Figura 63 - Diferentes contenidos dependiendo el contexto. El contexto es jerárquico.

#### 7.4.4.4.3 Visualización de Entidad

Al contar con modelos de datos genéricos y extensibles, la visualización de las entidades es un desafío ya que el tipo de datos recién se conoce en tiempo de ejecución a partir de la metadata definida por el tenant.

Una posible solución a esto es contar con controles que encapsulen la visualización de un cierto tipo de dato. Por ejemplo, si una entidad es de tipo fecha, habría un control que muestre un campo de texto y un calendario. El código encargado del rendering de la página leería la metadata asociada a la entidad y según el tipo del campo mostraría el control correspondiente.

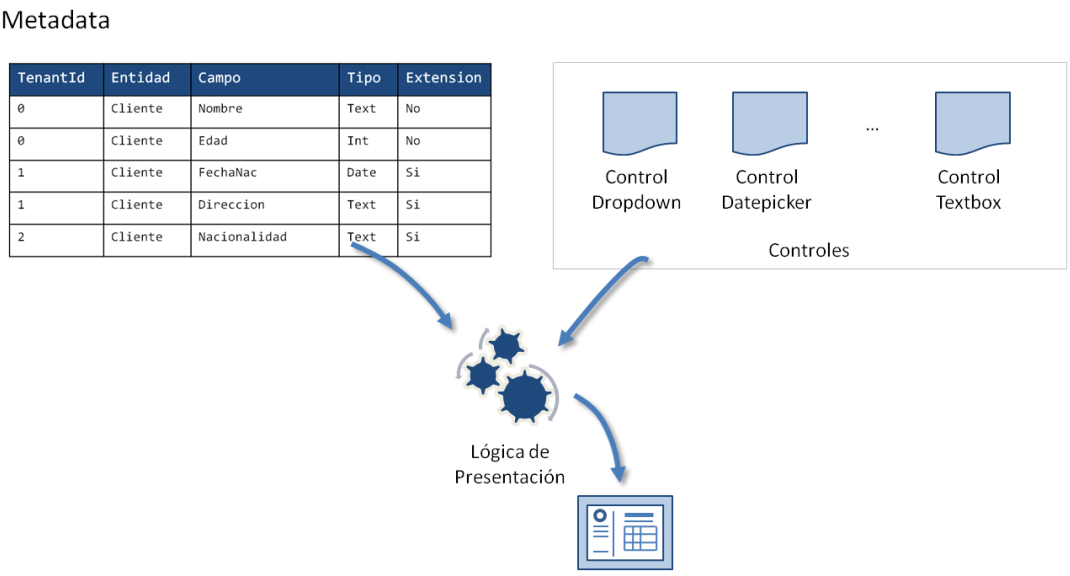



Figura 64 - Visualización de entidades genéricas a partir de metadata

### *7.4.4.5 Customización de Comportamiento*

Luego de haber presentado la customización del modelo de datos y la visualización, el último punto que analiza esta tesis es la customización de comportamiento. El comportamiento dentro de una aplicación es la lógica con que se ejecutan cada una de las partes.

La customización del comportamiento puede llegar a perjudicar a otros clientes dependiendo del nivel de exposición de las herramientas. Las opciones pueden ir desde un simple panel de control con unas pocas opciones de configuración hasta un lenguaje de programación para modificar el comportamiento de la aplicación.

La figura a continuación ilustra el espectro de soluciones para modelar la variabilidad de un producto [Overview of Generative Software Development, 2005]. Los wizards, a diferencia de una simple lista de parámetros que modifica el comportamiento de una aplicación, captura la dependencia entre las opciones. Por lo general cuando se utilizan wizards, se espera que toda la funcionalidad esté definida a priori y la cantidad de puntos de variación es generalmente poca. Los modelos de características (*feature model*), en cambio, son utilizados para manejar un gran número de variaciones, aunque sigue siendo una cantidad limitada y por naturaleza la variabilidad no se representa únicamente con estructuras estáticas como un árbol. No sirve para expresar, por ejemplo, orden de ejecución. Eso lleva al tercer escenario, los lenguajes de dominio específico o DSL (Domain Specific Language). Desde el punto de vista comparativo, los DSLs expresan la

variabilidad como parte del lenguaje en sí y atacan no solo lo estático sino la parte dinámica de la variabilidad.

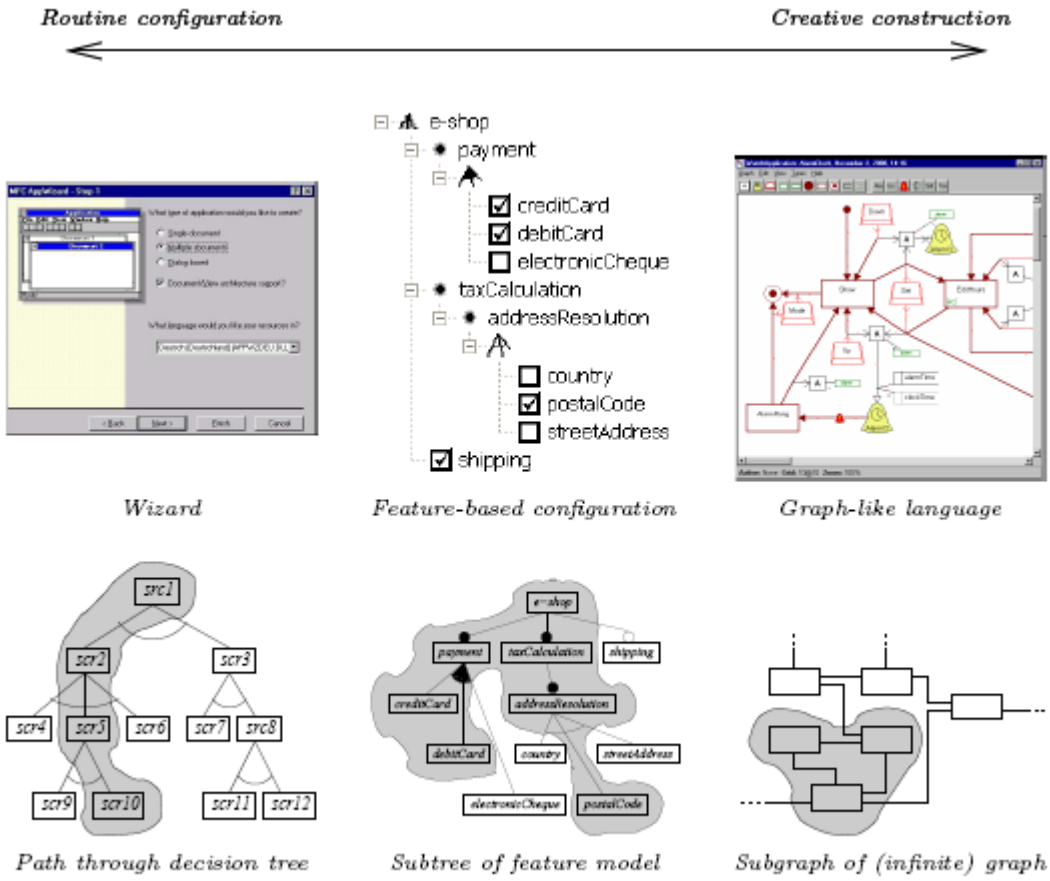



Figura 65 - Espectro de soluciones para resolver variabilidad de un producto (Krzysztof Czarnecki)

Dicho esto, a continuación se presenta algunos patrones y mecanismos a tener en cuenta para implementar la customización de comportamiento.

#### 7.4.4.5.1 Customización de Flujo de Trabajo y Reglas de Negocio

Para poder modificar un componente de comportamiento en tiempo de ejecución, es necesario que exponga una interfaz programática para modificar su estructura o que tenga una representación textual y que el motor de ejecución del componente interprete y ejecute su contenido. La representación textual es interesante si se quiere almacenar el comportamiento e incluso versionar.

Los lenguajes dinámicos a comparación de los lenguajes estáticos representan una alternativa interesante ya que su naturaleza de interpretación en tiempo de ejecución y su representación textual permitiría la modificación en runtime ya sea por el proveedor o el tenant.

Como se adelantó al principio de esta sección, una alternativa aun más interesante desprendida de los lenguajes dinámicos son los DSL (Domain Specific Languages). Los DSLs son lenguajes dedicados a un dominio en particular. Esto no es un concepto nuevo en la industria del software, el lenguaje T-SQL es un lenguaje específico para hacer consultas a una base de datos, o el lenguaje de scripting de UNIX es específico para organizar datos. El lenguaje de macros de Excel es un DSL que permite al usuario cambiar el comportamiento del producto. Se puede trazar una analogía entre la aplicación Excel y las macros y una aplicación multi tenant customizable y un DSL textual que permita al tenant customizar el comportamiento modificando este texto.

El siguiente fragmento de texto es un DSL para crear maquinas de estado [Nathan, 2008]. En este caso se está utilizando para definir los estados y eventos por los que pasa una orden de compra. La palabra *trigger* define los eventos y la palabra *state* define los estados. Las transiciones se definen con la palabra *when* y el estado inicial y final. En este ejemplo, el proveedor podría agregar lógica de programación utilizando el concepto de tarea. Cada vez que se cambia de estado se llama a una tarea "Log" que recibirá el contexto de la máquina de estado.

```
on_change_state      @Log, "on_change_state"

trigger @OrderPlaced
trigger @CreditCardApproved
trigger @CreditCardDenied
trigger @OrderCancelledByCustomer

state @AwaitingOrder:
    when @OrderPlaced              >> @AwaitingPayment

state @AwaitingPayment:
    when @CreditCardApproved       >> @AwaitingShipment
    when @CreditCardDenied         >> @OrderCancelled
    when @OrderCancelledByCustomer >> @OrderCancelled

…
```

El tenant, o el proveedor en representación del tenant, podría definir su propia maquina de estados para el manejo de órdenes de compra y así estaría cambiando el comportamiento de la aplicación.

Lenguajes como Ruby, Boo, Python entre otros permiten definir DSLs gracias al soporte de *metaprogramming* que ofrecen. Otra opción un poco más costosa sería crear un parser.

La definición de reglas también se puede implementar con la técnica mencionada anteriormente. No obstante, existen otros mecanismos que cumplen con el mismo objetivo. Los motores de reglas permiten definir un conjunto de reglas y ejecutar acciones a partir de la evaluación de esas reglas.

La siguiente figura muestra el editor de reglas que es parte de la plataforma Microsoft llamado Windows Workflow Foundation [Microsoft].

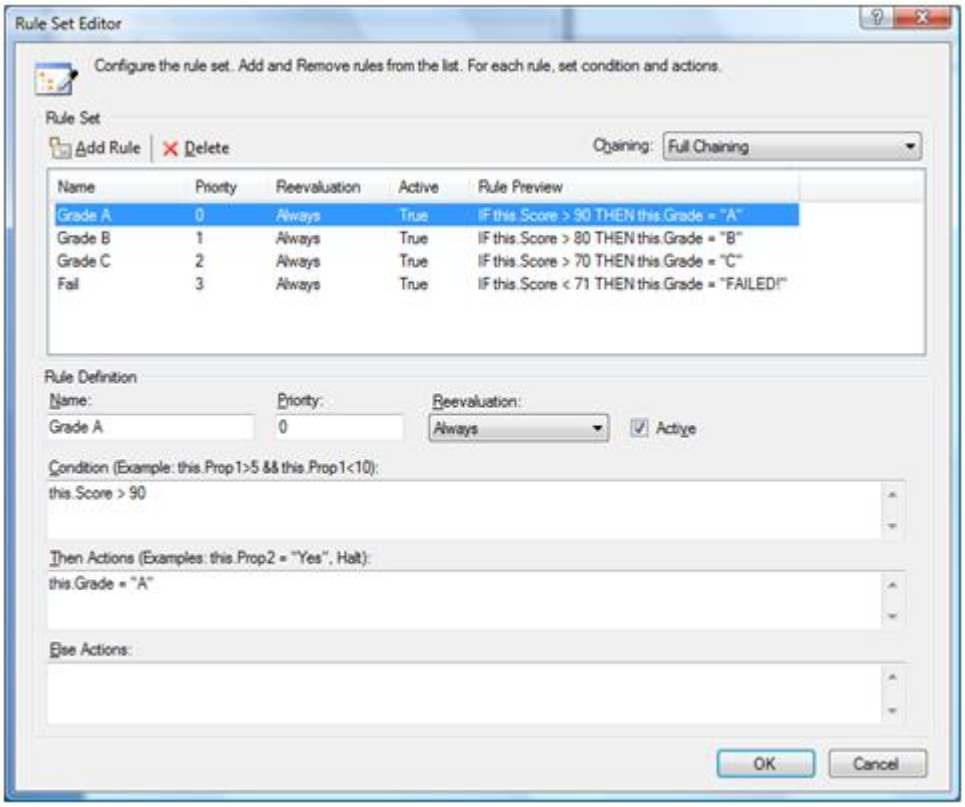


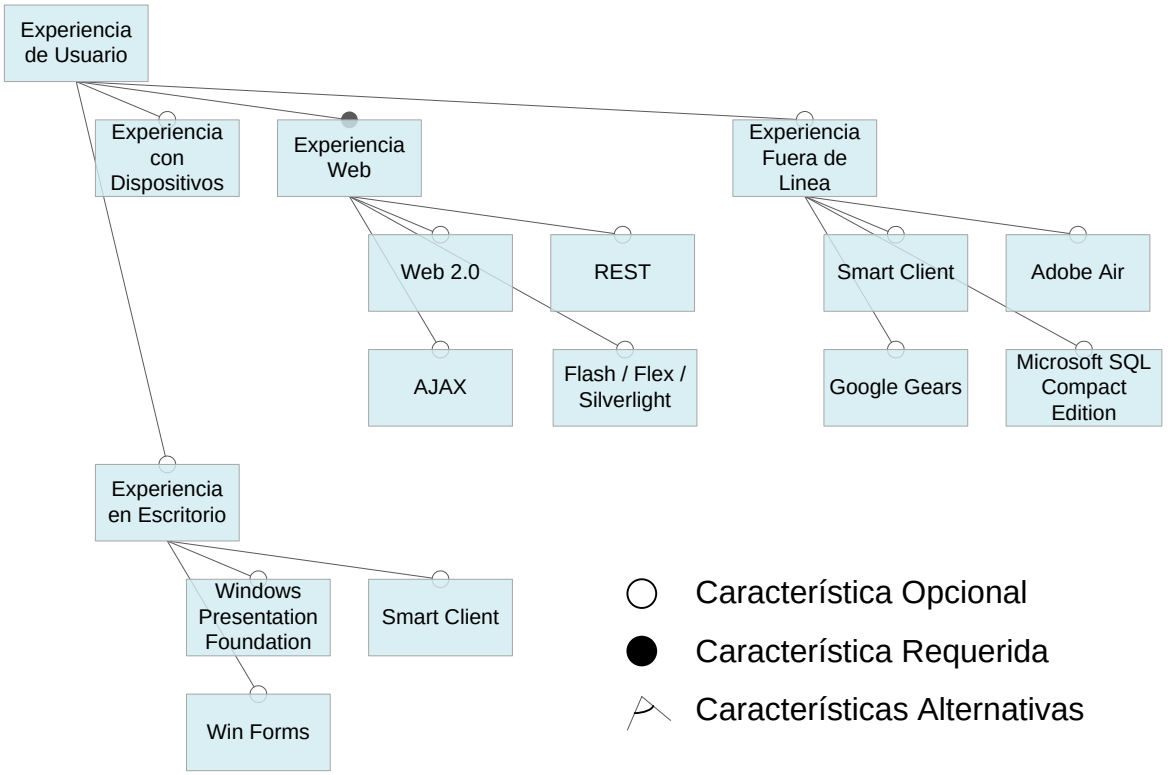

Figura 66 - Motor de reglas de Microsoft Windows Workflow Foundation

Aquí se define un conjunto de reglas para evaluar un examen. Si obtuvo más de 90, es A; más de 80 es B; más de 70 es C; menos de 70 o igual falla.

Una característica de este motor que hace que se pueda utilizar en aplicaciones de estas características es que las reglas se representan textualmente en un formato XML. Esto permite que se pueda guardar en un repositorio, permitir al tenant modificar las reglas y que el proveedor almacene la versión modificada.

Por último, un patrón más avanzado para permitir modificar el comportamiento de una aplicación es mediante una arquitectura de eventos. Los eventos deberían ser expuestos por la web y el tenant debería poder subscribirse a estos eventos ejecutando cierta lógica. Por ejemplo, en una maquina de estado de un proceso de reclutamiento, se podría lanzar un evento cada vez que el candidato pasa a otra etapa (estado). El tenant podría suscribirse al cambio de estado, ejecutar cierta lógica y lanzar otro evento que el proveedor estaría escuchando para cambiar al candidato a otro estado.

### 7.4.5 Experiencia de Usuario




#### *7.4.5.1 Experiencia Web*

La web ha evolucionado de ser una enrome biblioteca de solo lectura a ser una plataforma para hacer negocios. Las tecnologías web siguen evolucionando y permiten incrementar la experiencia del usuario y usabilidad. Las *aplicaciones web* son un escalón más en este proceso.

Las siguientes guías y prácticas son recomendaciones y deberían ser validadas con feedback del usuario y tests de usabilidad. Las guías están basadas en otros trabajos relacionados con diseño web y usabilidad [Hoggvliet, 2008]. En su libro "About Face", Alan Cooper, opina que la diferencia entre una aplicación de escritorio y una aplicación web debe tratar de minimizarse. Cooper dice que el diseño de una aplicación web (no un sitio web), es más exitoso si se lo toma como una aplicación de escritorio. Sin embargo, el diseñador debe entender claramente las limitaciones del browser y tenerlas en cuenta en el proceso. Con esto en mente, a continuación se exponen algunas guías a tener en cuenta:

- **Diferenciación entre web site y aplicación**
  Una aplicación web por lo general es lanzada desde el website de la empresa. Para

distinguir la aplicación del website, la transición de uno a otra debe ser clara. El usuario tiene que sentir que no está más en la web.

- **Abrir la aplicación en pantalla completa**
  El hecho de abrir la aplicación en pantalla completa hace que el usuario sienta que está en una aplicación de escritorio. Pantalla completa no significa que hay que tapar los elementos del sistema operativo, sino que sea una nueva ventana y totalmente maximizada.
- **Esconder los menúes y controles del browser**
  El usuario percibe el browser como una ventana a la web. En el contexto de una aplicación el objetivo es que el usuario vea la ventana como una aplicación independiente del browser.
- **No usar múltiples tabs o ventanas del browser**
  La recomendación es usar una única ventana de manera tal que el usuario no se pierda entre ventanas abiertas de la misma aplicación.
- **Utilizar elementos web conocidos**
  Los usuarios utilizan el conocimiento adquirido en la web. Es bueno construir sobre esa base de ese conocimiento. Hacer uso de elementos web como el hyperlink, áreas clickeables, etc. es una buena opción para la adopción.
- **Utilizar tamaños constantes en fuentes, tablas y otros elementos visuales**
  Para hacer consistente el uso de la aplicación desde diferentes máquinas y diferentes dimensiones.
- **Atajos de teclado**
  Permitir ejecutar acciones con atajos de teclado para usuarios expertos.
- **Utilizar overlays**
  El *overlay* es un efecto que simula un popup pero sin abrir otra ventana. Se maneja como un panel y se muestra en la página con una transparencia que deja ver lo que hay atrás. Esto se puede usar para funciones como *Login* o *Help* para que el usuario sienta "que sus datos no desaparecieron".
- **Animaciones y tiempo de carga**
  Informar al usuario el progreso mientras se cargan los datos.
- **Limitar el uso de animaciones/arte**
  El arte puede ser placentero a los ojos, pero en la mayoría de los casos distrae al usuario de la aplicación. El diseño visual es importante pero en una aplicación lo más importante es la funcionalidad. El diseño debe estar influenciado por la funcionalidad.

Existe un conjunto de tecnologías para desarrollar aplicaciones web, nombradas a continuación:

- AJAX (Asyncronous Javascript and XML)
  AJAX es la combinación de varias tecnologías existentes, incluyendo Javascript, XML, XSLT y DOM. Todas son parte intrínseca del browser y ayudan a una experiencia de usuario más rica, evitando refresco de página y brindando más interactividad. Las aplicaciones más conocidas que utilizan AJAX son GMail y Google Maps.
- Flex/Flash/Silverlight
  Flash está generalmente asociado a diseño y no a la ingeniería del software. Sin embargo, a lo largo de los años, Adobe expandió las capacidades de Flash. Un ejemplo de esto es Flex. Por otro lado Microsoft hace poco comenzó a ofrecer Silverlight como parte de su plataforma para atacar este mismo segmento. Estas tecnologías utilizan el poder de la PC de escritorio y permiten visualizaciones e interacciones que serían más complejas de hacer en HTML y AJAX
- REST
  REST en el contexto de aplicaciones web significa URLs con un significado semántico e intuitivo para el usuario. Por ejemplo, si el usuario quiere ver la lista de clientes la url seria "/clientes/lista". Si quiere ver uno de los clientes la URL seria "/clientes/3201" (el identificador único del cliente). Del lado de la ingeniería de la aplicación, REST propone el uso de caching, intrínseco del protocolo HTTP.

#### *7.4.5.2 Experiencia de Escritorio y Offline*

La utilización de tecnologías para aplicaciones de escritorio puede darle al usuario una experiencia diferente a la que encuentra en la web. La versión escritorio de una aplicación *Software como Servicio* tiene que entenderse simplemente como un front end diferente que reutilice los mismos servicios que usa la aplicación web. Este tipo de aplicaciones se conoce como *Smart Client*, un intermedio entre *Rich Client* o cliente-servidor y *Thin Client* o web browser.

Existen ejemplos de aplicaciones Smart Client en el modelo de *Software como Servicio*. Microsoft Office Live Meeting, WebEx y otros menos conocidos como Entellium que ofrece un CRM con un cliente de escritorio. Entellium analizó alguno de los problemas que tienen los usuarios en aplicaciones similares ofrecidas por la web [Entellium]:

- Los usuarios se mueven más rápido que el tiempo de respuesta que brinda la aplicación CRM online
- Los usuarios se tienen que mover entre múltiples tabs y pantallas (fatiga de "click")

- Los usuarios no tienen acceso a los datos cuando están offline (o tienen que pagar extra y tienen que decidir de antemano con que registros quieren trabajar)

Existen diferentes tecnologías y patrones para implementar la experiencia de escritorio y offline. Plataformas como Windows Presentation Foundation de Microsoft, habilitan nuevos formatos de visualización y animación que no existían en aplicaciones de escritorio antes.

El siguiente diagrama muestra la sincronización de datos entre el cliente y el proveedor. Este proceso puede ocurrir cuando se inicia la aplicación. Se chequeua el repositorio local y se lo compara con los datos del servicio del proveedor. Para implementar esto se puede usar Microsoft Syncronization Framework y el protocolo FeedSync [Microsoft].

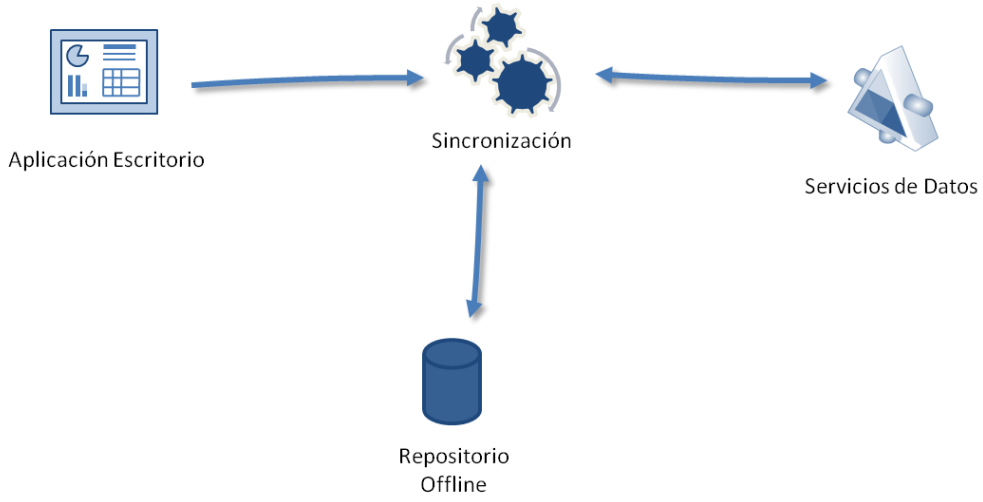



Figura 67 - Sincronización de datos en aplicación Smart Client

Durante la operación de la aplicación se realizarán llamadas a los servicios del proveedor. La aplicación Smart Cliente debe tener un mecanismo para detectar la conexión y en caso de no haber, guardar el mensaje saliente (similar a lo que hace Outlook cuando no tiene conexión con el Outbox). Para eso se necesita una cola que pueda ser persisitida en un repositorio local. Este repositorio puede ser desde un file system hasta una base de datos liviana como Microsoft SQL Compact Edition, SQLite [SQLite], etc.

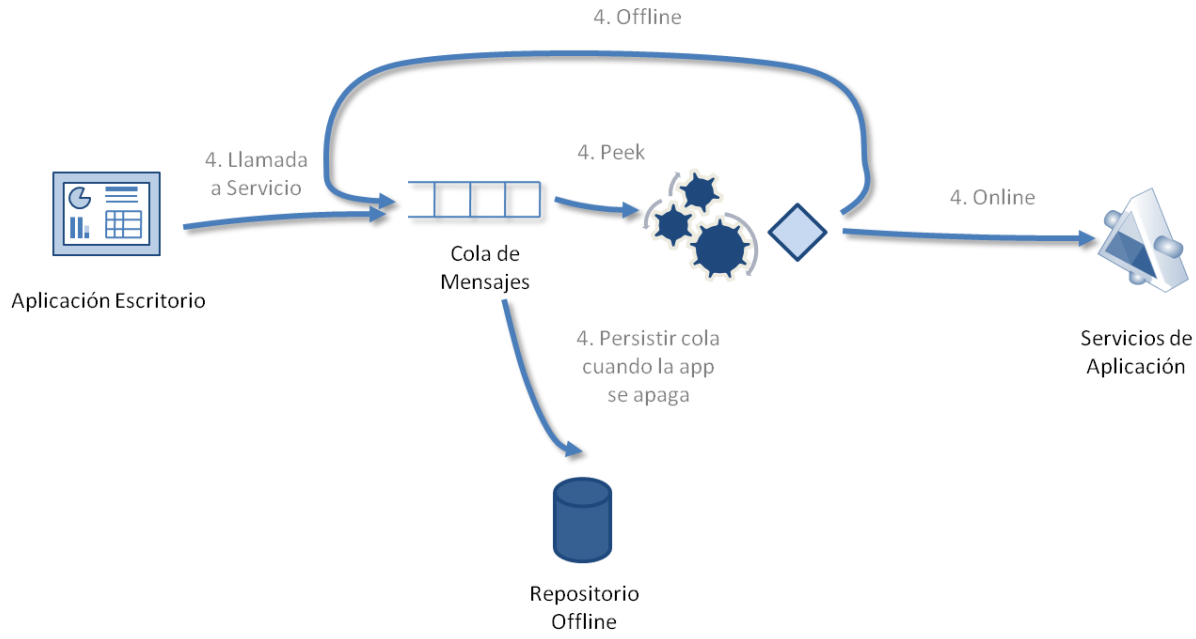



Figura 68 - Mecanismo de envío de mensajes en una aplicación de escritorio con soporte offline

Recientemente Adobe lanzó Adobe Air y Google sacó Google Gears. Estas tecnologías permiten implementar soporte offline pero en aplicaciones que son basadas en web.

# 8 Aplicación del modelo

A continuación se ensayará un ejemplo práctico que pondrá a prueba el modelo con una aplicación de ejemplo *LitwareHr* (http://www.codeplex.com/LitwareHr). *LitwareHr* es una aplicación de referencia de *Software como Servicio* creada por Microsoft. Se trata de una aplicación de reclutamiento en línea, que permite abrir posiciones, aplicar para un trabajo y hacer el seguimiento del proceso de reclutamiento. La siguiente figura ilustra el modelo de características a nivel de capacidades de esta aplicación.

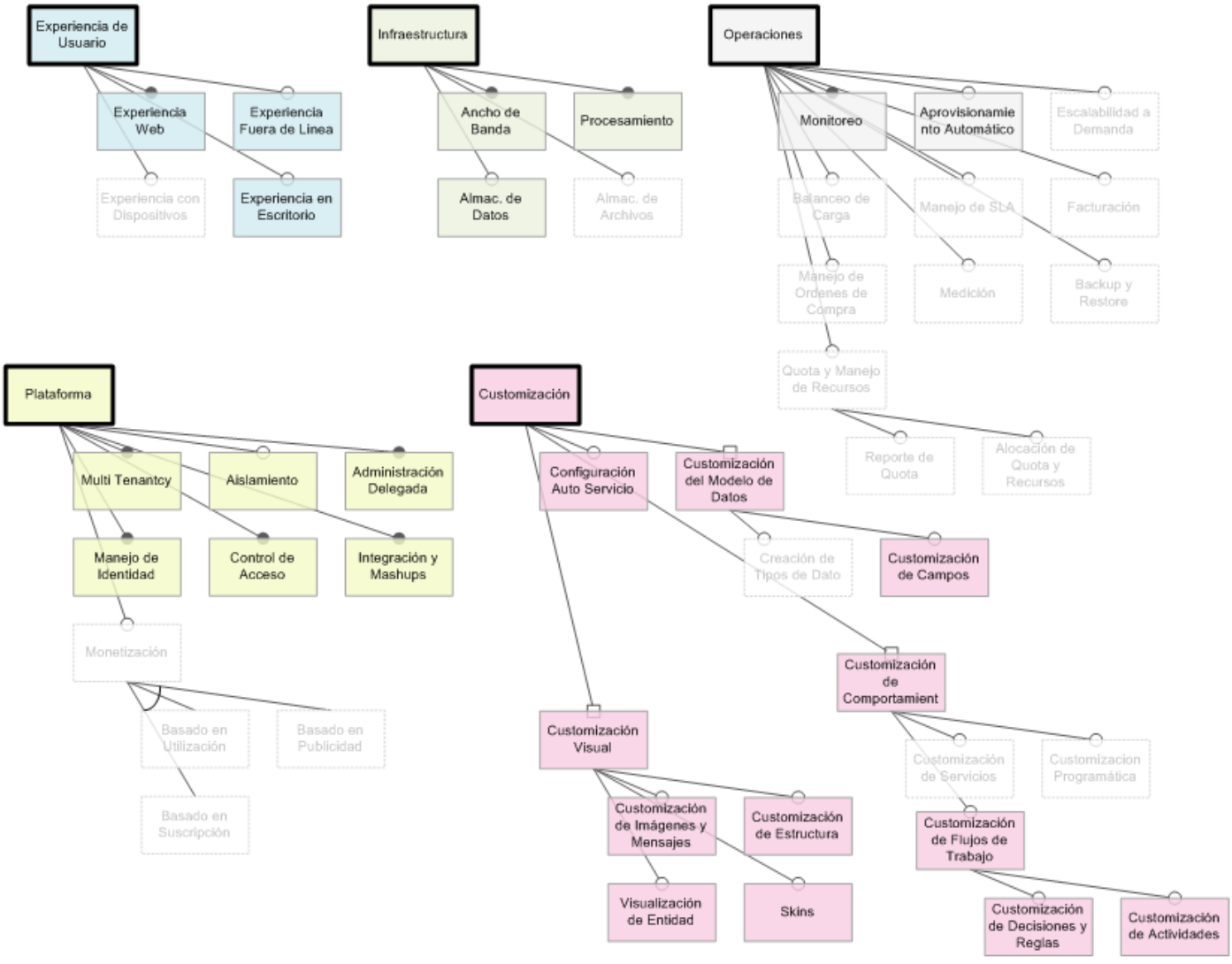



**Figura 69 - Modelo de características de una aplicación de ejemplo (LitwareHr). Las capacidades griseadas son aquellas que no son provistas por la aplicación**

El modelo de esta aplicación indica que existe un fuerte componente de customización y experiencia de usuario. Por otro lado, el aspecto operacional no está del todo explotado y la

aplicación cuenta con características de plataforma como multi tenancy, aislamiento, integración, dejando de lado aspectos como monetización.

Las características de implementación se detallan a continuación utilizando una notación textual para el modelo de características, en lugar de visual, para favorecer la claridad. El texto gris claro significa que esa característica no es parte de la aplicación.

```
Característica: Infraestructura
Descripción: La aplicación utiliza una infraestructura hibrida. El servidor web y
de aplicación es desplegado on premise utilizando virtualización. La base de
datos, en cambio, es utilizada mediante un servicio (Microsoft SQL Server Data
Services). La decisión de utilizar la base de datos como servicio se basa en dos
aspectos: la necesidad de escalar a medida que aumente la cantidad de tenants y la
flexibilidad que provee para almacenar entidades flexibles (ver característica de
Customización).
Tipo: Implementación
Componentes:
   Base de Datos: Proveedor, Microsoft SQL Server Data Services
   Procesamiento: Manejada, Virtualización
```

```
Característica: Plataforma
Descripción: La aplicación utiliza un único código base y un repositorio de
metadata para almacenar la variabilidad de cada tenant. En cuanto al aislamiento,
se provee una única instancia por tenant utilizando directorios virtuales que
brindan un Application Domain para cada uno. La administración delegada se logra
mediante una aplicación que permite configurar diferentes aspectos de la
aplicación al tenant (ver Customización). Los tenants pueden utilizar los
servicios web (SOAP y RSS) como estrategia de integración. Se utiliza un Security
Token Service para el manejo de identidad y el control de acceso en la capa de
servicios. En la capa de presentación se utilizan roles.
Tipo: Implementación
Componentes:
   Multi Tenancy: Codigo Base Unico, Metadata
   Aislamiento: Unica Instancia, AppDomain por tenant con Virtual Directory
   Administración Delegada: Aplicación
   Integración y Mashups: RSS, Web Services, SOAP
   Manejo de Identidad:
        Control de Acceso (Presentación): Role Based
        Control de Acceso (Servicios): Claim Based
        Almacenamiento (Usuarios): Servicio de Directorio
        Almacenamiento (Roles y Permisos): Base de Datos
        Federación: Security Token Service Custom
```

```
Característica: Operaciones
Descripción: La aplicación tiene sólo algunos componentes para la operación. Se
utilizan técnicas de DFO, como contadores de rendimiento a nivel de servicio y
manejo de excepciones a nivel de servicio e interfaz de usuario. La capa de
servicios se implementa con Windows Communication Foundation que permite la
intercepción de los mensajes para realizar potenciales mediciones y cobrar por
utilización.
Tipo: Implementación
```

**Componentes**:
**Monitoreo**:
Diseño para Operaciones (DFO): Contadores de Rendimiento, Manejo de Excepciones, Logging y Tracing
**Medición y Facturación**:
Intercepción: Pipe & Filters (Windows Communication Foundation)

**Característica**: Customización
**Descripción**: Se implementa customización en todos los niveles. Para comportamiento se utiliza Windows Workflow Foundation y para los datos de utiliza entidades flexibles Xml con el motor Microsoft SQL Server Data Services. En cuanto a la customización visual se implementan plantillas con ASP.NET Master Pages y estilos con CSS.
**Tipo**: Implementación
**Componentes**:
**Comportamiento**:
Motor de Reglas: Windows Workflow Foundation Rules Eng
Motor de Workflow: Windows Workflow Foundation Rules Eng
**Data Model Extension Patterns**: Patrón Repository
Cloud Database: Entidad Flexible (XML)
**Customización Visual**:
Plantillas: ASP.NET MasterPage
Estilos: CSS

**Característica**: Experiencia de Usuario
**Descripción**: La aplicación cuenta con dos clientes, web y smart client. El cliente web es utilizado para crear nuevas posiciones, monitorear los procesos de reclutamiento y customización de estos. El cliente smart client permite el acceso fuera de línea utilizando Microsoft SQL CE para almacenar los datos y cola de mensajes.
**Tipo**: Implementación
**Componentes**:
**Experiencia en Escritorio**: Smart Client, WPF, Win Forms, Otros
**Experiencia Web**: AJAX, Model View Presenter
**Experiencia Fuera de Linea**: Smart Client, Microsoft SQL CE

Esta aplicación se encuentra disponible en línea con su código fuente en

http://www.codeplex.com/LitwareHr.

# 9 Conclusiones

## 9.1 Sobre la adopción de *Software como Servicio*

Según las encuestas realizadas por importantes consultoras como Gartner, Forrester y McKensey (ver 5.4 Adopción y Difusión), *Software como Servicio* es un modelo de distribución de software que llegó para quedarse. Con un promedio actual del 30% de adopción por parte de las empresas pequeñas, medianas y grandes de acuerdo a los diferentes reportes de las consultoras mencionadas anteriormente y un mercado con un 50% de proveedores en un nivel de madurez medio alto y un 50% en niveles embriónicos y emergentes, la tecnología está atravesando la barrera para llegar a ser adoptado por el mainstream. Las empresas que adoptan, elijen aplicaciones que no son de misión crítica para su negocio pero que tienen un valor agregado y en donde los requerimientos de integración no son complejos.

Dentro de las preocupaciones que enuncian las empresas que adoptarían, se menciona integración y seguridad como los impedimentos para hacerlo. Todavía queda mucho por recorrer para lograr que la nube comience a ser parte de la infraestructura tecnológica de una empresa, sin embargo las empresas que brindan plataforma más importante comenzaron a incluir a la nube y el concepto de *Software como Servicio* como un ciudadano de primera clase en su estrategia.

## 9.2 Sobre el modelo de características planteado

A partir del problema planteado (la falta de un cuerpo de conocimiento sobre las características de *Software como Servicio*) se llegó a un modelo de características basado en *Feature Modeling*, que engloba las capacidades típicas de un proyecto de *Software como Servicio*. *Feature Modeling* es una práctica de desarrollo de software generativo utilizada para definir familias de sistemas similares mediante el análisis de la variabilidad. El objetivo fue separar el *espacio del problema* del *espacio de solución*, de manera tal que se identifiquen las características del modelo de *Software como Servicio* a nivel conceptual y se planteen alternativas en el espacio de solución (patrones, estilos de arquitectura, sub sistemas y tecnologías). Para identificar las características se realizó un análisis del dominio en cuestión teniendo en cuenta lo que ofrece el mercado y la propia experiencia.

El valor del modelo planteado radica en:

- Proveer un lenguaje consistente para describir el problema en cuestión dejando en claro los conceptos. Cuando se habla de aislamiento de un tenant los involucrados tienen en claro el contexto en el que se está hablando, cuál es el problema y qué alternativas existen.
- Distinguir y concentrarse en las características que son importantes para la solución que se quiera realizar dejando de lado las que no lo son. Si el problema está bien modelado, los involucrados en el diseño y la implementación tienen más chances de resolver sin distraerse en problemas linderos.
- Sirve como guía para organizar y tomar decisiones y dejarlas registradas.
- Alienta la reutilización. Tener un modelo ayuda a los involucrados a reutilizar efectivamente los conceptos a lo largo del ciclo de vida del proyecto.

Por último, el modelo debe ser un elemento que se integre dentro de un marco más grande en donde se incluya el análisis de los objetivos de negocios y los requerimientos funcionales.

# 10 Futuras líneas de investigación

Esta tesis abarca una gran cantidad de sub temas que están circunscriptos en el modelo de *Software como Servicio*. A continuación se detallan algunos de estos:

- **Extensión del modelo**
  La ventaja de tener un modelo es que se puede construir, extender y modificar. En los próximos años el *Software como Servicio* va a seguir evolucionando y por lo tanto el modelo también. El espacio de solución es donde más potencial de crecimiento hay. Las nuevas tecnologías, la evolución de la plataforma y la madurez en los que construyan este tipo de software, desembocará en una mayor riqueza para la capa de implementación del modelo.
- **Plataforma de *Software como Servicio* privada**
  Muchas grandes empresas poseen recursos, tanto de hardware como humanos, tal como para construir su propia plataforma de *Software como Servicio*. La idea de un departamento de IT que genere software bajo este modelo y lo comercialice a las diferentes subsidiarias podría traer beneficios a las grandes corporaciones que generalmente no reutilizan el software entre subsidiarias, hasta incluso, de un mismo país. Esta es una línea de investigación abierta en la cual se puede analizar la especialización del modelo para "Intra SaaS" y hacer un estudio de retorno de inversión en construir una plataforma de estas características.
- **Domain Specific Languages para la variabilidad por tenant**
  La utilización de DSLs para customizar la lógica y reglas de negocios es un campo interesante para analizar en profundidad. La naturaleza y expresividad que se puede alcanzar abriría nuevos horizontes y un valor agregado para el tenant.
- **Otros tipos de almacenamiento**
  La customización del modelo de datos es otro de los desafíos mencionados y analizados en esta tesis. Si bien se trataron los repositorios como base de datos relacionales y hash tables distribuidas, quedaron por analizar los triplestore utilizados en el campo de la semántica, las bases de datos orientadas a objetos y otros repositorios híbridos como Microsoft Research Output Platform. Relacionado con este tema, la performance y escalabilidad de cada una de estas opciones sería un análisis importante.
- **Infraestructura y plataforma "on the cloud" en el contexto corporativo**
  Cloud Computing es un concepto todavía más extenso que *Software como Servicio*. Es la idea de incluir a la web como parte de la infraestructura de delivery de una empresa. Un

análisis más exhaustivo se puede realizar sobre las ventajas y desventajas de utilizar recursos de la web en el contexto corporativo.

- **Seguridad y manejo de identidad**

  El manejo de identidad en el contexto de aplicaciones de *Software como Servicio* es un tema que todavía está poco explorado. En esta tesis se propuso la implementación de identidad federada utilizando security token services y los estándares de WS-Trust y SAML, sin embargo hay pocos casos de aplicación en el mundo real. Posiblemente por la poca información y la dificultad que estos mecanismos representan. Es interesante notar que esta es una de las mayores preocupaciones de las empresas a la hora de adoptar SaaS ya que no desean otro silo de identidad en la organización.

- **Costo de marketing y ventas**

  El modelo de suscripción parece ser una ventaja competitiva a la hora de vender *Software como Servicio*. Sin embargo, las compañías que construyen y comercializan SaaS están teniendo problemas de cashflow por el costo que implica conseguir un nuevo cliente (para salesforce.com esto representa el 50% de sus ingresos). El análisis de por qué esto sucede y qué alternativas de solución existen es otra línea de investigación.

# 11 Bibliografía

*A research manifesto for services science.* **Henry Chesbrough, Jim Spohrer. 2006.** s.l. : Communications of the ACM, 2006.

*Agility in Virtualized Utility Computing.* **Qian, Hangwei, y otros. 2007.** Reno, NV USA : Proceedings of the Second International Workshop on Virtualization Technology in Distributed Computing, 2007.

**Amazon.com. 2006.** Success Story: SmugMug. *Amazon Web Services @ Amazon.com.* [En línea] 2006. http://www.amazon.com/b?ie=UTF8&node=206910011.

**Bhavini, Desai y Currie, Wendy. 2003.** *Application Service Providers: A model in Evolution.* Pittsburgh, PA : ACM, 2003. Paper.

**Carraro, Gianpaolo. 2007.** Cost per feature vs. cost per tenant (or how to choose whether to go multi-tenant or not). *Gianpaolo's Blog.* [En línea] 25 de Enero de 2007. http://blogs.msdn.com/gianpaolo/archive/2007/01/25/cost-per-feature-vs-cost-per-tenant-or-how-to-choose-whether-to-go-multi-tenant-or-not.aspx.

**Carraro, Gianpaolo y Pace, Eugenio. 2007.** Gianpaolo's Blog - Build - Run - Consume - Monetize. *Gianpaolo's Blog.* [En línea] 21 de 05 de 2007. http://blogs.msdn.com/gianpaolo/archive/2007/05/21/build-run-consume-monetize.aspx.

**Chong, Frederik, y otros. 2007.** ISVs are from Mars, and Hosters are from Venus. *MSDN Architecture.* [En línea] Noviembre de 2007. http://msdn.microsoft.com/en-us/architecture/bb891759.aspx.

*Concepts and Guidelines of Feature Modeling for Product Line Software Engineering.* **Lee, Kwanwoo, Kyang, Kyo C. y Lee, Jaejoon. 2002.** London : Springer-Verlag, 2002.

**Entellium.** Entellium eDesktop Unlocks User Effectiveness for OnDemand CRM. [En línea] http://www.entellium.com/press/Entellium_Desktop_Release.pdf.

**Factor, Alexander. 2001.** *Analyzing Application Service Provider.* s.l. : Prentice Hall PTR, 2001. 0130894257.

**Fielding, Roy Thomas. 2000.** *Architectural Styles and the Design of Network-based Software Architectures.* Irvine : University of California, 2000.

*FORM: A Feature-Oriented Reuse Method with Domain Specific Reference Architectures.* **Kang, Kyo, y otros. 1998.** Victoria : J. C. Baltzer AG, Science Publishers, 1998. págs. 354-355. 1022-7091.

**Forrester Research. 2008.** *Competing in the Fast-Growing SaaS Market.* 2008.

**—. 2008.** *The State of Enterprise Software Adoption: 2007 to 2008.* s.l. : Forrester Research, 2008.

**—. 2008.** *What Evolving SaaS Options Mean for Buyers.* s.l. : Forrester Research, 2008.

**Gartner Inc. 2008.** *Hype Cycle for Software as a Service 2008.* s.l. : Gartner Inc., 2008.

**—. 2008.** *Predicts 2008: SaaS Gathers Momentum and Impact.* s.l. : Gartner Inc., 2008.

**Giurata, Paul. 2008.** Application Strategy and Design for a Profitable SaaS. *ebiz.* [En línea] 16 de Abril de 2008. http://www.ebizq.net/hot_topics/web20/features/9365.html.

**Google Inc. 2006.** Bigtable: A Distributed Storage System for Structured Data. *Google Labs.* [En línea] Noviembre de 2006. http://labs.google.com/papers/bigtable.html.

**Gottfrid, Derek. 2007.** Self-service, Prorated Super Computing Fun! *The New York Times Company.* [En línea] 1 de Noviembre de 2007. http://open.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/.

**Greschler, David y Mangan, Tim. 2002.** *Networking lessons in delivering 'Software as a Service' - Part I.* 2002. págs. 317-321.

**Hoggvliet, M.T. 2008.** *SaaS Interface Design.* Netherlands : Rotterdam University, 2008.

**IDC. 2008.** *IDC's Worldwide Software as a Service Taxonomy, 2008.* s.l. : IDC, 2008.

**Kang K., Cohen S., Hess J., Nowak W., Peterson S. 1990.** *Feature Oriented Domain Analysis (FODA) feasibility study.* Pittsburgh : Software Engineering Institute, Carnegie Mellon Univerity, 1990.

**Lacy, Sarah. 2008.** On-Demand Computing: A Brutal Slog. *BusinessWeek.* [En línea] 18 de Julio de 2008. http://www.businessweek.com/technology/content/jul2008/tc20080717_362776.htm.

**Microsoft Corp. 2008.** Software + Services: Bring it all together. *Microsoft.* [En línea] Microsoft Corp., 2008. http://www.microsoft.com/softwareplusservices/.

**Microsoft.** FeedSync. *Windows Live Dev.* [En línea] http://dev.live.com/feedsync/.

**Microsoft Patterns & Practices. 2008.** *Design for Operations, Designing Manageable Applications.* 2008.

**Microsoft.** Windows Workflow Foundation. *Microsoft MSDN.* [En línea] http://msdn.microsoft.com/en-us/netframework/aa663328.aspx.

**Moore, Geoffrey A. 1991.** *Crossing the Chasm.* s.l. : Harper Business Essential, 1991.

**Nathan. 2008.** SimpleStateMachine. *Codeplex.* [En línea] 2008. http://www.codeplex.com/SimpleStateMachine.

**Nielsen Online. 2008.** US Broadband Penetration Breaks 90% among Active Internet Users - July 2008 Bandwidth Report. *WebSiteOptimization.com.* [En línea] 23 de Julio de 2008. http://www.websiteoptimization.com/bw/0807/.

**O'Reilly, Tim. 2006.** REST vs SOAP at Amazon. *xml.com.* [En línea] 2006. http://www.oreillynet.com/pub/wlg/3005.

**—. 2005.** What Is Web 2.0. *O'Reilly.* [En línea] 9 de September de 2005. http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html.

*Overview of Generative Software Development.* **Czarnecki, Krzysztof. 2005.** Mont Saint-Michel : s.n., 2005. Unconventional Programming Paradigms (UPP) 2004. págs. 313-328.

**Ozzie, Ray. 2005.** The Internet Services Disruption. [En línea] 28 de Octubre de 2005. http://www.scripting.com/disruption/ozzie/TheInternetServicesDisruptio.htm.

**Pace, Eugenio, Carraro, Gianpaolo y Woloski, Matias. 2007.** Multitenant Database Performance Guide. *Codeplex LitwareHr.* [En línea] Noviembre de 2007. http://www.codeplex.com/LitwareHR/Release/ProjectReleases.aspx?ReleaseId=8440.

**ProgrammableWeb. 2006.** Salesforce.com Launches ApexConnect. *ProgrammableWeb.* [En línea] 27 de November de 2006. http://blog.programmableweb.com/2006/11/27/salesforcecom-launches-apexconnect/.

**Rogers, Everett. 2003.** *Diffusion of Innovations, Fifth Edition.* New York : Free Press, 2003.

**Salesforce.com.** Motorola Optimizes its CRM System with Enterprisewide Salesforce Implementation. *Salesforce.com.* [En línea] http://www.salesforce.com/customers/hi-tech-hardware/symboltechnologies.jsp.

**Saugatuck Technology. 2008.** *Saugatuck's 2008 SaaS research .* s.l. : Saugatuck Technology, 2008.

**Seeking Alpha. 2008.** Salesforce.com F2Q09 (Qtr End 7/31/08) Earnings Call Transcript. *Seeking Alpha.* [En línea] 20 de Agosto de 2008. http://seekingalpha.com/article/91899-salesforce-com-f2q09-qtr-end-7-31-08-earnings-call-transcript?page=1.

**SQLite.** SQLite Home Page. *SQLite.* [En línea] http://www.sqlite.org/.

**Weissman, Craig y Wong, Simon. 2005.** *Custom entities and fields in a multi-tenant database system. WO/2005/098593* United States of America, 20 de Octubre de 2005. Software.