Arquitectura de Proyectos de IT

Apunte:

Arquitecturas de Integración

Autores:

Santiago M. Blanco

santiago.blanco@gmail.com

Versión: 1.0

Mayo, 2011

1.Integración de sistemas

Comenzaré el texto comentando sobre la importancia de tener en cuenta ciertos aspectos en la integración de sistemas, más que nada en áreas de arquitectura empresarial (aunque la integración ya no se aplica solo a empresas, sino que cualquier desarrollo de SW debe ser capaz de integrarse e interactuar con otros).

En primer lugar, como dije anteriormente, hoy en día ya es virtualmente imposible que un sistema funcione totalmente aislado. El entorno complejo en el que nos encontramos hace que un desarrollo, para poder cumplir con la funcionalidad requerida, depende de otros aplicativos (APIs de herramientas de redes sociales, servicios expuestos por otros sistemas de la empresa), lo que genera la necesidad de brindar soluciones complejas, formada por sub-soluciones software separadas que forman un conjunto en base a integración.

Por otro lado, hay que tener en cuenta que la complejidad de las soluciones y desarrollos, sumado a ciertas estrategias de vendors, y la problemática de negocio cada vez más intrincada, hace que una se desarrollen soluciones de SW "de nicho" especializados (CRM, Sales, Billing, Gestor de campañas, Inventario y logística), estrategia encontrada con lo que se veía hace unos años atrás, con el surgimiento de las grandes moles ERP. Esta tendencia, por un lado hace que las soluciones se encuentren enfocadas en resolver problemáticas puntuales y específicas de una manera muchas veces mejor que las soluciones genéricas, pero por otro lado hace necesario unir estas soluciones separadas, para poder lograr una coherencia funcional entre todas ellas.

La última oración me lleva al último punto, el hecho de que las soluciones a las problemáticas de negocio en las empresas no están dadas en su mayoría por soluciones separadas e aisladas, sino que el negocio funciona en base a una serie de sistemas coexistiendo en forma colaborativa.

Entonces, la integración de sistemas me gusta decir que son "Las decisiones de arquitectura que se toman durante el ciclo de desarrollo de un software, para permitir la coexistencia y colaboración con otras soluciones de SW, teniendo en cuenta atributos de calidad definidos para la solución, sin perder en cuenta cuestiones como independencia funcional, o coherencia semántica".

Cuando hablamos de integración de sistemas, o de la definición de los mecanismos mediante los cuales diferentes soluciones SW se integran para poder cubrir en conjunto los requerimientos de negocio, es necesario tener en cuenta atributos de calidad que guíen dichas integraciones, junto a los requerimientos funcionales que fomentan el desarrollo.

2. Atributos de calidad a tener en cuenta

Estos atributos de calidad, a modo ilustrativo, pueden dividirse de la siguiente manera:

2.1. Calidades asociadas al diseño

Estas calidades están relacionadas a las tareas de definiciones de interfaces o conexiones entre sistemas, sumado al diseño de los protocolos, mecanismos y formatos de interacción. Si bien no impactan en forma directa en el funcionamiento de las interfaces, es muy importante tener en cuenta estos aspectos para hacer más feliz la vida de aquellas personas involucradas en el desarrollo.

2.1.1. Coherencia semántica

La coherencia semántica es mi atributo de calidad preferido. Cuando hablamos de integración, la coherencia semántica se refiere a la separación de incumbencias entre los diferentes sistemas. Se debe tener en cuenta que la información expuesta sea de importancia para los consumidores de la misma, pero sin dar a conocer información técnica de implementación. Así, por ejemplo, dar a conocer un id interno de una base de datos, si este no tiene ningún significado de negocio, es una mala práctica.

2.1.2. Unificación de mensajes

Es preciso que buscar que los protocolos definidos para las integraciones sigan una regla coherente para definir sus protocolos de integración. Si bien parece más un atributo que aumenta la reusabilidad, es bueno tenerlo en cuenta por separado, para que lo tengamos siempre presente. Es muy incómodo ver, en un sistema, dos integraciones distintas que hacen referencia a un nombre de dos maneras diferentes, uno como "customer_name", y otro como "nombre_cliente". Definir estándares hace más feliz la vida de los involucrados en la integración.

2.1.3. Reusabilidad

Es necesario que pensar las integraciones de forma tal que puedan utilizarse para cubrir diferentes problemáticas, y que no sean aplicables a un solo caso en particular. Para lograr esto, es imprescindible pensar a las integraciones en pos de que las mismas tengan un sentido de negocio, y que una interfaz no dependa funcionalmente de otra. Por ejemplo, si tenemos dos integraciones, una denominada "cálculo de importe" que tenga como parámetro el importe básico y los impuestos, y otro "cálculo de impuestos", que también tenga como parámetro el importe básico, pero además le sumamos la categoría impositiva del cliente, es una mala práctica hacer que el consumidor de la funcionalidad deba orquestar ambas integraciones (obteniendo el impuesto de "cálculo de impuestos", para luego pasarlo a "cálculo de importe". Sería mejor definir una única integración, llamada "cálculo de importe", con parámetros importe básico y categoría impositiva, que internamente obtenga el importe y calcule el total.

2.2. Calidades asociadas al tiempo de ejecución

Diferentes cuestiones o decisiones que tomemos van a impactar en forma perjudicial o beneficiosa en el tiempo de ejecución de los servicios. A continuación voy a describir cada una de las calidades relacionadas con las integraciones.

2.2.1. Disponibilidad

La disponibilidad se define como la proporción de tiempo que un elemento SW se encuentra funcionando y trabajando. Es necesario conocer los requerimientos de disponibilidad de cada uno de las integraciones, y no aplicar una única estrategia a todo el sistema en su conjunto. Me gustaría que quede claro, en este punto, que no es necesaria siempre una disponibilidad del 100%, y que no todos los consumidores de las integraciones requieren la misma disponibilidad.

Recordemos que a mayor disponibilidad, mayor costo y complejidad. No sumemos complejidad sin una necesidad real.

2.2.2. Interoperabilidad

Es vital tener en cuenta la necesidad de interoperabilidad que tendrá el sistema en cuestión para poder pensar en las cuestiones correctas. Se llama interoperabilidad a la capacidad que tiene uno o más sistemas para intercomunicarse entre sí, de forma tal que sea simple reemplazar un componente por otro, o modificar un sistema internamente sin impactar en el resto de los participantes de la integración. Además, está relacionado con la independencia tecnológica entre diferentes sistemas. Por ejemplo, si tengo aplicaciones desarrolladas en .NET, otras en J2EE y otras en Ruby, utilizaré mecanismos de integración agnóstico a las tecnologías, mientras que si todos mis sistemas se desarrollan en una única tecnología, podrá utilizar mecanismos propietarios (convengamos que este caso es poco probable).

2.2.3. Facilidad de administración

Este es un aspecto pocas veces tenido en cuenta a la hora de desarrollar una aplicación en general, y también aplica muy bien cuando pensamos en integraciones entre sistemas. Debemos tener en cuenta los requerimientos de administración, gestión de incidentes, trazabilidad y seguimiento de las ejecuciones. Es vital en entornos distribuidos, donde los logs de las aplicaciones serán muy diferentes entre sí, y muy pocas veces nos permitirá seguir con detalle la ejecución de un flujo de integración.

2.2.4. Performance

La performance, rendimiento, o la medida que nos permite conocer el nivel de respuesta ante una petición a una integración es algo que siempre será visto como "de vital importancia", siendo el requerimiento que la respuesta "sea presentada lo más rápido posible". Lo que tenemos que saber es ¿cómo mido "lo más rápido posible"? ¿Cómo impactará el rendimiento solicitado en el resto del sistema? ¿Cuál es realmente el requerimiento de performance? Debemos tener en cuenta todo el conjunto (consumidor, proveedor, medio, entorno) a la hora de hablar de performance en integraciones, y actuar en base a las mismas.

2.2.5. Robustez

La robustez es la capacidad de mantenerse un sistema (o componente SW) en funcionamiento ante determinadas condiciones de falla o inesperadas. En el caso de las integraciones, está íntimamente relacionado con la capacidad de mantener los mensajes o pedidos hasta que sean tratados, manejar transaccionabilidad de las interacciones, o devolver alguna respuesta al sistema consumidor aunque no sea más que un mensaje de error (por ejemplo, twitter y su famosa ballena).

2.2.6. Escalabilidad

La escalabilidad es la habilidad que tiene un sistema para aumentar su capacidad de procesamiento para soportar aumentos en la carga del mismo, sin perjudicar el rendimiento de las integraciones implementadas. Para lograrlo, hay que tener en cuenta, entre otras cosas, la capacidad de aumento de HW para lograr soportar mayor carga, el soporte a multithreading y multi core, y el aprovechamiento al máximo de los recursos utilizados.

2.2.7. Seguridad

La seguridad es la capacidad de prevenir acciones que perjudiquen al sistema, ya sean estas realizadas en forma maliciosa o accidental. Un sistema seguro es aquel que previene de accesos indebidos, al mismo tiempo que evita la modificación de los activos del sistema. Este es un atributo de calidad que no se lleva del todo bien con las integraciones, ya que puede penalizar el rendimiento, complicar la mensajería, y requerir cuestiones administrativas que son un tanto tediosas. Pero es necesario y muy importante tener en cuenta estas cuestiones, más cuando las

integraciones que exponga nuestro sistema serán la puerta de acceso a la modificación de datos y valores.

2.3. Calidades del sistema

Estos son atributos de calidad que deben tenerse en cuenta para poder dar una visión completa al sistema, no solo desde el punto de vista de mejoras en el diseño, desarrollo o tiempo de ejecución, sino también en el resto del ciclo de vida de un desarrollo.

2.3.1. Simplicidad de test

Las integraciones, al involucrar diferentes sistemas, grupos de desarrollo y equipo de personas, además de diferentes tecnologías y desarrollos diversos, presentan mucha dificultad para ser testeadas. Es necesario tener en cuenta los requerimientos de creación de escenarios de prueba y criterios de aceptación, y facilitar la ejecución de los mismos para comprobar la calidad del desarrollo.

2.4. Calidades de comunicación

Estamos hablando de integración entre sistemas, OK. La experiencia me dice que cuando pensamos en este tipo de integraciones, nos olvidamos que las mismas son desarrolladas por personas, y si las personas no pueden comunicarse, integrarse y trabajar en conjunto, estos desarrollos no llegan a ninguna parte. Es vital la capacidad de comunicación en estos equipos de trabajo, que por otro lado son multidisciplinarios, y con diferentes intereses. Estas calidades no impactan directamente en el desarrollo, ni en la ejecución, pero tenerlas en cuenta realmente nos hace la vida más feliz.

2.4.1. Utilidad para comunicación al usuario final

Al hablar de integraciones, entendemos muchas veces que no es necesario tener en cuenta al usuario final, ya que todo lo relacionado con integrar sistemas pasa "entre bambalinas", y el no debería enterarse, en el mejor de los casos. En la práctica, este escenario feliz no pasa, y muchas veces debemos dar explicaciones sobre porque el sistema de venta vende pero el CRM no se entera de las nuevas altas, o simplemente mostrar la complejidad del cambio que nos están pidiendo, al impactar en muchos sistemas de nuestra arquitectura. En estos casos, debemos pensar las integraciones (o solo los nombres) de manera tal que sea simple comunicarlas a usuarios no técnicos. Tal vez solo baste con tener una planilla donde se listen todas las interfaces y sus nombres funcionales, aunque otras veces es algo más complejo.

2.4.2. Utilidad para documentación

Que mejor que tener un desarrollo que se auto-documente. Eso nos ahorra mucho trabajo, y encima es mejor cuando este trabajo es algo que no es muy feliz hacer. Si la metodología de desarrollo que estamos utilizando no requiere muchísima documentación innecesaria, es muy útil pensar en que los diseños o desarrollos de las integraciones se auto-documenten (por ejemplo, utilizando WSDL o REST para definir los protocolos).

2.5. Notas finales

En este apartado intente explayarme sobre aquellos atributos de calidad con los que me topé en mi experiencia al desarrollar integraciones. No quiere decir que todos existan en todos los casos, ni que debamos aplicar todos a todos los desarrollos, ni siquiera que sean los únicos. Pero creo que sirve como guía para comenzar, o ayuda-memoria para no olvidarnos de cosas que creo importantes.

3.Mecanismos de integración

En esta sección voy a comentar, desde mi punto de vista y experiencia, los diferentes mecanismos que existen para resolver esta cuestión, y cuál es la relación entre estos y los atributos de calidad anteriormente mencionados.

Antes de empezar, voy a dividir los mecanismos, solo por claridad en la comunicación, entre mecanismos puntuales y masivos. Los primeros son aquellos mecanismos que permiten intercambiar información entre un sistema y otro en forma puntual, digamos de una interacción a la vez (por ejemplo, la solicitud del alta de un cliente, o la consulta de tweets ingresados por un usuario el último mes). Los segundos permiten la integración entre dos sistemas intercambiando grandes volúmenes de información.

Los primeros generalmente son utilizados para interacciones online, mientras que los segundos son utilizados para integración batch. Los primeros son generalmente sincrónicos (o el usuario los ve de esa manera), mientras los segundos son naturalmente asincrónicos.

Ahora, una vez establecida esta división, transcribiré en resumidas cuentas cada uno de los mecanismos.

3.1. Mecanismos puntuales

3.1.1.Web Services

Los web services son un mecanismo de integración basado en una colección muy amplia de protocolos y estándares, basados todos sobre los mismos estándares sobre los que funciona la web (aunque existen implementaciones de los mismos sobre tecnologías de mensajería). El principal protocolo y estándar en el que se basa es SOAP (Simple Object Access Protocol), que define el formato en que se intercambia la información, dividiéndolo básicamente entre header (cabecera que permite configurar el transporte), y el body (que contiene los datos a transmitir). Otro estándar importante es WSDL (Web Service Description Language), que define la estructura del mensaje, estableciendo los tipos de datos, campos y estructura de información a transmitir, sumado a UDDI (Universal Description, Discovery and Integration) para el registro de servicios web.

Con el paso del tiempo, se sumaron otros estándares al stack, para cubrir diferentes funcionalidades o atributos de calidad. De esta forma, tenemos por ejemplo WS-Security, WS-Reliablemessaging, o WS-transaction, que cubrieron aspectos que fueron dejados de lado por la definición inicial de SOAP, aunque agregándole complejidad.

Desde mi punto de vista, es una buena alternativa para interoperabilidad en ambientes heterogéneos, siempre y cuando nos basemos en el estándar SOAP sin ningún adicional del stack WS-*, ya que estos últimos no fueron implementado de la misma forma por todos los vendors. Aunque, si los sistemas a integrar son desarrollados en la misma tecnología y vendor tecnológico, el uso del stack WS-* soluciona diferentes cuestiones que son necesarias tener en cuenta.

Con respecto a la relación con los atributos de calidad, si se aplican bien los conceptos de WS, este mecanismo de integración se encuentra muy a favor de la coherencia semántica, unificación de mensajes y reusabilidad (heredado de la orientación a servicios). En lo que respecta a interoperabilidad, esto es una promesa, que no siempre es bien cumplida y en diferentes ocasiones presenta sus dificultades. El rendimiento de las integraciones se ve menguado debido a lo verboso de los mensajes (se estima que del total de un mensaje WS, solo el 30% representa datos con significado funcional). En lo que respecta a robustez, escalabilidad, facilidad de administración y disponibilidad, los mismos dependerán mucho de las plataformas donde se implementen los servicios (ESB, Application Server, Web Server).

En lo que respecta a los tests, tener el contrato (WSDL) separado de la implementación permite crear servicios "mockeados" que implementen la funcionalidad necesaria para realizar los

tests, mientras que los contratos pueden verse como auto-documentados, un beneficio que da XML y el estándar WSDL a este tipo de integraciones.

3.1.1.1. Cuando utilizarlo

Debido a la carga en cuanto a performance, lo utilizaría en aquellos casos en los que se requiera seguridad, trazabilidad, y cuando la performance no sea un requerimiento fuerte. Si se requieren interfaces autodocumentadas, es la opción por la que se debe ir. También es apto en aquellos desarrollos donde se requieran facilidad de pruebas.

3.1.2. REST

Este mecanismo de integración, cuyas siglas significan REpresentational State Transfer, aparece como una alternativa a los web services, para integraciones basadas en la arquitectura WEB y HTTP como transporte. Es muy utilizado por los grandes jugadores de la WEB 2.0 (Facebook, Google, Twitter y otras lo utilizan para sus APIS).

REST presenta las integraciones basado en los recursos de los sistemas (principal diferencia con respecto a los web services, que se basan en funcionalidades), basándose en la arquitectura HTTP, e implementando los siguientes principios:

- Identificación de recursos a través de URIs (Uniform Resource Identifier)
- La manipulación de los recursos se realiza a través de sus representaciones, utilizando los métodos HTTP de manera explícita (GET - consultar un recurso, POST - crear un recurso, PUT - modificar un recurso, DELETE - eliminar un recurso), y basándose en la información obtenida de la representación de los recursos.
- Mensajes auto-descriptivos, de forma tal que cada mensaje tenga la información necesaria para procesarlo (por ejemplo, estableciendo su MIME type y gestión de cache)
- La hypermedia se usa como motor de gestión de estado. Si bien es un mecanismo de integración stateless (sin estado), la interacción entre consumidor y proveedor se realiza a través de hyperlinks o hypertext presente en la representación.

Como se basa en un estándar ampliamente aceptado (HTTP), intercambiando mensajes JSON (Java Script Object Notation), XML o YAML(Yet Another Markup Language), una de sus principales ventajas son la interoperabilidad, reusabilidad, unificación de mensajes y coherencia semántica (heredados de la orientación a servicios), sumado a un gran soporte de la escalabilidad (heredado de HTTP). Representa una gran mejora en lo que respecta a performance comparado con los web services, ya que disminuye la cantidad de datos innecesarios y sin significado de negocio.

Otra vez, la robustez, facilidad de administración y disponibilidad, los mismos dependerán mucho de las plataformas donde se implementen los servicios (web server). La seguridad es una de sus falencias, ya que como mecanismo básico tiene lo soportado por HTTPS (Secure HTTP), requiriendo agregados como proxies o firewalls para mayor complejidad (por ejemplo, un appliance de HW como Datapower podría cubrir estos requerimientos).

El formato de mensaje, si bien no es tan autodocumentado como en el caso de web services, da un cierto soporte a la documentación y comunicación, siempre y cuando se establezcan criterios claros entre las partes.

3.1.2.1. Cuando utilizarlo

Este mecanismo es una muy buena alternativa a web services. Es fácil de usar cuando los recursos del sistema proveedor están claramente identificados, y no se tengan grandes requerimientos en cuanto a seguridad, trazabilidad y transaccionabilidad (ya que agregar estas características complejizarían el desarrollo). Si lo que se requiere es performance e interoperabilidad, es una excelente alternativa.

3.1.3. XML over HTTP

Este mecanismo se podría ver como un subconjunto de REST, ya que se basa en HTTP como transporte, y para la definición de los datos sensibles al negocio se establece XML para representar los mismos.

La única diferencia entre REST y este mecanismo, es que no cumple los principios establecidos por REST, ya que por ejemplo las interfaces o servicios expuestos pueden no estar representados como recursos, ni hacer uso de URIs para identificarlos. Esta diferencia es vital, y es necesario tenerla en cuenta a la hora de decidir por uno u otro.

También puede verse como una simplificación de los Web Services, ya que si bien utilizan XML para la transmisión de datos, no supone la carga de SOAP y WSDL.

Tiene, entonces, una mejor performance que los web services, pero no cuenta con la ventaja de ser autodocumentado, ya que no se requiere un estándar de definición de protocolos, y los mensajes muchas veces son definidos previamente entre las partes. La interoperabilidad estará dada, mientras se mantengan los acuerdos entre las partes, pero nada de los protocolos estándar utilizados juega en contra de esta calidad.

Otra vez, la robustez, facilidad de administración y disponibilidad, los mismos dependerán mucho de las plataformas donde se implementen los servicios (web server, application server).

La seguridad estará dada por las plataformas subyacentes, mientras que la claridad de documentación y comunicación no es una característica de este mecanismo.

3.1.3.1. Cuando utilizarlo

No recomendaría utilizarlo, ya que es una versión mala de REST. En caso de aparecer como una alternativa, evaluaría seriamente el uso de REST antes que este.

3.1.4. RSS

RSS (Really Simple Sindication), es básicamente un estándar XML para sindicar o compartir contenidos en la WEB. Su principal uso es la difusión de información a usuarios que se subscriben a una fuente de contenidos.

Su principal uso es la lectura de noticias o novedades en sitios web o blogs, mediante herramientas "agregadoras" (web browsers, Google reader, Bloglines).

La arquitectura que implementa este mecanismo es básicamente un publish-subscribe, en el cual un ente publica información, que interesa a uno o más consumidores, los cuales no son conocidos por el publicador.

Es muy útil para sistemas que requieran compartir información multimedia, o solo-texto, a diferentes aplicaciones implementadas en diferentes tecnologías, y desconocidas. No es simple realizar este tipo de integraciones, pero existen frameworks como RSS 2.0 (en C#), o RSS Framework (en PHP).

No es un mecanismo que esté acorde a la comunicación entre personas o documentación, y cuyo soporte a la performance, disponibilidad, robustez dependerán del tipo de desarrollo implementado.

Por su poca o nula políticas de seguridad, debe usarse en aquellos casos en que la información sea pública y poco sensible.

3.1.4.1. Cuando utilizarlo

Implementarlo en aquellos casos donde se quiera dar a conocer información que no requiera demasiada seguridad, y que interesa a diferentes consumidores desconocidos, distribuidos en internet.

3.1.5. LDAP

LDAP (Lightweight Direct Access Protocol), permite en su utilización más común la comunicación con un directorio de identidades, utilizando esta integración para autenticación y autorización.

Pero, por la forma de estructuración basada en una entidad principal, diferentes roles o características de dichas entidades, y dependencia entre entidades, es muy útil en algunos casos. Por ejemplo, en una empresa proveedora de internet, podría tenerse una estructura por ubicación geográfica (AMBA, NORTE, SUR), dentro de cada uno podría establecerse una división por tipo de cliente (gran cuenta, individuo), de los cuales se desprenderían los diferentes servicios que tiene activo el cliente, para su uso por la red, o los sistemas de CRM. ¿Cómo se conectarían estos sistemas con el repositorio? a través de LDAP.

Tiene un uso muy acotado a ciertas funcionalidades, pero es bueno tenerlo en cuenta para no realizar un desarrollo importante teniendo herramientas que puedan resolver la problemática. Las herramientas van desde componentes software pagos (Microsoft Active Directory, Oracle Identity Management) a aplicación gratuitas (Microsoft ADAM, open LDAP).

Es aplicable solo a intranet, debido a la simpleza del protocolo. Las calidades de tiempo de ejecución dependerán de la solución de directorios seleccionada, mientras que las calidades de comunicación no son muy bien implementadas.

3.1.5.1. Cuando utilizarlo

En aquellos casos en que se requiera implementar una funcionalidad basada en una entidad, que permita establecer una estructura jerárquica centrada en la misma, estableciendo características en dicha entidad, implementar un motor LDAP y realizar la integración mediante dicho protocolo.

3.1.6. Sockets TCP

El protocolo TCP basa su fortaleza en la conexión de sistemas a bajo nivel, siendo uno de los protocolos fundamentales de internet. El protocolo garantiza que los datos serán entregados en su destino sin errores y en el orden en que fueron transmitidos.

Al ser un protocolo de bajo nivel, las funcionalidades que soporten las calidades de tiempo de ejecución deben implementarse desde cero, siendo una solución poco simple de desarrollar. No es una buena solución para cuestiones asociadas al diseño de interfaces (salvo un gran esfuerzo de estandarización y acuerdos entre los participantes), ni para documentación.

Es uno de los mecanismos que mejor performance tiene, siempre y cuando el mensaje a enviar sea simple y no verboso (recordemos que los web services, por debajo, hacen uso de TCP).

La interoperabilidad puede verse como un fuerte, ya que este protocolo es aceptado por aplicaciones que van desde C o C++ hasta Ruby, .NET o Java.

3.1.6.1. Cuando utilizarlo

Cuando se requiera integraciones simples entre diferentes sistemas, sin requerimientos de seguridad, y cuando la performance sea una necesidad imperiosa.

3.1.7. CORBA

CORBA (Common Object Request Broker Architecture) es un estándar que permite el desarrollo de sistemas distribuidos facilitando la integración mediante invocación a métodos remotos, siendo los sistemas orientados a objetos.

Definido por el OMG (Open Management Group), establece las APIs, los protocolos y mecanismos necesarios para asegurar la interoperabilidad entre aplicaciones. Básicamente lo que hace es transformar la especificación de los servicios en cada tecnología a un IDL

establecido y estándar, realizando las transformaciones necesarias. Existen implementaciones estándares para ADA, C, C++, Smalltalk, Java, Python, Perl y Tcl. Existen también proyectos como Remoting.Corba, que intenta integrar aplicaciones .NET con CORBA, o Corba-Ruby para aplicaciones en Ruby.

Su gran fuerte y objetivo es la interoperabilidad, aunque ya no se ve mucho en los nuevos desarrollos, ya que producen un detrimento de la performance y no siempre es tan independiente como se dice de la tecnología subyacente.

Produce dependencia entre las aplicaciones, ya que hace que un sistema conozca los objetos de otro sistema, lo que no es nada bueno para las calidades de diseño, y en tiempo de ejecución tiene diferentes cuestiones que depende de la tecnología en la que se encuentre implementada cada una de las aplicaciones involucradas.

3.1.7.1. Cuando utilizarlo

Cuando exista una aplicación CORBA que deba integrarse con otros sistemas.

3.1.8. Otros mecanismos

Existen otros mecanismos, que por su poca importancia, o implementaciones no vale la pena bajar a detalle. Estos son por ejemplo Etch, creado por CISCO como un reemplazo a los web services, sin demasiado éxito; o DCOM (Distributed Component Object Model), tecnología propietaria de Microsoft ampliamente usada en VB 6, caída en desuso en nuevas implementaciones.

3.2. Mecanismos masivos

Una problemática que se tiene a la hora de decidir utilizar este tipo de integraciones es la definición de cuál es el volumen de información que me hace utilizar un medio masivo y cuando usar un online. A los efectos prácticos, vamos a decir que cuando la necesidad de obtener la información es puntual, o puede deberse a una única operatoria de negocio en un corto lapso de tiempo, y cuando dichas operatorias pueden darse desde diferentes fuentes - o usuarios, en un mismo momento, utilizaremos medios puntuales. En cambio, cuando las solicitudes o información provengan en un mismo momento, en forma masiva y en bloques, utilizaremos mecanismos masivos.

3.2.1. ETL

ETL (Extract, Transform and Load) se refiere al proceso implementado para extraer información de una fuente de datos, transformarla, reformatearlos, limpiarlos e introducirlos en otro medio de almacenamiento (DataMart, base de datos, archivos).

Es ampliamente usado en DataWarehouse, pero es muy útil para realizar integraciones que requieran un movimiento de un volumen grande de información.

Desde el punto de vista de diseño son buenas alternativas, ya que se pueden reutilizar los componentes de extracción, transformación y carga, sirviendo los mismos para definir diferentes flujos de integración, aunque hace que sea necesario conocer la estructura de datos de los sistemas origen y destino, lo que hace muy grande el acoplamiento. La performance dependerá de la forma en que se implemente, siendo muy buenas estas soluciones para asegurar robustez, seguridad, disponibilidad e interoperabilidad.

En cuestiones de diseño no es tan bueno, ya que pocas soluciones son auto-documentadas.

Existen en el mercado diferentes soluciones que implementan este tipo de integraciones, como Informática PowerCenter, Oracle WareHouse Builder, o Microsoft SQL Integration Services.

3.2.1.1. Cuando utilizarlo

Cuando se requiera integración mediante movimiento masivo de información entre diferentes plataformas heterogéneas utilizando diferentes fuentes de datos, y en los que no sea inconveniente el acoplamiento entre sistemas (cabe aclarar que las soluciones de ETL del mercado tienen la capacidad de desarrollar conectores, lo que minimiza el acoplamiento).

3.3. Mecanismos híbridos

Los mecanismos incluidos en este apartado podrían bien ser catalogados como mecanismos puntuales o masivos, por lo que se incluyeron en una sección propia para mayor claridad.

3.3.1. Colas / Mensajería

Un sistema de mensajería permite la comunicación entre aplicaciones de forma indirecta.

Esto se logra mediante el envío de mensajes a un administrador de mensajes, al cual están asociadas las aplicaciones, y es este último quien se encarga de administrarlos y distribuirlos según diversas configuraciones.

No es necesario especificar una aplicación de destino a la hora de enviar mensajes, sino que especifica el nombre de una cola.

Una aplicación puede tener una o más colas de entrada y una o diversas colas de salida.

No es necesario preocuparse por de la tecnología existente en la aplicación destino, o si la aplicación destino no se encuentra disponible (en el caso de los mensajes asincrónicos), sino que el motor de colas se encarga de reenviar el mensaje si la misma se encuentra detenida o bien de iniciarla dependiendo del caso.

Un mensaje consta de dos partes básicas: el header (que contiene información de control que facilita el funcionamiento del administrador), y la data (información a intercambiar entre las aplicaciones).

En la implementación básica, una aplicación entrega un mensaje al administrador (motor) de mensajería, el cual mantiene el mensaje vivo hasta que llegue su tiempo de expiración, o hasta que el consumidor del mensaje lo tenga en su poder.

La integración vía mensajería se basa en la robustez mediante la persistencia de los mensajes enviados, la gestión de prioridades de entrega (por defecto, el mecanismo de entrega es FIFO, pero existen formas de adaptarlo), o expiración de los mensajes (ya que los mismos no pueden estar presentes en la infraestructura del administrador por siempre).

La implementación de colas de mensajería aseguran un bajo acoplamiento entre los sistemas (ya que implementa un modo de comunicación asincrónico, siendo el administrador quien se encargue de entregar el mensaje), al mismo tiempo que asegura la interoperabilidad (siempre y cuando se utilicen motores de colas estándares, estándares de facto o con conectores multi-tecnológicos, como IBM MQ, o implementaciones JMS como ActiveMQ o RabbitMQ).

Las calidades relacionadas a diseño dependerán del buen criterio que se implemente para crear las integraciones, lo mismo que sucede con las cuestiones de comunicación. No es una solución buena para el test.

Existen diferentes motores en el mercado, pasando de los propietarios (Microsoft MSMQ), a los estándares "de facto" (IBM MQ, Oracle AQ), y los estándares tecnológicos (Rabbit MQ, Apache Active MQ).

3.3.1.1. Cuando utilizarlo

Siempre que se pueda, y cuando el manejo de mensajería asincrónica no sea una contra (ya que, por ejemplo, se requiera conocer en el momento la entrega del mensaje y su respuesta). Tener en cuenta las tecnologías involucradas para seleccionar el motor de colas.

3.3.2. Archivos

La integración a través de archivos es una de los mecanismos más utilizados desde hace ya mucho tiempo. Se basa en el acuerdo entre las partes involucradas acerca del contenido y formato del archivo, para que luego el proveedor de la información genere un archivo basado en el mismo ante cada necesidad de integración, y lo deposite en un repositorio pre-acordado. Luego, el o los consumidores de la información proceden a la lectura, interpretación, y actuación en consecuencia de la información contenida.

Este mecanismo de integración permite manejar la independencia entre aplicaciones, siempre y cuando los datos involucrados en los archivos no incluyan información relacionada a la implementación de un determinado sistema, y asegura la robustez y disponibilidad de la información (aunque la interoperabilidad está asegurada). La seguridad es manejada a través de permisos en los repositorios, y mecanismos de encriptación (como PGP - Prety Good Privacy). No es bueno para documentación, ya que no es un mecanismo auto-documentable, salvo en los casos en que se utilicen estándares de integración masiva, como por ejemplo EDIFACT (Estándar desarrollado por la ONU para intercambio de documentos comerciales).

3.3.2.1. Cuando utilizarlo

Siempre que se pueda usar un ETL, lo priorizaría ante este mecanismo. En caso de no poder utilizarse ETL (por costos, por ejemplo), usarlo en cualquier integración masiva que no pueda reemplazarse por colas.

3.3.3. Base de datos

No voy a explayarme mucho sobre el uso de una base de datos para integrar varias aplicaciones. Solo voy a decir que, si bien es simple de realizar (ya que la conexión a base de datos es lo segundo más estandarizado luego de los archivos), y que es un mecanismo robusto y seguro, no recomiendo su uso, ya que genera una dependencia muy fuerte entre los sistemas involucrados, hecho que luego es muy difícil de cambiar. Cualquier integración que se piense a través de base de datos, se puede reemplazar por los mecanismos antedichos.

4. Estilos arquitectónicos para integración

Anteriormente hablé sobre mecanismos de integración, y los dividí entre masivos y puntuales. Para estos últimos, la problemática es tan particular y compleja (mas aún cuando sumamos conceptos como ruteo, transformación, enriquecimiento y composición de mensajes), que la industria llegó a definir estilos arquitectónicos, denominados a través de acrónimos, que muchas veces confunden en lugar de ayudar. Mi intención en este post es volcar mi parecer sobre SOA, ROA, WOA y [lo que venga después]OA, comenzando primero explicando cada uno de ellos, y luego resumiendo la relación que tienen.

4.1. SOA (Service Oriented Architecture)

La descripción de SOA me gusta pensarla de dos maneras diferentes. La primera es la visión de los grandes vendors de software, y la segunda es una visión conceptual, más cercana a la realidad práctica.

Según la definición de los proveedores, SOA es la solución a todos los problemas que se nos presentan durante el ciclo de desarrollo de sistemas, acelerando la implementación de proyectos IT, al mismo tiempo que se reducen los costos a medida que las necesidades de negocio cambian. También proveen la capacidad de facilitar la comunicación entre los usuarios finales y los desarrolladores, definiendo un único lenguaje representado por servicios que tienen relación con actividades de negocio. Todo eso es posible, según ellos, haciendo uso de principios y patrones

arquitectónicos, sumado a metodologías de desarrollo, soportados obviamente por muchas herramientas, pasando por el ESB (Enterprise Service BUS), BPM (Business Process Management) , el registro de servicios y las herramientas de monitoreo. Obviamente, a nivel comercial es muy bueno para esas empresas, ya que obtienen mayor ganancias cuando mas productos venden.

El tema es que a los potenciales clientes de esos productos no les sirve demasiado comprar por comprar o adquirir productos por promesas o presentaciones powerpoints (donde las soluciones siempre compilan y son el camino mas feliz a seguir), sino que para las empresas en las que trabajamos la gran mayoría, debemos tener en cuenta cuestiones que van mas allá de las opiniones de un determinado proveedor, y si no es así estamos en un grave inconveniente.

Lo que si sirve, y mucho, de SOA, es la orientación a servicios. O sea, el concepto de tener a los diferentes sistemas como entidades aisladas e independientes, que proveen puntos de conexión, o servicios (nótese que digo servicios, y no web services). Pensar a los sistemas y la relación entre los mismos de esa manera nos mejora muchísimo la vida, ya que minimiza muchísimo el acoplamiento y dependencia entre los sistemas, y por otro lado reduce los tiempos de desarrollo, al poder ahorrarnos el tiempo de desarrollar dos o mas veces la misma integración.

Obviamente todas estas ventajas requieren de una forma de trabajo en particular, y un compromiso de todas las áreas involucradas, de forma tal que los servicios que se desarrollen tengan un significado desde el punto de vista de negocios. Este punto es crucial en SOA, ya que si al analizar los servicios a crear no tenemos en cuenta el negocio como es actualmente, y las necesidades que pueden a surgir a futuro, los mismos serán un dolor de cabeza mas que una solución.

Centrándome en los pilares de la orientación a servicios, desde mi punto de vista estos son:

- **Coherencia semántica**, o la delimitación clara de los límites funcionales de cada sistema, de forma tal que un único sistema sea el dueño de una determinada entidad de negocios, y que cada funcionalidad sea provisto por un único sistema, o en caso contrario, que la situación negativa sea una cuestión temporal.
- **Puntos de conexión defi**nidos. Cada uno de los sistemas debe definir en forma clara los puntos de conexión que provea para poder explotar sus funcionalidades.
- **Lenguaje canónico**. Este punto muchas veces se estigmatiza, y no se le da la importancia necesaria. Es de vital importancia definir un modelo de entidades, en el que se establezcan no solo la relación entre las mismas y los atributos, sino que sistema las administra, y el formato de cada uno de los atributos. Y lo mas importante, es utilizar este mapas de entidades para definir un lenguaje canónico (por ejemplo, realizar los XSD en los que se basen los WSDL) que definan los protocolos de las integraciones. Al realizar esto, no solo ganamos en facilidad de diseño, sino también en performance, ya que al hacer que todos los sistemas comprendan el mismo lenguaje, nos ahorramos muchísimo tiempo de procesamiento en transformaciones.
- **Conocimiento del negocio**. Para que la orientación a servicios tenga éxito, es necesario contar con analistas muy experimentados y con amplio conocimiento en el negocio. Este es el punto más complicado de entender por los managers de las empresas (ya que no entienden porque es importante tener analistas experimentados para integrar sistemas). Desde mi experiencia y punto de vista, el no contar con estas capacidades hace necesario mucho re trabajo en la definición de los servicios, o se minimiza muchísimo la capacidad de reutilización ya que al momento de la creación de un determinado servicio, no se tuvieron en cuenta muchas cuestiones de contexto, que son necesarias a futuro.

Una vez establecidos los pilares, aparecen las necesidades que van fortaleciendo la orientación a servicios. Estos son el conocimiento de la existencia de servicios, que puede ser soportada por una registry UDDI comercial, una open source, o simplemente una planilla Excel (ya que lo importante es que esta información pueda ser accedida por quienes la precisen, y no que se descubran servicios en tiempo de ejecución), la orquestación y transformación de mensajes, que pueden ayudar a orientar a servicios a sistemas legados o desarrollados en tecnologías propietarias (que puede cubrirse con un ESB), o cuestiones de medición, trazabilidad, orquestación y transaccionabilidad (con un motor BPEL).

La conclusión es no dejarse llevar por lo que nos quieren vender los proveedores, y tratar de centrarse en las necesidades que tengamos, siempre teniendo en cuenta a la orientación a servicios en los casos en que aplique para hacernos mas feliz la vida.

4.2. ROA (Resource Oriented Architecture)

Según wikipedia, ROA (Resource Orientede Architecture) , es un conjunto de especificaciones sobre la implementación de una arquitectura REST. Por tal motivo, la explicación de este modelo estará basada en los conceptos de dicha arquitectura.

REST es un estilo arquitectónico híbrido, derivado de diferentes estilos basados en la red, y combinados con algunas restricciones adicionales, que definen una interface uniforme. REST es una parte importante de WOA, agregando a los principios de SOA las siguientes restricciones arquitecturales:

4.2.1. Cliente/Servidor

Este es el estilo arquitectónico utilizado para la integración. Separando los aspectos de interface de la implementación o los datos, se mejora la portabilidad de los servicios entre plataformas diferentes, además de mejorar la escalabilidad por la simplificación de los componentes del servidor. De igual modo, mejora la escalabilidad (lo que más mejora con esta restricción) ya que no hace falta infraestructura para almacenar información de estado entre los pedidos. Como contra, además de la seguridad antes mencionada, tiene implicancias sobre la performance de la red, por la necesidad de envío de gran cantidad de información.

4.2.2. Servidor Stateless

No se mantiene sesión de estado en el servidor. Todas las interacciones entre el cliente y el servidor tiene toda la información necesaria para entender el pedido, sin tener que depender de información almacenada en el servidor. El estado es mantenido entonces en forma entera en el cliente. De esta forma se mejora la visibilidad ya que un sistema de monitoreo solo debe basarse en los mensajes para saber la naturaleza de los pedidos (esto juega en contra de la seguridad). La disponibilidad es mejorada también, porque se facilita la recuperación de fallas parciales (reintentando).

4.2.3. Cache

Las respuestas pueden estar implícita o explícitamente marcadas como cacheables o no cacheable. Implementando una caché, se salva en parte las debilidades tenidas por la arquitectura de comunicación stateless. Si una respuesta es cacheable, entonces el cliente puede usar los datos de esa respuesta a futuro, sin hacer una nueva petición. Al eliminarse en forma potencial muchas interacciones se mejora la eficiencia, escalabilidad y la performance percibida por el cliente.

4.2.4. Code-on-demand

El pedido de la representación de un recurso puede retornar código embebido (como JavaScript).

4.2.5. Interface Uniforme

Es el núcleo de ROA. La característica principal que caracteriza a REST es el énfasis en una interface uniforme entre los componentes.

Aplicando el principio de generalidad a la interface de los componentes, la arquitectura se simplifica y se mejora la visibilidad e interacción. Mediante la aplicación del principio de generalidad a la interface se simplifica la arquitectura completa, y se mejora la visibilidad de las interacciones. Las implementaciones (lógica) se desacoplan de los servicios provistos. Como contrapartida, se tiene una penalización en eficiencia, ya que la información es transmitida de una forma estándar en lugar de hacerse en una dependiente de la aplicación. Para poder lograr interfaces uniformes, se agregan múltiples restricciones a la arquitectura, a saber: identificación de recursos, manipulación de los mismos a través de su representación, mensajes auto descriptivos, e hipermedia como motor de estados.

Todos los métodos que se utilizan sirven para trabajar con cualquier recurso. Por ejemplo:

- GET: consulta por una representación del estado de un recurso
- PUT: modifica el estado de un recurso
- POST: procesamiento (creación)
- DELETE: elimina la representación

4.2.6. Capas

Se pueden agregar n-capas, cada una de las cuales oculta a la anterior (n-layered). Restringiendo el conocimiento a una sola capa, se oculta la complejidad del mismo, exponiendo la funcionalidad en una forma simple a quienes lo necesiten. La principal desventaja es que agrega overhead y latencia al procesamiento de datos, reduciendo la performance detectada por el usuario. Es útil resaltar que la combinación de una arquitectura en capas con la restricción impuesta por una interface uniforme, se logra un estilo similar a pipe-and-filter, ya que los datos van pasando por diferentes filtros (capas) a través de la red (pipes).

4.3. WOA (Web Oriented Architecture)

WOA Es un subestilo de SOA, basado en la web, igualando la integración A2A (Application to Application) y U2A (User to Application), utilizando HTTP y la arquitectura de la web como núcleo del estilo arquitectónico.

WOA se puede definir como un conjunto entre SOA, REST y la web.

El objetivo que persigue WOA es cambiar el paradigma de diseño de interfaces. Pasar de pensar y diseñar interfaces especializadas a modelar interfaces genéricas, basadas en hipermedia, basándose en el estilo de la web., mediante las cuales se integren sistemas y usuarios a través de una red de hypermedia linkeada, basada en la arquitectura de la web (www.w3.org/TR/2004/REC-webarch-20041215).

Esta arquitectura enfatiza la generalidad de interfaces (UIs y APIs) para lograr efectos de red en base a las siguientes restricciones:

4.3.1. Identificación de recursos

Los recursos deben ser identificados por un Uniform Resource Locator (URL). Llamando recurso a un mapeo abstracto entre una URL y un conjunto de representaciones.

El uso de http: para la identificación de los recursos es una de las restricciones mas importantes de WOA. Pero WOA también define que se reutilicen todos los esquemas de URL (mailto:, ftp: y http: en especial). Como http es el identificador mas utilizado en la web, es el que se debería utilizar principalmente para aumentar el efecto de red.

Los diseñadores necesitan estandarizar los templates de URL genéricos basandose en jerarquias y espacios. Por ejemplo, porque un site de fotos usaría una url como {categoria}/{usuario}/photo/{fecha}, y otro photo/{fecha}/{categoria}/{usuario}. Este tipo de definición de urls hace mas difícil la obtención de fotos de diferentes sitios, en este caso. Es imprescindible nombrar a las cosas por una URI, pero tener cuidado de no nombrar a muchas cosas con la misma URI.

4.3.2. Manipulación de recursos mediante representaciones

El estado del recurso debe ser modificado por la creación o cambio en una representación del recurso en cuestión (por ejemplo, XML), y luego crear o modificar el mismo mediante un conjunto acotado de operaciones genéricas.

Las interfaces WOA usan principalmente las 4 operaciones primarias de HTTP: GET, PUT, POST y DELETE. Lo importante no es cuantos sean los verbos, sino cuan neutrales son con respecto a aplicaciones. Si las operaciones se especializan en un recurso, como get_customer, entonces es poco reutilizable.

En general se comprende la importancia de un lenguaje de marcado genérico (HTML, XML) para brindar generalidad a las interfaces. WOA obliga a los diseñadores a reutilizar las representaciones genéricas disponibles de ser posible, y, sino, crear una representación alternativa lo más genérica posible. Un ejemplo es Atom, inicialmente creado como un reemplazo genérico de RSS, y hoy utilizado por muchas interfaces WOA. Otro ejemplo es XBRL (eXtensible Business Reporting Language).

4.3.3. Mensajes auto descriptivos

Que un mensaje sea auto descriptivo significa que el significado de un mensaje esta incluido en forma completa en su especificación pública.

Los mensajes en una aplicación REST son auto descriptivos por dos motivos: el primero es por el uso de una interface uniforme, y el segundo, es por el hecho de utilizar headers HTTP que describe el contenido de los mensajes además de implementar diferentes aspectos de HTTP como negociación de contenido, pedidos condicionales o cacheo.

Esta característica se logra en REST debido a, por un lado, que el estado completo viaja en el mensaje; la semántica que se utiliza es uniforme (HTTP 1.1); el uso de tipos de datos estándar (utilizan los tipos MIME definidos por IANA); y la funcionalidad de uso de cache en forma explícita.

4.3.4. Hypermedia como motor del estado de la aplicación

Este es la restricción más enigmática, ya que se emparenta con la distinción tradicional entre datos pasivos y código activo. Hypermedia as the Engine of Application State (HATEOAS) requiere que una representación de un recurso contiene más que la simple descripción del recurso, sino que debe tener la descripción de las acciones que pueden ser realizadas sobre el recurso o los recursos asociados.

En otras palabras, todas las interfaces WOA se diseñan para ser generadas dinámicamente de interface a interface. Nosotros interactuamos de esta manera a través de los links en las

páginas. WIkipedia esta pensado de esta manera, por ejemplo.

4.3.5. Neutralidad de aplicaciones

Las primeras cuatro restricciones las heredan de REST, mientras que la última es propia de WOA. Este estilo arquitectónico tiene como premisa la existencia generalizada de HTTP y la arquitectura de la web, lo que hace muy simple y conocida la implementación de nuevas interfaces genéricas sobre protocolos y tecnologías probadas y estándares, haciendo que la implementación sea totalmente neutral a las aplicaciones.

El principal problema de la especificación WS-* es su énfasis en la neutralidad de implementación. Esta se enfoca en generalizar los detalles de las tecnologías de middleware. Este requerimiento muy pocas veces se cumple, ya que en muchos casos los proveedores piensan que no es necesario.

La neutralidad de aplicaciones debe ser el principal objetivo de una interface, ya que esta característica permita la reusabilidad. Los diseñadores de las interfaces deben crear interfaces genéricas, neutrales a las aplicaciones.

La clave de la reutilización es un protocolo genérico, neutral a las aplicaciones, como Atom Publishing Protocol, or Google GData Protocol.

Al ser un subestilo de SOA, WOA sigue los mismos principios:

- Modularidad: Los sistemas deben ser modulares (divide y conquistaras)
- Facilidad de distribución: Los módulos deben ser distribuibles
- Facilidad de descubrimiento: Las interfaces deben ser descubribles
- Posibilidad de intercambio: Un módulo que implementa un servicio debe poder intercambiarse por otro que implemente el mismo servicio
- Reusabilidad: Los módulos proveedores deben ser compartibles, o sea, no debe acoplarse a un solo cliente.

Pero WOA va mas allá, mejorándolos. Así, por ejemplo la modularidad la explota mejor haciendo que cada recurso sea único e independiente; Cada recurso tiene una representación en un formato bien conocido, e identificado de forma pública y acordada; los recursos están visibles solo a través de sus identificadores, de forma tal que son fácilmente descubribles; se aumenta el intercambio ya que no existe detalles de la implementación que soporta a los recursos, lo que también aumenta la reusabilidad.

4.3. Conclusiones finales

En las secciones anteriores se vió que pueden existir diferentes acrónimos que prometan facilitarnos la vida en lo que respecta a integración entre sistemas. Lo realmente importante, es basarse en los pilares definidos para la orientación a servicios, y luego definir si dicha orientación a servicios va a estar enfocada en actividades (o servicios SOA) o en recursos (ROA o WOA), dependiendo de las características de cada empresa, y hasta de los conocimientos funcionales o técnicos con los que se cuente.