Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Diseño e implementación de un protocolo de transferencia confiable de archivos usando arquitectura REST

Memoria para optar al título de
Ingeniero Civil en Computación

Claudio Alejandro Trujillo Muñoz

Profesor Guía:
Agustín Antonio Villena Moya

Miembros de la Comisión:
Claudio Gutiérrez Gallardo
Eduardo Godoy Vega

Santiago de Chile
Enero 2008

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE:
INGENIERO CIVIL EN COMPUTACIÓN
POR: CLAUDIO TRUJILLO M.
FECHA: 15/11/2007
PROF. GUIA: Sr. AGUSTÍN VILLENA M.

"Diseño e implementación de un protocolo de transferencia confiable de archivos usando arquitectura REST"

La presente memoria tiene por objetivo diseñar un protocolo de un sistema que se enmarca en el conjunto de los servicios Web relacionados con la transferencia de archivos, además se debe generar una implementación completa del protocolo diseñado de forma de poder verificar las propiedades que posea el protocolo al estar basado en una arquitectura REST, la cual, de acuerdo a la opinión generalizada del ambiente tecnológico aporta gran cantidad de cualidades a los sistemas que la siguen.

Actualmente el campo de los servicios web de transferencia de archivos que usan la arquitectura REST como base ha sido muy pobremente explorado, pues existen un par de protocolos planteados, los cuales al ser analizados se aprecia que no poseen las características necesarias para ser adoptados seriamente, lo que es ratificado al testear las implementaciones dadas de estos sistemas, ya que no muestran todas las funciones prometidas.

El presente trabajo se abordó de la siguiente forma: primero se realizó un estudio de la arquitectura REST de forma de extraer las propiedades fundamentales y que fuesen un aporte al diseño del protocolo. Luego se estudiaron las alternativas existentes de forma de rescatar lo útil e identificar sus limitantes. Posterior a esto se diseñó el protocolo, denominado Restfull Reliable Transfer Protocol(RRTP) de acuerdo a la arquitectura REST y siguiendo el estándar HTTP/1.1, teniendo especial énfasis en las características de confiabilidad y seguridad de éste. Finalmente se realizó la implementación completa del protocolo, construyendo un cliente y un servidor compatibles, usando para ello tecnologías bastante populares en el mundo del desarrollo, partiendo por el lenguaje de programación Java y todas las tecnologías relacionadas con éste, y que además cumplen con la propiedad de ser de libre acceso y poseer una documentación abundante, de forma de no crear barreras para su adopción.

Se considera que se cumplieron a cabalidad los objetivos planteados en esta memoria, pues como se detalla en ella, se presenta un completo protocolo RRTP de transferencia de archivos más su implementación hecha en Java. El desafío que le queda al protocolo es poder hacerse conocido en el ambiente y comenzar a ser usado por usuario de forma de verificar el éxito de este.

Agradecimientos

Quiero dedicar esta memoria principalmente a mi familia, en especial a mi querida madre, que con su ejemplo de esfuerzo me supo mostrar el correcto camino a seguir al enfrentar los grandes desafíos de la vida. Agradezco a mi padre y hermanos también, los cuales siempre me dieron su incondicional apoyo, sobretodo en los momentos más complicados durante este largo camino de estudios, supieron comprenderme y ayudarme, los amo mucho a todos.

Quiero agradecer a mi profesor Agustín Villena por todo el apoyo brindado en la elaboración de esta memoria.

Quiero agradecer también a mis buenos y grandes amigos, a todos, desde aquellos que hice en mi etapa escolar, a todos aquellos que han pasado por mi vida y yo he pasado por la de ellos. Gracias Maikel, Víctor y Juan Pablo por haber compartido desde un principio tanto los momentos complicados, como los mejores momentos de la carrera y mantener hasta el día de hoy una gran amistad. Gracias Leonardo por ser siempre un gran amigo a pesar del tiempo, gracias por haber compartido conmigo grandes momentos junto a todos los chiquillos Carlitos, Juanjo, Roy, Diego, Mauro, Jano.

Quiero agradecer a todos mis grandes amigos de mi carrera, gracias Cristhian por ser una gran persona y haberme entregado conocimiento muy valioso para la vida. Gracias Felipe S. por ser un gran amigo y compañero de mil batallas en la carrera. Muchas gracias a Daniel, Coya, Osvaldo, Fabián, Héctor, Iván, Mauricio M., Mauricio A., Marcelo, Felipe G., Parra, Esteban, muchos de ellos ya flamantes ingenieros, les deseo a todos la mejor de las suertes en lo que emprendan y en su vida en general.

Quiero agradecer muy especialmente a mi querida Maricelle, la cual me da fuerzas para trabajar gracias al incondicional cariño que me entrega, le agradezco también por comprenderme en los momentos de mayor estrés de la carrera, en los cuales no me caracterícé por ser muy comprensivo, muchas gracias y te quiero mucho mi niña.

No quiero dejar de lado en este agradecimiento a todos aquellos maestros que me formaron, en especial a mis profesores de la etapa escolar, los recuerdo a todos con especial cariño pues creo que han sido los pilares fundamentales de mi formación y de los logros que he obtenido hasta el día de hoy.

Finalmente quiero agradecer a quien ha hecho posible que todo esto suceda, quiero agradecer a Dios por haberme dado la oportunidad de vivir esta vida y cada uno de los momentos que ella ha tenido, me siento muy afortunado por ello y por toda la gente que aquí he nombrado y que sólo Dios pudo haber puesto en mi camino.

Claudio Alejandro Trujillo Muñoz

Índice

Índice de ilustraciones

Índice de tablas

1. Introducción

El actual desarrollo de la Web ha posicionado a esta tecnología como una de las más importantes a nivel mundial. Dicha importancia ha sido obtenida gracias a la gran diversificación de usos que se le ha dado a la Web, dentro de los cuales se puede nombrar: Portal de presentación para empresas de múltiples rubros, comunidades electrónicas, lectura de noticias, Chat, Blog, puntos de compra de productos y servicios, pago de cuentas, etc.

Dada su masificación, existe una enorme cantidad de personas que se dedican a desarrollar para la Web, personas que van desde Ingenieros que han estudiado el tema, hasta personas autodidactas que se han transformado en desarrolladores gracias a un computador conectado a internet y su inquietud por aprender.

Esta diversidad de servicios brindados en la Web, que la ha convertido en una tecnología bastante versátil, sumado a la amplia gama de perfiles de desarrolladores Web existentes, ha acarreado consigo que el desarrollo Web se haya difundido sin tener muy en cuenta las características que la hicieron exitosa. Estas características tienen relación con la arquitectura que se definió en ella. La arquitectura que guarda los principios que la han hecho exitosa recibe el nombre **REST**[4]**,** acrónimo inglés que significa *Representational State Transfer*(Transferencia de Estado Representacional), esta arquitectura fue propuesta por Roy Fielding en su tesis doctoral, el cual, a partir de cómo era la Web, pudo conceptualizarla y llevarla a un modelo arquitectónico formal, describiendo sus características a cabalidad y señalando las ventajas y desventajas que cada una de éstas tenía. REST posee principios básicos como los siguientes:

- Todos los elementos de información son llamados *recursos* y sus *representaciones* pueden ser accedidas utilizando un identificador global(**URI**[10]).
- Como se señaló, cada cliente accede a una *Representación* del recurso, ya que el servidor debe enviar la información a cada cliente cuando se accede a un recurso, luego cada cliente accede a la *Representación* de éste. Luego cuando se hable de *recursos* se entenderá que se hace referencia a su *Representación.*
- No contempla *estados* para el procesamiento de peticiones, pues la petición **HTTP**[17] contiene toda la información necesaria para ser comprendida.
- Todos los *recursos* comparten una interfaz uniforme para la transferencia de estados entre el cliente y la *representación* del *recurso*, la cual consiste en: un conjunto bien definido de operaciones: **POST, PUT, GET, DELETE**[17] y un conjunto de tipos de contenido(content types).

- El uso de enlaces entre los *recursos* lo que le permite al usuario tener acceso a mayor cantidad de recursos sólo siguiendo lo enlaces(*"links"*) de los demás *recursos* existentes.

Los principios que se han nombrado han permitido grandes logros de la tecnología Web, entre los cuales podemos nombrar:

- Proveer mejores tiempos de respuesta gracias al soporte del "*caching".*
- Mejorar la escalabilidad del servidor, reduciendo la necesidad de mantener un estado de comunicación, con lo que diferentes servidores pueden manejar las peticiones siguientes de un cliente.
- En el lado del cliente el requerimiento de software el mínimo, pues basta con un *Browser* para acceder a cualquier aplicación o recurso.
- No requiere un mecanismo de descubrimiento de nuevos recursos, pues estos poseen hyperlinks dentro del contenido actual.
- Provee funcionalidad equivalente cuando es comparada con intentos alternativos de comunicación.

Como se ve, el uso de la arquitectura **REST** para el desarrollo de aplicaciones Web es una buena elección ya que posee principios simples y resultados que han sido verificados por el gran éxito de este tipo de tecnologías. El objetivo de esta memoria es basarse en esta arquitectura para diseñar e implementar un protocolo de intercambio de archivos entre un cliente y un servidor en la Web, que tenga las buenas propiedades nombradas anteriormente.

Tenemos que tener en cuenta para este diseño que el protocolo **HTTP** de por sí NO es confiable, es decir, ni el servidor ni el cliente se aseguran de que sus mensajes son realmente recibidos por su contraparte. Si bien en la gran mayoría de los casos los mensajes se transfieren exitosamente, no son parte del protocolo las peticiones de confirmación de envío/recepción. Este es uno de los desafíos a abordar al diseñar un protocolo nuevo que trabaje sobre **HTTP**, puesto que no es una funcionalidad por defecto de éste.

2. Justificación

Los protocolos de intercambio de archivos vía Web son un tema abordado en la actualidad desde 2 frentes: **SOAP**[12] y **REST**. Actualmente el estándar es la implementación de estos protocolos usando **SOAP**, un ejemplo de esto es **WS-Reliable Messaging**[3], el cual en su definición presenta las distintas estructuras en formato **XML**[14] que deben ser implementadas para tener compatibilidad entre los sistemas. El problema es que las actuales implementaciones existentes no poseen todas las funcionalidades especificadas, puesto que las definiciones son muy extensas, por lo que generalmente se implementan las funcionalidades básicas, esto se puede ver en implementaciones como **Project Tango** de Sun[8], **RAMP** de IBM[7] y **Sandesha** de Apache[6], los cuales señalan que sus implementaciones poseen sólo partes del protocolo, siendo **Sandesha** la más completa hasta el momento, pero continuamente en desarrollo. Esto produce que la interoperabilidad de los sistemas no sea 100% efectiva, lo cual acarrea eventuales costos de adopción e implementación de funcionalidades por cada empresa que quiere sumarse al uso de este protocolo en pos de la interoperabilidad de los sistemas que posea.

Por lo tanto es interesante que el diseño y posterior implementación de este protocolo se haga cumpliendo con la arquitectura **REST**, lo cual entrega una ventaja en el ámbito de la interoperabilidad de los sistemas. Esto se puede apreciar analizando los actuales diseños existentes, tales como **HTTPLR**[1] y **RRMTP**[2] cuyas especificaciones muestran que los mensajes de comunicación entre los participantes son mucho más simples, haciéndolos más eficientes e igual de efectivos. Luego hacer interoperables sistemas **REST** es más directo y mucho menos costoso en caso de que quiera ser adoptado por la empresa.

El desarrollo de esta memoria, como se ha mencionado antes, consta de 2 grandes partes, una de diseño y otra de implementación. Se procedió de tal forma que en un principio se estudió el estado del arte de las actuales arquitecturas existentes, y de **REST** en particular.

El diseño permite que la transferencia de archivos cumpla con las propiedades de ser:

- **Segura**: que los actores dentro de la transferencia estén claramente identificados y que además los datos no puedan ser atrapados por un externo y reconstruidos fácilmente por éste.

- **Confiable**: dice relación con que tanto los avisos de éxito o fracaso de una transferencia sean siempre coherentes para ambos participantes. Además de la implementación de técnicas que permitan la verificación de la transferencia íntegra de los datos en ambos sentidos(Cliente->Servidor y viceversa).

Para la empresa, uno de los principales problemas hoy en día para la adopción de una tecnología, es la alta complejidad de los protocolos definidos para realizar operaciones muchas veces simples. Para el caso de la transferencia de archivos vía Web el estándar WS-Reliable Messaging ofrece una solución bastante compleja y que, por este mismo motivo, no posee implementaciones completas y agrega una casi nula compatibilidad con actuales herramientas de uso común sobre la Web, por ejemplo un browser como cliente, lo cual, de ser factible aportaría una facilidad de adopción increíble y un ahorro de costo considerable para la empresa.

Esta implementación estará orientada a resolver el problema de la interoperabilidad de Web Services de sistemas Business-To-Business, es decir, interoperar sistemas de distintas compañías, problema que hoy se ve reflejado en la demora de ciertas transacciones realizadas, como por ejemplo en sistemas bancarios, las cuales por lógica deberían tomar un tiempo a lo más igual que realizar la transacción presencialmente en alguna sucursal, pero hoy en día toman un tiempo mayor, lo cual se explica por la poca confianza entre los sistemas y la cantidad de validaciones por las que tienen que pasar transacciones realizadas entre distintos bancos.

Con todo esto lo que se busca es brindar una nueva alternativa al mundo de protocolos de transferencia de archivos vía Web, pero que cumpla con la arquitectura **REST**, cumpliendo además con las características de ser **Seguro** y **Confiable,** intentando brindar una real alternativa al actual estándar **SOAP**, permitiendo que la interoperabilidad de los sistemas sea algo directo, valiéndose para esto, de las características que han hecho que la Web sea tan exitosa como lo es hoy en día. Teniendo en cuenta que mientras más simple y efectivo sea el protocolo, su adopción será menos costosa, facilitando su difusión, lo cual se facilita aún más al entregar una implementación de referencia del protocolo, usando las tecnologías que han mostrado ser de mayor uso en el mundo de la Web.

3. Objetivos

3.1. Objetivo General

Diseñar e implementar un protocolo de transferencia de archivos vía Web que cumpla con ser **seguro** y **confiable** y que además cumpla con las propiedades de la arquitectura Web **REST**, obteniendo de esta forma una implementación que cumpla con las características que han hecho que la Web tenga el éxito que posee actualmente. El protocolo deberá ser claro en su especificación e implementado íntegramente de acuerdo a ésta.

3.2. Objetivos Específicos

- Establecer las propiedades de la arquitectura REST que serán usadas en el diseño en implementación del sistema, a través del estudio de las características de la arquitectura **REST** que hacen de la Web una tecnología exitosa.
- Diseñar un protocolo de transferencia confiable a partir del estudio del estado del arte de los protocolos propuestos actualmente en el tema de transferencia de archivos vía Web, evaluando sus fortalezas y debilidades, en particular **HTTPLR** y **RRMTP**.
- Seleccionar de las tecnologías que permitirán cumplir con los requerimientos de seguridad en el diseño y de fácil adopción, a través del estudio de las tecnologías actuales de Firma Electrónica, de **HTTPS**[13], clientes HTTP, servidores de aplicación y persistencia de la información.
- Generar una implementación de referencia del protocolo diseñado de manera íntegra tanto del lado del Servidor como del cliente haciendo uso de las tecnologías seleccionadas, las cuales cumplirán con la propiedad de ser de libre acceso.

4. Metodología

La metodología a seguir para alcanzar los objetivos planteados consta de los siguientes puntos:

- **Marco Teórico:** Investigación sobre la arquitectura **REST** recolectando sus propiedades más importantes, que permitieron al posterior diseño del protocolo adoptar la mejores propiedades de acuerdo a las características del sistema particular.
- **Investigación de Soluciones Existentes:** Estudio de los actuales protocolos de transferencia de archivos de manera de obtener una idea completa del estado del arte en el tema y obtener las fortalezas y debilidades de cada una, las cuales se tuvieron en cuenta al momento de realizar el diseño final del protocolo.
- **Diseño del protocolo:** Definición del diseño del protocolo tomando en cuenta todo el estudio previo, de forma de definir las funciones a usar, las componentes del protocolo y cuales serán cada uno de los pasos a realizar para completar correctamente las transferencias. Definición de los comportamientos en casos de eventuales errores, los cuales aparecerán en algún momento pues el sistema opera sobre redes.
- **Investigación de tecnologías**: Estudio de las tecnologías asociadas con la implementación a realizar, en este caso el estudio de **HTTPS**, y las tecnologías asociadas con la Firma Electrónica, de manera de cumplir con lo prometido en el diseño del protocolo y brindar de buena forma las funcionalidades planteadas. Además de la selección del lenguaje de programación, servidor de aplicaciones y manejo de persistencia de los datos, siempre enfocándose en la fácil adopción de la implementación a posterior por algún interesado.
- **Implementación del protocolo:** Implementación del protocolo a cabalidad, siendo rigurosos con el detalle expresado en el diseño. Implementación de un servidor y un cliente compatibles sobre un servidor Web de uso público de manera que esta implementación pueda ser difundida de manera gratuita y libre a través de Internet de forma de fomentar una masificación expedita.

5. Marco Teórico

Una de las grandes bases teóricas para el desarrollo de esta memoria es la arquitectura REST, a continuación se exponen en detalle sus características con el objetivo de clarificar sus cualidades y justificar el por qué de su elección.

5.1. Definiciones básicas

Se presentan las definiciones de conceptos básicos para el completo entendimiento de los puntos posteriores:

a) Componente

Una componente es una unidad abstracta de instrucciones y estados interno de software que proveen una transformación de datos vía su interfaz. Ejemplos de transformación incluyen cargar en memoria desde un dispositivo de almacenamiento secundario, realizar cálculos, trasladar a un formato diferente, etc. En otras palabras una componente se define por su interfaz y los servicios que ésta provee a otras componentes.

b) Conector

Un conector es un mecanismo abstracto que media la comunicación, coordinación o cooperación entre componentes. Una forma de entenderlo es contrastándolo con una componente. Los conectores permiten la comunicación entre componentes transfiriendo elementos de datos de una interfaz a otra sin cambiar los datos, en cambio una componente por lo general transforma datos y los entrega.

c) Dato

Un dato es un elemento de información que es transferido desde una componente, o recibido por una, vía un conector. Ejemplos incluyen secuencias de bytes, mensajes, objetos serializados, pero no incluye información que es residente permanente. La naturaleza de los elementos de datos dentro de una arquitectura de aplicación basada en la red a menudo determinará si es que un estilo de arquitectura dado es o no apropiado.

5.2. Propiedades arquitectónicas claves

Se procede a explicar cada una de las propiedades de una arquitectura que se han usado para diferenciar y clasificar estilos de arquitectura.

a) Rendimiento

Una de las principales razones para centrarse en estilos de aplicaciones basadas en la red es porque la interacción de las componentes puede ser el factor dominante en determinar el rendimiento percibido por el usuario y la eficiencia de la red. El rendimiento de una aplicación basada en la red es limitado primero por los requerimientos de la aplicación, luego por el estilo de interacción elegido, seguido por la arquitectura realizada y finalmente por la implementación de cada componente. Una arquitectura no puede ser más eficiente que lo permitido por su estilo de interacción.

a.1) Rendimiento de la red

El rendimiento de la red es usado para medir ciertos atributos de comunicación. *Throughput* es la tasa a la cual la información, incluyendo datos de aplicación y la sobrecarga de comunicación, es transferida entre las componentes. *Overhead* puede ser separada en overhead de configuración inicial, y overhead por interacción, distinción que es útil para identificar conectores que pueden compartir overhead de configuración a través de interacciones múltiples. *Bandwidth* es una medida del máximo throughput disponible sobre un conexión de red. *Bandwidth usable*, se refiere a la porción de bandwidth que está actualmente disponible para la aplicación. El estilo de arquitectura usado impacta en el rendimiento de la red por su influencia en el número de interacciones por cada acción del usuario y la granularidad de los elementos de dato. Un estilo que privilegia pequeñas y fuertes interacciones será eficiente en aplicaciones que involucran pequeñas transferencias de datos entre componentes conocidas, pero causará excesivo overhead en aplicaciones que involucran transferencias de grandes cantidades de datos o muchas negociaciones entre las interfaces.

a.2) Rendimiento percibido por el usuario

Las medidas primarias del rendimiento percibido por el usuario son la *latencia* y el *tiempo de completitud*. La latencia es el periodo de tiempo entre el primer estímulo y la primera indicación de respuesta. Tiempo de completitud es el tiempo tomado para

completar una acción de la aplicación. Es importante notar que consideraciones de diseño para mejorar la latencia tendrán, en ocasiones, efectos de degradamiento en el tiempo de completitud y vice-versa. Por ejemplo el comprimir datos antes de enviarlos por la red puede mejorar la latencia en acciones de envío de datos, pero si la cantidad de datos a comprimir es demasiado, el aumento del tiempo de completitud puede ser excesivo.

a.3) Eficiencia de la red

Una observación interesante acerca de las aplicaciones basadas en la red es que el mejor rendimiento de una aplicación es obtenido al no usar la red. Esto significa que el estilo de arquitectura más eficiente para una aplicación basada en la red, es aquella que puede efectivamente minimizar el uso de la red cuando es posible hacerlo, a través de uso de interacciones previas, reduciendo la frecuencia de interacciones con la red en relación a las acciones del usuario.

b) Escalabilidad

Escalabilidad se refiere a la habilidad de la arquitectura para soportar un gran número de componentes, o interacciones entre los componentes, dentro de una configuración activa. La escalabilidad puede ser mejorada simplificando componentes, distribuyendo servicios a través de muchas componentes y controlando las interacciones y configuraciones como resultado de un monitoreo. La escalabilidad es impactada por la frecuencia de las interacciones, si la carga sobre una componente está distribuida uniformemente sobre el tiempo u ocurre en peaks, si la interacción requiere de una entrega garantizada o del mejor esfuerzo.

c) Simplicidad

El medio básico por el cual los estilos de arquitectura inducen simplicidad es aplicando el principio de separación temática para la asignación de funcionalidades entre las componentes. Si la funcionalidad puede ser asignada de forma tal que las componentes individuales son sustancialmente menos complejas, entonces ellas serán más fáciles de entender e implementar. Además esta separación facilita la tarea de razonar acerca de la arquitectura completa.

d) Modificabilidad

Modificabilidad es la facilidad con la cual un cambio puede ser hecho a una

arquitectura de aplicación. Esta característica puede ser descompuesta en las siguientes:

d.1) Evolucionabilidad: representa el grado en el cual una implementación de una componente, puede ser cambiado sin afectar negativamente a otras componentes.

d.2) Extensibilidad: es definida como la habilidad para adherirle funcionalidad a un sistema. La extensibilidad dinámica implica que la funcionalidad se puede agregar a un sistema desplegado sin afectar el resto del sistema.

d.3) Adaptabilidad: se refiere a la habilidad de especializar temporalmente el comportamiento de un elemento arquitectónico, de tal forma que pueda efectuar un servicio inusual.

d.4) Configurabilidad: se relaciona tanto con extensibilidad como con reusabilidad en que se refiere a modificación de componentes post-despliegue, o configuración de componentes, tales que ellos son capaces de usar un nuevo servicio o tipo de elemento.

d.5) Reusabilidad: es una propiedad de una arquitectura de aplicación, se da si sus componentes, conectores, o elementos de dato pueden ser reusados, sin modificación, en otra aplicación.

e) Visibilidad

Visibilidad en este caso se refiere a la habilidad de una componente para monitorear o mediar la interacción entre otras dos componentes. Visibilidad puede habilitar una mejora de rendimiento vía interacciones de cache compartida, escalabilidad a través servicios en capa, confiabilidad con la supervisión, y seguridad permitiendo que las interacciones sean examinadas por mediadores.

f) Portabilidad

El software es portable cuando puede correr en diferentes ambientes. Estilos que inducen portabilidad también incluyen aquellos que mueven código entre los datos para ser procesado, como los estilos de máquina virtual y agente móvil.

g) Confiabilidad

Confiabilidad, dentro de la perspectiva de arquitecturas de aplicación, puede ser visto como el grado en el cual una arquitectura es susceptible a fallas en el nivel de sistema en la presencia de fallas parciales en las componentes, conectores o datos.

5.3. Estilos arquitectónicos basados en la red

Luego de presentar las propiedades arquitectónicas relevantes para el diseño de aplicaciones basadas en la red, se procede a detallar los estilos arquitectónicos para aplicaciones basadas en la red, señalando las características principales de cada una.

a) Estilos de Replicación

a.1) Repositorio replicado

Sistemas basados en repositorios replicados mejoran la accesibilidad de los datos y la escalabilidad de servicios teniendo más de un proceso proveyendo el mismo servicio. Estos servidores descentralizados interactúan para proveerle al cliente la ilusión de que hay sólo un servicio centralizado. Sistemas de archivo distribuídos y sistemas de versionamiento remoto(CVS) son los ejemplos primarios.

La primera ventaja es que mejora el rendimiento percibido por el usuario, ya que reduce la latencia de una petición normal y habilita operaciones desconectado, frente a posibles fallas del servidor o búsquedas intencionales de redes. El mantener la consistencia de los datos es la principal tarea.

a.2) Caché

Una variante del repositorio replicado es encontrado en el estilo de caché: replicación del resultado de una petición individual de tal forma que ésta pueda ser reusada por posteriores peticiones. Esta forma de replicación es comúnmente encontrada en casos en los que el conjunto de datos excede largamente la capacidad de cualquier cliente, como es el caso de la web, o donde el completo acceso al repositorio no es necesario.

Caching provee una levemente menor mejora que el repositorio replicado en términos de rendimiento percibido por el usuario, ya que muchas peticiones perderán el caché y sólo datos recientemente accesados estarán disponibles para operaciones desconectadas. Por otro lado caching es mucho más fácil de implementar, no requiere mucho procesamiento ni almacenamiento y es más eficiente pues los datos son transmitidos sólo cuando son solicitados.

b) Estilos Jerárquicos

b.1) Cliente-Servidor

Este estilo es el más frecuentemente encontrado de los estilos arquitectónicos para aplicaciones basadas en la red. Una componente servidor, ofreciendo un conjunto de servicios, espera a escuchar peticiones sobre esos servicios. Una componente cliente, deseando que un servicio sea realizado, enviando una petición al servidor vía un conector. El servidor o bien rechaza o ejecuta la petición y envía la respuesta de vuelta al cliente.

La separación de rubros es el principio detrás de la restricción cliente-servidor. Una apropiada separación de las funcionalidades debería simplificar la componente servidor y así mejorar la escalabilidad. Esta simplificación usualmente se realiza moviendo todas las funcionalidades de la interfaz de usuario en la componente cliente. La separación también permite que ambos tipos de componentes evolucionen independientemente, a condición de que la interfaz no cambia.

b.2) Sistema de capas y Cliente-Servidor en capas

Un sistema en capas es organizado jerárquicamente, cada capa provee servicios a la capa que está sobre ella, y usa los servicios de la capa que está bajo ella. Aunque el sistema de capas es considerado un estilo puro, su uso dentro de los sistemas basados en la red está limitado a su combinación con el estilo cliente-servidor para producir el estilo cliente-servidor en capas.

Sistemas en capas reducen el acoplamiento entre múltiples capas escondiendo las capas internas de todas, excepto de la capa adyacente, así se mejora la evolucionabilidad y la reusabilidad. La desventaja principal es que suman overhead y latencia al procesamiento de los datos, reduciendo el rendimiento percibido por el usuario.

Cliente-servidor en capas agrega las componentes proxy y gateway al estilo cliente-servidor. Un proxy actúa como un servidor compartido para una o más componentes cliente, tomando requerimientos y reenviándolos a componentes servidor. Una componente gateway aparenta ser un servidor normal para clientes o proxies que requieren sus servicios, pero realmente está reenviando esas peticiones a sus servidores en capas internas. Esta componente adicional mediadora puede ser agregada en múltiples capas para agregar características como balance de carga o chequeos de seguridad al sistema.

b.3) Cliente-SinEstado-Servidor

Este estilo deriva del cliente-servidor con la restricción adicional de que *estados de sesión* no son permitidos en la componente servidor. Cada petición desde el cliente hacia el

servidor debe contener toda la información necesaria para ser comprendido, y no puede tomar ventaja de ningún contexto guardado en el servidor. Estado de la sesión es mantenido enteramente en el cliente.

Estas restricciones mejoran las propiedades de visibilidad, confiabilidad y escalabilidad. Visibilidad es mejorada pues un sistema de monitoreo no tiene que mirar más allá de los datos de una petición simple para determinar la naturaleza completa de la petición. Confiabilidad es mejorada porque facilita la tarea de recuperación desde fallos parciales. Escalabilidad es mejorada porque no tiene que guardar los estados entre peticiones, permitiendo a la componente servidor liberar los recursos rápidamente y además simplificar la implementación.

La desventaja de este estilo es que puede decrecer el rendimiento de red al incrementar los datos repetidos enviados en una serie de peticiones, ya que los datos no pueden ser dejados en el servidor en un contexto compartido.

b.4) Cliente-Cache-SinEstado-Servidor

Este estilo deriva del cliente-sinEstado-servidor y del estilo caché, vía la agregación de la componente de caché. Caché actúa como mediador entre el cliente y el servidor en quienes las respuestas a peticiones previas pueden, de ser consideradas cacheables, ser reusadas en respuesta a peticiones posteriores que sean equivalentes a la que ha sido guardada.

La ventaja de agregar la componente caché, es que se tiene el potencial para parcial o completamente eliminar algunas interacciones, mejorando la eficiencia y el rendimiento percibido por el usuario.

b.5) Capas-Cliente-Cache-SinEstado-Servidor

Este estilo deriva de cliente-servidor en capas y cliente-cache-sinEstado-servidor a través de la adición de una componentes proxy y/o gateway. Las ventajas y desventajas de este estilo provienen de la combinación de ambos estilos que lo originan.

b.6) Sesión Remota

Este estilo es una variante del cliente-servidor que intenta minimizar la complejidad, o maximizar el reuso, de las componentes del cliente en vez de la componente servidor. Cada cliente inicia una sesión en el servidor y luego invoca una serie de servicios, finalmente termina la sesión. El estado de la aplicación es mantenido enteramente en el servidor.

Las ventajas de este estilo es que es fácil centralmente mantener la interfaz del servidor, reduciendo temas de inconsistencia en clientes desplegados cuando una

funcionalidad es extendida, y mejora la eficiencia si las interacciones hacen uso del contexto de sesión extendida en el servidor. La desventaja es que reduce la escalabilidad del servidor, debido a los estados de aplicaciones guardados y reduce la visibilidad de las interacciones, ya que un monitor tendría que saber el estado completo del servidor.

5.4. Descripción del estilo arquitectónico REST

Luego de presentados los estilos arquitectónicos y sus características, se procede a describir a cabalidad el estilo arquitectónico REST, construyéndolo incrementalmente, al indicar cada una de las propiedades arquitectónicas que serán incluidas en él.

El estilo arquitectónico REST es un estilo híbrido derivado de varios de los estilos arquitectónicos nombrados previamente y combinado con restricciones adicionales que definen un conector de interfaz uniforme. Se detalla a continuación incrementalmente las propiedades que generarán el estilo.

a) Estilo nulo

El estilo nulo es simplemente un conjunto vacío de restricciones. Desde la perspectiva de las arquitecturas, el estilo nulo describe un sistema en el cual no hay límites descritos entre las componentes. Este es el punto de partida para la descripción de REST.

b) Cliente-Servidor

Las primeras restricciones agregadas a la arquitectura son las del estilo arquitectónico cliente-servidor, descrito previamente. La separación de funciones es el principio detrás de las restricciones del cliente-servidor. Separando las funciones de la interfaz de usuario, de las funciones de almacenamiento de datos, se mejora la portabilidad de la interfaz de usuario a través de múltiples plataformas y mejora la escalabilidad simplificando las componentes del servidor. Quizás los más significativo para la Web, es que la separación permite a las componentes evolucionar independientemente, soportando así los requerimientos de la escala de internet de múltiples dominios organizacionales.

c) Sin Estado

Se necesita agregar una restricción a la interacción cliente-servidor: la comunicación debe ser naturalmente sin estado, como en el estilo cliente-sinEstado-servidor, de tal forma que cada petición del cliente al servidor contenga toda la información necesaria para ser entendida, y no puede tomar ventaja de ningún contexto guardado en el servidor. El estado de sesión es enteramente guardado en el cliente.

Esta restricción induce las propiedades de visibilidad, confiabilidad y escalabilidad.

Visibilidad es mejorada pues un sistema de monitoreo no tiene que mirar más allá de un simple dato de una petición para determinar la naturaleza completa de la petición. Confiabilidad es mejorada pues facilita la tarea de recuperación a fallas parciales. Escalabilidad es mejorada pues no se tienen que guardar los estados entre peticiones, permitiéndole a la componente servidor liberar recursos rápidamente, y además simplificar su implementación pues el servidor no tiene que administrar el uso de recursos a través de peticiones.

La desventaja es que puede disminuir el rendimiento de la red al incrementar la cantidad de datos repetidos que tienen que enviarse entre peticiones, ya que los datos no pueden ser dejados en el servidor en un contexto compartido. Además, al dejar el estado de la aplicación en el lado del cliente reduce el control del servidor sobre el comportamiento consistente de la aplicación, ya que la aplicación se hace dependiente de la correcta implementación de la semántica sobre múltiples versiones de clientes.

d) Cache

De forma de mejorar la eficiencia de la red, se agrega la restricción para formar el estilo cliente-cache-sinEstado-servidor. La restricción de cache requiere que los datos dentro de la respuesta a una petición sean clasificados como cacheable o no cacheable. Si la respuesta es cacheable, entonces al cliente cache se le da el derecho a reusar los datos de la respuesta para posteriores peticiones equivalentes.

La ventaja de agregar la restricción de cache es que tiene el potencial de poder parcial o completamente eliminar algunas interacciones, mejorando la eficiencia, escalabilidad y rendimiento percibido por el usuario al reducir la latencia promedio de una serie de interacciones. El trade-off es que el cache puede disminuir la confiabilidad si es que los datos guardados en el cache difieren significativamente de los datos que habrían sido obtenidos si es que la petición hubiese sido enviada directamente al servidor.

e) Interfaz uniforme

La característica central que distingue al estilo arquitectónico REST de otros estilos basados en la red, es su énfasis en una interfaz uniforme entre sus componentes. Aplicando el principio de software de generalidad a la componente interfaz la arquitectura sobre todo el sistema es simplificada y la visibilidad de las interacciones es mejorada. Implementaciones son desacopladas de los servicios que proveen, lo que anima una evolucionabilidad independiente. El trade-off es que una interfaz uniforme degrada la eficiencia, ya que la información es transferida en una forma estándar en vez de una forma que sea específica a las necesidades de la aplicación. La interfaz de REST es diseñada para ser eficiente para transferencias largas y granuladas de datos de hypermedia, optimizando el caso común de la Web, pero resultando en una interfaz que no es óptima para otras

formas de interacción arquitectónica.

f) Sistema en capas

En orden de mejorar el comportamiento para los requerimientos de escala de internet, se agrega la restricción de sistema en capas. El estilo de sistema de capas permite a una arquitectura ser compuesta por capas jerárquicas, restringiendo el comportamiento de las componentes a que cada una de ellas no puede ver más allá de la capa inmediata con la que está interactuando. Restringiendo el conocimiento del sistema a una sola capa, se coloca un límite a la complejidad del sistema completo. Las capas pueden ser usadas para encapsular servicios viejos, y para proteger nuevos servicios de clientes viejos, simplificando componentes al mover las funcionalidades menos frecuentemente usadas a intermediarios compartidos. Intermediarios pueden también ser usados para mejorar la escalabilidad del sistema permitiendo el balance de carga de servicios a través de múltiples redes y procesadores.

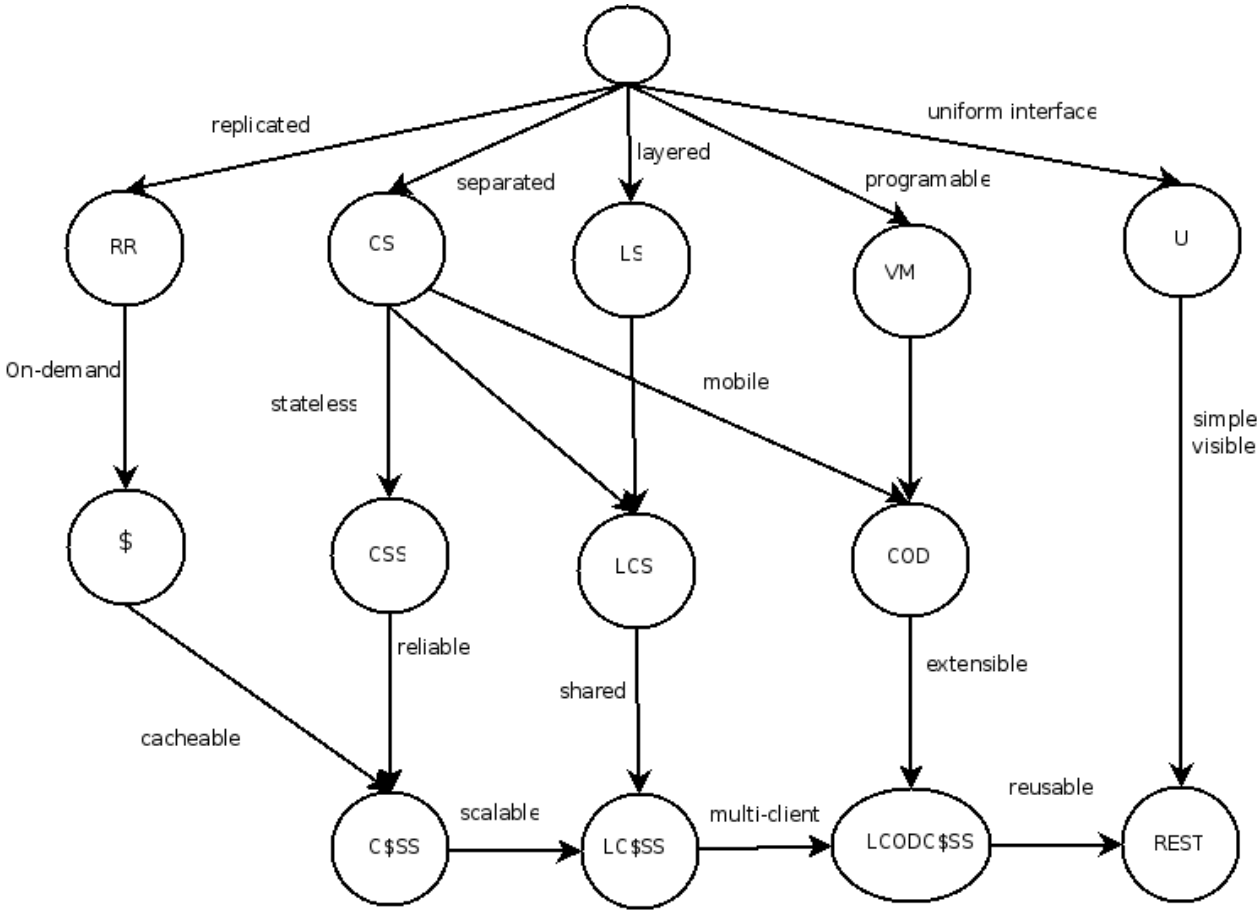
La desventaja primaria es que se agrega sobrecarga y latencia al procesamiento de los datos, reduciendo el rendimiento percibido por el usuario.

g) Código en demanda(Code on demand)

La última agregación al conjunto de restricciones para REST viene del estilo de código en demanda. REST permite que las funcionalidades del cliente sean extendidas a través de la descarga y ejecución de código en forma de applets o scripts. Esto simplifica a los clientes reduciendo el número de características requeridas para ser pre-implementadas. Permitiendo que las características sean descargadas después del despliegue, lo que mejora la extensibilidad del sistema. Aunque, también reduce la visibilidad, por lo que es sólo una restricción opcional en REST.

5.5 Diagrama de derivación de REST

Se presenta a continuación un diagrama que muestra los estilos arquitectónicos de los que proviene REST con las respectivas características que los van formando:



Leyenda:

RR:Replicated Repositorie
CS:Client-Server
LS:Layered System
VM:Virtual Machine
U:Uniform Interface
$:Cache
CSS:Client-Stateless-Server
LCS:Layered-Client-Server
COD:Code On Demand
C$SS:Client Cache Stateless Server
LC$SS:Layered Client Cache Stateless Server
LCODC$SS:Layered Code On Demand Client Cache Stateless Server

Ilustración 1 - Derivación arquitectura REST

6. Sistemas actuales

Se procede a detallar el análisis realizado a las 2 alternativas existentes hoy en día en la industria de sistemas basados en la arquitectura REST, que permiten realizar transferencia de archivos. Las alternativas analizadas son RRMTP y HTTPLR. Se presentan los detalles mas significativos de cada protocolo de manera de ver las fortalezas y debilidades de cada uno.

Al final del capítulo se detalla una descripción general de una tercera alternativa existente hoy, WS-Reliable Messaging, la cual es el estándar de comunicación en la industria, la cual no se basa en la arquitectura REST, sino que usa SOAP. Es importante mostrar sus características pues es el actual dominador del campo de mensajería entre servicios Web.

6.1. RRMTP

Este protocolo fue creado por el gobierno Irlandés, específicamente la entidad de gobierno llamada Reach.

Se procederá a explicar el funcionamiento del protocolo en lo que respecta a las 2 opciones de transacción que considera, obtener un mensaje del servidor y poner un mensaje en él.

6.1.1. Creación del intercambio de mensaje

La creación del intercambio de mensaje es el primer paso requerido tanto para enviar como para recibir un mensaje usando RRMTP. El mensaje de intercambio es creado usando un mensaje vacío HTTP POST al servidor RRMTP. Una vez que el intercambio de mensaje existe, su tipo es determinado por la acción subsecuente pedida al servidor, HTTP GET o PUT.

El intercambio de mensaje es creado usando una petición HTTP POST estándar y especificando la *<ruta-servidor>* del servidor RRMTP. El valor de la *<ruta-servidor>* no debe ser asumido como una URI relativa, de acuerdo a al especificación HTTP.

Si la *<ruta-servidor>* fue válida, y el servidor creó el intercambio de mensaje exitosamente, el servidor debe retornar una respuesta 201 CREATED con *Content-Length* igual a cero y con las cabeceras *Location* y *Expires.*

La transacción posee un identificador único llamado MXID, generado por el servidor. El valor de MXID será un sub-elemento de la URL especificando la *<ruta-servidor>.* El valor completo de la cabecera *Location* es, por lo tanto, garantizado de ser

único dentro del contexto de espacio de nombre de URI para el servidor.

La cabecera *Expires* indica al cliente el tiempo de vida del intercambio de mensaje. Las peticiones HTTP al lugar especificado en la cabecera *Location* después del tiempo especificado en la cabecera *Expires* debe generar una respuesta 404 Not Found desde el servidor.

6.1.2. Obteniendo un mensaje del servidor

Obtener un mensaje del servidor es uno de los 2 posibles segundos pasos en un intercambio de mensaje RRMTP. Para recibir un mensaje, el cliente debe tener un identificador de intercambio de mensaje válido. Este identificador es normalmente obtenido justo antes de intentar obtener el mensaje, pero cualquier otro medio podría ser usado, por ejemplo un email.

Los mensajes son recibidos usando una petición HTTP GET estándar hacia una URL identificadora de intercambio de mensaje:*<message-exchange-location>*. Si la *<message-exchange-location>* representa un intercambio de mensaje en el estado *Creado,* el servidor debe intentar reservar el próximo mensaje disponible para entregarlo con este intercambio de mensaje. Si un mensaje está disponible y fue exitosamente reservado, el servidor debe responder con una respuesta 200 *OK.* Así el mensaje reservado no debe estar disponible a ningún otro cliente mientras el intercambio de mensaje es válido.

Si la *<message-exchange-location>* representa un intercambio de mensaje en el estado *GET,* y el servidor está disponible de localizar el mensaje sin errores, el servidor debe responder con una respuesta *200 OK* conteniendo el mensaje a ser transferido en el cuerpo de la respuesta.

6.1.3. Poniendo un mensaje en el servidor

Poner un mensaje en el servidor es la segunda de las 2 posibles operaciones que pueden ser ejecutadas en un nuevo intercambio de mensajes. Para poner un mensaje en el servidor, el cliente debe tener un identificador de intercambio de mensaje válido, de la misma forma que para obtener un mensaje.

Los mensajes son enviados al servidor usando una operación HTTP PUT estándar apuntando a la URL *<message-exchange-location>* dada, la cual normalmente incluye la ruta al servidor RRMTP y el identificador de intercambio de mensaje.

Si la *<message-exchange-location>* representa un intercambio de mensaje en el estado *Creado*, el servidor debe grabar que este intercambio de mensaje es para aceptar un mensaje desde el cliente. El servidor debe tratar el *<message-body>* transferido con la

petición como el mensaje a ser asociado con el actual intercambio y debe asegurarse que una vez recibido, el *<message-body>* es confiablemente guardado hasta que el intercambio de mensaje es cerrado o expira. Si la operación PUT es exitosa, el servidor debe responder 201 *Created* de acuerdo a la especificación HTTP/1.1.

Si la *<message-exchange-location>* representa un intercambio de mensaje en el estado *PUT*, el servidor debe guardar el *<message-body>* recibido desde el cliente en lugar de cualquiera recibido previamente desde el cliente para este intercambio de mensaje. Si la operación es exitosa, el servidor debe responder con 204 *No Content Reply.*

6.1.4. Cerrando el intercambio de mensaje

El paso final en un intercambio de mensaje normal es cerrarlo o hacer *commit*. Esta operación tiene esencialmente la misma semántica para intercambio de mensajes que para el *commit* de una transacción de base de datos. Una transferencia de mensaje debería ser cerrada si y sólo si el cliente ha recibido exitosamente un mensaje o ha transferido exitosamente un mensaje al servidor. Si, por cualquier razón, el cliente no está seguro que una operación GET o PUT fue exitosa, el cliente no debe cerrar el intercambio de mensaje ya que esto podría potencialmente resultar en pérdida de mensajes.

Los intercambios de mensajes son cerrados usando la operación HTTP DELETE con un *<message-exchange-location>* objetivo.

Si la *<message-exchange-location>* representa un intercambio de mensaje en el estado *Creado o GET*, el servidor debe tomar las acciones apropiadas para prevenir cualquier acción posterior sobre este intercambio de mensaje, y responder con un 204 *No Content Reply,* liberando cualquier recurso guardado para el intercambio de mensaje.

Haciendo un resumen de propiedades se tiene que para el protocolo **RRMTP:**

- Uso de URL de intercambio.
- Se garantiza que la transferencia se realice una sola vez.
- Generación de un ID(MXID) el cual debe ser único para cada transferencia.
- Se define tiempo de vida para la transferencia, por defecto es muy pequeño, y se aloja en el servidor.
- Uso de HTTPS en 1 y 2 sentidos.
- No deja en claro donde los guarda(cliente o servidor), pero usa estados para las transferencias.
- Posee implementación de referencia.

6.2. HTTPLR

Protocolo creado por Bill de hÓra, ingeniero inglés, basado en la arquitectura REST. Se presenta un análisis de la versión publicada en Febrero del año 2005. Al igual que el protocolo anterior se detallará el proceso de subida de mensajes al servidor como de obtención de mensajes, las cuales son las únicas 2 opciones.

6.2.1. El protocolo de subida

6.2.1.1. Paso uno: Establecer la URL de intercambio

El estado del intercambio de mensajes es coordinado a través de un recurso compartido llamada la "URL de intercambio". Este recurso es distinto del mensaje intercambiado. La URL de intercambio debe ser única. Esto implica que una URL de intercambio no debe ser reciclada.

Una petición-respuesta intercambiada entre cliente y servidor a una URL bien conocida establece la URL de intercambio. Cómo el cliente y el servidor determinan la URL no es especificado. El cliente debe iniciar el intercambio usando POST. Si el servidor está dispuesto a aceptar la petición de intercambio, éste debe usar el código de respuesta 201 *Created.* El identificador provisto por el servidor debe ser una URL, la cual debe aparecer en la cabecera *Location* de la respuesta. Esta es la URL de intercambio.

El servidor podría retornar una representación (entidad body) en la respuesta. En el caso que una petición de apertura falla, el cliente podría repetidamente pedir una URL de intercambio hasta que reciba una respuesta.

Después de recibida la petición y antes de enviar la respuesta, el servidor debe mantener el estado de la URL de intercambio consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/created.

Después de recibida la respuesta, el cliente debe mantener el estado de esta URL consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/created.

6.2.1.2. Paso dos: enviar el mensaje

El cliente podría usar o PUT o POST para enviar su mensaje a la URL de intercambio entregada por el servidor. El cliente podría usar una petición HEAD para determinar qué métodos están soportados a cualquier punto del intercambio. El servidor debe soportar peticiones HEAD contra la URL de intercambio a cualquier punto del intercambio.

Dada la opción, el cliente debería preferir PUT sobre POST. El servidor debe soportar el envío de la opción POST y debería soportar PUT.

La petición del cliente debe contener una entidad body(el mensaje). La respuesta del servidor debería incluir una cabecera *Location* nombrando la URL de intercambio en su respuesta. La respuesta del servidor debe incluir una cabecera *Allow* indicando cuales métodos el cliente podría usar para continuar el intercambio.

Después de recibir la petición y antes de enviar la respuesta, el servidor debe mantener el estado de esta URL de transferencia consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/accepted.

Después de recibida la respuesta, el cliente debe mantener el estado de esta URL consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/accepted.

El cliente no debe enviar mensajes a una URL que ha grabado como URL/httplr/state/accepted.

6.2.1.3. Paso tres: Mensaje de reconciliación

Hasta este punto el servidor sabe que el mensaje fue enviado a él, pero no sabe si el cliente está de acuerdo que éste ha sido enviado(y no sabe por cierto si el cliente recibió la respuesta). El cliente debe informar al servidor con una petición DELETE o POST que está de acuerdo que el mensaje fue entregado, notar que el servidor tendrá que indicar qué métodos están soportados. Dadas las opciones, el cliente debería preferir DELETE.

La petición de reconciliación del cliente no debe contener una entidad body. En el caso que el método POST es usado para ambos, la petición de entrega y de reconciliación, el servidor debe usar la ausencia de la entidad para distinguir el orden de las peticiones. Cuando el cliente recibe la respuesta del servidor, éste podría liberar estados grabados de la URI de intercambio.

La respuesta del servidor debería contener una cabecera *Location* para indicar la URL de intercambio. Al recibir una petición de reconciliación el servidor debe responder a posteriores peticiones con el código de respuesta 410 *Gone*.

Después de recibir la petición y antes de enviar la respuesta, el servidor debe mantener el estado de esta URL de transferencia consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/finished.

Después de recibida la respuesta, el cliente debe mantener el estado de esta URL consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/finished.

6.2.2. El protocolo de descarga

6.2.2.1. Paso uno: Establecer URL alimento(feed)

El cliente cuando descarga mensajes necesita determinar o bien la URL del mensaje a descargar o una lista de tales URL's. Estas URL's son llamadas “URL's de mensaje”. Ellas son encontradas en otra URL llamada “URL de alimento”. Cómo el cliente descubre la URL de alimento de un servidor no es especificado.

Para obtener la lista de los mensajes disponibles, el cliente debe usar una petición GET contra la URL de alimento. El servidor debe al menos soportar el formato Atom syndication[15] para publicar los mensajes disponibles. El cliente debe ser capaz de procesar el formato Atom syndication a lo menos.

6.2.2.2. Paso dos: Descargar el mensaje

El cliente debe usar GET para recolectar un mensaje desde la URL de mensaje entregada por el servidor. La respuesta del servidor debe contener una cabecera *Location* para indicar la URL de intercambio. La URL de mensaje de la cual la representación del mensaje es recolectada debe ser distinta de la URL de intercambio que será usada por el cliente para verificar la recepción del mensaje.

Antes de recibir la petición GET sobre la URL de mensaje, el servidor debe mantener el estado de esta URL de transferencia consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/created.

Después de recibir la petición y antes de enviar la respuesta, el servidor debe mantener el estado de esta URL de transferencia consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/accepted.

Después de recibida la respuesta con el mensaje, el cliente debe mantener el estado de esta URL consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/accepted.

6.2.2.3. Paso tres: Mensaje de reconciliación

Una indicación del cliente al servidor que éste ha aceptado el mensaje, fue recolectado exitosamente. Hasta este punto el servidor sabe que el mensaje fue enviado al cliente, pero no sabe si el cliente aceptó que éste ha llegado(y tampoco sabe si el cliente recibió la respuesta). El cliente debe informar al servidor con una petición DELETE o POST a la URI de intercambio que éste ha aceptado que el mensaje fue entregado.

El servidor debe soportar las opciones DELETE y POST. El servidor debe proveer

la URL de mensaje en la cabecera *Location* de la respuesta. El cliente debería preferir la opción DELETE.

Después de recibir la petición y antes de enviar la respuesta, el servidor debe mantener el estado de esta URL de transferencia consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/finished.

Después de recibida la respuesta con el mensaje, el cliente debe mantener el estado de esta URL consistentemente: Guardar el estado de la URL de intercambio como URL/httplr/state/finished.

En el caso que el servidor no está respondiendo como se espera o hay una falla en la red, el cliente podría reenviar la reconciliación en la ausencia de respuesta.

Una vez que el servidor ha grabado una URL de intercambio en el estado *finished* éste debe responder a cualquier petición posterior a la URL de mensaje con la respuesta 410 *Gone* y no debe retornar una entidad en la respuesta.

El cliente que ve un código de respuesta 410 para una URL de mensaje debería aceptar que el mensaje fue intercambiado y no debería enviar posteriores peticiones de reconciliación al servidor.

Finalmente detallamos una lista de las características de **HTTPLR:**

- Transferencia se realiza una y sólo una vez.
- Usa URL de intercambio, la cual debe ser única, además de indicar el estado de la transferencia y es distinta de la URL donde se guardará el archivo.
- Uso de identificador único para cada transferencia.
- Timeout o retries no los estandariza y los deja a elección del cliente.
- Uso de URL Feed en la cual el servidor lista los recursos que pueden ser descargados.
- Tiene la opción de poder consultar en cualquier punto de la transferencia cuales son los comandos aceptados en cada paso.
- No especifica como un cliente llega a la URL Feed del Servidor.
- Establece que todos los recursos sean visibles para cualquier cliente.

6.3. WS-Reliable Messaging

Este protocolo fue establecido en conjunto por las empresas BEA, IBM, Microsoft y TIBCO Software. Se diferencia de las anteriores por no usar una arquitectura REST en su diseño. Por el contrario WS-RM hace uso del protocolo SOAP, la cual establece el uso de un esquema XML particular para la comunicación entre los componentes.

El presente protocolo, por estar auspiciado por las grandes firmas tecnológicas actuales, y ante la falta de competencias reales en la industria, se ha transformado en el protocolo de mayor uso para la comunicación de servicios Web.

6.3.1 XML Namespace

Para este protocolo el namespace de XML para la implementación de esta especificación es el siguiente:

http://schemas.xmlsoap.org/ws/2005/02/rm

El prefijo de namespace "wsrm" es usado en esta especificación para esta URI.

6.3.2 Definiciones

Las siguientes definiciones son usadas en la especificación:

- **Endpoint**: Una entidad, recurso o proceso referenciable donde los mensajes del servicio Web son originados o enviados.
- **Application Source**: el endpoint que envía el mensaje.
- **Application Destination**: el endpoint al cual el mensaje es enviado.
- **Delivery Assurance**: la garantía que la infraestructura de mensajería en la entrega de un mensaje.
- **RM Source**: el endpoint que transmite el mensaje.
- **RM Destination**: el endpoint que recibe el mensaje.
- **Send**: el acto de enviar un mensaje el RM Source para entrega confiable. La garantía de confiabilidad comienza en este punto.
- **Deliver**: el acto de transferir un mensaje desde el RM Destination a la Application Destination.
- **Acknowledgement**: la comunicación desde el RM Destination al RM Source indicando la recepción exitosa de un mensaje.

6.3.3 Precondiciones del protocolo

La correcta operación del protocolo requiere que un número de precondiciones deba ser establecido antes de procesar el mensaje inicial de la secuencia:

- El RM Source debe tener una referencia al endpoint que identifica únicamente el RM Destination endpoint.
- El RM Source debe tener conocimiento de las políticas del destino, si es que hay, y el RM Source debe ser capaz que generar mensajes que se adhieren a esas políticas.
- Si es requerida una transferencia segura de mensajes, entonces el RM Source y el RM Destination deben tener un contexto de seguridad.

6.3.4 Invariantes del protocolo

Durante el tiempo de vida del protocolo 2 invariantes son requeridas por correctitud:

- El RM Source debe asignar a cada mensaje confiable un número de secuencia, comenzando en valor 1 e incrementándose exactamente en 1 para cada subsecuente mensaje seguro.
- Cada mensaje de acknowledgement del RM Destination debe incluir el rango de los números de secuencia de todos los mensajes recibidos con éxito, y debe excluir los números de secuencia de los mensajes no recibidos aún.

6.3.5 Esquema general del protocolo

Se muestra a continuación un ejemplo para detallar el funcionamiento del protocolo de forma general. La siguiente figura presenta un caso particular de como se procesa una transferencia de mensaje.

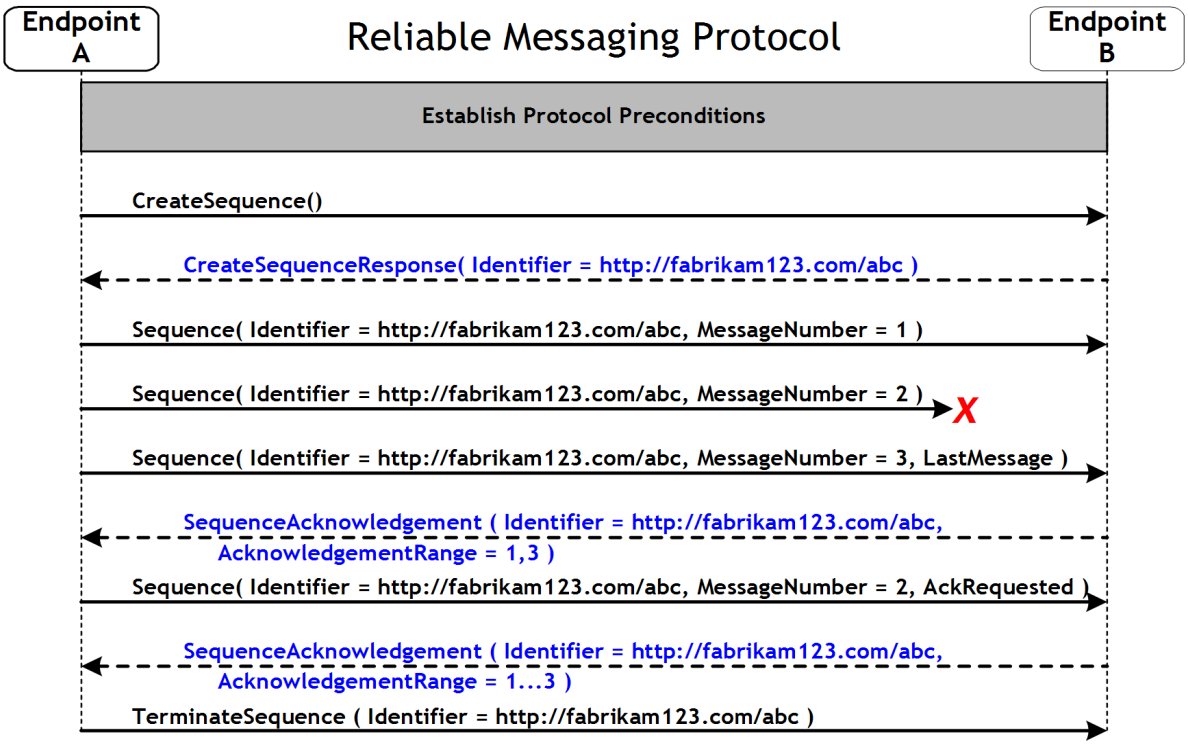


Ilustración 2 - Ejemplo protocolo WS-RM

Se detalla paso a paso lo que la figura anterior muestra:

1. Las precondiciones del protocolo son establecidas. Éstas incluyen intercambio de políticas, resolución de endpoints, establecer la confianza de ambas partes.
2. El RM Source pide la creación de una nueva secuencia.
3. El RM Destination crea una secuencia retornando el identificador único global de ella.
4. El RM Source parte enviando mensajes comenzando con el MessageNumber 1. En la figura el RM Source envía 3 mensajes.
5. Ya que el tercer mensaje es el último en este intercambio, el RM Source incluye un token `<LastMessage>` .
6. El segundo mensaje es perdido en el camino.
7. El RM Destination reconoce la llegada de los mensajes número 1 y 3 en respuesta al token `<LastMessage>` enviado por RM Source.
8. El RM Source retransmite el segundo mensaje. Este es un nuevo mensaje en el proceso, pero como tiene el mismo identificador de secuencia y número de mensaje entonces el RM Destination puede reconocerlo como equivalente al mensaje previo

en el caso en que ambos hayan sido recibidos.

9. El RM Source incluye un elemento `<AckRequest>` entonces el RM Destination enviará un mensaje acknowledgement.
10. El RM Destination recibe la segunda transmisión del mensaje con MessageNumber 2 y envía el mensaje acknowledgement de recepción de los mensajes 1, 2 y 3 el cual traía el token `<LastMessage>`.
11. El RM Source recibe el mensaje acknowledgement y envía un mensaje TerminateSequence al RM Destination indicando que la secuencia está completa.
12. El RM Destination recibe el mensaje TerminateSequence indicando que el RM Source no enviará más mensajes.

En general este protocolo no hace uso de las características propias del protocolo HTTP, puesto que fue diseñado pensando en ser independiente del protocolo de transferencia, luego, basa todo su manejo en la información transferida. Hace uso del esquema XML señalado anteriormente por lo que la cantidad de información adicional a lo que se quiere transmitir es considerable. Si bien XML permite que la información sea de más fácil lectura para una persona, por otro lado, el proceso de parseo de la información y posterior toma de datos sea más lenta.

En el apartado de anexos se mostrarán las estructuras XML involucradas en el procesamiento de las peticiones usando este protocolo, de forma de dejar en claro la completa forma de poder implementar el protocolo, teniendo que implementar clientes y servidor exclusivos, sin la posibilidad de reutilizar con un costo menor tecnologías asociadas que participan del proceso, por ejemplo algún cliente HTTP normal.

7. Diseño del sistema

En el presente capítulo se especifica en detalle el diseño del protocolo propuesto. Se ha escogido la arquitectura REST para ello por las buenas propiedades que esta presenta, no se quiere establecer que sea mejor o peor que otras arquitecturas de servicios Web, como SOAP por ejemplo, sólo se escoge al ver que las propiedades que posee muestran ser atractivas y en un actual auge en la industria actual. Uno de los principales ejemplos de este auge a gran escala es el servicio que brinda la empresa Amazon.com, el cual en estos momentos posee implementados ambos enfoques en su portal de ventas.

Dentro de las principales ventajas de REST está su interoperabilidad, la cual permite que distintos sistemas puedan interactuar con el menor costo posible para cada empresa en términos de adaptación de los sistemas involucrados. Esto gracias a la manera natural de referenciar los recursos, a través de URL de lectura directa y de fácil procesamiento. Otra de las ventajas importantes es su interfaz uniforme para operar sobre los recursos(métodos GET, PUT, POST, DELETE) lo que permite que se puedan utilizar herramientas estándar como navegadores WEB, clientes HTTP ya existentes, teniendo que respetarse la semántica de cada uno de los métodos simplemente para mantener los sistemas operando normalmente.

Se procede a detallar entonces el diseño del sistema propuesto, denominado Restfull Reliable Transfer Protocol, en adelante RRTP, sistema de transferencia de archivos de forma confiable, el cual usa el estándar HTTP/1.1 que es descrito en el RFC 2616[17].

7.1. Terminología

Las palabras claves “DEBE”, “NO DEBE”, “REQUERIDO”, “SERÁ”, “NO SERÁ”, “DEBERÍA”, “NO DEBERÍA”, “SERÍA” y “OPCIONAL” en este documento tienen que ser interpretadas como está descrito en el RFC 2119[18].

7.2. Componentes

Componentes son las entidades que forman parte del sistema y en conjunto lo definen completamente. Para el sistema RRTP se tiene 4 componentes: Recurso, Método, Servidor, Cliente, Recurso y Transacción.

7.2.1. Recurso

El recurso es la componente que representa a todo archivo que sea insertado en el sistema RRTP. No se hacen restricciones sobre el tipo de archivo o su tamaño. Si bien el

sistema está orientado para su uso como servicio Web se consideran archivos de tamaño, en general, no mayor a 50 Mb, como son boletas electrónicas, facturas, certificados, etc. Se deja en claro que igualmente no hay restricción en el tamaño de archivos, pero archivos de mayor tamaño son claramente más susceptibles a caer en problemas de red en medio de la transferencia en casos que la red se presente inestable.

7.2.2. Método

Los métodos aceptados dentro del sistema son los métodos HTTP: GET, PUT, POST y DELETE, los cuales se implementan de acuerdo al RFC 2616[17]. Se tiene en cuenta la idempotencia de los métodos GET, PUT y DELETE, y además se considera la implementación de POST de acuerdo a lo señalado en el ítem Confiabilidad. No se consideran otros métodos de forma de mantener una interfaz uniforme de operación.

7.2.3. Servidor

El servidor es la componente encargada del procesamiento de las peticiones que llegan a él y de la persistencia de los recursos. El servidor recibe los archivos y los guarda en un medio físico que garantice su persistencia. Está encargado de verificar la autenticidad de los clientes que ingresan, de recibir los recursos subidos, y enviar los recursos pedidos por los clientes. El servidor debe soportar el protocolo de seguridad HTTPS como se señala en el ítem de seguridad.

El servidor debe enviar un mensaje de confirmación por cada acción realizada, sea exitosa o no exitosamente ejecutada. El servidor debe generar un identificador único para cada transacción solicitada por los clientes. El servidor debe asegurar la idempotencia de los métodos HTTP GET, PUT, DELETE e implementar POST de acuerdo a lo señalado en el ítem de confiabilidad.

7.2.4. Cliente

El cliente es la componente encargada de comenzar las transacciones de recursos con el servidor. El cliente, por cada petición realizada al servidor, debe esperar el correspondiente mensaje de confirmación, de no recibirlo, la petición debe ser reenviada las veces que sea necesario de forma de asegurar la confiabilidad del protocolo, por lo tanto se establece que sobre el cliente recae la responsabilidad de verificación de procesamiento exitoso de cada petición, teniendo que esperar siempre las confirmaciones del servidor y solicitándolas nuevamente en los casos que esto sea necesario.

El cliente debería tener acceso sólo a los recursos subidos anteriormente por él

mismo, de forma de hacer la información confidencial en los casos que esto se requiera.

7.2.5. Transacción

La transacción es la componente que representa el intercambio de recursos entre cliente y servidor. Cada transacción posee un identificador único que la representa, el cual es generado por el servidor, de forma de facilitar su unicidad cuando se trabaja con muchas transacciones. Además posee un tiempo de vida que indica el tiempo máximo que puede durar la transacción hasta completarse.

La transacción posee estados que indican en que parte del proceso de transferencia de datos se encuentra y además señala cuales métodos son válidos para un momento determinado de la transacción. Los estados para una transacción son los siguientes: *Creada*, *Descarga*, *Subida*, *Borrado, Corrupta* y *Cerrada*.

- **Creada**: la transacción posee este estado justo en el momento en que se crea en el servidor, luego de llegada la petición POST que lo requiere. Cuando una transacción está en este estado son válidos los métodos GET, PUT y DELETE. Cuando se recibe el método GET sobre una URL de recurso en una transacción *Creada*, el estado pasa a *Descarga*. Si se recibe el método GET sobre una URL que pide el listado de recursos para un cliente, sobre una transacción en estado *Creada,* esta mantiene el mismo estado. Si se recibe el método PUT en una transacción *Creada*, el estado de ésta pasa a *Subida*. Si se recibe el método DELETE sobre una URL de recurso en este estado, la transacción pasa al estado *Borrado*.
- **Descarga**: la transacción posee este estado cuando, al estar en estado *Creado,* el servidor recibe el método GET sobre una URL de recurso de parte del cliente, lo que indica que se realizará la descarga de algún recurso. Cuando una transacción está en este estado son válidos los métodos GET y DELETE. Cuando se recibe un método GET en una transacción en *Descarga*, se mantiene el estado de la transacción intacto, esto se puede dar en casos en que ha habido una caída de la red, y el cliente reenvía la petición GET para reanudar la descarga del recurso, y también se da cuando el cliente solicita listados de los recursos para buscar el que necesita descargar. Cuando se recibe un método DELETE en una transacción en *Descarga*, ésta pasa al estado *Cerrada.*
- **Subida**: la transacción posee este estado cuando, al estar en estado *Creada,* el servidor recibe el método PUT desde el cliente, lo que indica que se realizará la subida de un recurso. Al estar en este estado son válidos los métodos PUT y DELETE. Cuando se recibe un método PUT en una transacción en estado *Subida,* se mantiene el estado intacto de la transacción, esto se puede dar en casos que el

cliente reintenta la subida por eventuales caídas de la red. Cuando se recibe el método DELETE en una transacción en *Subida*, ésta pasa al estado *Cerrada.*

- **Borrado**: la transacción posee este estado cuando, al estar en estado *Creada,* el servidor recibe el método DELETE sobre una URL de recurso desde el cliente, lo que indica que se realizará la eliminación de un recurso ya existente en el servidor. Al estar en este estado es válido recibir el método DELETE del cliente. Cuando se recibe el método DELETE sobre una URL de recurso en una transacción en estado *Borrado,* quiere decir que el cliente realiza un reintento debido a posibles fallas de red, aquí se mantiene el estado *Borrado* de la transacción. Si se recibe el método DELETE sobre una URL de transacción, ésta pasa al estado *Cerrada.*
- **Cerrada**: la transacción posee este estado cuando, al estar en uno de los estados *Descarga, Subida* o *Borrado,* se recibe el método DELETE sobre una URL de transacción desde el cliente, lo que indica que la transferencia del recurso ha terminado de forma satisfactoria.
- **Corrupta**: la transacción posee este estado cuando, al estar en el estado *Descarga*, el servidor recibe un método distinto de GET y DELETE, el cual no tiene sentido para la transacción en curso. Lo mismo sucede cuando se está en el estado *Subida* y se recibe un método distinto de PUT y DELETE, lo cual no corresponde. También pasa *a Corrupta* si al estar en estado *Borrado* se recibe algún método distinto de DELETE. También una transacción pasa a estado *Corrupta* cuando esta supera su tiempo de vida sin haber llegado al estado *Cerrada,* esto se puede dar en los casos que la red ha caído y no ha habido reintentos para reanudar la transacción. Este estado finaliza la vida de la transacción indicando que hubo problemas, por lo que no se puede volver a reanudar la transacción.

A continuación se presenta un diagrama que muestra el ciclo de estados posibles para una transacción:

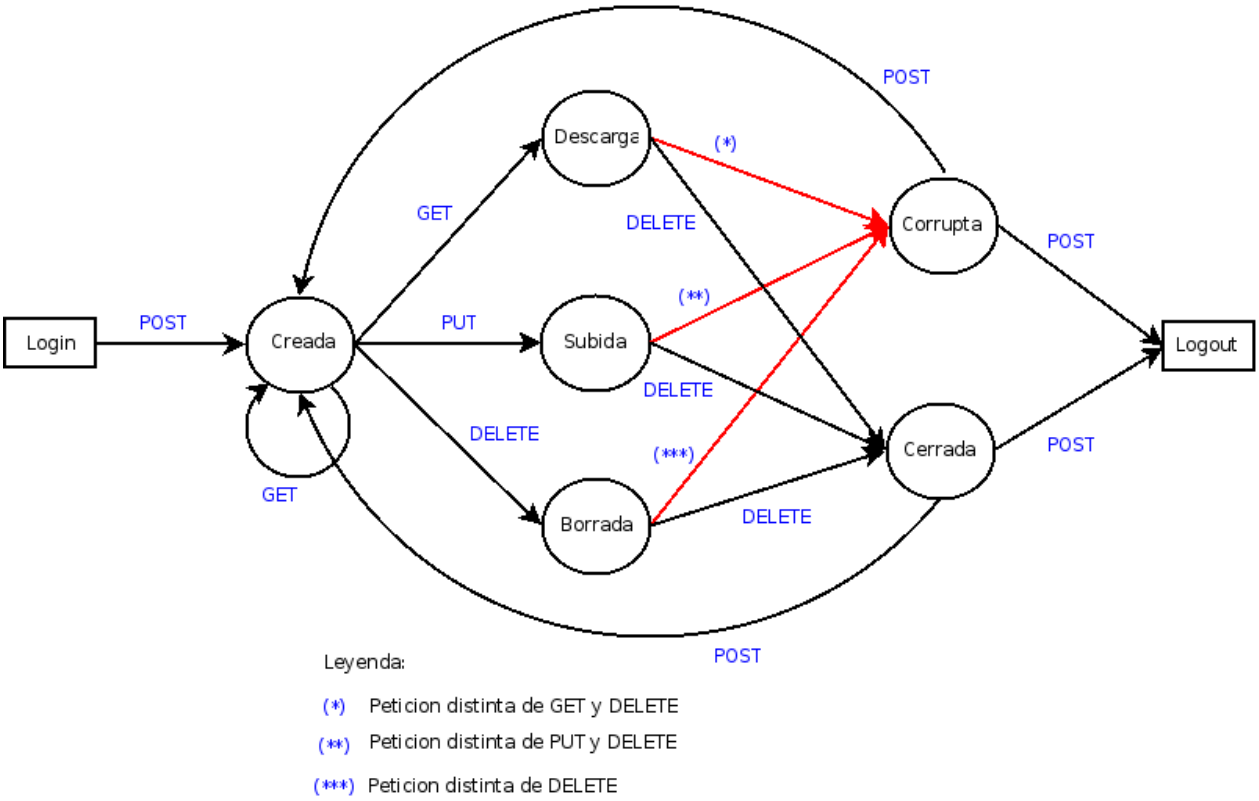


Ilustración 3 - Diagrama estados protocolo RRTP

En el diagrama los círculos representan los estados, las flechas indican los métodos HTTP que producen los cambios de estado, y los cuadrados indican los estados del cliente.

7.3. Seguridad

La característica de seguridad no forma parte directa del protocolo, pero el diseño de éste permite que el sistema sea compatible con cualquier protocolo de seguridad que quiera usarse sobre HTTP/1.1. Se recomienda el uso de **HTTPS**[13]. Para que el nivel de seguridad del sistema sea óptimo, tanto el cliente como el servidor deberían implementar HTTPS, pero basta que el servidor implemente HTTPS para que la seguridad del sistema sea aceptable.

HTTPS se vale del uso del protocolo criptográfico SSL(Secure Socket Layer)[19], el cual brinda autenticación de las partes y privacidad de la información transferida. La autenticación de las partes se obtiene a través del uso de sistemas de cifrado de claves públicas, en el cual se intercambian las claves públicas de ambas partes para la autenticación. Para garantizar la privacidad de la información SSL especifica el uso de criptografía híbrida, puesto que tiene una parte asimétrica y otra simétrica, la parte asimétrica es usada para intercambiar entre las partes la clave simétrica que usaran ambas partes para cifrar y descifrar la información. El motivo por el cual no se usa criptografía

asimétrica para autenticación y transferencia de los datos posteriores es, básicamente, por la mejor eficiencia que posee la criptografía simétrica al usar 1 sola clave en ambas partes y que es igual de segura pues esta clave se ha transferido usando criptografía asimétrica.

7.4. Confiabilidad

Para obtener la confiabilidad del protocolo, primero dejamos claro que este tema hace referencia a la confiabilidad de la transmisión que se quiera realizar, ya que el protocolo HTTP no garantiza que la información transmitida efectivamente llegó a destino.

En una transmisión de información a través de internet hay muchos intermediarios que eventualmente podrían fallar, desde los cables en algún punto del viaje, como los procesos de enrutamiento(realizado por dispositivos como Routers o Switches) de los datos, inclusive la ausencia de electricidad en algún equipo intermediario podría darse. Si el problema de red ocurre mientras quien envía el mensaje no puede completar el envío, este se da cuenta del fallo y puede reenviarlo. Pero si el problema ocurre cuando el receptor del mensaje está respondiendo hay cierta ambigüedad en lo que debe responder finalmente, éxito o falla, o si realmente recibió el mensaje correctamente.

Una forma de estar seguro que el mensaje fue recibido exitosamente, es reenviarlo hasta obtener una respuesta satisfactoria del receptor. Esto se puede lograr en este caso si es que la acción que estamos realizando es *idempotente,* lo que garantiza que es seguro repetir la acción. Los métodos HTTP GET, PUT y DELETE cumplen con ser idempotentes, lo que quiere decir que, independiente de cuantas veces realice la operación, esta siempre tendrá el mismo efecto. Para entender mejor esta situación se puede hacer una analogía entre los recursos de internet y archivos de un sistema operativo, y del método GET con la lectura de archivos, PUT con la escritura y DELETE con el borrado. Como se ve, no importa cuantas veces lea un mismo archivo, el resultado siempre será el mismo. Tampoco importa escribir un archivo muchas veces, siempre y cuando se escriba la misma información y borrar un archivo muchas veces a lo más entrega un mensaje de error luego de borrarlo efectivamente.

Por otra parte el método HTTP POST no es idempotente y puede ser comparado con la acción de añadir información a un archivo. Puede verse que adherir información a un archivo repetidas veces es distinto a sobreescribir el archivo con la misma información(PUT), puesto que la información se acumula. Por lo tanto se detallará la estrategia a seguir por el sistema para evitar que múltiples ejecuciones de la misma llamada POST generen comportamientos diferentes.

La forma de evadir este problema será insertando un identificador(ID) en un header

del mensaje enviado. Se tendrá un registro de este ID por lo que se podrá ignorar cualquier mensaje POST que viene de un ID que ya ha sido visto previamente.

La responsabilidad de generar los identificadores queda en el servidor, puesto que la generación de identificadores únicos es compleja, y múltiples clientes distintos pueden generar problemas en caso de generar ellos mismos los ID. Además que es el servidor quien manejará los ID y verificará cuando corresponda. El servidor no debe ignorar este identificador.

Esto funciona pues se usa la función POST *no segura* sólo una vez, en el momento que se hace la primera petición. Además que los identificadores son baratos de generar y su registro puede ser eliminado luego de ser usado.

El cliente no tiene que preocuparse por que accidentalmente crea 2 identificadores de intercambio, puesto que el que queda sin usar es irrelevante. Tampoco tiene que preocuparse en caso que se ejecute 2 veces una operación POST sobre una URL, ya que el servidor debe ignorar los intentos posteriores al primero, y su respuesta debe ser la misma generada en la primera petición recibida, lo que garantiza que las funciones POST se ejecutarán una y sólo una vez sobre una URL para un identificador dado, generándose con esto una especie de idempotencia que ayuda a la confiabilidad del sistema completo, sobre todo en los casos que se necesita reenviar una petición cuando la red ha fallado.

Tomando en cuenta estas consideraciones la estrategia será la siguiente: el cliente cada vez que realice una operación sobre el servidor esperará la respectiva confirmación por parte de éste, de no recibir la confirmación, el cliente reenviará la petición tantas veces sea necesario. Por el lado del servidor este al recibir una petición la procesará y enviará el correspondiente mensaje de respuesta sin esperar alguna confirmación por parte del cliente. Se deja entonces la responsabilidad de verificación de la integridad de la transacción en el lado del cliente.

7.4.1. Integridad de datos transferidos

Otra parte importante que dice relación con la confiabilidad del protocolo se refiere a la verificación de la integridad de los datos. Se establece que el servidor debe usar el algoritmo de tipo hash **MD5**[21] para verificar los datos de las transferencias. Para el caso que el servidor envía a un cliente un recurso solicitado, éste debe enviar en el header HTTP ***Content-MD5*** de la respuesta, la cadena de caracteres con el valor del hash de los bytes transferidos, en cuyo caso el cliente tiene la opción de usar este header para verificar si los bytes transferidos son correctos, el cliente debería realizar la verificación en orden de mejorar la confiabilidad del sistema. En caso que el cliente encuentre error en la verificación, debe reenviar la petición al servidor para obtener el recurso correctamente.

En el caso que el servidor recibe un recurso subido por un cliente, el servidor debe buscar dentro de la petición el header ***Content-MD5,*** que en caso de tener un valor distinto de vacío, el servidor deberá verificar la integridad de los bytes recibidos desde el cliente. En caso de obtener problemas con la verificación el servidor debe entregar un código de respuesta *500 Internal Server Error,* código que es usado para informar de errores del servidor y para pedir reenvíos del cliente y en este caso indica el error de datos corruptos.

7.5. Descripción del protocolo

Se procede a exponer en detalle, paso a paso, el funcionamiento del protocolo RRTP. Se explica el rol que cada componente juega dentro de los procesos, siempre teniendo en cuenta de cumplir con las propiedades antes señaladas.

7.5.1. Autenticación

Para la autenticación, ambas partes deben identificarse, es decir, tanto cliente ante el servidor y viceversa. Al hacer uso de HTTPS con SSL, se exige el uso de certificados de firma digital válidos, puesto que a parte de proveer seguridad al momento del ingreso al sistema, proveen seguridad al cifrar la información que viaje por la red.

Para realizar el ingreso, el cliente deberá enviar un mensaje POST a la dirección URL de login del servidor RRTP compatible. El encabezado del mensaje debe tener la siguiente estructura:

```
POST <ruta-servidor>/rrtp/login HTTP/1.1
```

Además de incluir el certificado digital que identifica al cliente. Aquí se produce el proceso *de Handshake* entre cliente y servidor, proceso en el cual ambos intercambian su respectiva llave pública, enviándola en certificados de formato X.509[22], la validan sobre su repositorio de cliente/servidor confiable, proceso que de ser exitoso, pasa a la fase de generación de una llave simétrica mediante la cual se encriptará la comunicación entre ambos puntos. Luego de lo cual, el servidor recibe la petición y verifica que exista un usuario en el sistema para esa llave pública de cliente, el servidor debe retornar el código de respuesta *200 OK* en caso de verificación satisfactoria. Si la validación del servidor no es satisfactoria, debe retornar el código *401 Unauthorized* al cliente, rechazando cualquier intento posterior de ejecución desde ese cliente, sobre URL's distintas a la de login.

Por su parte el cliente al recibir el código de respuesta *200 OK*, sabe que puede proceder a realizar operaciones sobre el cliente sin ningún problema y con la confianza de que la información viajará encriptada sobre la red.

7.5.2. Creación de transacción

El primer paso luego del login, es la creación de una transacción, que es el proceso común a cualquiera de las 3 opciones de transacción existentes(subida, bajada o borrado).

Este proceso parte con la petición del cliente para la creación de transacción, esto se traduce en una petición POST sobre la URL de creación de transacciones en el servidor RRTP. Un ejemplo del encabezado de esta petición es la siguiente:

```
POST <ruta-servidor>/rrtp/ HTTP/1.1
```

El servidor, al recibir esta petición de un cliente válido, debe generar un identificador único para la transacción (en adelante IDT). El servidor debe enviar una respuesta de código *201 Created* al cliente incluyendo el IDT para su posterior uso, luego de lo cual el estado de la transacción es *Creada*. En caso que el servidor no pueda generar el IDT debe enviar una respuesta de código *500 Internal Server Error*.

En caso que el servidor, en esta etapa, reciba un petición de tipo distinta a POST sobre la URL de creación de transacción, éste debe enviar el código de respuesta *403 Forbidden* indicando que el método recibido no puede ser ejecutado en esta etapa.

Un ejemplo de una respuesta satisfactoria para la creación de IDT del servidor RRTP sería la siguiente:

```
HTTP/1.1 201 Created
Connection: close
Date: <HTTP-date>
Server: <server-identifier>
Content-Length: 0
Location: <ruta-servidor>/rrtp/<IDT>
Expires: <HTTP-date><CRLF><CRLF>
```

Para garantizar la confiabilidad del protocolo hay que señalar que si el cliente no recibe respuesta alguna del servidor, debe insistir enviando tantas peticiones de creación de transacción como sea necesario, puesto que es barata la generación de identificadores, que de no ser usados mueren al fin del tiempo de vida de la transacción.

Luego de creada la transacción el header *Location* será usado para guardar el identificador de ésta para cada una de las peticiones posteriores del cliente y de las respuestas del servidor, siendo este header exclusivo para la transferencia del IDT.

7.5.3. Subida de recurso

La subida de recurso es una de las 3 alternativas posibles luego de creada la transacción correctamente. El cliente debe enviar un comando PUT para poder subir el archivo a una URL con la siguiente estructura: <ruta-servidor>/rrtp/AAAA/MM/DD/file.ext. Donde AAAA indica el año actual, MM el mes, DD el día en que se sube el recurso y file.ext es el nombre del archivo a subir. Un ejemplo del encabezado de esta petición es:

```
PUT <ruta-servidor>/rrtp/2007/08/22/file.ext HTTP/1.1
```

El cliente debe incluir en el cuerpo del mensaje el contenido del recurso a subir. La fecha de la URL debe ser la fecha actual al momento de realizada la petición por el cliente. El servidor debe validar esta fecha, y debe aceptar un error de a lo más un día de diferencia para los casos límite en que se inicia una subida de archivo en un horario próximo al cambio de fecha, llegando al lado del servidor cuando se ha producido este cambio. En este caso la fecha del cliente es la que manda, pues es la fecha en la que éste sabe que envió el archivo. En caso que la validación de fecha arroje error por tratarse de fechas muy distantes a la actual del servidor, éste debe retornar el código de error *400 Bad Request.* El cliente puede incluir el header *Content-MD5* con el valor del hashing de los datos transferidos para que el servidor pueda realizar la verificación de integridad de datos.

El servidor luego de recibir la petición debe marcar el estado de la transacción en *Subida.* Para cada nuevo archivo subido el servidor debe enviar en la respuesta el header *Content-Location* con la URL en que queda guardado el recurso. Además el servidor debe verificar la existencia del header *Content-MD5* y proceder a realizar la validación de integridad en caso que corresponda, en caso que la validación no sea satisfactoria, el servidor debe enviar el código de respuesta *500 Internal Server Error,* el cual le indica al cliente que debe realizar el reintento en el envío de los datos. En caso de recibir satisfactoriamente la petición con su recurso respectivo, el servidor debe responder al cliente con el código de respuesta *201 Created.* Un ejemplo de la respuesta para el caso válido de un servidor RRTP sería:

```
HTTP/1.1 201 Created
Location:<ruta-servidor>/rrtp/IDT
Content-Location:<ruta-servidor>/rrtp/2007/08/22/file.ext
Expires: <HTTP-date><CRLF><CRLF>
```

En caso que el servidor no reciba satisfactoriamente el recurso, debe solicitar al cliente la retransmisión de los datos enviándole el código de respuesta *500 Internal Server Error.* Se permiten tantos reintentos como sean necesarios dentro del tiempo de vida de la transacción, si este tiempo de vida es superado, el servidor debe marcar el estado de la transacción en *Corrupta,* terminando de esta forma la transacción. Luego de esto cualquier método que el cliente intente realizar sobre la esa IDT debe ser respondido por el servidor con el código de respuesta *401 Unauthorized.*

El cliente al recibir la respuesta satisfactoria del servidor debe proceder a cerrar la transacción normalmente, para ello debe enviar al servidor el método DELETE sobre la url con la siguiente estructura: <ruta-servidor>/rrtp/IDT. El servidor al recibir el método DELETE en la url señalada, sabe inmediatamente la transacción involucrada por lo que procede a cambiar su estado a *Cerrada* y enviándole al cliente la confirmación de que se ha cerrado la transacción satisfactoriamente con el código de respuesta *200 OK.* En este punto el cliente de no recibir respuesta del servidor debe realizar los reintentos necesarios para poder cerrar la transacción satisfactoriamente, al igual que si recibe el código de respuesta *500 Internal Server Error*, el que siempre indica al cliente que debe reenviar la petición actual.

7.5.4. Descarga de recurso

La descarga de un recurso es la segunda alternativa posible a realizar luego de creada la transacción. El cliente podría no saber la URL exacta del recurso que desea descargar, para obtener esta información, el cliente puede hacer consultas al servidor a través de métodos GET a las siguientes URL's:

<ruta-servidor>/rrtp/ : entrega lista completa de recursos.

<ruta-servidor>/rrtp/AAAA/: entrega lista de recursos para el año señalado

<ruta-servidor>/rrtp/AAAA/MM[/DD]: entrega lista de recursos para el mes o día si es que se especifica.

El servidor debe implementar el formato Atom syndication[20] como mínimo para la entrega de las listas de recursos que el cliente puede descargar. Un ejemplo de la petición GET del cliente y la posterior respuesta del servidor sería la siguiente:

Cliente:

```
GET <ruta-servidor>/rrtp/2007/08 HTTP/1.1
```

Servidor:

```
 HTTP/1.1 200 Ok
 Content-Type: application/x.atom+xml
 [crlf]
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
    <title>Lista de recursos</title>
    <subtitle>Recursos para usuario</subtitle>
    <link href="URL_SERVER/rrtp/"/>
    <author>
            <name>Client Name</name>
    </author>
    <entry>
            <title>video.avi</title>
            <link href="URL_SERVER/rrtp/2007/08/01/video.avi"/>
            <id>7</id>
            <updated>2007/08/01</updated>
            <summary>3545992 bytes</summary>
    </entry>
    <entry>
            <title>P-08:31:2007-16:42:38.jpg</title>
            <link href="URL_SERVER/rrtp/2007/8/20/P-08:31.jpg"/>
            <id>6</id>
            <updated>2007/08/20</updated>
            <summary>19688 bytes</summary>
    </entry>
</feed>
```

El cliente debe como mínimo poder procesar los listados de recursos en el formato Atom syndication. Como se aprecia el código de respuesta del servidor es un código *200 OK.* En caso de que no se encuentren recursos para el usuario en la fecha consultada se debe enviar el mismo código de respuesta anterior, en conjunto con el xml que sólo posee encabezados y ninguna entrada correspondiente a recursos.

Luego de tener la lista de todos los recursos disponibles para descargar, el cliente procede a descargar uno de ellos realizando una petición GET sobre la url que apunta al recurso, en el ejemplo anterior una URL válida de petición sería:

```
GET <ruta-servidor>/rrtp/2007/08/01/video.avi HTTP/1.1
```

Además el cliente, como se señaló anteriormente, en el header *Location* de la petición debe ingresar el IDT que ha sido creado por el servidor para la transacción en curso. Si el recurso efectivamente existe y la transacción estaba en estado *Creada* o *Descarga,* el servidor envía al cliente con el código de respuesta *200 OK,* enviando en el cuerpo de la respuesta el recurso pedido por el cliente. El servidor actualiza el estado de la transacción a *Descarga.* El cliente puede repetir la petición en caso de no recibir respuesta

del servidor y en pos de garantizar la confiabilidad del protocolo. También puede repetir la petición en caso de recibir el código de respuesta *500 Internal Server Error.*

En caso que la URL no sea válida, el servidor debe responder con el código de respuesta *404 Not Found.* lo que le indicará al cliente que debe enviar una nueva la petición con una URL de recurso válida, igualmente debe marcar el estado de la transacción como *Descarga*, para que el cliente realice los reintentos necesarios.

El cliente al recibir la respuesta satisfactoria del servidor debe proceder a cerrar la transacción normalmente, para ello debe enviar al servidor el método DELETE sobre la url con la siguiente estructura: <ruta-servidor>/rrtp/IDT . El servidor al recibir el método DELETE en la URL señalada, sabe inmediatamente la transacción involucrada por lo que procede a cambiar su estado a *Cerrada* y enviándole al cliente la confirmación de que se ha cerrado la transacción satisfactoriamente con el código de respuesta *200 OK.* En este punto el cliente de no recibir respuesta del servidor debe realizar los reintentos necesarios para poder cerrar la transacción satisfactoriamente, al igual que si recibe el código de respuesta *500 Internal Server Error*, el que siempre indica al cliente que debe reenviar la petición actual.

7.5.5. Borrado de recurso

El borrado de un recurso es la tercera alternativa posible de ejecutar luego de creada la transacción. Al igual que en el proceso de descarga de un recurso, el cliente puede consultar al servidor por los recursos existentes a través de peticiones GET con la estructura antes señalada. Tanto cliente como servidor deben a lo menos soportar el formato Atom syndication para la comunicación en este punto.

Luego de obtenidos los recursos existentes el cliente procede a enviar la petición de borrado del recurso enviando un método DELETE al servidor sobre la URL completa que apunta al recurso. Un ejemplo de petición de borrado de un cliente RRTP sería la siguiente:

```
DELETE <ruta-servidor>/rrtp/2007/08/01/file.ext HTTP/1.1
```

Además el cliente en el header *Location* de la petición debe ingresar el IDT que ha sido creado por el servidor para la transacción en curso, como se ha señalado anteriormente. En caso de existir el recurso el servidor lo debe borrar físicamente de su sistema, sin tener el cliente posibilidad de recuperarlo, además el servidor debe marcar el estado de la transacción como *Borrada* y enviar una respuesta al cliente con el código de retorno *204 No Content* de acuerdo al estándar HTTP/1.1. En caso de que el recurso solicitado para ser borrado no existe en el servidor, se le debe enviar al cliente el código de respuesta *404 Not Found*, marcando igualmente la transacción en estado *Borrado*, para

darle la opción al cliente de que efectúe el reintento sin ningún problema.

El cliente debe esperar la respuesta de confirmación del servidor, en caso de una respuesta satisfactoria(código 204) el cliente debe cerrar la transacción enviando una petición DELETE sobre la URL que incluye el IDT de la transferencia actual. El servidor al recibir esta petición marca el estado de la transacción en *Cerrada,* terminando la transacción en curso. En caso que el cliente reciba el código de respuesta 404, este puede volver a enviar una petición GET para obtener la lista de recursos disponibles y sus URL's respectivas e intentar nuevamente borrar el recurso con la petición DELETE correspondiente.

7.5.6. Salida del sistema

Luego de que la transacción ha llegado a un estado final, es decir, o *Cerrada o Corrupta,* el cliente tiene sólo un comando HTTP válido, POST, el cual puede ser usado tanto para crear una nueva transacción, en caso de querer comenzar una nueva transferencia, o simplemente para salir del sistema. Para el primer caso basta con realizar la petición con la estructura que se señaló en el ítem *3.5.2. Creación de Transacción.*

Para salir del sistema basta con que el cliente envíe una petición POST a la URL de salida del servidor RRTP. Un ejemplo del encabezado de una correcta petición de logout al servidor es la siguiente:

```
POST <ruta-servidor>/rrtp/logout HTTP/1.1
```

El servidor luego de recibir la petición de logout debe eliminar los datos del usuario activo y enviar la respuesta de código *200 OK*. El cliente al recibir la respuesta termina su ejecución. En caso límite de que no reciba la respuesta enviada por el servidor, puede deberse a que el servidor no recibió la petición, caso en que se reenvía y espera por respuesta. El caso en que hay que tener cuidado es cuando el servidor recibe la petición y saca del sistema al usuario, pero la respuesta del servidor no llega al cliente, en este caso el cliente al reintentar el servidor rechazará sus peticiones por no estar logeado, por lo que sólo en este caso el servidor responderá con el código de error *410 Gone,* el cual el cliente debe interpretar con que ya fue deslogeado del sistema satisfactoriamente en una petición previa, por lo que termina su ejecución normalmente.

7.5.7. Tabla resumen métodos del protocolo

Se presenta a continuación una tabla en la que se muestran las funciones descritas en el protocolo, mostrando el método HTTP, la URL sobre la cual se aplica y un detalle sobre la función específica que se ejecuta en el servidor RRTP:

Método HTTP	URL	Función ejecutada
POST	http://server/rrtp/	Creación de transacción
GET	http://server/rrtp/	Listado de recursos
DELETE	http://server/rrtp/IDT	Cierra transacción
POST	http://server/rrtp/login/	Negocia ingreso
POST	http://server/rrtp/logout/	Negocia salida
GET	http://server/rrtp/AAAA/MM/DD	Lista recursos filtrados
GET	http://server/rrtp/AAAA/MM/DD/FILE	Descarga recurso
PUT	http://server/rrtp/AAAA/MM/DD/FILE	Sube recurso
DELETE	http://server/rrtp/AAAA/MM/DD/FILE	Borra recurso

Tabla 1 - Resumen protocolo RRTP

En la tabla IDT es el identificador de la transacción como señaló anteriormente. Para los métodos GET que entregan listado de recursos, estos se entregan en el formato Atom syndication.

7.6 Comparación de diseños

Se procede a realizar una comparación entre el diseño propuesto y las 2 alternativas de diseños REST existentes. Se excluye de esta comparación el protocolo WS-RelibleMessaging por poseer una arquitectura distinta y cuyas diferencias ya fueron señaladas previamente.

	RRMTP	**HTTPLR**	**RRTP**
Transferencia se realiza una y solo una vez	OK	OK	OK
Uso de URL de Intercambio	OK	OK	OK
Identificador único por transferencia	OK	OK	OK
Tiempo de vida para transferencia	Definido en servidor, numero fijo para todas las transferencias	Se contempla, pero no se estandariza	Dependerá de la magnitud de datos a transferir
Uso de HTTPS	OK	NO	OK
Uso de URL Feed para obtener archivos	No, no señala como obtener archivo subido, implementación no lo permite.	OK	OK
Uso de estados en la transferencia	OK	OK	OK
Uso de sistema de autenticación	OK	No, solo propone alternativas	OK(certificados)
Implementación de referencia	OK, pobre	NO	OK

Tabla 2 - Comparación protocolos REST

En el caso del protocolo RRMTP, este hace uso de identificadores únicos de transferencia, el cual podría ser obtenido a través de distintos medios por el cliente, incluso a través de correo electrónico, se señala como ejemplo, lo cual afecta considerablemente la seguridad del sistema, y le hereda los problemas de seguridad existentes actualmente en el sistema de correo electrónico.

Otro punto a considerar, es que ninguna de las 2 alternativas al protocolo propuesto considera la posibilidad de borrar un recurso subido, dejándose esta opción olvidada completamente, lo cual es completamente normal de querer realizar en un protocolo de transferencia de archivos que tiene una arquitectura de cliente-servidor.

Además que, de ambas propuestas, sólo una (RRMTP) posee una implementación de referencia, la cual se analizó encontrando una pobre funcionalidad y una grave falta al no poder obtener una referencia a los recursos subidos por el cliente, lo cual hace que tenga poca utilidad y que su masificación no haya sido para nada buena.

Se ve que el protocolo propuesto (RRTP) cumple con todas las características señaladas, si bien la diferencias no son radicales, lo más significativo del nuevo protocolo es su completa implementación de referencia, la cual se procede a detallar posteriormente. Además de considerar las propiedades esenciales para su adopción en ambientes serios, como son la confiabilidad y la seguridad en la transmisión de la información.

8. Implementación protocolo RRTP

En el presente capítulo se procede a detallar la implementación realizada del protocolo RRTP. Para realizar esta implementación se ha seguido estrictamente el protocolo presentado en el capítulo 7, implementándolo a cabalidad. Se presentarán las tecnologías usadas en la implementación, para luego explicar por separado el diseño que poseen tanto el servidor como el cliente.

Los principios que se tuvieron en cuenta para la selección de las tecnologías fueron básicamente, que fueran tecnologías de libre acceso y de uso popular en el actual mundo de los servicios Web. Esto es principalmente con el objetivo que favorecer la adopción de la implementación de la forma menos dolorosa para quien quiera usarlo, puesto que el uso de tecnologías populares y sólidas a la vez permite que, en casos de complicaciones, a parte de tener la documentación directa del protocolo, se puede acceder a múltiples sitios y foros relacionados con el tema, donde obtener ayuda es un asunto bastante sencillo y a su vez rápido.

Al ser un protocolo referente a un servicio Web es conveniente que las tecnologías que se utilicen sean independientes del sistema operativo donde, tanto el servidor como el cliente, sean ejecutados, no siendo el sistema operativo un restricción en este caso.

8.1 Tecnologías seleccionadas

8.1.1 Lenguaje de programación

El lenguaje de programación seleccionado para la implementación tanto del cliente como del servidor es Java, haciendo uso del Java Development Kit en su versión 1.5, de la empresa Sun Microsystems.

Siendo seleccionado ya que posee la característica de ser multiplataforma, con lo cual se tiene la independencia del sistema operativo. Otra de las características relevantes para que fuera seleccionado es su popularidad por ser un lenguaje ampliamente aceptado en el mundo tecnológico ya que posee propiedades que hacen la vida del programador más fácil. Además que su popularidad produce que la documentación existente sea abundante en la Web.

8.1.2 Servidor de aplicaciones

En este ítem se seleccionó un servidor con la capacidad de procesar sistemas desarrollados con el lenguaje Java. Se seleccionó el servidor Apache Tomcat[11] en su versión 5.5.23.

Esta selección se rige por los mismos principios señalados anteriormente, su libre acceso y su gran popularidad, siendo uno de los servidores de aplicaciones más usados hoy en día. Además que es el núcleo de muchos otros servidores de aplicaciones, por lo que un sistema desarrollado para Tomcat es directamente compatible con una amplia gama de servidores de aplicaciones derivados de éste, favoreciendo así la portabilidad del sistema desarrollado.

8.1.3 Certificados digitales

Para la generación de los certificados digitales se han usado varias herramientas que se detallan a continuación:

- Openssl[23]: es una librería que implementa de manera completa el protocolo SSL, proveyendo de las herramientas necesarias para aplicaciones tanto comerciales como no comerciales, entregando las herramientas como código abierto.
- Keytool: es una librería de Java que viene incluida por defecto en la versión de este lenguaje usada. Provee del manejo de certificados digitales y la creación de repositorios de llaves, tanto propias como de entidades en las que se confía.
- Librería Jetty[24]: Jetty es un servidor web de código abierto, con todas las características propias de un servidor implementadas enteramente en el lenguaje Java. Se ha usado de este servidor una clase llamada PKCS12Import, para realizar el correcto traspaso de certificados generados por openssl a los repositorios generados por keytool.

8.1.4 Cliente HTTP

Con el fin de mostrar que la implementación del protocolo es portable con un costo bajo, se hizo uso de un cliente HTTP construido en Java, el cual provee las funcionalidades básicas para realizar las consultas necesarias del protocolo. Este cliente es el perteneciente al proyecto Jakarta HTTP Components[25], cuyo nombre es HTTPClient[26], en su versión 3.0.1. Este cliente es el comúnmente usado para construir

aplicaciones que operan sobre HTTP, contando con el respaldo de pertenecer al proyecto Jakarta.

8.1.5 Persistencia de datos

Para el caso de la persistencia de los datos se escogió un motor de base de datos sólido y reconocido por la comunidad en general, este fue el motor Postgresql[27] en su versión 8.2. Si bien el protocolo RRTP propuesto es independiente de cómo se guarde la información en el lado del servidor, para así dejar libertad en ese sentido, la implementación que aquí se presenta hace uso de un motor de base de datos relacional.

8.2 Servidor

Se detalla a continuación la implementación realizada de un servidor HTTP compatible con el protocolo RRTP haciendo uso de las tecnologías señaladas anteriormente.

Como se señala en la especificación del protocolo, la implementación del servidor se debe guiar por obtener las características de ser confiable y seguro, teniendo esto en cuenta el diseño del servidor debe considerar siempre las posibilidades de error en cualquier punto del procesamiento y resguardar la seguridad de los datos apoyados con el uso de SSL.

8.2.1 Arquitectura

La arquitectura del servidor cuenta, a grandes rasgos, con 2 capas, una capa lógica, la cual está encargada del procesamiento de las peticiones y todo lo que involucra el verificar qué se pide, identificar la transacción y usuario involucrado y generar las respuestas respectivas. La segunda capa es una capa física que se encarga de la persistencia de los datos, tanto guardar, obtener como borrar los datos involucrados. Los datos los recibe de la capa lógica y los envía a esta capa de acuerdo a la función que se esté realizando.

En la arquitectura, por tratarse de un sistema que opera como servicio Web, no requiere de una capa visualización en el lado del servidor, ya que la única interfaz de comunicación con un cliente es a través de peticiones HTTP.

8.2.1.1 Diagrama de clases capa lógica

Para el servidor se presenta el diagrama de clases general de la capa lógica, de procesamiento de las peticiones. Primero se presenta un diagrama general, para luego ir detallando cada clase por separado.

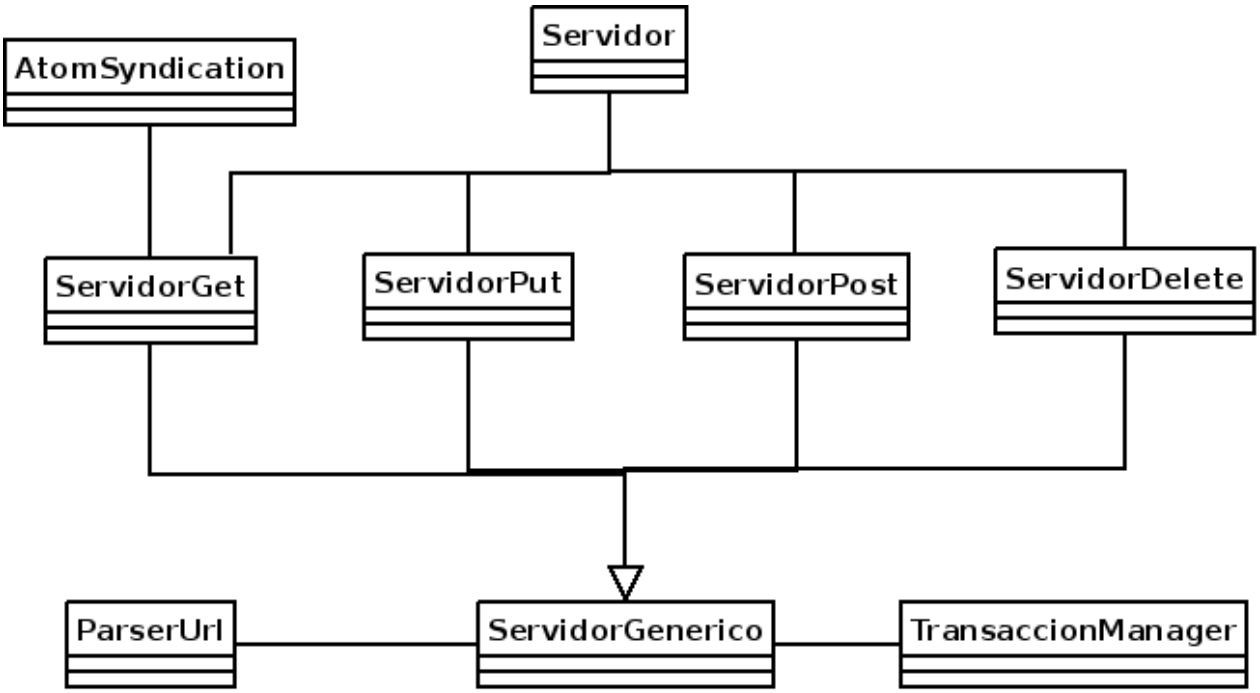


Ilustración 4 - Diagrama de clases capa lógica

En el diagrama se han presentado las clases más relevantes para el procesamiento de las peticiones, sin considerarse clases que contienen métodos utilitarios clásicos, los cuales son usados de forma transversal por la mayoría de las demás clases del servidor, y hay clases en las que se guardan constantes del servidor, nombres de propiedades, códigos de retorno, etc. Éstas clases serán nombradas y referenciadas en la sección de anexos.

a.- Clase Servidor

Se presenta a continuación el detalle de la clase Servidor, presentando sus métodos y variables de instancia necesarias:

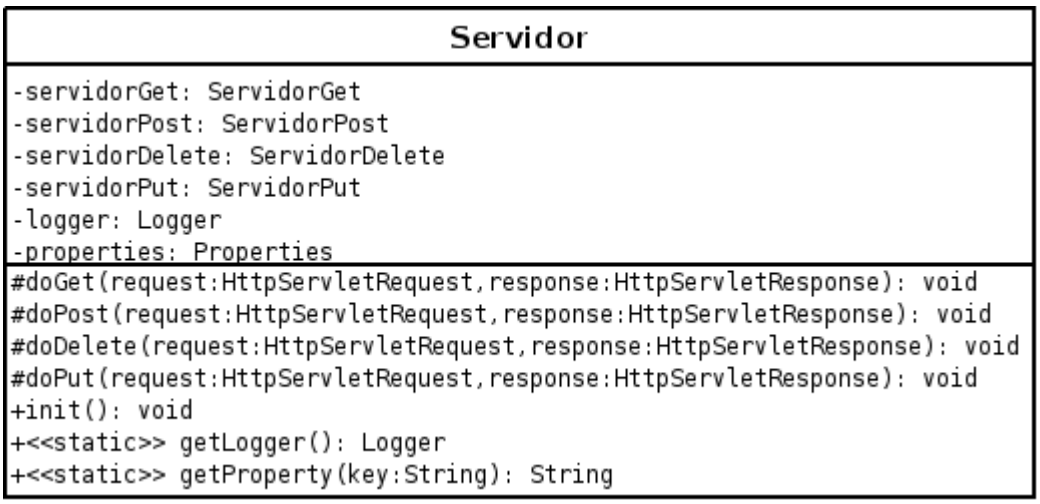


Ilustración 5 - Diagrama de clase Servidor

Esta clase servidor extiende a la clase `javax.servlet.http.HttpServlet` e implementa la interfaz `javax.servlet.Servlet` lo cual permite que el filtro de las peticiones HTTP(GET,PUT,DELETE,POST) se realice automáticamente. Las variables de instancia son del tipo ServidorXX, los cuales, dependiendo del tipo de petición que se recibe, son invocados. Además posee un objeto de tipo Logger, el cual es usado para el proceso de debugging del servidor, proceso que es muy importante para obtener una clara visualización de lo que está sucediendo en la ejecución interna. Por último posee un objeto de tipo Properties en el cual se guardan valores de configuración del servidor entregados a través de un archivo de configuración que se detallará en el apartado de anexos.

Se proceden a detallar los métodos más importantes de esta clase:

Método	Descripción
void init()	Método encargado de la carga de las propiedades por defecto del servidor, presentes en el archivo de configuración de este. Además inicializa el objeto Logger encargado del servidor.
void doGet(request,response)	Método que recibe las peticiones HTTP GET, crea un objeto ServidorGet el cual realiza el procesamiento. Los demás métodos del tipo doXX(request,response) tienen el mismo comportamiento.

Tabla 3 - Métodos clase Servidor

b.- Clase ServidorGenerico

Se presenta a continuación la clase ServidorGenérico con sus métodos y variables de instancia:

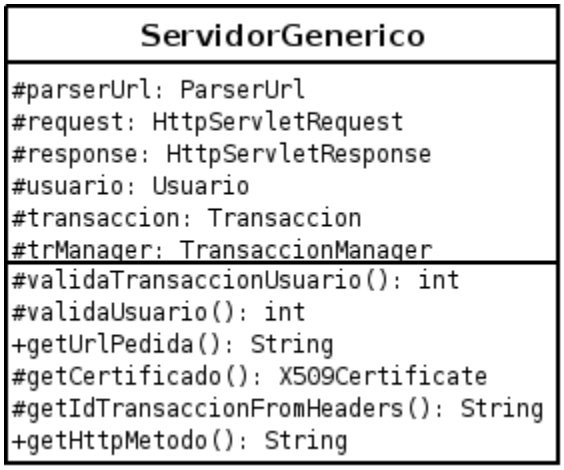


Ilustración 6 - Diagrama de clase ServidorGenerico

Esta clase es una de las más importantes pues es la clase padre de las clases ServidorGet, ServidorPut, ServidorPost y ServidorDelete. Esta clase posee en sus variables de instancia al objeto que representa a la transacción en curso, y que es usada por las clases hijas, además de guardar la petición http en el objeto request, el cual también es visible por las clases que la extienden. Además posee un objeto ParserUrl encargado de analizar la petición y saber que método es el que se debe ejecutar.

A continuación se explican los métodos más importantes de esta clase:

Método	Descripción
X509Certificate getCertificado()	Método encargado de la obtención del certificado que identifica al usuario que realiza la petición, procesa el objeto request y obtiene el certificado, retornándolo. Entrega nulo de no encontrarlo en la petición.
String getIdTransaccionFromHeaders()	Método que busca en el objeto request, dentro del header *Location* el identificador de la transacción en curso, retornándolo como String.
String getUrlPedida()	Método que entrega la Url sobre la que se realiza la petición desde el objeto request. Realiza el decoding de

	la URL.
int validaTransaccionUsuario()	Método que valida que el usuario que realiza la petición exista y que su certificado corresponda. Además valida que la transacción involucrada exista y que no haya expirado.
int validaUsuario()	Método que sólo realiza la validación de la existencia del usuario y la correctitud de su certificado, usado en los casos de peticiones en las que aún no existe una transacción creada.

Tabla 4 - Métodos clase ServidorGenerico

c.- Clase ServidorGet

Se detalla a continuación el diagrama específico de esta clase, con las variables de instancia que la componen y los métodos implementados:

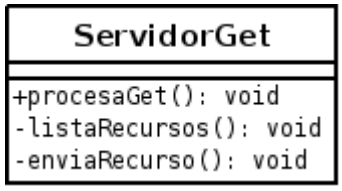


Ilustración 7 - Diagrama de clase ServidorGet

Esta clase extiende a ServidorGenerico, por lo que hereda sus métodos y variables de instancia, ya que se ve que no posee variables de instancia particulares. Esta clase es la encargada de procesar las peticiones HTTP de tipo GET, por lo que implementa las funciones de entrega de recursos al cliente o el envío de un lista de los recursos de acuerdo a la fecha pedida, como se detalla en el protocolo RRTP.

A continuación se explican los métodos de esta clase:

Método	Descripción
void procesaGet()	Método encargado de procesamiento de la petición HTTP, tomando la decisión de cuál es el método invocado por el servidor, validando que el usuario y la transacción se encuentren en el estado correcto. Hace uso del objeto ParserUrl para procesar la URL y seleccionar la acción a ejecutar.
void listaRecursos()	Método que está encargado de entregar un listado de los recursos que posee el cliente en formato XML, siguiendo el estándar AtomSyndication. De acuerdo a la fecha pasada en la URL son los archivos que se incluyen en la lista a enviar.
void enviaRecurso()	Método encargado de enviar el recurso pedido por el usuario en la URL, este debe coincidir en nombre y fecha en que fue subido en el servidor, es por esto que es importante una consulta previa pidiendo el listado de recursos si no se está seguro de los datos del recurso.

Tabla 5 - Métodos clase ServidorGet

d.- Clase ServidorPut

A continuación se muestra el diagrama particular de la clase ServidorPut, con sus métodos implementados:

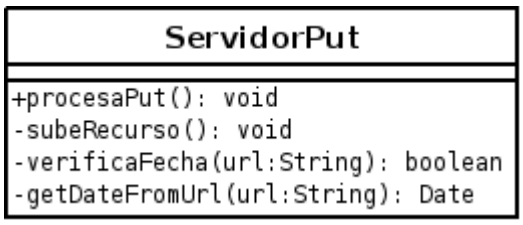


Ilustración 8 - Diagrama de clase ServidorPut

La clase ServidorPut es la encargada de procesar las peticiones HTTP PUT, estas peticiones tienen una sola funcionalidad dentro del sistema, esta es la de subida de un recurso al servidor. Esta clase se encarga de verificar la correctitud de los elementos

relacionados con la transacción, ya sea, el tiempo de vida de esta, que los datos lleguen íntegros y que la fecha de subida sea correcta de acuerdo a lo especificado en el protocolo RRTP.

Se proceden a detallar los métodos de esta clase:

Método	Descripción
void procesaPut()	Método encargado de recibir las peticiones y realizar el procesamiento de estas, verifica que es usuario sea válido, así como la transacción involucrada. Parsea la petición y llama al método que maneja la subida de recurso cuando así sea pedido.
void subeRecurso()	Método encargado de gestionar la recepción de un recurso enviado por un cliente válido, este método verifica la integridad de los datos y en caso exitoso guarda el recurso de manera persistente enviando la correspondiente respuesta.
boolean verificaFecha(String url)	Método encargado de verificar que la fecha con la que se sube el recurso al servidor no distancia en más de 1 día con la fecha actual, esto de acuerdo a la especificación del protocolo RRTP.
Date getDateFromUrl(String url)	Método que obtiene la fecha en que se indica se debe subir el recurso, la cual es usada para su posterior verificación y registra la fecha para posteriores consultas del cliente.

Tabla 6 - Métodos clase ServidorPut

e.- Clase ServidorPost

Se procede a mostrar el diagrama de esta clase, mostrando los métodos que la componen:

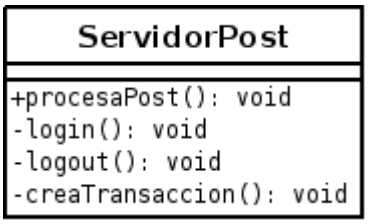


Ilustración 9 - Diagrama de clase ServidorPost

Esta clase es la encargada de procesar las peticiones HTTP POST que llegan al servidor, en este caso hay 3 funciones distintas que el protocolo señala que pueden ser realizadas a través de una petición POST, estas son la de ingreso al sistema, creación de una transacción y la salida del sistema. La elección de que función realizar se hace al mirar las URL de destino que cada petición POST tiene, luego de vista la URL se determina cual fue la función requerida por el cliente.

Se detallan a continuación los métodos de esta clase:

Método	Descripción
void procesaPost()	Método encargado de determinar a partir de la URL, cuál fue la función requerida por el cliente, además de verificar que el cliente sea valido para la creación de transacción, puesto que para el caso de login recién estoy ingresando el usuario al sistema.
void login()	Método que a partir del certificado del cliente, verifica la existencia de este para el servidor y de encontrarlo lo registra como logeado en el sistema. Si el usuario es nuevo y pasa el handshake, lo ingresa al sistema.
void logout()	Método encargado de actualizar la información necesaria para la correcta salida del cliente del sistema, registrándolo como logout y eliminando cualquier dato relacionado con este.

void creaTransaccion()	Método que crea una nueva transacción en el sistema a petición del cliente, esta se crea con el identificador del cliente y el tiempo de vida que se obtiene del archivo de configuración del servidor, que será detallado en los anexos.

Tabla 7 - Métodos clase ServidorPost

f.- Clase ServidorDelete

Se presenta el diagrama de clases particular de esta clase detallando los métodos que la componen:

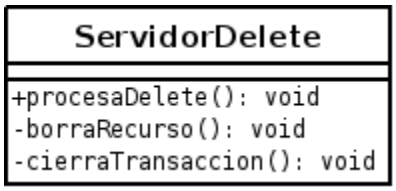


Ilustración 10 - Diagrama de clase ServidorDelete

ServidorDelete es la clase encargada de procesar las peticiones HTTP DELETE que llegan al servidor. En este caso hay 2 funciones determinadas en el protocolo RRTP que pueden ser realizadas con este tipo de petición, estas son borrar un recurso del servidor y la otra es cerrar una transacción. Como se ha señalado anteriormente la distinción de funcionalidad pedida por el cliente se realiza analizando la URL objetivo del método DELETE como buen sistema de arquitectura REST.

Se detallan a continuación los métodos de esta clase:

Método	Descripción
void procesaDelete()	Método encargado de procesar la URL objetivo y discriminar cuál es el método solicitado por el cliente, si borrar un recurso o el cierre de una transacción. Además verifica que tanto el cliente como la transacción sean válidas en la petición.

void borraRecurso()	Método encargado de eliminar un recurso que ha sido indicado por el cliente, este recurso es eliminado completamente de los registros sin que tenga el cliente posibilidad de recuperarlo.
void cierraTransaccion()	Método encargado de actualizar el estado de la transacción en curso al estado *Cerrada,* de acuerdo a los especificado en el protocolo RRTP, no permitiéndose más operaciones sobre esa transacción.

Tabla 8 - Métodos clase ServidorDelete

g.- Clase ParserUrl

Se presenta el diagrama de clases con los métodos y variables de instancia propios de la clase:

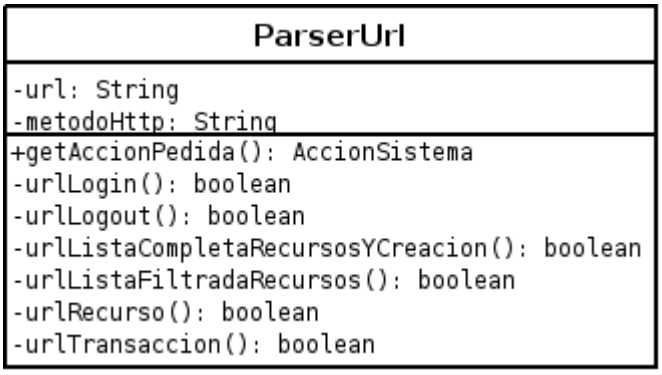


Ilustración 11 - Diagrama de clase ParserUrl

Un objeto de esta clase es usado por todas las clases que extienden a ServidorGenerico, y es usado en su función de discriminar de acuerdo a la URL y el tipo de petición HTTP, cual es la acción que el servidor debe realizar. Las variables de instancia guardan tanto la URL objetivo, como el método HTTP enviado por el cliente.

A continuación se detallan los métodos de la clase:

Método	Descripción
AccionSistema getAccionPedida()	Método encargado de determinar de acuerdo al método HTTP entregado y la URL objetivo cual es la acción que debe realizar el servidor. Retorna una instancia de AccionSistema, el cual es una enumeración que contiene las posibles acciones a realizar.
boolean url*()	Los métodos url* de esta clase, retornan verdadero en los casos que la sintaxis de la URL objetivo coincide con la que el método busca. Dos métodos distintos no pueden retornar verdadero al mismo tiempo, pues de no ser así habría ambigüedad en la función pedida.

Tabla 9 - Métodos clase ParserUrl

h.- Clase TransaccionManager

Se muestra a continuación el diagrama particular de esta clase, mostrando sus métodos más importantes y variables de instancia:

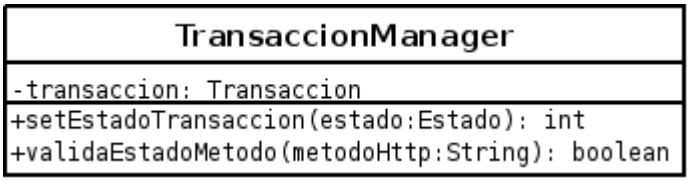


Ilustración 12 - Diagrama de clase TransaccionManager

Esta clase está encargada de determinar si el método que el cliente quiere aplicar sobre la transacción involucrada es válido de acuerdo al diagrama de estados de la transacción definido en el protocolo RRTP. De acuerdo al estado actual de la transacción se verifica el tipo de petición del cliente y aprueba o no la ejecución de la nueva petición.

A continuación se detallan los métodos de la clase:

Método	Descripción
int setEstadoTransaccion(Estado estado)	Método encargado de actualizar el estado de la transacción en curso, luego de validado el método a ejecutarse sobre ella, y que este es haya cumplido exitosamente, el método actualiza este estado de manera persistente.
boolean validaEstadoMetodo(String metodo)	Método encargado de realizar la validación del método solicitado por el cliente, de acuerdo al estado que posea. Retorna verdadero o falso según corresponda.

Tabla 10 - Métodos clase TransaccionManager

i.- Clase AtomSyndication

Se presenta a continuación el diagrama de la clase AtomSyndication:

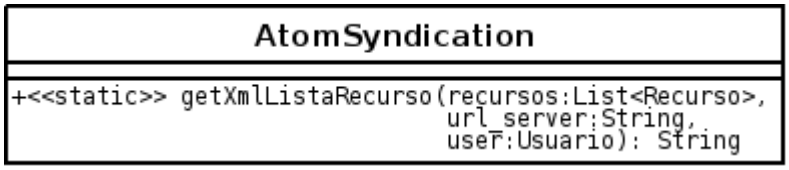


Ilustración 13 - Diagrama de clase AtomSyndication

Esta clase es la encargada de generar el documento XML siguiendo el estándar Atom Syndication con los recursos que son pedidos por el usuario, siguiendo el estándar de campos especificados en el protocolo RRTP. La clase posee sólo el método estático que genera el XML, recibiendo en sus parámetros toda la información necesaria para ello. El primer parámetro es la lista de recursos, esta lista se recorre obteniendo objetos de tipo Recurso, los cuales poseen los datos necesarios para describir un recurso, el siguiente parámetro es la URL del servidor usada para llenar un campo exigido por el protocolo, y el último parámetro es un objeto de tipo Usuario, el cual posee los datos del cliente que está realizando la consulta, de forma de poner esta información en el XML de respuesta, identificando de esta forma que los recursos enviados pertenecen a él.

8.2.1.2 Diagrama de clases capa física

Se presenta a continuación el diagrama de clases de la capa física del servidor, esta capa es la encargada de mantener la persistencia de los datos que así lo requieran, fundamentalmente los datos de los recursos:

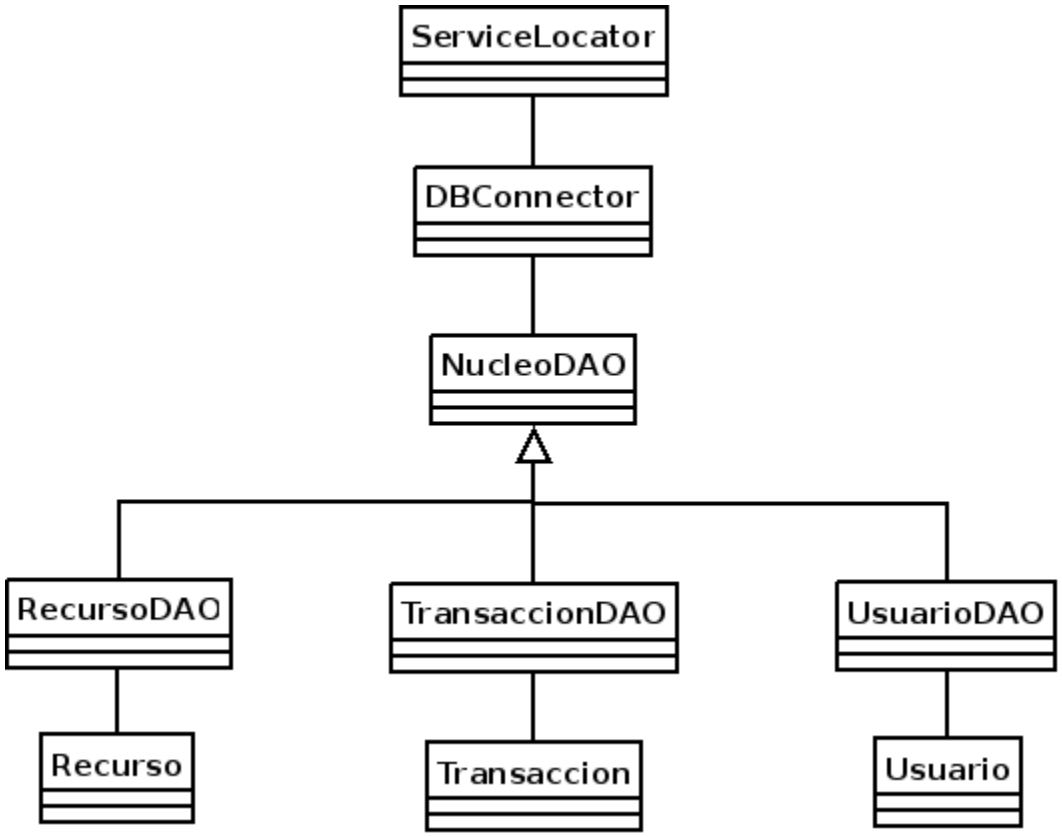


Ilustración 14 - Diagrama de clases capa física

La capa física del servidor es la que maneja las conexiones con el motor relacional de base de datos usado. En esta implementación se construyó una aplicación Web inserta en un contenedor Web llamado Tomcat, luego las conexiones con la base de datos se realizan a través de DataSources, que son creados a través de un archivo de configuración de formato XML, llamado *context.xml* el cual es detallado en la sección de anexos. Luego la aplicación servidor siempre tiene una cantidad de conexiones con la base de datos a las cuales puede acceder de manera más eficiente.

Se procede a detallar cada una de las clases relevantes de esta capa.

a.- Clase ServiceLocator

Se presenta el diagrama específico de esta clase, para detallar su funcionamiento:

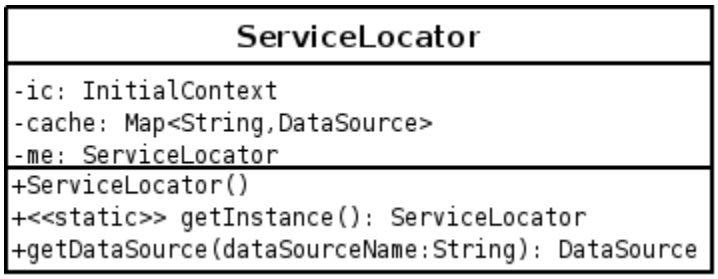


Ilustración 15 - Diagrama de clase ServiceLocator

Esta es la clase responsable de obtener los DataSources que se conectan con el motor de base de datos. Esta clase se referencia estáticamente, y como se ve en el diagrama de la ilustración 14, se relaciona con la clase DBConector que es la que realiza el llamado de estos métodos. En esta clase se guardan los DataSources que ya han sido pedidos, en la variable cache, el cual es un objeto Map, que guarda los DataSources de acuerdo al nombre con el que fueron solicitados, de forma de optimizar esta obtención y sacarlos de memoria en casos que hay varios accesos a base de dato de un mismo cliente.

Se detallan a continuación los métodos de esta clase:

Método	Descripción
ServiceLocator()	Método constructor que se encarga de inicializar las variables de instancia de la clase, tanto el objeto de tipo InitialContext, que es el que busca sobre los DataSources del servidor Tomcat, como el objeto Map que guarda en memoria los DataSources ya pedidos.
ServiceLocator getInstance()	Método encargado de retornar la instancia estática del objeto ServiceLocator.
DataSource getDataSource(String dataSourceName)	Método encargado de buscar un DataSource de acuerdo al nombre entregado en el contexto de la aplicación, todo esto dentro del servidor de aplicaciones tomcat en este caso. De ser

	encontrado lo guarda en el objeto Map, para así tener un acceso más rápido a él ante eventuales peticiones posteriores.

Tabla 11 - Métodos clase ServiceLocator

b.- Clase DBConnector

Se presenta a continuación el diagrama de clases particular que detalla la estructura de esta clase:

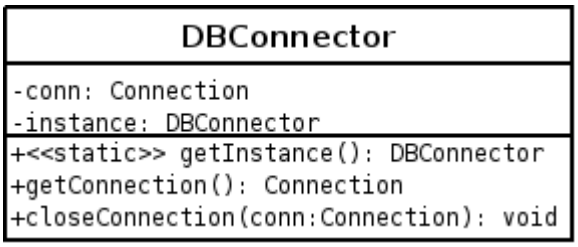


Ilustración 16 - Diagrama de clase DBConnector

Clase encargada de entregar los Objetos Connection a partir de los DataSources obtenidos de la clase ServiceLocator, estos objetos son los que entregan conexión directa con la base de datos y sobre éstos se pueden realizar las consultas de inserción, actualización o borrado según corresponda.

Se detallan cada uno de los métodos de esta clase:

Método	Descripción
DBConnector getInstance()	Método que entrega el objeto estático DBConnector a través del cual se accede a los demás métodos de la clase.
Connection getConnection()	Método encargado de obtener el objeto Connection a partir de un DataSource obtenido de la clase ServiceLocator.
void closeConnection(Connection conn)	Método que se encarga de cerrar una conexión con la base de datos, obtenida previamente de

	un DataSource existente.

Tabla 12 - Métodos clase DBConnector

c.- Clase NucleoDAO

Se muestra a continuación el diagrama de esta clase con sus métodos y variables de instancia:

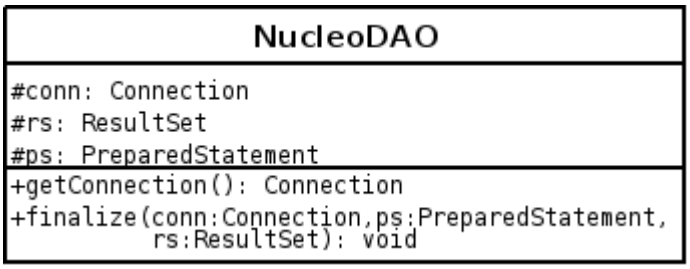


Ilustración 17 - Diagrama de clase NucleoDAO

Esta es la clase padre de todos las otras clases DAO, en ella se presentan los métodos para obtener una conexión con la base de datos y en sus variables de instancia están los objetos usados para preparar y realizar consultas(PreparedStatement) y el objeto en el que se guarda lo retornado por la base de datos(ResultSet), tanto los métodos como variables de instancia son usados en la mayoría de los métodos de las clases que la extienden.

Se detallan a continuación los métodos presentes en la clase:

Método	Descripción
Connection getConnection()	Método encargado de obtener una conexión con la base de datos, haciendo uso del objeto DBConnector para esto.
void finalize(Connection conn, PreparedStatement ps, ResultSet rs)	Método encargado de cerrar una conexión con la base de datos además de eliminar los datos asociados a esa consulta, ya sea el objeto retornado por la base de datos o la consulta misma, las

	cuales podrían permanecer en las variables de instancia respectivas.

Tabla 13 - Métodos clase NucleoDAO

d.- Clase RecursoDAO

Se presenta el diagrama particular de la clase RecursoDAO, dejando en claro sus métodos:

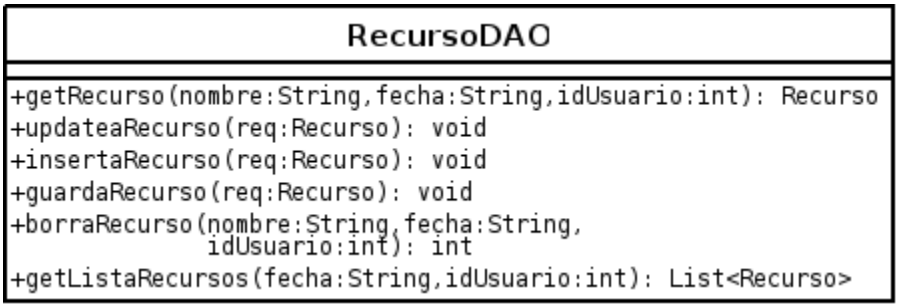


Ilustración 18 - Diagrama de clase RecursoDAO

Esta clase es la encargada de gestionar todo el manejo de persistencia de los recursos relacionados con cada petición. Esta clase extiende a NucleoDAO, haciendo uso en cada método propio de las variables de instancia de su padre y métodos también. Hace uso del objeto Recurso para hacer todo el traspaso de los datos con la capa lógica, permitiendo que el proceso sea transparente.

Se detallan a continuación cada uno de los métodos de esta clase:

Método	Descripción
Recurso getRecurso(String nombre, String fecha, int idUsuario)	Método encargado de obtener un recurso de la base de datos, que posea el nombre pasado como parámetro en la fecha indicada en el parámetro y que sea para el usuario indicado por idUsuario, de no cumplirse alguna de las condiciones se retorna el objeto nulo.

void updateaRecurso(Recurso req)	Método encargado de actualizar la información de un recurso, esto se refiere a actualizar sólo los datos del recurso que se ha subido por las restricciones que existen en el protocolo.
void insertaRecurso(Recurso req)	Método encargado de realizar la inserción en la base de datos de un nuevo recurso subido por un cliente, se recibe el objeto Recurso con todos los datos seteados, de forma que solo resta realizar la inserción.
void guardaRecurso(Recurso req)	Método que discrimina cuando el recurso que se ha subido con la petición HTTP PUT es nuevo o ya existía, para así hacer una inserción de este o un updateo de los datos respectivamente.
int borraRecurso(String nombre, String fecha, int idUsuario)	Método encargado de eliminar un recurso de la base de datos. Tiene que coincidir con el nombre de recurso entregado, con la fecha de subida dada y pertenecer al usuario que realiza la petición.
List<Recurso> getListaRecursos(String fecha, int idUsuario)	Método que retorna la lista de recursos que pertenecen a un usuario particular, filtrado por la fecha indicada de acuerdo a lo especificado en el protocolo RRTP, este método es usado para formar el XML en formato Atom Syndication que lista los recursos.

Tabla 14 - Métodos clase RecursoDAO

e.- Clase TransaccionDAO

Se presenta a continuación el diagrama de esta clase, mostrando los métodos que la componen:

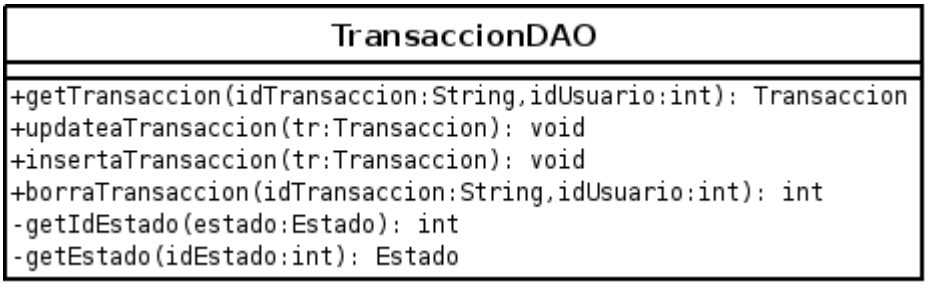


Ilustración 19 - Diagrama de clase TransaccionDAO

Esta clase es la encargada del manejo de las transacciones a medida que se van realizando peticiones, se pueden crear nuevas, buscar las que están en curso y actualizar el estado y borrar una transacción finalizada. El objeto de tipo Transacción es el que hace la comunicación de los datos entre la capa física y la lógica en este caso.

Se detallan a continuación los métodos de esta clase:

Método	Descripción
Transaccion getTransaccion(String idTransaccion, int idUsuario)	Método encargado de obtener una transacción que posea el idTransaccion señalado y que pertenezca al usuario de posea el identificador entregado como parámetro.
void updateaTransaccion(Transaccion tr)	Método encargado de actualizar la información de una transacción, específicamente se actualiza su estado, siendo esta la única información que necesite ser actualizada.
void insertaTransaccion(Transaccion tr)	Método que inserta una nueva transacción en la base de datos. Esta transacción es creada en la capa lógica luego de la petición respectiva del cliente.
int borraTransaccion(String idTransaccion, int idUsuario)	Método que elimina una transacción que posea el identificador pasado como parámetro y pertenezca al usuario con ese identificador. Retorna un entero con el número de filas borradas.
int getIdEstado(Estado estado)	Método privado que retorna el entero que identifica el estado de una transacción.
Estado getEstado(int idEstado)	Método que retorna el Estado que es representado

	por el identificador que se entrega como parámetro.

Tabla 15 - Métodos clase TransaccionDAO

f.- Clase UsuarioDAO

Se presenta a continuación el diagrama de clase particular con los métodos que componen esta clase:

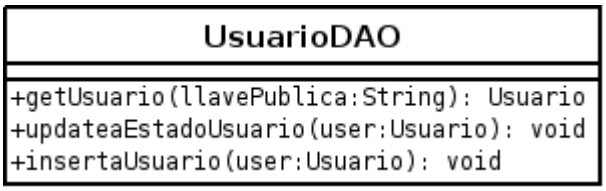


Ilustración 20 - Diagrama de clase UsuarioDAO

Esta clase es la encargada del manejo de la información de los usuarios del sistema, también llamados clientes. En este caso la implementación hace uso de SSL, por lo que si un nuevo usuario tiene un certificado válido, es decir, emitido por una autoridad en la que el servidor confía, entonces este usuario es ingresado automáticamente a la base de datos del servidor, de esta forma se hace más expedito el registro de los usuarios. Se deja en claro que este comportamiento es particular de esta implementación y no es parte del protocolo.

Se detallan a continuación los métodos de esta clase:

Método	Descripción
Usuario getUsuario(String llavePublica)	Método encargado de obtener un objeto Usuario a partir de la llave pública de este, de manera de verificar que exista en la base de datos del servidor.
void updateaEstadoUsuario(Usuario user)	Método encargado de actualizar el estado del usuario en la base de datos. Esto se realiza cuando un usuario ingresa o sale del servidor para realizar peticiones

void insertaUsuario(Usuario user)	Método encargado de guardar en la base de datos del servidor al nuevo usuario que al pasar el handshake de SSL puede ingresar al sistema.

Tabla 16 - Métodos clase UsuarioDAO

g.- Clase Recurso

Se presenta a continuación el diagrama de clases particular que detalla la estructura de esta clase:

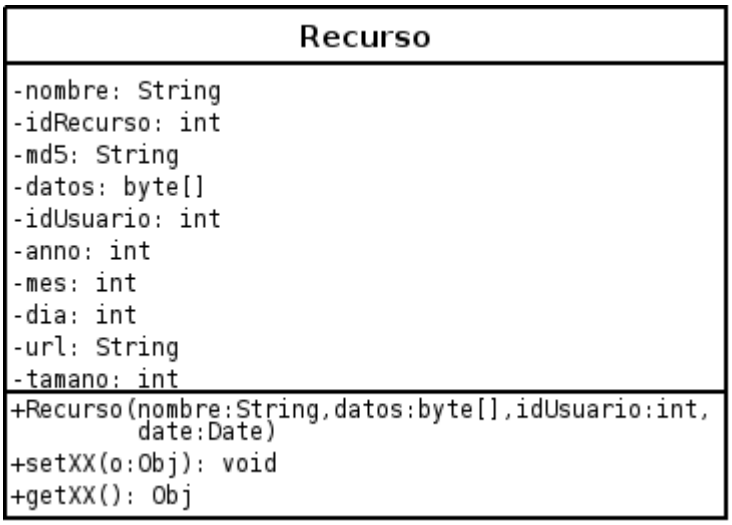


Ilustración 21 - Diagrama de clase Recurso

Esta clase es la que representa la información de un recurso dentro del sistema, es usada para pasar la información entre las 2 capas del sistema. Posee una cantidad de variables de instancia considerable, la primera de ellas guarda el nombre del recurso, la segunda, *idRecurso*, guarda el identificador del recurso para el sistema, *md5* guarda la cadena de caracteres que se obtiene de aplicar el algoritmo MD5 al arreglo de bytes del recurso. Precisamente es la variable *datos* la que guarda los bytes que componen el recurso, *idUsuario* identifica al usuario o cliente dueño del recurso. *Anno, mes* y *dia* guardan la fecha en la que el recurso fue subido al servidor. *Url* guarda la URL dentro del servidor donde está ubicado el recurso finalmente *tamano* guarda el tamaño en bytes del recurso, el cual es usado como uno de los datos que se entregan en el listado de recursos cuando es pedido por el cliente.

Se procede a detallar los métodos que posee esta clase:

Método	Descripción
Recurso(String nombre, byte[] datos, int idUsuario, Date date)	Método constructor de la clase, el cual recibe el nombre del recurso enviado por el usuario, un arreglo de bytes con todos los datos que componen el recurso, el identificador del usuario dueño del recurso y la fecha en la que se realiza la operación.
void setXX(Obj o)	Todas las variables de instancia por ser variables privadas poseen un método setXX, con XX como el nombre de la variable, de forma de poder ingresar información a las variables, a partir del parámetro ingresado que debe corresponder al tipo de la variable.
Obj getXX()	Método encargado de retornar el valor de una variable de instancia del objeto, retorna el valor y tipo que coincide con la variable pedida.

Tabla 17 - Métodos clase Recurso

h.- Clase Transaccion

Se presenta a continuación el diagrama de clases particular que detalla la estructura de esta clase:

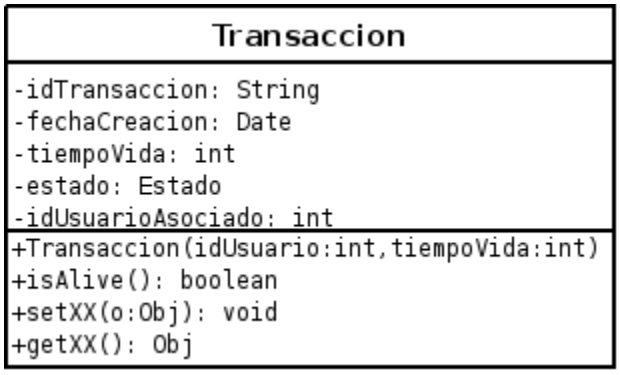


Ilustración 22 - Diagrama de clase Transaccion

Esta es la clase que posee toda la información relativa a una transacción del sistema. Sus variables de instancia guardan: *idTransaccion* guarda el identificador único de la transacción, el cual es generado usando la clase de Java java.util.UUID, la cual

genera un identificador único usando un algoritmo criptográfico que asegura con una probabilidad bastante alta que el identificador será único; *fechaCreacion* guarda la fecha en la cual el recurso fue subido por el usuario al servidor, no siendo posible cambiar esta fecha; *tiempoVida* representa el tiempo de vida en minutos de la transacción, este valor es entregado al servidor a través del archivo de configuración; *estado* guarda el estado en el que se encuentra la transacción, este estado puede tomar valores de acuerdo a lo especificado en el protocolo RRTP; *idUsuarioAsociado* guarda el identificador del usuario que está realizando la actual transacción.

Se detallan a continuación los métodos de esta clase:

Método	Descripción
Transaccion(int idUsuario, int tiempoVida)	Método constructor de la clase, el cual recibe el usuario que ha hecho la petición de creación de transacción y el tiempo de vida en minutos que es obtenido de una propiedad que se obtiene del archivo de configuración del servidor.
boolean isAlive()	Método que retorna verdadero o falso de acuerdo a si el tiempo transcurrido desde la creación de la transacción ha superado o no al tiempo de vida de ésta.
void setXX(Obj o)	Todas las variables de instancia por ser variables privadas poseen un método setXX, con XX como el nombre de la variable, de forma de poder ingresar información a las variables, a partir del parámetro ingresado que debe corresponder al tipo de la variable.
Obj getXX()	Método encargado de retornar el valor de una variable de instancia del objeto, retorna el valor y tipo que coincide con la variable pedida.

Tabla 18 - Métodos clase Transaccion

i.- Clase Usuario

Se presenta a continuación el diagrama de clases particular que detalla la estructura de esta clase:

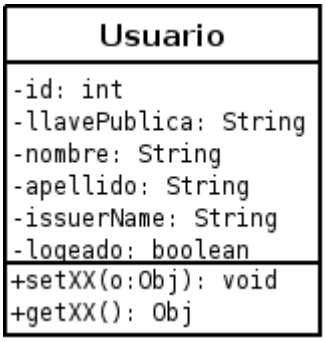


Ilustración 23 - Diagrama de clase Usuario

Esta clase maneja la información relacionada a los usuarios, también llamados clientes, que posee registrados el servidor. Sus variables de instancia son las siguiente: *id* guarda el identificador del usuario para la base de datos, nada tiene que ver con su certificado digital; *llavePublica* guarda la cadena de caracteres que representa la llave pública del usuario, que lo identifica unívocamente en el sistema; *nombre* guarda el nombre del usuario; *apellido* guarda el apellido del usuario; *issuerName* es el nombre que viene como campo en el certificado digital, incluye el nombre del usuario y de la organización a la que pertenece; *logeado* es un campo que dice el estado del usuario en el servidor.

Se detallan a continuación los métodos de la clase:

Método	Descripción
void setXX(Obj o)	Método que se aplica a cada una de las variables de instancia de forma de poder darles algún valor pues son variables privadas.
Obj getXX()	Método encargado de retornar el valor de una variable de instancia del objeto, retorna el valor y tipo que coincide con la variable pedida.

Tabla 19 - Métodos clase Usuario

8.2.2 Modelo físico base de datos

Se presenta a continuación el esquema de tablas del motor de base de datos usado en la presente implementación del protocolo RRTP

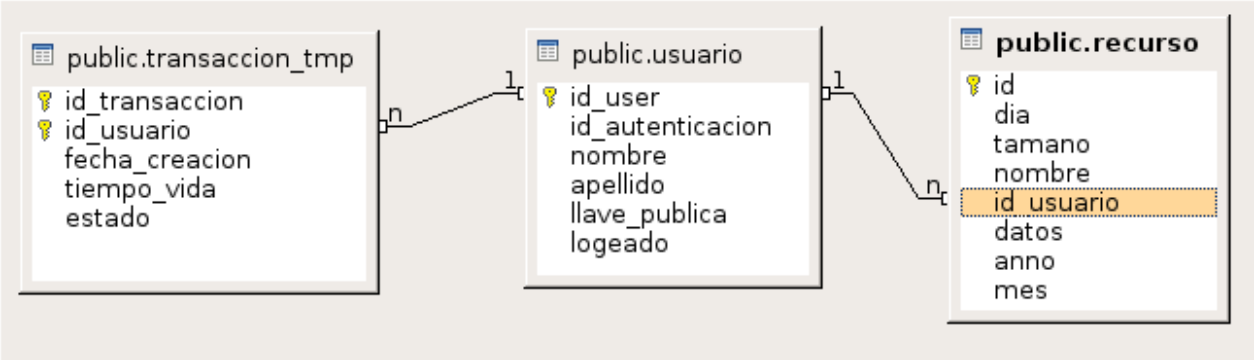


Ilustración 24 – Modelo físico base de datos

Para el modelo físico se usaron estas 3 tablas, hay que señalar que la persistencia en un motor relacional no es parte del protocolo RRTP, dejando abierto a cada implementación de servidor el manejo de la persistencia de los datos.

Procedemos a detallar cada tabla:

- **Usuario**: tabla que guarda la información que describe a un cliente o usuario del sistema. Sus atributos son: *id_user* es un número entero, el cual es un identificador interno de la tabla para cada usuario; *id_autenticacion* es una cadena de caracteres que guarda el issuerName del certificado del usuario, en el que aparece el nombre de éste como el de la organización emisora del certificado; *nombre, apellido* guardan el nombre y apellido del usuario respectivamente; *llave_publica* es una cadena de caracteres que guarda la llave publica del usuario para el proceso de validación al entrar al sistema; *logeado* es un entero que indica si el usuario está o no logeado en el sistema.
- **Recurso**: tabla que guarda toda la información referente a los recursos subidos al servidor. Sus atributos son: *id* es un número entero que es un identificador interno de la tabla para cada recurso; *dia, mes y anno* son enteros que guardan el día, mes y año en que el recurso fue subido al servidor; *tamano* es un entero que guarda el número de bytes del recurso; *nombre* es una cadena de caracteres que guarda el nombre del recurso; *id_usuario* es un entero que es llave foránea y apunta al identificador de la tabla usuario, específicamente al usuario que subió el recurso;

datos es un tipo especial de postgresql que guarda una cadena de bytes, permite guardar los datos que forman el recurso.

- **Transaccion_tmp**: tabla que guarda las transacciones desde que son creadas, hasta que son finalizadas. Sus atributos son: *id_transaccion* es una cadena de caracteres que identifica únicamente a una transacción, es generada por la clase Transaccion dentro del servidor; *id_usuario* es un entero, llave foránea que apunta al usuario que ha creado la transacción; *fecha_creacion* es un timestamp que señala la fecha y hora en que se creó la transacción; *tiempo_vida* es un entero que indica el tiempo de vida que posee esa transacción en minutos; *estado* entero que representa uno de los estados que puede tener una transacción de acuerdo a lo especificado por el protocolo RRTP.

8.3 Cliente

A continuación se detalla la implementación del cliente compatible con el protocolo RRTP. En pos de verificar la interoperabilidad del protocolo y por consiguiente de la implementación se ha usado un cliente HTTP escrito en Java, este es el clienteHttp[26] del proyecto Jakarta HTTP Components[25]. Luego la implementación se ha centrado solamente en un nivel superior en el que se ordenan las secuencias de peticiones y se entregan ciertos valores en los headers especificados por el protocolo, dejando todo el tema de las peticiones HTTP mismas encapsuladas en el cliente que las provee por defecto. Esto nos señala que podría ser cualquier cliente HTTP compatible el que podría haber sido usado, siendo esta una de las ventajas de sistemas diseñados bajo la arquitectura REST.

8.3.1 Arquitectura

Para el cliente la arquitectura consta de 2 capas, una de las cuales es la capa de la transferencia de los datos a través de las peticiones HTTP, la cuales por haber sido importadas de un cliente ya hecho no será detallada. La capa que se detallará es la implementada específicamente para el cliente RRTP.

Se presenta a continuación un diagrama de clases del cliente RRTP implementado:

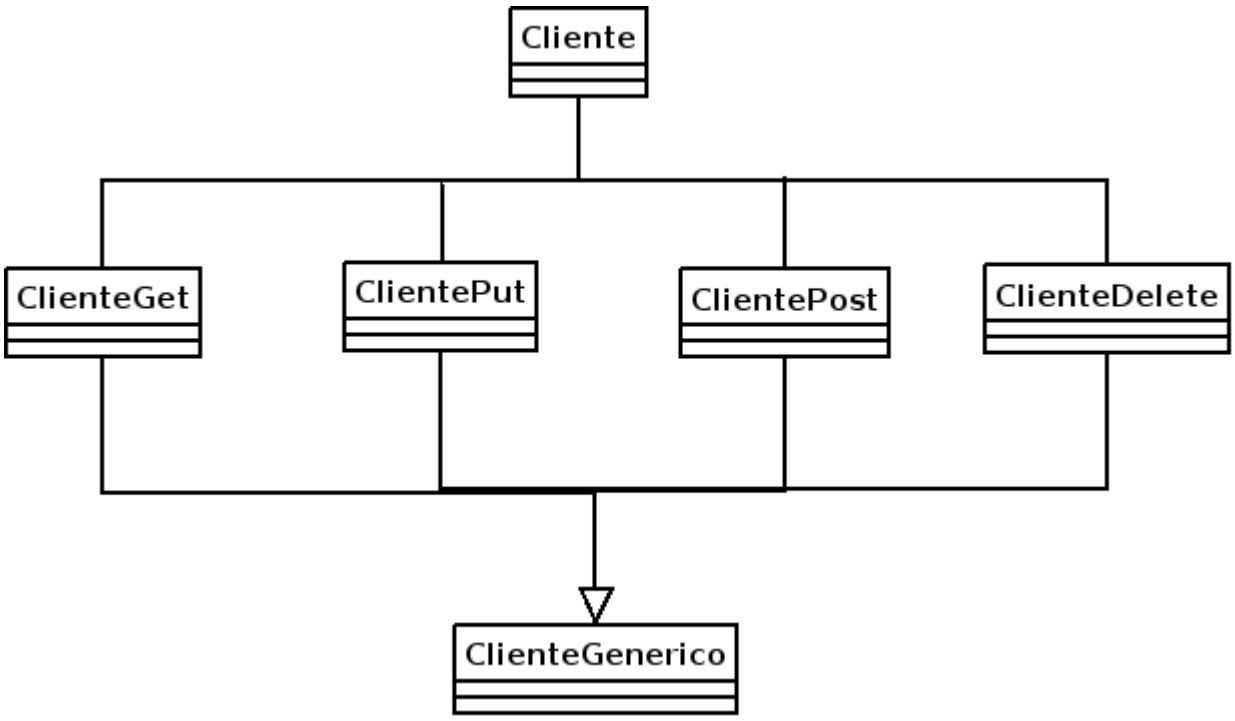


Ilustración 25 – Arquitectura cliente RRTP

Se ve en el diagrama las clases más relevantes de la implementación del cliente. Se han dejado de lado clases de métodos utilitarios que son usadas transversalmente por casi todas las demás clases y que no cumplen una función especifica importante. Además se han dejado de lado clases que guardan valores constantes que se referencian en varias clases distintas. En general, para la implementación del cliente, el tema más importante a considerar fue que sobre éste recae la responsabilidad de generar peticiones de reintento ante eventuales fallas de la red o peticiones respondidas fallidamente por el servidor, por lo que se tuvo especial cuidado en este tema. Existe un archivo de configuración del cliente, el cual es leído cuando se ejecuta algún comando, el detalle de este archivo se da en la sección de anexos.

Se detallan a continuación las clases y sus diagramas específicos:

a.- Clase Cliente

Se presenta a continuación el diagrama de clases detallado, con los métodos y variables de instancia que componen esta clase:

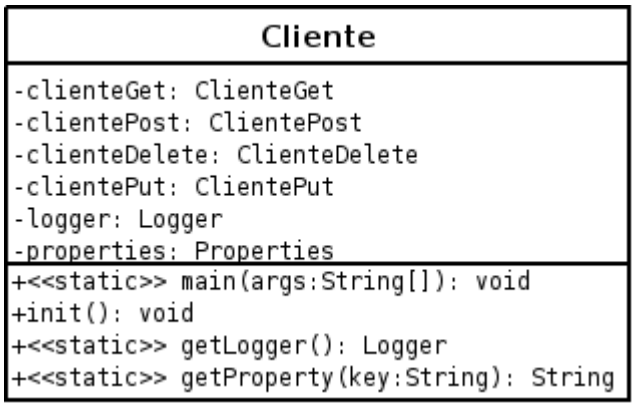


Ilustración 26 – Diagrama clase Cliente

Esta es la clase que es ejecutada para realizar las peticiones sobre el servidor RRTP. Se ve que la clase posee el método *main* el que es el ejecutable clásico en Java. Esta es la clase de partida, la cual carga a memoria ciertas propiedades existentes en el archivo de configuración del cliente, las cuales queden registradas en la variable de instancia *Properties*. Además esta clase posee un objeto de tipo *Logger* que permite realizar el proceso de debugging al igual que se realiza en la implementación del servidor. El cliente también posee en sus variables de instancia objetos del tipo ClienteXX, los cuales son los encargados de organizar los datos y enviar las peticiones al servidor.

Se detallan a continuación los métodos que conforman esta clase:

Método	Descripción
void int()	Método encargado de realizar la inicialización de los datos, es decir cargar las propiedades de un archivo de configuración y configurar el objeto que realiza el proceso de debugging leyendo el archivo que tiene esta configuración.
static void main(String[] args)	Método principal que realiza la verificación de los parámetros entregados en la línea de comandos, verifica que estos coincidan con la sintaxis de funciones implementadas del protocolo RRTP. Además configura SSL antes de realizar las peticiones, siempre se usa SSL en ambos sentidos, tanto cliente como servidor se autentican.
Logger getLogger()	Método estático que retorna el objeto de tipo

	Logger que es usado por las demás clases para realizar le proceso de debugging.
String getProperty(String key)	Método estático que entrega el valor de propiedades que fueron cargadas desde el archivo de configuración del cliente.

Tabla 20 - Métodos clase Cliente

b.- Clase ClienteGenerico

Se presenta a continuación el diagrama de clases específico, con los métodos y variables que conforman esta clase:

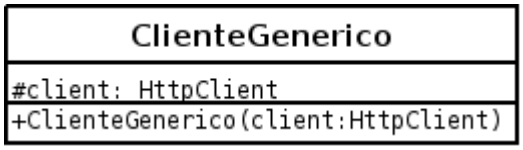


Ilustración 27 – Diagrama clase ClienteGenerico

Esta es la clase padre de todas las demás clases ClienteXX, con XX como alguno de los métodos HTTP, contiene simplemente la variable de instancia de tipo *HttpClient* del cliente que se ha importado, siendo esta variable usada por todos los demás clientes en el procesamiento de cualquiera de las peticiones. El único método que posee es un constructor que recibe como parámetro un objeto *HttpClient* que es guardado en la variable de instancia.

c.- Clase ClienteGet

Se presenta a continuación el diagrama de clases específico, con los métodos que conforman esta clase:

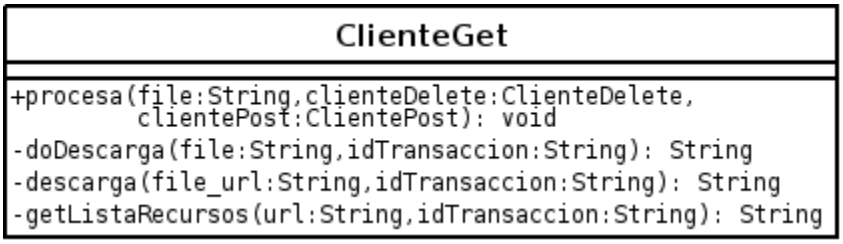


Ilustración 28 – Diagrama clase ClienteGet

Esta clase es la encargada de procesar las peticiones realizadas que involucren una petición HTTP GET, es decir, cuando se quiere descargar un recurso existente en el servidor o cuando se quiere obtener el listado de recursos que el cliente ha subido al servidor en una determinada fecha. Esta clase extiende a ClienteGenerico por lo que puede hacer uso del objeto de tipo *HttpClient* para las peticiones GET.

Se detallan a continuación los métodos de esta clase:

Método	Descripción
void procesa(String file, ClienteDelete clienteDelete, ClientePost clientePost)	Método que realiza todas las peticiones necesarias para cumplir con el protocolo RRTP. Realiza las peticiones usando el cliente que corresponda para cada una de ellas. Este método logea al cliente en el servidor, luego crea una transacción, para luego realizar la petición GET, después cierra la transacción y finalmente deslogea al cliente del servidor.
String doDescarga(String file, String idTransaccion)	Método encargado de discriminar si el método pedido es una descarga de recurso o una petición de listado de recursos, manejando los reintentos en casos de fallas.
String descarga(String fileUrl, String idTransaccion)	Método encargado de realizar la petición de descarga de un recurso, definido por la URL de este pasada como parámetro y el identificador de la transacción en curso. Setea los datos requeridos en los headers correspondientes y usa el objeto *HttpClient* para enviar la petición GET.

String getListaRecursos(String url, String idTransaccion)	Método encargado de enviar la petición GET que solicita el listado de recursos del cliente para una fecha dada en el parámetro *url*, usando la transacción que también se entrega como parámetro.

Tabla 21 - Métodos clase ClienteGet

d.- Clase ClientePut

Se presenta a continuación el diagrama de clases específico con los métodos que conforman esta clase:

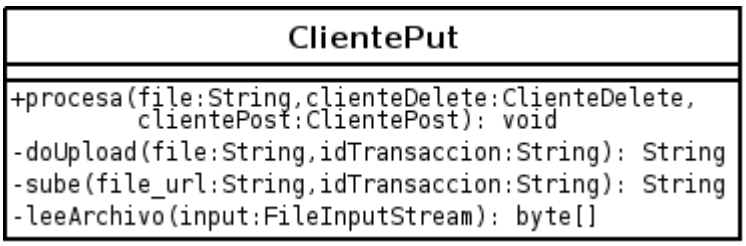


Ilustración 29 – Diagrama clase ClientePut

Esta es la clase responsable de realizar las peticiones que involucren un método HTTP PUT, es decir, para el cliente RRTP, se refiere a la subida de un recurso al servidor.

A continuación se detallan los métodos de esta clase:

Método	Descripción
void procesa(String file, ClienteDelete clienteDelete, ClientePost clientePost)	Método que realiza todas las peticiones necesarias para cumplir con el protocolo RRTP. Realiza las peticiones usando el cliente que corresponda para cada una de ellas. Este método logea al cliente en el servidor, luego crea una transacción, para luego realizar la petición PUT, después cierra la transacción y finalmente deslogea al cliente del servidor.

String doUpload(String file, String idTransaccion)	Método encargado de invocar el método de subida de archivo y además de manejar los reintentos en casos de retorno fallido por parte del servidor.
String sube(String file, String idTransaccion)	Método encargado de realizar la petición de subida de un recurso, recibe como parámetro la ruta absoluta del recurso en la máquina origen, leyendo sus datos y seteandolas en el cuerpo de la petición PUT como lo señala el protocolo RRTP.
byte[] leeArchivo(FileInputStream input)	Método encargado leer el archivo y retorna un arreglo de bytes con su información, este es usado para calcular el algoritmo MD5 para verificación de integridad de datos.

Tabla 22 - Métodos clase ClientePut

e.- Clase ClienteDelete

Se presenta a continuación el diagrama de clases específico con los métodos que conforman esta clase:

Ilustración 30 – Diagrama clase ClienteDelete

Clase encargada del procesamiento de las peticiones que involucran un llamado al método HTTP DELETE, ellas son el borrado de un recurso y el cierre de una transacción, las cuales son ejecutadas dependiendo de la URL objetivo, la cual es la que le dice al servidor si se está borrando un recurso o cerrando la transacción.

Se detallan a continuación los métodos de esta clase:

Método	Descripción
void procesa(String file, ClientePost clientePost)	Método que realiza todas las peticiones necesarias para que se realice correctamente el borrado de un recurso cumpliendo con el protocolo RRTP. Primero logea al usuario, luego se crea la transacción, se procede entonces a borrar el recurso y finalmente se deslogea al usuario.
String doDelete(String file, String idTransaccion)	Método encargado de invocar el método de borrado de un recurso y además de manejar los reintentos en casos de retorno fallido por parte del servidor.
String borrar(String file, String idTransaccion)	Método encargado de enviar la petición HTTP DELETE pidiendo el borrado de un recurso, cuya URL viene como parámetro en *file*, usando la transacción pasada como parámetro.
void doCloseTransaccion(String idTransaccion)	Método encargado de invocar el método que cierra una transacción activa. Este método se encarga de manejar los reintentos en casos de retorno fallido por parte del servidor.
int cierraTransaccion(String idTransaccion)	Método que envía la petición HTTP DELETE que ordena el cierre de la transacción que se le entrega como parámetro.

Tabla 23 - Métodos clase ClienteDelete

f.- Clase ClientePost

Se presenta a continuación el diagrama particular de esta clase con los métodos que la componen:

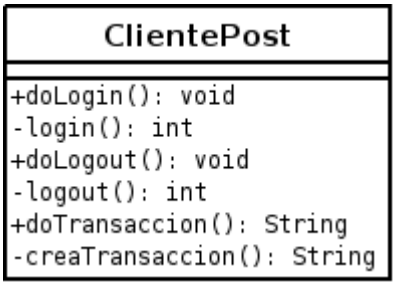


Ilustración 31 – Diagrama clase ClientePost

Clase encargada de realizar todas los métodos que involucren peticiones HTTP POST, que según el protocolo RRTP son las de ingreso al servidor, salida del servidor y la creación de una transacción. Esta clase por sí sola no realiza peticiones relacionadas con el manejo de recursos, sino que implementa peticiones que son necesarias para cumplir con las especificaciones del protocolo RRTP y que son usadas por todos los demás clientes del sistema.

Se detallan a continuación los métodos de esta clase:

Método	Descripción
void doLogin()	Método encargado de invocar al método que realiza la petición HTTP POST que pide el ingreso del cliente sobre el servidor. Se encarga de manejar los reintentos en caso de eventuales fallos de la petición.
int login()	Método que envía la petición HTTP POST con todos los headers necesarios para que sea interpretado por el servidor como una petición de ingreso al sistema.
void doLogout()	Método encargado de invocar el método que envía la petición HTTP POST que indica que el cliente se deslogea del servidor. Se encarga de manejar los reintentos en casos de retorno fallido por parte del servidor.
int logout()	Método encargado de enviar la petición HTTP POST pidiendo el deslogeo del usuario luego de haber completado las transacciones enviadas al

	servidor.
void doTransaccion()	Método encargado de invocar el método que crea una nueva transacción. Este método se encarga de manejar los reintentos en casos de retorno fallido por parte del servidor.
int creaTransaccion()	Método que envía la petición HTTP POST que ordena la creación de una nueva transacción, enviando en la petición los headers necesario para cumplir con lo pedido por le protocolo RRTP.

Tabla 24 - Métodos clase ClientePost

8.3.2 Uso del cliente

Para el correcto uso del cliente implementado, se necesita especificar cuál es la sintaxis de comandos correcta para su buen funcionamiento.

En estos momentos el cliente implementa todos los métodos especificados en el protocolo RRTP, es decir, puede subir un recurso a un servidor, puede bajar un recurso, puede eliminar un recurso y puede realizar una consulta de la lista de recursos válidos para un cliente en una fecha determinada.

Se procede a explicar la forma de invocar al cliente para realizar las operaciones antes señaladas. Se parte de la base que el cliente posee correctamente configurados sus repositorios de certificados para el uso de SSL, esto se detalla en la sección de anexos. Se tiene como base también que se posee el archivo empaquetado JAR con el cliente dentro, luego el comando *cliente* ejecuta la clase *Cliente* que es la única que posee el archivo con un método *main* en el sistema del cliente.

a.- Peticiones GET

La sintaxis para la obtención de la lista de recursos de un cliente es la siguiente:

```
$cliente GET AAAA/MM/DD [ip_server[port_server]]
```

Comando en el que AAAA indica el año, MM el mes y DD el día en que se quiere consultar. Se pueden realizar consultas parciales poniendo sólo el año, año más el mes o completa, en los casos parciales se buscarán los archivos que correspondan. Hay que

señalar que para obtener la lista completa de recursos de un cliente la sintaxis es la siguiente:

```
$cliente GET / [ip_server[port_server]]
```

En todos los casos es opcional dar como parámetro la dirección IP del servidor y el puerto en que está escuchando las peticiones, siempre por defecto se toman estos datos del archivos de configuración del cliente, el cual se detalla en la sección de anexos.

Ejemplos de peticiones de este tipo son:

```
$cliente GET 2007/11/1
$cliente GET 2007/11 server.rrtp.org
```

El primer caso el comando solicita los recursos subidos por el cliente el primero de Noviembre del 2007 al servidor por defecto. El segundo comando pide todos los recursos subidos por el cliente en Noviembre del 2007 sobre el servidor especificado.

El segundo tipo de petición GET con el cliente es la descarga directa de un recurso, en cuyo caso la sintaxis es la siguiente:

```
$cliente GET AAAA/MM/DD/FILE [ip_server[port_server]]
```

En este caso se debe entregar la ruta completa del recurso, la cual señala la fecha exacta en que fue subido al servidor, un ejemplo de esta petición sería:

```
$cliente GET 2007/11/1/src.zip
```

En el ejemplo se solicita un recurso subido el primero de Noviembre del 2007 de nombre *src.zip* al servidor por defecto existente en el archivo de configuración.

b.- Petición PUT

En el caso de una petición del tipo PUT existe sólo una alternativa para el cliente, la cual es la subida de un recurso al servidor. La sintaxis de esta petición es la siguiente:

```
$cliente PUT /ruta/completa/a/FILE [ip_server[port_server]]
```

Como se ve, en este caso sólo se le debe entregar como parámetro al cliente la ruta completa del recurso que se quiere subir al servidor, el cliente internamente creará la URL objetivo de la petición con la fecha correspondiente, lo cual asegura que no se puedan realizar intentos de cambio de fecha en la subida de recursos.

Un ejemplo de esta petición es:

```
$cliente PUT /home/user/doc/src.zip
```

En caso que se haya subido correctamente el recurso, el cliente en la línea de comandos entrega la URL en que se ha subido en el servidor, la cual también puede ser

consultada en cualquier momento usando el comando GET respectivo.

c.- Petición DELETE

Para el caso de una petición DELETE la única función relacionada en el cliente es la del borrado de un recurso existente en el servidor. En este caso la sintaxis es similar a la de la descarga de un recurso, cambiando sólo la función HTTP, como vemos a continuación:

```
$cliente DELETE AAAA/MM/DD/FILE [ip_server[port_server]]
```

Al igual que en los casos anteriores son opcionales la IP y puerto del servidor. La fecha y nombre del recurso deben coincidir exactamente para que sea posible realizar el borrado del recurso, estos datos pueden ser obtenidos usando la petición de consulta para una fecha determinada.

Un ejemplo de esta petición es:

```
$cliente DELETE 2007/11/1/src.zip
```

En caso de éxito o fracaso el cliente entregará en la línea de comandos un mensaje que le deje claro esto al usuario.

8.4 Flujo de procesamiento de implementación

Se presenta a continuación una explicación detallada del procesamiento que realiza tanto el cliente como el servidor de una petición GET de descarga de un recurso, se detallará cada una de las clases involucradas y de los métodos usados por cada una. Finalmente se presenta un diagrama con el proceso completo mostrando las clases involucradas. Los pasos del flujo son los siguientes:

1. **Comienzo petición**: el usuario ejecuta una petición GET sobre una URL de recurso válida. La clase *Cliente* en su método *main()* ejecuta la petición, setea los valores de seguridad, y selección cual clase debe continuar con el procesamiento, en este caso la clase es *ClienteGet.*
2. **ClienteGet procesa petición**: se ejecuta el método *procesa(file, clienteDelete, clientePost),* el cual se encarga de realizar todas las peticiones necesarias para cumplir con el protocolo RRTP.
3. **ClientePost realiza login**: dentro del método procesa, el objeto *clientePost* realiza el llamado al método de login sobre el servidor.
4. **Servidor procesa login**: en el servidor, la clase *Servidor* recibe la petición POST de login por lo que llama a la clase *ServidorPost* para que lo procese.
5. **ServidorPost procesa login**: la clase *ServidorPost* ejecuta el método *procesaPost()* en el cual usa el objeto *ParserUrl* y su método *getAccionPedida()*, la cual entrega que método se está ejecutando por la URL objetivo, luego de lo cual se ejecuta le método *login()* de *ServidorPost.*
6. **ServidorPost hace login del usuario**: ServidorPost toma los datos de la petición, recibiendo el certificado del cliente y verificando su autenticidad. Luego de lo cual setea su estado como logeado para el sistema, enviando el código de respuesta pedido por el protocolo.
7. **ClienteGet crea transacción**: luego de recibida la respuesta de login exitoso, *ClienteGet* nuevamente hace uso del objeto *clientePost* y envía una petición de creación de transacción ejecutando el método *doTransaccion().*
8. **Servidor entrega petición a ServidorPost**: la petición nuevamente es derivada a *ServidorPost*, el cual realiza el mismo procesamiento anterior usando la clase *ParserUrl*, la cual nos dice que se está pidiendo la creación de una nueva transacción. Se ejecuta el método *creaTransaccion()* el cual inserta la nueva transacción en la tabla de transacciones y la asocia con el cliente logeado, y responde lo pedido.

9. **ClienteGet envía petición de descarga**: luego de tener el identificador de transacción válido retornado en la petición anterior, la clase *ClienteGet* puede enviar la petición GET para la descarga del recurso, se ejecuta el método *doDescarga()* el cual maneja el proceso.

10. **Servidor entrega la petición a ServidorGet**: al recibir la petición GET de descarga de recurso, el Servidor entrega esta a la clase *ServidorGet*, la cual realiza el procesamiento usando *ParserUrl* para determinar cual es el método pedido de acuerdo a la URL objetivo, determinando que se debe procesar una descarga de recurso, ejecutándose el método *enviaRecurso()* el cual analiza la URL objetivo y obtiene los datos necesarios para ir a buscar el recurso a la base de datos usando el objeto *RecursoDAO*. Luego de lo cual setea los *headers* necesarios y envía el recurso en el cuerpo de la respuesta como lo pide el protocolo.

11. **ClienteGet recibe el recurso**: la clase recibe el recurso enviado por el servidor en el método descarga() y lo escribe al disco en la ruta configurada en el archivo del cliente, la cual es la ruta por defecto para guardar los archivos recibidos. El método escribe a la salida estándar la ruta en disco en que es escrito el recurso.

12. **ClienteGet cierra transacción**: luego de recibido el archivo *ClienteGet* hace uso del objeto *clienteDelete* para que realice el cierre de la transacción en curso, para ello *clienteDelete* ejecuta el método *doCloseTransaccion()*, el cual envía una petición HTTP DELETE para el cierre de la transacción.

13. **Servidor recibe la petición de cierre**: la clase *Servidor* recibe la petición HTTP DELETE por el cierre de la transacción, llamando a la clase *ServidorDelete* y su método *procesaDelete(),* esta clase realiza la determinación de qué método se debe ejecutar usando la clase *ParserUrl,* ejecutando finalmente el método *cierraTransaccion()*. Este método actualiza el estado de la transacción como cerrada y envía la respuesta al cliente.

14. **ClienteGet envía petición de logout**: luego de recibida la confirmación del cierre de la transacción *ClienteGet* hace uso del objeto *clientePost* para enviar la petición del salida del sistema ejecutando el método doLogout(), el cual envía una petición HTTP POST a la URL destinada al deslogeo del cliente.

15. **Servidor deslogea al cliente**: nuevamente la clase *Servidor* recibe la petición POST y llama al objeto *ServidorPost* para que la procese ejecutando el método *procesaPost()*, este método usa la clase *ParserUrl* y determina que es una petición de salida del sistema, por lo que ejecuta el método *logout()*, el cual identifica al usuario por su llave privada y actualiza su estado como corresponde, enviando la respuesta al cliente.

16. **ClienteGet termina ejecución**: luego de recibida la respuesta satisfactoria *ClienteGet* termina la ejecución de su método *procesa()* por lo que la ejecución de la petición está completa de acuerdo a lo requerido por el protocolo RRTP

8.5 Extensibilidad del sistema

Se explica a continuación la forma de poder extender la actual implementación de forma de agregar funciones que se necesiten en casos particulares o ante la eventual extensión del protocolo RRTP.

La actual implementación permite que se extiendan las funcionalidades tanto para el servidor como el cliente, modificando clases específicas de forma de integrarlas naturalmente al actual sistema.

8.5.1 Servidor

Para extender las funciones soportadas por el servidor las siguientes clases tienen que ser modificadas: Servidor, ParserUrl, AccionSistema, TransaccionManager, y ServidorXX, con XX indicando el método HTTP que se quiere extender o crear(GET, PUT, POST, etc).

La primera clase que se debería modificar es Servidor, esto se debe hacer sólo en el caso en que la extensión del sistema se relación con integrar una nueva función HTTP para que sea manejada, como OPTIONS, HEAD, etc. Sólo en estos casos es necesario modificar esta clase.

La forma correcta de modificar el servidor en este caso es añadiendo un nuevo método de la forma: `protected void doFuncion(request, response)`

Aquí se debe cambiar *Funcion* por el nombre del nuevo método HTTP a añadir, por ejemplo OPTIONS, obteniendo: `protected void doOptions(request, response)`. Además se debe crear una nueva clase ServidorOptions(en este ejemplo) la cual es la encargada de procesar las peticiones OPTIONS. En el caso de sólo querer extender los métodos ya implementados, no es necesario tocar la clase Servidor.

La clase ParserUrl debe ser modificada para extender las funcionalidades, ya que la principal tarea de ésta es analizar qué función se pide. Para esto la clase mira cuál es la función HTTP ejecutada y luego de eso ve la sintaxis que posee la URL objetivo y teniendo estos datos puede determinar exactamente cual es la función a ejecutar.

La gran restricción para extender la funcionalidad es que la sintaxis de la URL destino no debe coincidir con alguna de las sintaxis ya existentes en el sistema. Al poder crear una nueva sintaxis, basta con agregar la condición en el método *getAccionPedida()* y se podrá obtener la nueva función agregada.

El paso paralelo al de modificar ParserUrl es agregar a la clase enumeración, AccionSistema, la nueva funcionalidad para que pueda ser retornada por el método *getAccionPedida* de la clase ParserUrl.

Luego se necesita modificar el método *validaEstadoMetodo* de la clase TransaccionManager el cual debe considerar en qué estados de la transacción es válido el nuevo método, teniendo que agregarse la nueva restricción dentro del método señalado para mantener la estructura.

Finalmente se debe modifcar la clase ServidorXX relacionada, o crear una nueva, en el caso en que se añade el manejo de un nuevo método HTTP distinto a GET, PUT, POST y DELETE. Se debe agregar la posibilidad de obtener esta nueva función en el método *procesaXX()* que cada ServidorXX posee, de forma de seguir con el estándar definido en la implementación. Luego de lo cual se debe implementar la nueva función que maneja las peticiones del cliente.

Si se quiere tener acceso a la persistencia de los recursos basta con modificar las clases DAO correspondientes o usar las ya existentes, dependiendo qué es lo que se necesita hacer.

Se ve que, en el mejor de los casos, para agregar una funcionalidad al servidor basta sólo con modificar 4 clases, lo que muestra que el sistema ofrece una muy buena extensibilidad en los casos que se necesite agregar funciones para hacer más versátil tanto el servidor como el cliente.

8.5.2 Cliente

Para agregar funcionalidades al cliente es más simple, puesto que son menos las clases involucradas en cada petición.

El primer caso es en el que se quiere agregar un nuevo método HTTP para ser soportado se debe crear una nueva clase *ClienteXX*, siendo XX el nombre del nuevo método HTTP. Esta nueva clase debe extender a *ClienteGenerico* para así seguir el estándar definido en la implementación. Además se debe incluir el método *procesa()* mediante el cual son ejecutados los procesos pertenecientes a cada una de estas clases.

También en el caso en que se crea una nueva clase *ClienteXX*, se debe modificar el método *main()* de la clase *Cliente* agregando al final de este la condición para que sea ejecutado cuando el nuevo método sea llamado en la línea de comandos.

En los casos que se quiera extender la funcionalidad de un método HTTP ya existente, sólo se debe modificar la clase *ClienteXX* añadiendo el nuevo método que realiza

la operación y modificando el método *procesa()* para que llame cuando corresponda al nuevo método implementado.

Como se ve es bastante simple la extensión del cliente RRTP implementado, luego la evolución del protocolo está bien sustentada por una implementación que da facilidades para ser extendida o modificada en su funcionalidad.

9. Conclusiones

El presente trabajo de título incluyó, en una primera parte, el diseño de un protocolo de transferencia de archivos, denominados recursos, haciendo uso de la arquitectura REST como base de éste, siendo la confiabilidad del protocolo su principal característica. Además se incluyó en el trabajo de título la implementación completa del protocolo diseñado, haciendo uso de herramientas tecnológicas de libre acceso, mostrando que es posible implementar un sistema de intercambio de recursos que cumpla con los estándares actuales de seguridad sin la necesidad de realizar inversiones monetarias en la adopción de las tecnologías a usar.

La primera parte del trabajo consistió en el estudio a cabalidad de la arquitectura REST de forma de establecer las propiedades de mayor aporte y que serían consideradas prioritarias en el posterior diseño.

El trabajo prosiguió con el análisis del estado del arte en sistemas de este tipo, encontrándose un pobre avance en el tema, ya que las propuestas encontradas carecían de un conjunto de funcionalidades mínimas deseables para su adopción y poseían implementaciones de referencia bastante limitadas o derechamente no poseían implementación alguna. Se encontraron 2 diseños propuestos usando arquitectura REST, los cuales fueron analizados rescatando las fortalezas y estableciendo las debilidades de cada uno. Este trabajo dejó bastante claro hacia dónde debía apuntar la construcción del sistema, de forma que el diseño usara las propiedades de la arquitectura REST que más aporte hicieran al sistema en términos de favorecer su confiabilidad, claridad, simpleza y, por consiguiente, su facilidad de adopción e interoperabilidad.

El proceso de diseño tuvo que enfrentarse a tener que cumplir con la característica de confiabilidad, siendo que su medio de transporte sería el protocolo HTTP, el cual no posee esta característica, por lo que se tuvo que tener especial cuidado con las peticiones HTTP de forma de asegurar esta característica. Se prosiguió el diseño teniendo en cuenta su compatibilidad con el protocolo de seguridad SSL, de forma que una implementación pudiera tener la opción de agregar la seguridad de forma directa. Las funciones consideradas por el protocolo son aquellas esenciales en cualquier diseño de sistema relacionado con la transferencia de archivos, las cuales son subida, descarga y borrado, que, aunque suene extraño, no habían sido consideradas en conjunto por ninguno de los diseños existentes.

El proceso de implementación del sistema consideró la utilización de tecnologías de libre acceso y de uso masivo, con el fin de confirmar que hoy en día este tipo de tecnologías pueden ser usadas en la implementación de sistemas completos que lleguen

fácilmente a ambientes de producción cumpliendo con los requisitos de seguridad necesarios. El uso del lenguaje de programación Java facilitó encontrar tecnologías relacionadas para el resto de la implementación, abarcando tanto el servidor de aplicaciones, la integración con certificados digitales, SSL y el uso de un núcleo de cliente HTTP que se encargó de las peticiones básicas en el lado Cliente del sistema, y demostrando que el diseño del sistema puede ser adoptado por clientes HTTP estándares hoy en día.

Luego de construidos el diseño y la implementación del sistema se consideran cumplidos a cabalidad los objetivos planteados para este trabajo de título, puesto que cumplen los objetivos planteados.

El trabajo realizado se enmarca en un terreno que ha sido poco explorado, ya que no existen sistemas REST de uso masivo en la transferencia de archivos puesto que los diseños propuesto no cumplen con las propiedades básicas que incentiven su adopción. Además la arquitectura REST ha tenido un auge recientemente, al dejar en claro su gran interoperabilidad al hacer uso de los métodos básicos HTTP para implementar toda la funcionalidad de los sistemas, lo cual permite la fácil implementación de módulos que interactúen con éstos, ya que la interfaz de comunicación uniforme favorece este proceso.

Tanto el protocolo como la implementación de este cumplen con todas las propiedades necesarias para ser una competencia real al actual estándar en la transferencia de archivos a través de servicios Web, que es WS-ReliableMessaging el cual usa el protocolo SOAP. Es una competencia real pues cumple las mismas funciones, sumado a que posee las ventajas de interoperabilidad, simplicidad, eficiencia y fácil adopción entregadas por la arquitectura REST. Siendo el principal desafío del protocolo su aceptación y entrada al mundo de los servicios de transferencia, de forma de posicionarse y ver las opiniones de la gente que adopta el sistema.

10. Bibliografía

[1] Bill de hÓra, “Reliable messaging specification (HTTPLR)”, Draft-httplr-01, 2005.

[2] Reach Ireland Goverment Project, “Reach Reliable Messaging Transport Protocol (RRMTP)”, Release 1.2, 2006.

[3] BEA Systems, IBM, Microsoft and TIBCO Software, “Web Services Reliable Messaging Protocol(WS-ReliableMessaging)”, Febrero, 2005.

[4] Roy T. Fielding, “Architectural styles and the design of network-based software architectures”, PhD Thesis, University of California, Irvine, 2000.

[5] Greg Goth, “Critics Say Web Services Need a REST”, IEEE Distributed Systems Online, vol. 5, no. 12, 2004.

[6] Sandesha de Apache. http://ws.apache.org/sandesha/

[7] RAMP de IBM. http://www.alphaworks.ibm.com/tech/ramptk

[8] Project Tango de Sun. http://weblogs.java.net/blog/haroldcarr/archive/2006/02/an_overview_of_1.html

[9] C. Alexander. “The Timeless Way of Building”. Oxford University Press, New York, 1979.

[10]URI definición. http://www.w3.org/Addressing/

[11]Servidor Apache Tomcat. http://tomcat.apache.org/

[12]SOAP especificación. http://www.w3.org/TR/soap/

[13]HTTPS especificación. http://www.ietf.org/rfc/rfc2660.txt

[14]XML especificación. http://www.w3.org/XML/

[15]M. and R. Sayre,“The Atom Syndication Format”,October 2004, Nottingham.

[16]Leonard Richardson, Sam Ruby, “Restfull Web Services”, First Edition, May 2007.

[17]Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1" RFC 2616, June 1999.

[18]Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels,” RFC 2119, March 1997.

[19]Alan O. Freier, Philip L. Karlton, Paul C. Kocher, “The SSL Protocol”, Version 3.0, November 1996.

[20]M. Nottingham, R. Sayre, “The Atom Syndication Format”, RFC 4287, December 2005.

[21]R. Rivest, “The MD5 Message-Digest Algorithm”, RFC 1321, April 1992.

[22] R. Housley, W. Polk, W. Ford, D. Solo, “Internet X.509 Public Key Infrastructure”, RFC 3280, April 2002.

[23]Openssl. http://www.openssl.org/

[24]Jetty. http://jetty.mortbay.org/

[25]Proyecto Jakarta HTTP Components. http://jakarta.apache.org/httpcomponents/

[26]HTTPClient. http://jakarta.apache.org/httpcomponents/httpclient-3.x/

[27]Motor base de datos Postgresql. http://www.postgresql.org/

11. Anexos

Anexos protocolo WS-ReliableMessaging

Se presentan a continuación formatos de los elementos XML que están presentes en las peticiones del protocolo WS-RM, que son necesarios para su correcto funcionamiento:

- Petición de creación de secuencia:

```
<wsrm:CreateSequence ...>
      <wsrm:AcksTo ...> wsa:EndpointReferenceType </wsrm:AcksTo>
      <wsrm:Expires ...> xs:duration </wsrm:Expires> ?
      <wsrm:Offer ...>
            <wsrm:Identifier ...> xs:anyURI </wsrm:Identifier>
            <wsrm:Expires ...> xs:duration </wsrm:Expires> ?
            ...
      </wsrm:Offer> ?
      ...
      <wsse:SecurityTokenReference>
      ...
      </wsse:SecurityTokenReference> ?
      ...
</wsrm:CreateSequence>
```

- Respuesta a creación de secuencia:

```
<wsrm:CreateSequenceResponse ...>
      <wsrm:Identifier ...> xs:anyURI </wsrm:Identifier>
      <wsrm:Expires> xs:duration </wsrm:Expires> ?
      <wsrm:Accept ...>
            <wsrm:AcksTo ...> wsa:EndpointReferenceType </wsrm:AcksTo>
            ...
      </wsrm:Accept> ?
      ...
</wsrm:CreateSequenceResponse>
```

- Sintaxis de un bloque de secuencia:

```
<wsrm:Sequence ...>
      <wsrm:Identifier ...> xs:anyURI </wsrm:Identifier>
      <wsrm:MessageNumber> xs:unsignedLong </wsrm:MessageNumber>
      <wsrm:LastMessage/>?
      ...
</wsrm:Sequence>
```

- Ejemplo de bloque SequenceAcknowledgement usado para informar recepción de mensajes exitosos:

```
<wsrm:SequenceAcknowledgement ...>
      <wsrm:Identifier ...> xs:anyURI </wsrm:Identifier>
      [ <wsrm:AcknowledgementRange ...
            Upper="xs:unsignedLong "
            Lower="xs:unsignedLong "/> +
      | <wsrm:Nack> xs:unsignedLong </wsrm:Nack> + ]
      ...
</wsrm:SequenceAcknowledgement>
```

- Ejemplo de bloque AckRequest, el cual solicita el bloque SequenceAcknowledgement:

```
<wsrm:AckRequested ...>
      <wsrm:Identifier ...> xs:anyURI </wsrm:Identifier>
      <wsrm:MessageNumber> xs:unsignedLong </wsrm:MessageNumber> ?
      ...
</wsrm:AckRequested>
```

- Bloque que se incluye en los casos de falla, para detallar el error presente en la transferencia:

```
<S:Envelope>
      <S:Header>
        <wsa:Action>
            http://schemas.xmlsoap.org/ws/2004/08/addressing/fault
        </wsa:Action>
        <!-- Headers elided for clarity. -->
      </S:Header>
      <S:Body>
        <S:Fault>
          <S:Code>
            <S:Value> [Code] </S:Value>
            <S:Subcode>
               <S:Value> [Subcode] </S:Value>
            </S:Subcode>
          </S:Code>
          <S:Reason>
             <S:Text xml:lang="en"> [Reason] </S:Text>
          </S:Reason>
          <S:Detail>
             [Detail]
             ...
          </S:Detail>
        </S:Fault>
      </S:Body>
</S:Envelope>
```

Clases adicionales implementación Servidor y Cliente

Para la implementación del servidor se usaron clases usadas transversalmente por el sistema completo, estas clases son: ***Utils.java,*** ***Constantes.java*** y ***HttpConstantes.java,*** las 2 clases involucradas con valores constantes de configuración de la implementación. El caso específico de ***HttpConstantes*** guarda los códigos de retorno HTTP, los cuales son usados por las clases *Servidor*. Se procede a detallar la clase Utils que tiene un aporte importante en la funcionalidad del sistema:

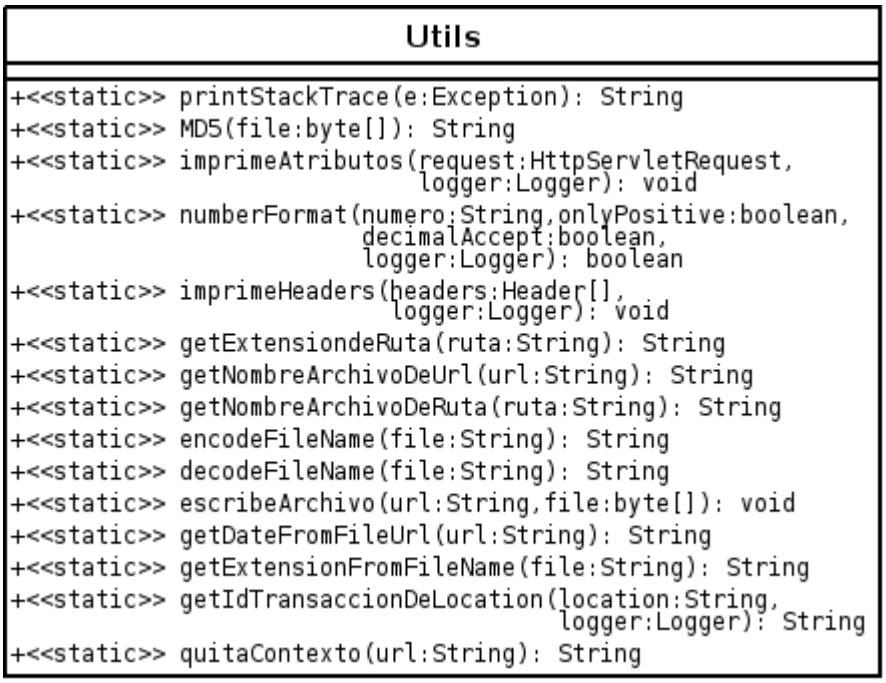


Ilustración 32 – Diagrama clase Utils

Los métodos señalados en su mayoría poseen funciones de obtención de datos a partir de las peticiones enviadas por los clientes, ya sea el identificador de transacción que se transfiere en el header *Location*, la fecha de un recurso que se quiere subir o descargar, el nombre del recurso obtenido de la URL, así como su extensión, etc.

Para el caso del cliente existe una sola clase adicional a la implementación y que no fue nombrada previamente en el diseño, este es una clase llamada ***ContantesCliente*** y que posee valores de configuración del este, y que son usados por la mayoría de las demás clases y que son centralizados en una ante la eventualidad de cambios de estos valores, lo que permite que ese cambio de extienda a cada parte donde son usados.

Archivos de Configuración Servidor

El servidor posee 2 archivos de configuración, uno es **servidor.properties** y el otro es el llamado **context.xml**. El primero de ellos guarda variables que son cargadas en el servidor al momento en que este parte, un ejemplo del archivo es el siguiente:

```
#jndi name para la busqueda de datasource
jdbc=java:/comp/env/jdbc/rrtpds
#tiempo de vida de transaccion en minutos
ttlt=30
```

Actualmente posee estas 2 propiedades que una indica el nombre de la conexión con la base de datos, y el segundo el tiempo de vida de una transacción en minutos.

El segundo archivo es el que realiza la conexión con la base de datos, creando el DataSource en el servidor, este archivo es el siguiente:

```
<Context     path="/rrtp"     docBase="rrtp"     debug="0"     reloadable="true"
crossContext="true">
<Resource
       auth="Container"
       description="conexion postgresql"
       name="jdbc/rrtpds"
       type="javax.sql.DataSource"
       username="user"
       password="password"
       driverClassName="org.postgresql.Driver"
       url="jdbc:postgresql://localhost:5432/rrtpdb"
       maxActive="10"
       maxIdle="5"
       maxWait="5000"
/>
</Context>
```

Se ve que posee todos los datos de configuración para acceder a la base de datos y hacer que esto sea flexible en casos de una instalación sobre una base de datos remota sobre ambientes de producción.

Configuración SSL

Para el uso de SSL se deben seguir los siguientes pasos tanto para el cliente como para el servidor. Estos pasos indican la creación de un repositorio keystore con la clave personal y un truststore con la clave pública de la entidad en que se confía.

Es necesario posee la herramienta Openssl instalada en la máquina, se procede entonces a crear la autoridad certificadora propia ejecutando:

```
openssl req -new -x509 -keyout private/cakey.pem -out cacert.pem
```

Comando que creará una autoridad propia con la cual se podrán firmar los certificados que así lo pidan. Luego de esto se procede a generar los certificados del cliente y del servidor. El proceso es el mismo para ambos, se ejecuta:

```
openssl req -new -keyout clientekey.pem -out clientecert-req.pem
```

Lo anterior genera un requerimiento para la autoridad certificadora, por lo que se autorizan los certificados de la siguiente forma:

```
openssl ca -in clientecert-req.pem -out clientecert.pem
```

Lo cual firma el certificado que ha hecho el requerimiento. Posteriormente se debe exportar el certificado firmado al formato PKCS12 para su posterior importación en el keystore, ejecutamos el siguiente código para ello:

```
openssl pkcs12 -export -in clientecert.pem -inkey clientekey.pem
                       -out clientecert.p12
```

Ahora ya tenemos el certificado en el formato necesario para ser importado. Se procede a usar la clase PKCS12Import del proyecto Jetty[24], para crear el keystore, ejecutamos:

```
java -classpath lib/org.mortbay.jetty.jar
org.mortbay.util.PKCS12Import clientecert.p12 cliente.keystore
```

Lo anterior crear el keystore, falta por último crear el truststore con la llave pública de la autoridad, para lo cual se usa la herramienta **keytool** de Java. ejecutamos:

```
keytool -import -v -keystore cliente.truststore -storepass 123456
-file ca-cert.pem
```

Con lo cual tenemos configurado SSL para ser usado por el cliente. Se debe repetir lo mismo para el servidor, siendo el único paso adicional el siguiente. Se debe ir al directorio de instalación de Tomcat y entrar al directorio **conf** y se debe modificar el archivo **server.xml** agregando los datos del conector ssl de la siguiente forma:

```
<!-- Define a SSL HTTP/1.1 Connector on port 8443 -->
     <Connector port="8443" maxHttpHeaderSize="8192"
```

```
                maxThreads="150" minSpareThreads="25"
maxSpareThreads="75"
                enableLookups="false" disableUploadTimeout="true"
                acceptCount="100" scheme="https" secure="true"
                clientAuth="true" sslProtocol="TLS"
        truststoreFile="/ruta/completa/tomcat/servidor_truststore"
truststorePass="passw"
keystoreFile="/ruta/completa/tomcat/servidor_keystore"
keystorePass="passw"/>
```

Este es el paso final, por lo que de estar bien configuradas las rutas y claves de acceso el sistema está listo para funcionar haciendo uso de SSL en ambos sentidos.

Archivo configuración Cliente

Para el cliente existe un archivo de configuración llamado **cliente.properties** el cual posee la siguiente información:

```
#ruta en disco keystore cliente
keystore=/ruta/al/archivo/de/cliente/cliente_keystore
#password de acceso a keystore
keystore_passw=pass_keystore
#ruta en disco truststore cliente
truststore=/ruta/al/archivo/de/cliente/cliente_truststore
#password de acceso a truststore
truststore_passw=pass_truststore
#directorio donde cliente guarda archivos recibidos
home_folder_cliente=/ruta/al/directorio/recibido_cliente/
#caracter separador de rutas, setear caso Unix o Ms
separador_ruta=/
#url/ip y puerto del servidor por defecto
url_default_server=172.17.69.49
port_default_server=8443
#numero de reintentos que hara el cliente en ejecuciones no exitosas
retries=5
#tiempo en segundos de espera entre cada reintento
delay=15
```

Este archivo posee toda la información necesaria para que el cliente pueda ser ejecutado sin problemas en cualquier máquina que posee Java. Se deben indicar la ruta del truststore y keystore del cliente así como las claves de acceso a estos. Además de los datos del servidor por defecto usado, tanto su dirección IP o URL y el puerto por defecto donde escucha. Finalmente se indican la cantidad de reintentos por petición que realiza el cliente y el margen de tiempo que existe entre peticiones indicado en segundos.