UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

Análisis, diseño e implementación de un sistema para envío de información de un servidor a una terminal remota

REPORTE DE INVESTIGACIÓN

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

Abraham Valle Alvarez.

TUTOR:

Dr. Benjamín Macías Pimentel.

2008

Datos del Alumno

Abraham Valle Alvarez
55 52 51 46
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la computación

Datos del tutor

Doctor
Benjamín
Macías
Pimentel

Datos del sinodal 1

Doctor
José de Jesús
Galaviz
Casas

Datos del sinodal 2

M. en C.
Verónica Esther
Arreola
Ríos

Datos del sinodal 3

M. en C.
Dante
Ortiz
Ancona

Datos del sinodal 4
L. en C. C.
Francisco Lorenzo
Solsona
Cruz

Agradecimientos

Para mi madre Rosa Álvarez Gadea y mi abuelo Francisco Valle Chavarría, que en el ocaso de sus vidas dieron suficiente luz para alumbrar el amanecer de la mía. Gracias a Huilver Nolasco y a Daniel Estévez porque encontré en ellos un gran complemento y porque no les basto con ser mis compañeros de clases, se volvieron mis compañeros de vida, mis amigos y mis hermanos.

A mi tutor Dr. Benjamín Macías Pimentel, porque me enseño cosas en un estilo muy particular, porque se volvió mi ejemplo académico a seguir, porque lo admiro, lo respeto y le guardo un gran cariño como mi tutor.

Gracias a todos los profesores que me dieron algún curso, pues en su mayoría no les basto con impartir sus enseñanzas dentro de las aulas, también me dieron clases de vida fuera de ellas. Particularmente un agradecimiento especial a José de Jesús Galaviz, porque me dio la oportunidad no solo de ser su alumno, también de ser su ayudante y de volverme uno más de todos los amigos que ha hecho en esta facultad; a María de la Luz Gasca, porque me dio las bases de lo que más amo de esta carrera, la teoría de algoritmos, y porque pase con ella todo tipo de momentos como alumno, desde algunos muy buenos hasta otros no tanto, siempre la tendré en mi recuerdo.

Gracias a todos los amigos que me han acompañado hasta este día, porque son ellos quienes han tenido la gran virtud de soportarme, impulsarme, el tiempo y el cariño para acompañar todas mis risas y mis llantos, la sabiduría para darme consejos y el amor que solo un amigo sabe dar. Mención especial a Pepe, sin ti amigo mi vida sería completamente caótica; a Miguel, porque jamás he conocido persona más noble en el mundo; a Job porque gracias a su experiencia sabe ayudarme a encontrar mi camino; a Adrian porque me hace darme cuenta que mi vida no es tan difícil como suelo pensarlo; a Ángel y Jouseff, por ser mis amigos a pesar de mi carácter; a Gaby porque sabe cuando no debe darme la razón; a Martha, porque es mi amiga y mi testigo desde mi infancia; a José Luis, Jorge, Israel y Gerardo, porque sin ellos no hubiera sobrevivido a la pubertad, y gracias a toda persona que se halla cruzado en mi camino, pero que por razones de espacio no puedo hacerles mención especial.

Yo me debo a toda la gente que esta o a estado a mi alrededor, a todos ellos muchas, pero muchas gracias por haber compartido un poco de su tiempo junto a mí.

1. Introducción General

Con telefonía celular y asistentes electrónicos PDA (agendas electrónicas) se ha incrementado el uso de diversas aplicaciones que permitan mantener a sus usuarios en comunicación y además guardar u obtener, de alguna manera, información importante para los usuarios y que ésta sea recuperable en todo momento.

Una manera de cumplir con ambas cosas, es utilizar Internet para mantener las terminales remotas en contacto con servidores (computadoras). Las terminales pueden realizar peticiones de una manera asíncrona, es decir, realizando la petición de una tarea a un servidor, sin que se mantenga un enlace permanente con la terminal y respondiendo con un estado final de la tarea a la terminal que realizó la petición una vez que esta finalice.

Se propuso un proyecto de investigación para estudiar el uso de terminales remotas para la comunicación distante con Internet. El propósito es el de producir una arquitectura que permita a un usuario la comunicación asíncrona vía SMS (short message services) con un servidor remoto conectado a Internet junto con el software que haga esto posible. Esto requiere el estudio y la implementación de una arquitectura básica, el estudio de filtros para reducir la variedad de formatos (HTML, XML, texto plano, etc.) a uno solo, software para manipular la variedad de tipos de interacción, y técnicas de lenguaje natural para extraer la información relevante y producir síntesis adecuadas para textos en español.

El presente documento se complementa con el documento [1], ambos documentos corresponden a la especificación de la primera fase del proyecto PAPIIT [2], por lo que se recomienda su lectura.

1.1 Objetivos

El presente proyecto tiene como objetivo el estudio y la construcción de una arquitectura que soporte sistemas SRII (Sistemas Remotos de Interacción con Internet), y demuestren la viabilidad del concepto para un grupo de servicios. Para llevar a cabo un proyecto exitoso, requerimos acotar un poco más el área de estudio. En especial, estamos interesados en aplicaciones informáticas del tipo:

- cliente-servidor
- comunicaciones inalámbricas

- transacciones destinadas al intercambio de información
- aplicaciones asíncronas a través de SMS

Para lograr el objetivo planteado consideramos los siguientes puntos:

I. Transmisión de datos entre las terminales remotas y el servidor, estudio de los distintos tipos de protocolos y lenguajes de programación, que permiten implantar sistemas de un modo flexible y portable.

II. Análisis, diseño e implementación de un sistema informático para transferencia de información entre el servidor y las terminales remotas a nivel básico. Este sistema se ejecutará sobre la funcionalidad más básica del punto anterior.

III. Diseño de una arquitectura cliente-servidor con base en el punto anterior que permite llevar a cabo los tipos básicos de transacciones contempladas en el proyecto.

2. Resumen de las terminales remotas disponibles en el mercado, sus ventajas y desventajas

2.1 Introducción

Definiré una terminal remota como aquella que permite acceder a los servicios de un servidor en Internet mediante algún protocolo (p. ej. *Telnet*, *ssh*). Dicha terminal, puede ser de varios tipos, como lo son: computadoras caseras, teléfonos celulares, agendas personales electrónicas entre otras.

Para explicar lo que es una terminal remota y como se comporta, podemos descomponerla en dos cosas: un *hardware* ajeno al servidor y un sistema remoto de interacción (*software*).

Los métodos de comunicación entre una terminal remota y un servidor son variados y una manera de clasificarlos seria como alambrados e inalámbricos. Las formas alambradas pueden ser mediante cables de red, USB, seriales, etc.; mientras que los inalámbricos funcionan a través de tarjetas del tipo *Bluetooth*, infrarrojas, WI-FI, entre otras.

Para objetivos de este proyecto, se utilizará dispositivos que dentro de sus capacidades de hardware se comunique con un servidor de manera inalámbrica, y que tenga pueda manejar SMS (*Short Message Service*).

Así mismo, éstas deben tener la capacidad de ser programables por un lenguaje de alto nivel, específicamente, Java en versión 5 o superior.

2.2. Tipos de terminales Remotas

Dados los requerimientos mencionados anteriormente, nos enfocaremos en las terminales remotas que cumplan con lo establecido en la introducción.

2.2.1 Terminales remotas de tipo PDA (*Personal Digital Assistant*)

Estas son terminales originalmente diseñadas como agendas electrónicas con un sistema de reconocimiento de escritura. Hoy en día se pueden usar como una computadora doméstica (ver películas, crear documentos, navegar por Internet, etc.).

Los dispositivos PDA, ofrecen la conectividad de un teléfono móvil, de un dispositivo de correo electrónico, de un explorador Web, así como un organizador, todo en una sola unidad.

2.2.1.1 Terminales Remotas (PDA) de tipo *Blackberry*

Software para cualquier dispositivo Blackberry.

Funciona con *BlackBerry Enterprise* Server

Compatible con:

Versión 2.1 o superior para Microsoft Exchange versión 2.0 con Service Pack 2 o superior para Lotus Domino Versión 4.0 o superior de Novell GroupWise.

Para desarrollar aplicaciones en JavaME, se tienen los siguientes requerimientos en este tipo de terminales.

Microsoft Windows 2000 SP1 or superior, o Microsoft Windows XP

PC con procesador Intel Pentium III o compatible (800 MHz o mayor, 512MB RAM)

Java 2 SDK, Standard Edition v1.4 (JDE 3.7, 4.0, 4.0.2)

Java 2 SDK, Standard Edition v5.0 (JDE 4.1)

Descripción General de la serie 7100g.

- Especificaciones del dispositivo.
  - Teléfono.
  - Correo electrónico.
  - SMS.
  - Explorador Web.
  - Libreta de direcciones, calendario, bloc de notas y lista de tareas integrados.
  - Compatibilidad Bluetooth.
  - Visualización integrada de archivos adjuntos.
  - Compatibilidad con software popular de administración de información personal PIM (*Personal Information Manager*).
  - 32 MB de memoria.
  - Memoria interna de 32 MB; SRAM de 4 MB.
  - Sistema operativo Propietario RIM TM.
  - Tecnología Java que soporta aplicaciones creadas en este lenguaje.

- Característica de la red.
  - GSM/GPRS (Global System for Mobile Communications/*General Packet Radio Service*) de 850, 900, 1800 y 1900 MHz.
- Servicios de datos
  - Conectividad GPRS que permite transmitir datos por medio de una conexión móvil y cable de sincronización.
  - Recibe y realiza llamadas sin perder la conexión.
  - WAP (*Wireless Application Protocol*) version 4.0.2. Este equipo cuenta con un micro navegador por lo que se podría utilizar el servicio de Conexión móvil.
  - XHTML.

Tiene como ventaja una buena capacidad de memoria interna, pero en desventaja está atado a un sistema operativo propietario y tiene un alto costo.

Descripción General de la serie 7280.

- Especificación del dispositivo.
  - Teléfono.
  - Correo electrónico.
  - SMS.
  - Explorador Web.
  - Visualización integrada de archivos adjuntos.
  - Compatibilidad con software popular de administración de información personal (PIM).
  - 16 MB de memoria flash mas 2MB de SRAM.
  - MODEM inalámbrico RIM incorporado.
  - Correo electrónico inalámbrico
  - Soporte para la sincronización con la PC.
- Características de la Red.
  - GSM/GPRS de 850/1800/1900 MHz.
- Servicios de datos.
  - GPRS.
  - WAP Versión 2.0.

Tiene como ventaja una buena capacidad de memoria, que puede ser incrementada usando memoria tipo flash, así como la posibilidad de comunicación vía un MODEM inalámbrico. En desventaja está atado a un sistema operativo propietario y un alto costo.

Descripción General de la serie 7290.

- Especificación del dispositivo.
  - Teléfono.
  - Correo electrónico.

    - SMS.
    - Bluetooth.
    - Memoria flash de 32 MB; SRAM de 4 MB.
    - Sistema operativo Propietario RIM TM.
    - Tecnología Java que soporta aplicaciones creadas en este lenguaje.

- Características de la red.
    - GSM/GPRS de 850, 900; 1800 y 1900 MHz.
- Servicios de datos.
    - GPRS.
    - WAP Versión 4.0.2.
    - XHTML

Tiene las mismas características que la serie 7280, excepto que de fábrica tiene una mayor memoria flash, pero conserva ventajas y desventajas.

2.2.1.2 Terminales Remotas (PDA) de tipo Palm

Para desarrollar aplicaciones en JavaME, se tienen los siguientes requerimientos en este tipo de terminales.

Windows2000 o Windows XP (versiones posteriores pueden soportarlo) para Palm.

Java 2 SDK, Standard Edition v1.4.

Java 2 SDK, Standard Edition v5.0.

Descripción general de la serie PalmOne Treo 600

- Especificación del dispositivo.
    - Conectividad vía Infrarrojo, Bluetooth o MMS.
    - Java soporta aplicaciones creadas con esta tecnología.
    - Joystick (Navegador) Navegador 5 vías.
    - Memoria 32 MB RAM (24 MB disponibles para el usuario).
    - Navegador Web incluido Blazer PDA.
    - Pantalla a color táctil (*touch screen*).
    - Procesador ARM de 144 MHz.
    - Sincronización SyncML HotSync de palmOne con una computadora.
    - Sistema Operativo Palm OS 5.2.1H.
    - Tarjeta de memoria SD/MMC, habilitado para SDIO.
- Características de la red
    - GSM/GPRS de 850, 900, 1800 y 1900 MHz.
- Servicios de datos

    - CSD (*Circuit Switched Data*). Te permite transmitir datos por medio de una conexión móvil a una velocidad máxima de 9.6 Kbps. Utiliza canales de voz.
    - GPRS.
- HTML Soporta múltiples estándares Web e inalámbricos: HTML 4.0, XHTML Basic y Mobile Profile, WML 1.3, HTML, una variedad de formatos de imágenes y Java Script.

Se tiene la ventaja de poder transmitir datos vía CSD, tiene un procesador más poderoso en comparación con otras terminales, y utiliza tarjetas de memoria SD/MMC, se tiene la desventaja en cuestiones de costo.

Descripción de la serie PalmOne Treo 650.

- Especificación del dispositivo.
    - Conectividad Vía Infrarrojo, Bluetooth o MMS.
    - Java Soporta aplicaciones creadas en esta tecnología.
    - Joystick (Navegador) Navegador 5 vías
    - Memoria 32 MB RAM (24 MB disponible para el usuario).
    - Mensajes Multimedia MMS.
    - Navegador Web incluido Blazer Pantalla Color TFT de 320 x 320 de alta resolución (*touch screen*).
    - Procesador Intel PXA270 de 312 MHz.
    - Sincronización SyncML HotSync de palmOne con una computadora.
    - Sistema Operativo Palm OS 5.4.
    - Tarjeta de memoria SD/MMC, habilitado para SDIO (*Secure Digital Input Output*).
- Características de la red.
    - GSM/GPRS de 850, 900, 1800 y 1900 MHz.
- Servicios de datos.
    - EDGE (*Enhaced Data Rates for Global Evolution*) te permite mayor rapidez en transmisión de datos hasta 384 kbps.
    - CSD (*Circuit Switched Data*).
    - GPRS.
    - HTML Soporta múltiples estándares Web e inalámbricos: HTML 4.0, XHTML Basic y Mobile Profile una variedad de formatos de imágenes y Java Script.

Mejora en relación con la serie Treo 600 en cuestiones de velocidad de procesamiento, salvo eso, mantiene las mismas ventajas y desventajas.

2.2.2 Otros tipos de dispositivos móviles

La empresa Motorola, tiene algunos modelos con las características necesarias para desarrollar los objetivos de este proyecto, por lo mismo, se integran algunos modelos.

Descripción de la serie Motorola E398.

- Especificación del dispositivo.
  - Bluetooth.
  - Conexión inalámbrica con gran capacidad de transmisión y recepción de datos la cual permite conectarse con otros dispositivos Bluetooth a una distancia máxima de 10 m.
  - Joystick (Navegador).
  - Memoria interna (capacidad) 5MB.
  - Mensajes Multimedia (MMS).
  - Modem.
  - Tarjeta de memoria Memoria Expandible 128Mb TransFlash.
  - Java soporta aplicaciones creadas sobre esta tecnología.

- Características de la Red.
  - GSM/GPRS de 900, 1800 y 1900 MHz.
- Servicios de Datos.
  - CSD con canales de voz.
  - GPRS.
  - WAP Versión 2.0.
  - XHTML.

Descripción de la serie Motorola L6.

- Especificaciones del dispositivo.
  - Bluetooth.
  - Conexión inalámbrica con gran capacidad de transmisión y recepción de datos la cual permite conectarse con otros dispositivos Bluetooth a una distancia máxima de 10 m.
  - Memoria interna (capacidad) 8 Mb (Compartida Java, MMS, Multimedia).
  - Mensajes Multimedia (MMS).
  - MODEM.
  - Permite la sincronización de información entre dispositivos y servicios de red.
  - Java, soporta aplicaciones creadas en esta tecnología.

- Características de la Red
  - GSM/GPRS de 850, 1800 y 1900 Mhz.

- Servicios de Datos
  - CSD con canales de voz.
  - HSCSD (High Speed Data) permite transmitir datos por medio de una conexión móvil a una velocidad máxima de 43.2 Kbps. Utiliza canales de voz.
  - GPRS.
  - WAP versión Wap 2.0.

2.3 Conclusión

Estos son los modelos más importantes de terminales remotas en el mercado. Cabe mencionar, que todas cubren la necesidad de SMS como medio de comunicación a través de algún protocolo.

También cabe señalar que se ha seleccionado la tecnología Bluetooth para manejar la comunicación entre la Terminal remota y el servidor.

Otras opciones eran WI-FI o infrarrojo, pero estas requerían de hardware extra en la terminales remotas o de opciones de pago a un proveedor del servicio; con Bluetooth, se puede montar incluso una pequeña red si sus máquinas no rebasan los 10 metros y cumple con los requerimientos de ser equipos inalámbricos, suficiente razón para que se puedan alcanzar los objetivos de este proyecto.

Es necesario comentar que para ningún modelo, la tecnología WAP será necesaria, pues esta se utiliza en los navegadores convencionales, es decir, los que mantienen una comunicación permanente con un servicio de Internet, sin embargo, los modelos del mercado actual, tienen incluido este tipo de servicio y no se encontró alguno con las características necesarias para trabajar en este proyecto y que no contuviera esta tecnología.

Así mismo, los PDAs, no se seleccionaron por varias razones, comenzando por su elevado costo, además de su dependencia a trabajar con sistemas que no manejan la filosofía de software libre. En otros casos, como en las terminales Blackberry, se tiene que desarrollar en ambientes de trabajo ya dispuestos para sus dispositivos, lo cual dificulta desarrollar la aplicación para el servidor en un ambiente Linux.

3. Análisis, diseño e implementación de un sistema informático para transferir información entre un servidor y terminales remotas a nivel básico

A continuación se describe el sistema entero que tuve que desarrollar durante el periodo de trabajo del proyecto. El énfasis estuvo en ir de la fase de especificación al diseño; a pesar de ello, algunas partes fueron implementadas para demostrar que el análisis era correcto.

3.1 Especificación, análisis y diseño alternativos del sistema que transfiere información entre el servidor y las terminales remotas a nivel básico

En esta sección se describe el análisis y el diseño de un sistema que permita la transferencia de información desde una terminal remota hacia un servidor y viceversa. Además del diseño de algunas clases necesarias para proveer servicios. Esta es una alternativa a la presentada por el tutor que se encuentra en el anexo A y B.

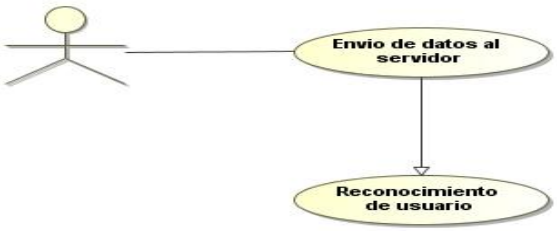
3.1.1 Análisis de requerimientos funcionales

I) Escenario 1.

Un usuario solicita un servicio desde una terminal remota vía SMS a un servidor. Cuando el mensaje sale del usuario, este llega a otra terminal remota, que mediante algún medio de comunicación (Bluetooth, cables, etc.) envía el contenido de la petición al servidor.

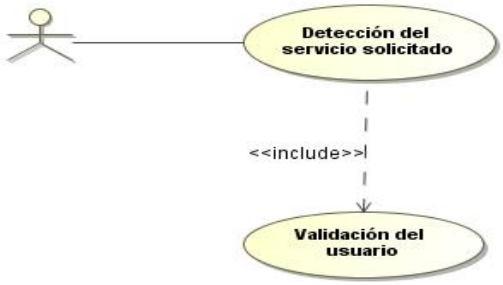
Nombraré puente a la terminal remota que llegan los SMS de las terminales remotas de los usuarios, ya que su utilidad radica en el hecho de ser un dispositivo que recibe SMS y los transmite al servidor encargado de satisfacer la petición.

Una primera necesidad, es identificar a los usuarios válidos para solicitar un servicio desde una terminal, para lo cual, necesitamos un sistema de validación de usuarios en el servidor. Cabe señalar que, inicialmente, cualquier terminal remota es potencialmente un cliente, pero solo se dará un servicio, a aquellos clientes que cuenten con ciertas características (que posean el software necesario por ejemplo). El siguiente diagrama, nos muestra cómo funciona este caso de uso.



Validacion de Usuario

Una vez que el usuario es validado y se establece una comunicación se analiza el tipo de petición que el cliente solicita, el servidor debe tener un proceso activo permanentemente que pueda identificar su tipo. Observe el siguiente diagrama.



Detección del servicio solicitado

Una vez identificado el tipo de servicio, esta solicitud se debe guardar en una bitácora dentro del servidor, para poder darle un seguimiento a los servicios que más se proporcionan, ver cuáles son más susceptibles a fallos y permitir el análisis futuro de los mismos. Las solicitudes se guardaran en forma de archivos (un archivo por solicitud). Observe el siguiente

Respuesta del servidor a la terminal remota

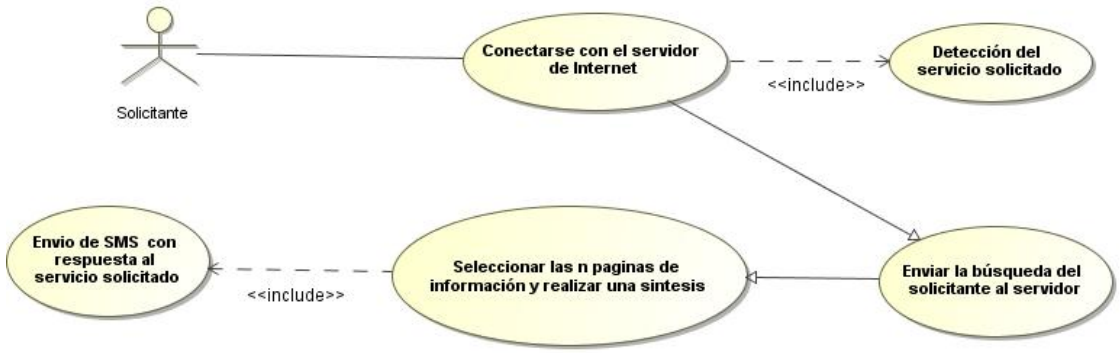
diagrama.

I.I). Descripción de servicios posibles.

Dentro del gran grupo de servicios que se pueden dar, se seleccionaron los siguientes, como parte del primer análisis. Cabe señalar que todos los servicios siempre deben terminar con el envío de un SMS al solicitante, el cual debe contener el resultado del mismo.

I.I.I) Solicitud de información.

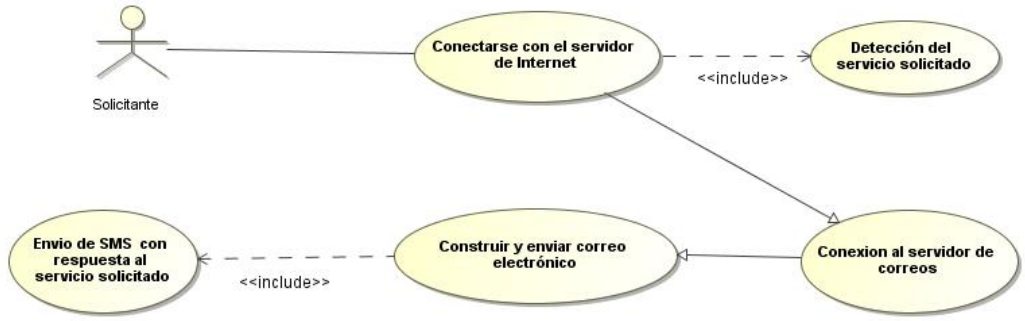
Un cliente, necesita información sobre un conjunto de palabras clave que definen los parámetros de su búsqueda; el servidor debe ser capaz de tomarlas y buscar mediante Internet las páginas relacionadas pudiendo realizar una síntesis del resultado de la búsqueda. Observe el siguiente diagrama.



Búsqueda de información via SMS

I.I.II) Solicitud de envío de correo electrónico.

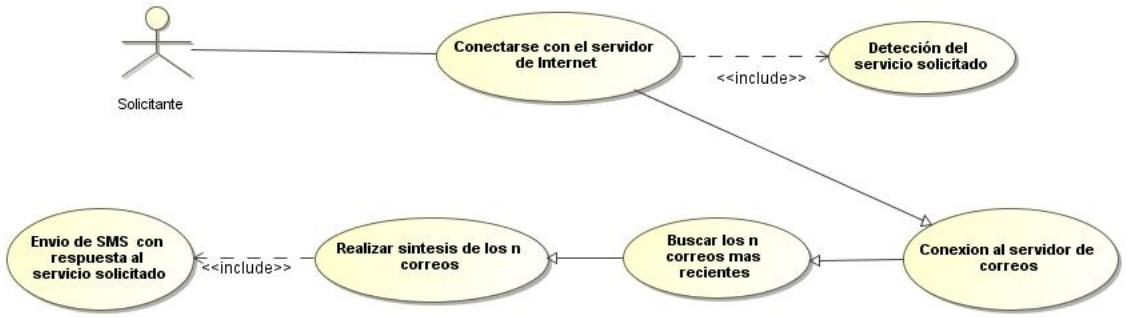
Un cliente, necesita enviar un correo electrónico desde una cuenta en un servidor de correos, el cliente provee el remitente, la clave del mismo, el destinatario y opcionalmente el encabezado y el cuerpo del correo. Observe el siguiente diagrama.



Solicitud de envio de correo ele ctrónico

I.I.III) Solicitud de recepción de correo electrónico.

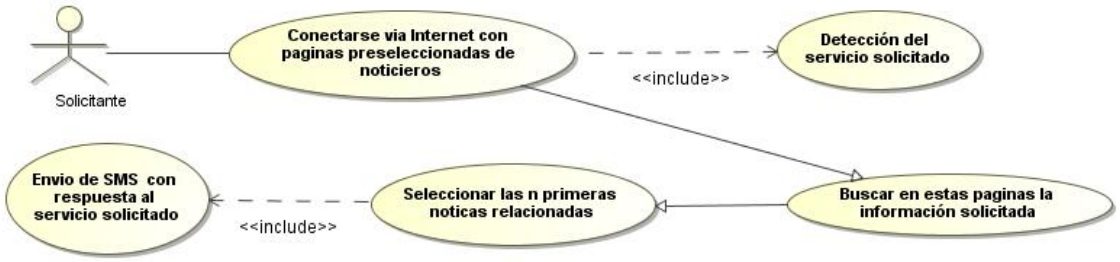
Un cliente, necesita revisar el correo electrónico de una cuenta en un servidor de correos, el cliente provee el correo, clave del mismo y número N de correos a revisar. El sistema debe generar una lista de los N correos. Observe el siguiente diagrama.



Solicitud de recepción de correos via SMS

I.I.IV) Solicitud de búsqueda de noticias.

Un cliente, necesita información sobre un conjunto de noticias definido por los parámetros de su búsqueda; el servidor debe ser capaz de tomarlos y buscar mediante un informador de noticias las que estén relacionadas pudiendo enviar el resultado de la búsqueda. Observe el siguiente diagrama.



Solicitud de búsqueda de noticias via SMS

I.I.V) Ejecución remota de un comando.

Un cliente, necesita ejecutar un conjunto de instrucciones en una computadora, la cual puede ser vista por el servidor vía Internet, el cliente debe dar además del conjunto de instrucciones, la dirección IP de la máquina y un usuario junto con su clave. Observe el siguiente diagrama.

Solicitud de ejecución remota de comando via SMS

3.1.2 Diseño de clases

A partir del análisis de los requerimientos funcionales, describiremos un diseño de clases que abstraigan los distintos casos de uso en base al paradigma de orientación a objetos.

En principio, se nota la necesidad de un intercambio de información entre la terminal remota que recibe los SMS de los clientes, y el servidor que satisface las peticiones. De esta manera, se establece una interfaz Conmutador, que definirá la forma de intercambiar información entre ambas cosas.

3.1.2.1 Interfaz Conmutador

Descripción

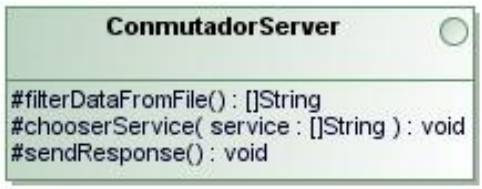
En esta interfaz, se debe tener el nombre del servidor y el puerto asignado para escuchar peticiones.

Para los atributos, se crearán métodos para asignarles valor y otros para solicitar los mismos; esta es la base para los procesos que estarán verificando la solicitud de servicios, por lo tanto, se debe tener un método que "escuche" cuando un servicio es solicitado.

Métodos

- setNameServer(String nameServer): asigna un nombre al servidor de aplicaciones.
- String getNameServer(): devuelve el valor del nombre del servidor.
- setNumPort(Integer numPort): asigna un número de puerto para conexión al servidor.
- getNumPort(): regresa el valor del puerto.
- boolean listen(String Received): verifica cuando ocurre un evento de llegada de información para un servicio solicitado.

3.1.2.2 Interfaz ConmutadorServer



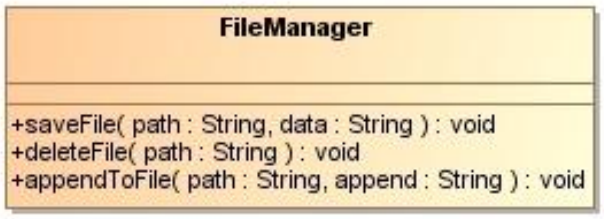
Descripción

Para que la interfaz Conmutador sea implementado del lado del servidor, esta debe ser extendido por otra interfaz ConmutadorServer, que declara métodos que permitan identificar si tiene un archivo en la bitácora con información para un nuevo servicio; en caso de ser así reconocerlo, asignar responsabilidad a una clase(s), que lleve(n) a cabo el servicio y enviar un archivo con respuesta del estado final del mismo.

Métodos

- String[] filterDataFromFile(String path): a partir de los datos de un archivo, construye un arreglo con los parámetros necesarios para que la(s) clases(s) que implementen un servicio, puedan llevarlo a cabo.
- chooserService(String[]): llama a la(s) clase(s) que correspondan al servicio.
- sendResponse(): envía la respuesta del estado final del servicio.

3.1.2.3 Clase FileManager



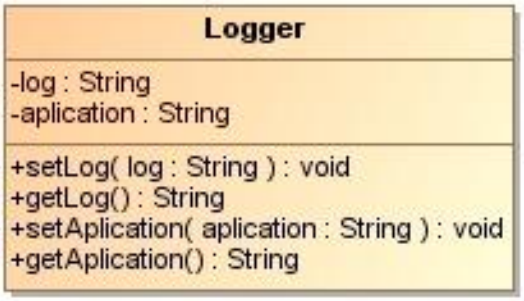
Descripción

La clase FileManager se encarga de manipular los archivos que se escriben en el servidor.

Métodos

- saveFile(String data, String path): guarda una cadena en un archivo, sobre una ruta específica.
- deleteFile(String path): borra el archivo de la ruta especificada como parámetro.
- appendToFile(String append, String path): agrega una cadena a un archivo especificado como parámetro.

3.1.2.4 Clase Logger



Descripción

Se encarga de controlar la bitácora del servidor, tanto de escribir el estado de los servicios en un archivo, como devolver dichos archivos.

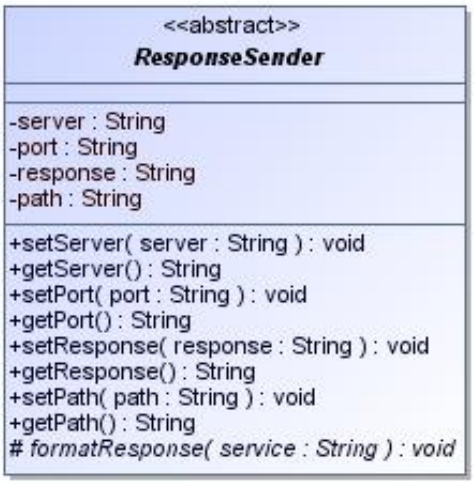
Atributos

- log:String Descripción: Cadena que define el contenido del archivo en la bitácora.
- aplication:String Descripción: Nombre del servicio para el cual se escribe en un archivo en la bitácora.

Métodos

- setLog(String log): asigna la cadena que se escribirá en el archivo de la bitácora, asignándola al atributo log.
- String getLog (): devuelve el valor del atributo log.
- setAplication(String aplication): asigna valor del atributo aplication.
- String getAplication(): devuelve el valor del atributo aplication.
- SaveLogFile(String path): guarda el valor del atributo log en un archivo que tiene la ruta especificada como parámetro.

3.1.2.5 Clase Abstracta ResponseSender



Descripción

Clase abstracta que guarda la respuesta que se da de un servicio en base al formato definido por alguna clase que la implemente.

Atributos

- server:String Descripción: Nombre del servidor que atendió la petición.
- port:Integer Descripción: Número del puerto de comunicación con la terminal remota.
- response:StringBuffer Descripción: Valor que será enviado a la terminal remota para poder construir un SMS.
- path:String Descripción: Valor de la ruta donde se mandará el flujo del mensaje.

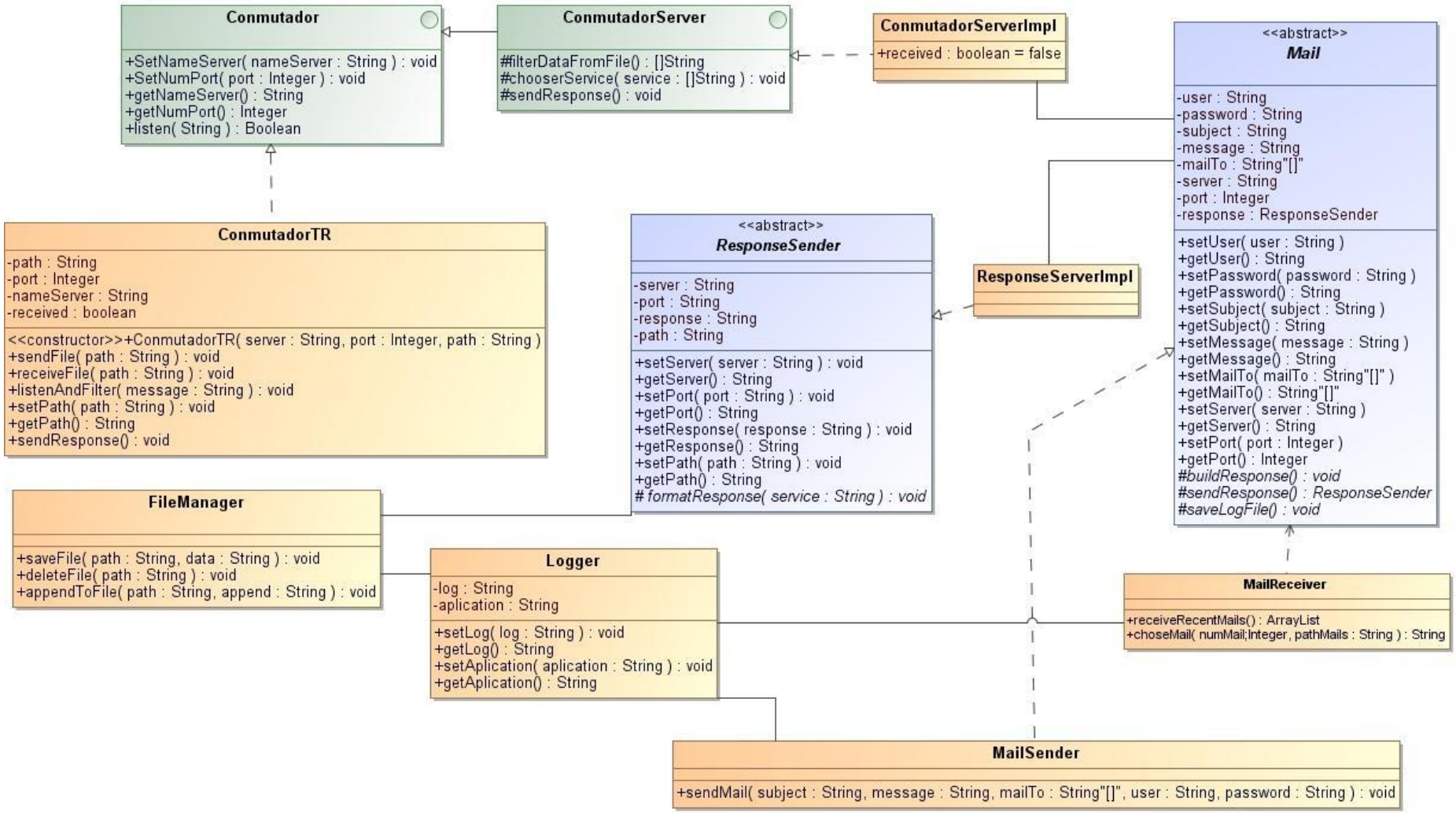
Métodos

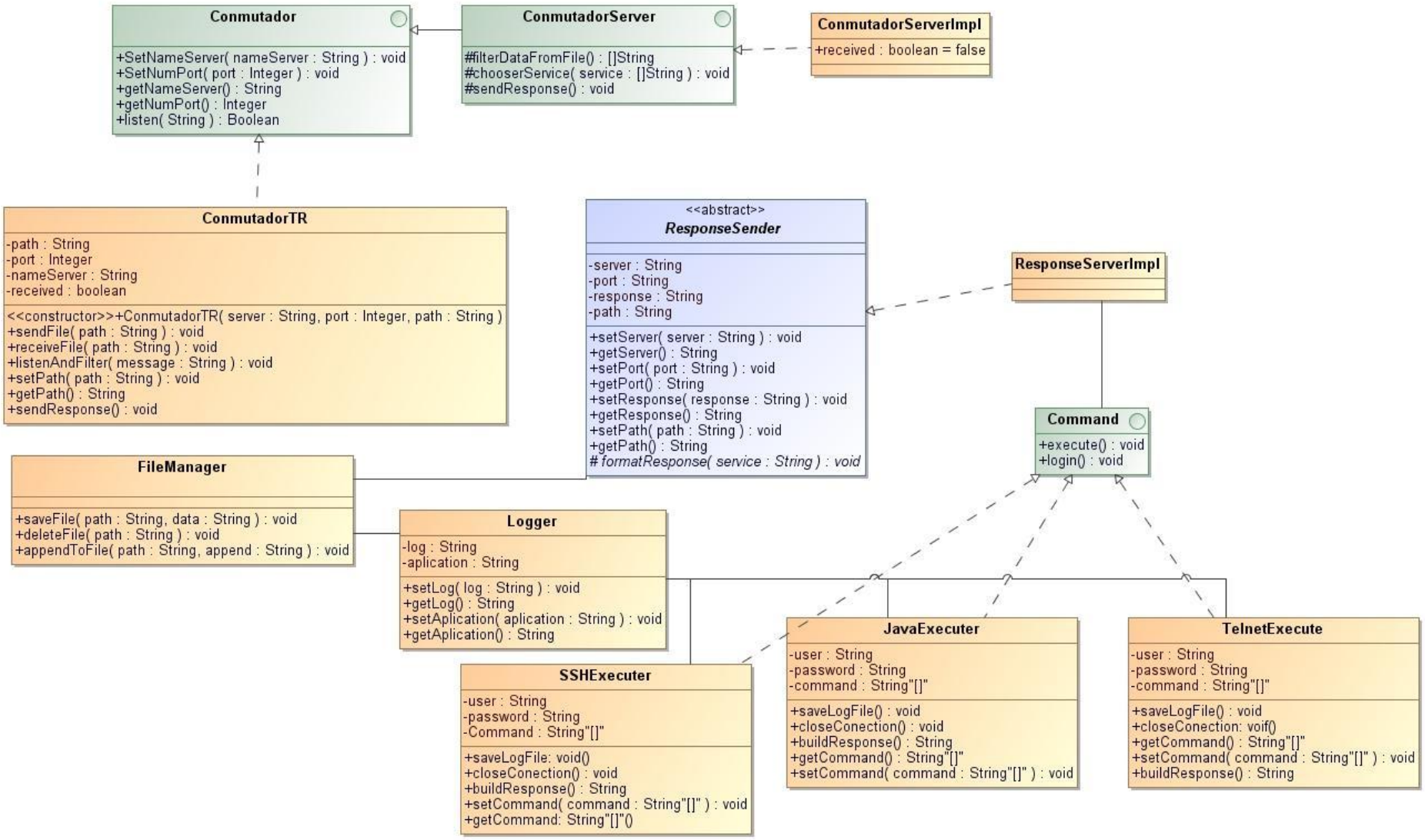
- setResponse(String response): asigna un valor del atributo response.
- String getResponse(): devuelve el valor del atributo response.
- saveResponse(String path): guarda la cadena de respuesta en un archivo con ruta especificada por el parámetro.
- sendResponse(): envía la respuesta de un servicio a la terminal remota.
- abstract formatResponse(String service): asegura que quien implemente esta clase, debe dar formato al mensaje de respuesta para el servicio que le corresponde.
- setPath(String path): asigna el valor del atributo path.
- String getPath(): devuelve el valor del atributo path.

3.1.3 Diagramas UML.

Los siguientes diagramas de clases, ejemplifican la manera en la cual interactúan las clases descritas con anterioridad, además de otras desarrolladas para el mismo proyecto, se sugiere consultar el texto referido en [3].

3.1.3.1 Diagramas de clases.


Conmutador
+SetNameServer(nameServer : String) : void
+SetNumPort(port : Integer) : void
+getNameServer() : String
+getNumPort() : Integer
+listen(String) : Boolean
ConmutadorServer
#filterDataFromFile() : []String
#chooserService(service : []String) : void
#sendResponse() : void
ConmutadorServerImpl
+received : boolean = false
<<abstract>>
Mail
-user : String
-password : String
-subject : String
-message : String
-mailTo : String"[]"
-server : String
-port : Integer
-response : ResponseSender
+setUser(user : String)
+getUser() : String
+setPassword(password : String)
+getPassword() : String
+setSubject(subject : String)
+getSubject() : String
+setMessage(message : String)
+getMessage() : String
+setMailTo(mailTo : String"[]")
+getMailTo() : String"[]"
+setServer(server : String)
+getServer() : String
+setPort(port : Integer)
+getPort() : Integer
#buildResponse() : void
#sendResponse() : ResponseSender
#saveLogFile() : void
ConmutadorTR
-path : String
-port : Integer
-nameServer : String
-received : boolean
<<constructor>>+ConmutadorTR(server : String, port : Integer, path : String)
+sendFile(path : String) : void
+receiveFile(path : String) : void
+listenAndFilter(message : String) : void
+setPath(path : String) : void
+getPath() : String
+sendResponse() : void
<<abstract>>
ResponseSender
-server : String
-port : String
-response : String
-path : String
+setServer(server : String) : void
+getServer() : String
+setPort(port : String) : void
+getPort() : String
+setResponse(response : String) : void
+getResponse() : String
+setPath(path : String) : void
+getPath() : String
formatResponse(service : String) : void
ResponseServerImpl
FileManager
+saveFile(path : String, data : String) : void
+deleteFile(path : String) : void
+appendToFile(path : String, append : String) : void
Logger
-log : String
-aplication : String
+setLog(log : String) : void
+getLog() : String
+setAplication(aplication : String) : void
+getAplication() : String
MailReceiver
+receiveRecentMails() : ArrayList
+choseMail(numMail;Integer, pathMails : String) : String
MailSender
+sendMail(subject : String, message : String, mailTo : String"[]", user : String, password : String) : void


Conmutador
+SetNameServer(nameServer : String) : void
+SetNumPort(port : Integer) : void
+getNameServer() : String
+getNumPort() : Integer
+listen(String) : Boolean
ConmutadorServer
#filterDataFromFile() : []String
#chooserService(service : []String) : void
#sendResponse() : void
ConmutadorServerImpl
+received : boolean = false
ConmutadorTR
-path : String
-port : Integer
-nameServer : String
-received : boolean
<<constructor>>+ConmutadorTR(server : String, port : Integer, path : String)
+sendFile(path : String) : void
+receiveFile(path : String) : void
+listenAndFilter(message : String) : void
+setPath(path : String) : void
+getPath() : String
+sendResponse() : void
<<abstract>>
ResponseSender
-server : String
-port : String
-response : String
-path : String
+setServer(server : String) : void
+getServer() : String
+setPort(port : String) : void
+getPort() : String
+setResponse(response : String) : void
+getResponse() : String
+setPath(path : String) : void
+getPath() : String
formatResponse(service : String) : void
ResponseServerImpl
Command
+execute() : void
+login() : void
FileManager
+saveFile(path : String, data : String) : void
+deleteFile(path : String) : void
+appendToFile(path : String, append : String) : void
Logger
-log : String
-aplication : String
+setLog(log : String) : void
+getLog() : String
+setAplication(aplication : String) : void
+getAplication() : String
SSHExecuter
-user : String
-password : String
-Command : String"[]"
+saveLogFile: void()
+closeConection() : void
+buildResponse() : String
+setCommand(command : String"[]") : void
+getCommand: String"[]"()
JavaExecuter
-user : String
-password : String
-command : String"[]"
+saveLogFile() : void
+closeConection() : void
+buildResponse() : String
+getCommand() : String"[]"
+setCommand(command : String"[]") : void
TelnetExecute
-user : String
-password : String
-command : String"[]"
+saveLogFile() : void
+closeConection: voif()
+getCommand() : String"[]"
+setCommand(command : String"[]") : void
+buildResponse() : String

3.1.3.2 Diagramas de secuencia.

En esta sección se muestra el flujo del sistema de transporte de información asociado con el proyecto. Para ello se describen diferentes escenarios asociados cada uno de ellos con un servicio que el sistema debería proporcionar.

La descripción del proceso que involucra cada escenario se hace mediante diagramas de secuencia definidos en el *Lenguaje Unificado de Modelado (UML)*.

La petición de servicios al servidor se hace mediante el *protocolo de comunicación* establecido para este sistema, por lo que es deseable leer dicho protocolo antes de proseguir con esta lectura.

Escenario: Recepción de correo electrónico.

La obtención de correos electrónicos se divide en dos etapas: la primera consiste en que el usuario envíe la petición de obtener correos, entonces el proceso regresa al usuario una lista de encabezados, después de este evento viene la segunda etapa, en la cual el usuario solicita el número de correos que se quiere obtener.

La petición de este servicio sería de la siguiente manera:

GMa

Email *valle.abraham@gmail.com*

Pass *micontraseña*

Respuesta del servidor*: recepción de los encabezados de los correos recibidos con un número asociado a cada encabezado.*

Envío del mensaje:

Nmail 2

Respuesta del servidor: *recepción del correo numero 2 a revisar*

Como este proceso está dividido en dos etapas, es conveniente separar la secuencia de procesos de cada etapa.

Etapa 1: petición inicial del servicio de recepción de correos electrónicos.

1. El usuario realiza la petición del servicio de recepción de correos electrónicos en base a la descripción mostrada anteriormente.

2. La clase *ConmutadorTR*, que reside en la TR (terminal remota) conectada al servidor por usb o tecnología Bluetooth, está en espera (en ciclo infinito) de mensajes SMS con el formato congruente con el protocolo de comunicación establecido. ConmutadorTR realiza lo anterior mediante su método *ConmutadorTR.ListenAndFilter(message).*

3. Se ejecuta *ConmutadorTR.saveFile(message, path*) el cual delega la responsabilidad de guardar el mensaje recibido a un archivo a *FileManager.saveFile (path, message).* Por comodidad llamaremos a este *archivo arch.in.*

4. Se ejecuta el método *ConmutadorTR.sendFile(path),* el cual permite enviar a *arch.in.*

5. Se ejecuta el método *ConmutadorTR.sendFile(path),* el cual permite enviar a *arch.in.*

6. La clase *ConmutadorServerImp*, que reside en nuestro servidor esta en espera (en ciclo infinito) de eventos generados por ConmutadorTR, el evento que espera es la recepción del archivo *arch.in*. Lo anterior se realiza mediante la ejecución del método *ConmutadorServerImp.Listen(path).*

7. Se ejecuta *ConmutadorServerImp. FilterDataFromFile(path*), el cual se encarga de extraer y filtrar los datos del archivo *arch.in*.

8. Se ejecuta *ConmutadorServerImp.chooseServise(String[]),* el cual se encarga de elegir el servicio solicitado.

9. Se ejecuta *MailReceiver.receiveRecentMails(eMail, pass),* el cual tiene como propósito obtener del servidor de correo electrónico los recientes y guardarlos en un archivo para su uso posterior.

10. Se ejecuta MailReceiver.buildResponse(), el cual tiene como propósito construir el archivo de respuesta de servicio, en este caso solo contendrá la notificación de ejecución de proceso, MailReceiver.buildResponse() delega esta responsabilidad a métodos de otras clases.

11. Cuando *MailReceiver.buildResponse()* es ejecutado, éste ejecuta a *ResponseSenderImp.setResponse(response)* el cual coloca la respuesta para su posterior formateo, mediante *ResponseSenderImp.formatResponse(tipoServicio).*

12. El usuario recibe una lista *de los encabezados de los correos recibidos con un número asociado a cada encabezado.*

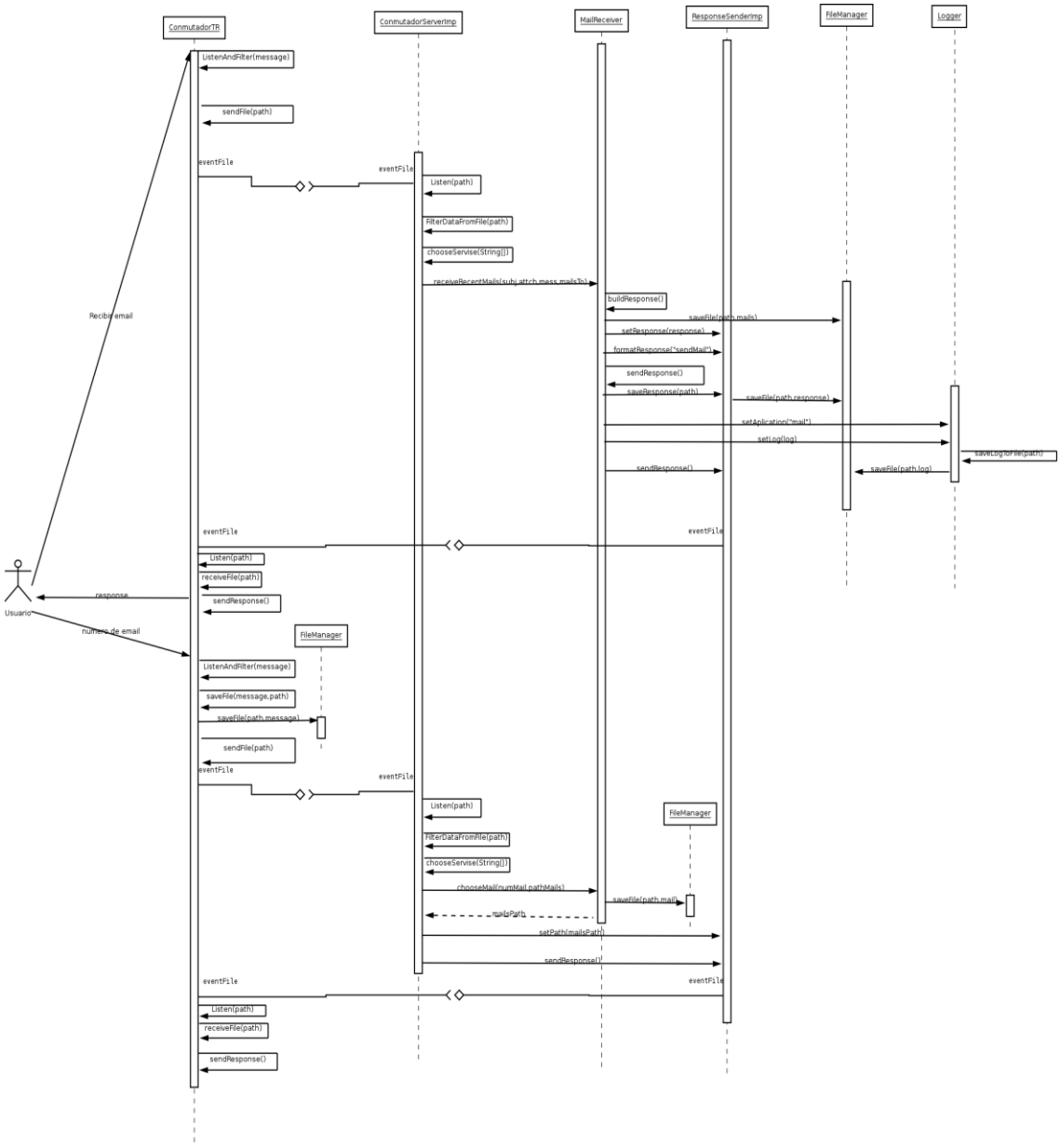
Etapa2: Solicitud de correos electrónicos que se quieren obtener.

Al igual que la etapa anterior, los pasos de secuencia de proceso coinciden en su mayoría con los pasos de secuencia de proceso del escenario de *envío de recepción de correo electrónico,* excepto en los pasos 9 y 12 que se describen a continuación.

9. Se ejecuta MailReceiver.chooseMail(numMail,pathMails), cuyo objetivo es elegir el correo electrónico correspondiente al número que se pasa como parámetro.

12. El usuario recibe el email que solicitó.

El diagrama de secuencia para este escenario es el siguiente.


ConmutadorTR
ConmutadorServerImp
MailReceiver
ResponseSenderImp
FileManager
Logger
ListenAndFilter(message)
sendFile(path)
eventFile
Listen(path)
FilterDataFromFile(path)
chooseServise(String[])
receiveRecentMails(subj,attch,mess,mailsTo)
buildResponse()
saveFile(path,mails)
setResponse(response)
formatResponse("sendMail")
sendResponse()
saveResponse(path)
saveFile(path,response)
setAplication("mail")
setLog(log)
saveLogToFile(path)
saveFile(path,log)
Recibir email
Usuario
response
numero de email
receiveFile(path)
saveFile(message,path)
saveFile(path,message)
chooseMail(numMail,pathMails)
mailsPath
setPath(mailsPath)

4. Diseño de una arquitectura que permita llevar a cabo los tipos básicos de transacciones contempladas en el proyecto

A continuación se presenta la implementación de las clases diseñadas para este proyecto, así como una breve descripción del funcionamiento y utilidad de cada una.

4.1 Implementación del análisis y diseño propuesto (proyecto PAPIIT) para envío de información desde un servidor hacia una terminal remota

El diseño propuesto por el Dr. Benjamín Macías Pimentel, está descrito en el anexo A y el anexo B. De este diseño se implementaron todas las clases necesarias para el envío de información desde una terminal remota hacia un servidor.

La implementación del sistema que permite transferir información entre el servidor y las terminales remotas a nivel básico, consta de tres "subproyectos" cada uno ubicado en una entidad del sistema.

Los "subproyectos" desarrollados son:

SRII-TRC llamado así porque reside en la terminal remota cliente (TRC) que es la que se encarga de interactuar con el usuario mediante una interfaz gráfica.

SRII-TRS llamado así porque reside en la terminal remota servidor (TRS) que se encarga de interactuar con SRII-TRC mediante la red telefónica y con SRII-SRV mediante Bluetooth.

SRII-SRV llamada así porque reside en el servidor (SRV) Se encarga de recibir las peticiones realizadas por SRII-TRS y de proveer el mecanismo necesario para responder a TRC usando como puente a TRS.

A continuación se presenta SRII-SRV.

4.1.1 SRII-SRV

El objetivo de este subproyecto es la recepción de la información enviada por SRII-TRS, su uso para atender el servicio pedido y enviar una respuesta del servicio solicitado a la terminal remota cliente.

A continuación se describirán todas las clases de éste subproyecto, agrupadas por su paquete respectivo.

4.1.1.1 Componente transporte

En este componente se encuentran los paquetes necesarios para llevar a cago el intercambio de información entre el servidor y una terminal remota.

4.1.1.1.1 Paquete mensaje

En este paquete se encuentran las clases relacionadas con el encapsulamiento de mensajes de entrada y salida del servidor.

4.1.1.1.1.1 MensajeEntrada

Encapsula la información de un mensaje de entrada a partir de un archivo de entrada.

Implementación:

```
/*
 * MensajeEntrada.java
 *
 */

package mensaje;

import java.io.*;
import java.util.*;
/**
 * @author Abraham Valle Alvarez
 */
public class MensajeEntrada {
    private long numeroTelefono;
    private int servicio;
    private int puertoMensaje;
    private String mensaje;
    private String pathMensajeEntrada;

    public MensajeEntrada(String pathMensajeEntrada) {
        this.pathMensajeEntrada=pathMensajeEntrada;
        try {

            BufferedReader in= new BufferedReader(new FileReader(pathMensajeEntrada));
            String line=in.readLine();
            line=line.substring(6);
            StringTokenizer st=new StringTokenizer(line,":");

            numeroTelefono=Long.parseLong(st.nextToken());
            puertoMensaje=Integer.parseInt(st.nextToken());
            StringTokenizer st2=new StringTokenizer(st.nextToken(),"|");
            servicio=Integer.parseInt(st2.nextToken());
            mensaje=st2.nextToken();
            in.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public long getnumeroTelefono(){
        return this.numeroTelefono;
    }

    public int getPuertoMensaje(){
        return this.puertoMensaje;

    }
```

```
    public int getServicio(){
        return this.servicio;

    }

    public String getMensaje(){
        return this.mensaje;
    }

    public String getPathFile(){
        return this.pathMensajeEntrada;
    }
    }
```

4.1.1.1.1.2 MensajeSalida

Encapsula la información de un mensaje de salida a partir de un archivo de salida.

Implementación:

```
/*
 * MensajeSalida.java
*/
package mensaje;

import java.io.*;
import java.util.*;
/**
 *
 * @author Abraham Valle Alvarez
 */

public class MensajeSalida {

    private long numeroTelefono;
    private int servicio;
    private int puertoMensaje;
    private String mensaje;
    private String pathMensajeSalida;

    public MensajeSalida(String pathMensajeSalida) {
        this.pathMensajeSalida=pathMensajeSalida;
        try {

            BufferedReader in= new BufferedReader(new FileReader(pathMensajeSalida));
            String line=in.readLine();
            line=line.substring(6);
            StringTokenizer st=new StringTokenizer(line,":");

            numeroTelefono=Long.parseLong(st.nextToken());
            puertoMensaje=Integer.parseInt(st.nextToken());
            mensaje =st.nextToken();

            in.close();

        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    public void setnumeroTelefono(long numeroTelefono){
        this.numeroTelefono=numeroTelefono;
    }

    public void setPuertoMensaje(int puertoMensaje){
        this.puertoMensaje=puertoMensaje;

    }
```

```
    public void setServicio(int servicio){
        this.servicio=servicio;

    }

    public void setMensaje(String mensaje){
        this.mensaje=mensaje;
    }

    public void setPathFile(String pathMensajeEntrada){
        this.pathMensajeSalida=pathMensajeSalida;
    }

    public long getnumeroTelefono(){
        return this.numeroTelefono;
    }

    public int getPuertoMensaje(){
        return this.puertoMensaje;

    }

    public int getServicio(){
        return this.servicio;

    }

    public String getMensaje(){
        return this.mensaje;
    }

    public String getPathFile(){
        return this.pathMensajeSalida;
    }
}
```

4.1.1.1.2 Paquete transporte.

Este paquete contiene las clases necesarias para la recepción y el envío de información desde SRII-TRS y hacia TRS.

Cabe resaltar que para el intercambio de la información entre TRS y SRV se hace uso de un servidor Obex provisto por el stack de Bluetooth para Linux llamado BlueZ; ademas de un paquete de KDE (K Desktop Enviroment) llamado KDEBluetooth que también hace uso de BlueZ.

En el apéndice se describe la configuración de SRV para el uso de BlueZ así como las bibliotecas usadas por este componente.

4.1.1.1.2.1 EnviadorSMS.

Clase que tiene por objetivo el envío de comandos AT (lenguaje de un terminal modem) hacia TRS mediante Bluetooth para que ésta a su vez envíe el mensaje SMS indicado por EnviadorSMS.

Para que esta clase pueda llevar a cabo su funcionalidad, es necesario el uso de las bibliotecas: rxtx y SMSLib; las primera para el envío de información vía Bluetooth hacia la terminal remota y la segunda para la codificación del mensaje como un comando AT.

Implementación:

```
/**Clase que tiene la funcionalidad de enviar mensajes sms a traves de el bluetooth
 *con comandos AT.
*@author Abraham Valle Alvarez
 */
package transporte;

import org.smslib.*;

class EnviadorSMS implements Runnable{

    private CService servicio;
    private String  mensaje;
    private String numero;

    /***Constructor de clase
     *@param puertoSerie - indica a que puerto debemos conectarnos ejemplo /dev/rfcomm2
     *@param velodidadModem - indica con la velocidad con la que trabaja el modem, en
nuestro caso es un celular
     que funje como modem - un valor de ejemplo es el 9600
     *@param centralSMS - La central SMS, en este caso es la de telcel, dada por el
siguiente codigo +5294100001410 (el + es necesario)
     *@param pim - es el numero pin con el cual se sincroniza el proceso y el dispositivo
bluetooth ejemplo 1234
     */
    public EnviadorSMS(String puertoSerie,int velocidadModem,
                       String centralSMS,String pin){
       servicio =new CService(puertoSerie, velocidadModem, "", "");
       servicio.setSimPin(pin);
       servicio.setSmscNumber(centralSMS);
    }

    /**Metodo que abre la conexion del servicio y envia el mensaje dado
     *@param message - mensaje que sera enviado
     *@param numero - numero de telefono al cual queremos mandar el mensaje ejemplo
+525536684795  (el + es necesario)
     */
    public synchronized void  connectAndSend(String mensaje,String numero){
       this.mensaje=mensaje;
       this.numero=numero;
       Thread t = new Thread(this);
       t.start();
    }


    /**Es la implementación del método run el cual es que ejecutan los threads*/
    public void run(){
       try {

           servicio.connect();

           COutgoingMessage msg =
              new COutgoingMessage(this.numero,this.mensaje);
           //codificación del mensaje
           msg.setMessageEncoding(CMessage.MessageEncoding.Enc7Bit);
           //pedimos status de envio
           msg.setStatusReport(true);
           msg.setValidityPeriod(8);

           //se envía el mensaje
           servicio.sendMessage(msg);

           //nos desconectamos del bluetooth
           servicio.disconnect();
       } catch (Exception e) {
           e.printStackTrace();
       } finally {
           if(servicio!=null&&servicio.getConnected())
              servicio.disconnect();
        }
```

```
    }
}
```

4.1.1.1.2.2 Transporte.

Clase que integra la funcionalidad de EnviadorSMS descrita arriba, DirFilterWatcher y DirWatcher descritas en el componente control.

Por ser una clase integradora, la clase Trasporte es la encargada del intercambio de información entre TRS y SRV. Esta clase detecta cuando el servidor Obex ha recibido un archivo de SRII-TRS, entonces toma el archivo y se lo pasa al componente de control; además detecta el envío de archivos del componente de control y los envía a TRS.

Esta clase utiliza una llamada a un programa por lotes llamado mataproceso.sh, que se utiliza para cerrar la primera ventana generada por KDEBluetooth. Dicho programa se puede ver en el apéndice.

Los métodos especificados en ésta clase son:

- recibeEntrada(MensajeEntrada msgE): permanece a la espera de mensajes enviados por SRII-TRS, cuando éste método detecta un envío de un mensaje, éste es enviado al componente de control.

- enviaEntrada(MensajeEntrada mensajeEntrada): envía el mensaje al componente de control.

- recibeSalida(MensajeSalida msjS):es invocado por el componente de control para que invoque al método de enviaSalida.

- enviaSalida(MensajeSalida msjS): envía el mensaje de respuesta al TRC.

Implementación:

```
/*
 * Transporte.java
 *
 *
 */

package transporte;

import utilerias.*;
import control.Control;
import mensaje.MensajeEntrada;
import mensaje.MensajeSalida;
import java.io.*;
import java.util.*;



/**
 *
 * @author Abraham Valle Alvarez
 */
```

```
public class Transporte {

    /**Dispositivo serial conectado al bluetooth*/
    private final String device="/dev/rfcomm0";
    /**velocidad del modem*/
    private final int velocidadModem=9600;
    /**centro de mensajería de sms*/
    private final String smsCenter="+5294100001410";
    /**pin para coordinarse con el celular*/
    private final String pin="1234";
    /**código internacional del país*/
    private final String codigoInternacional="+52";

    /** Creates a new instance of Transporte */
    public Transporte() {
    }
    /*
     *detecta el envió de un mensaje de entrada por parte de la clase Receptor;
     *Conceptualmente, esta función trabaja como un demonio en un ciclo
     * infinito
     * @param msgE - El objeto que envuelve el mensaje de entrada
     */
    public  void recibeEntrada(MensajeEntrada msgE){

        Timer timer = new Timer();
        TimerTask task = new DirWatcher("/tmp", "sms.txt" ) {
            protected void onChange( File file, String action ) {
                if(action.equals("add")){
                    System.out.println( "File "+ file.getAbsolutePath()+" action: " + action
);
                    MensajeEntrada me=new MensajeEntrada(file.getAbsolutePath());
                    //con esto cerramos la ventana de kdeobexserver
                    mataProceso("kbtobexsrv");
                    enviaEntrada(me);


                }
            }
        };
        timer.schedule( task , new Date(), 5000 );


    }

      /*
      * envía el mensaje recibido al componente de control para su proceso
      * esta función conceptualmente trabaja como un evento que se dispara cada
      * vez que recibeEntrada se activa.
      * @param mensajeEntrada El objeto que envuelve el mensaje de entrada
      */
    public void enviaEntrada(MensajeEntrada mensajeEntrada){

        Control ctl= new Control();
        ctl.recibeEntrada(mensajeEntrada);

    }


     /*
      * detecta el envió de un mensaje de salida por parte de Control
      * Conceptualmente, esta función trabaja como un demonio en un ciclo infinito
      * @param msgS EL objeto que envuelve el mensaje de salida
      */

    public  void recibeSalida(MensajeSalida msjS){

        enviaSalida(msjS);
    }


     /*
```

```
      *envía el mensaje recibido al Receptor; invoca al objeto Receptor
      * @param msgS EL objeto que envuelve el mensaje de salida
      *
      */
    public void enviaSalida(MensajeSalida msjS){
        System.out.println("device "+device+ "\n"+
                "smsCenter "+smsCenter+"\n"+
                "telefono "+codigoInternacional+msjS.getnumeroTelefono()+"\n"+
                "mensaje "+msjS.getMensaje()
                );
        EnviadorSMS sms=new EnviadorSMS(device,velocidadModem,smsCenter,pin);
        sms.connectAndSend(msjS.getMensaje(),codigoInternacional+msjS.getnumeroTelefono());
         File fileMsjS=new File(msjS.getPathFile());
                            if(fileMsjS.renameTo(new
File("/tmp/logSend/"+System.currentTimeMillis()+fileMsjS.getName()))
                               System.out.println("El mensaje fue procesado");
                            else
                                System.out.println("El mensaje no fue procesado");
    }



      /**Elimina el proceso más viejo que coincida con el proceso especificado
     * *****NOTA:Este metodo solo funciona par sistemas UNIX******
     *@param proceso - nombre de proceso ejemplo: java
     */
    private void mataProceso(String proceso){
        String bashCmmd="mataproceso.sh "+proceso;
        try{
            Runtime.getRuntime().exec(bashCmmd);
        }catch(IOException ioe){
            ioe.printStackTrace();
        }
    }

}
```

4.1.1.2 Componente control

Este componente contiene los paquetes necesarios para interactuar con el componente anterior, interpretar sus mensajes y envíale mensajes de salida al mismo

4.1.1.2.1 Paquete control

4.1.1.2.1.1 Control

El objetivo de esta clase es detectar los mensajes de entrada enviados por el componente de trasporte. A partir de la información contenida en él mensaje, ejecutar un servicio que genera un archivo de respuesta y por último, enviar al componente de transporte la respuesta generada.

Los métodos que esta clase contiene son:

- recibeEntrada(MensajeEntrada mensajeEntrada): detecta el envío de un mensaje por parte de la clase Transporte.
- procesaMensajeEntrada(MensajeEntrada mensajeEntrada): crea el servicio establecido por el mensaje de entrada y lo ejecuta.

- creaServicio(int tipoServicio,final MensajeEntrada mensajeEntrada,final String nameOut): crea un objeto de tipo Manipulador del cual heredan todos los servicios a partir del mensaje de entrada. El nombre de archivo de respuesta del servicio es especificado en nameOut.

- recibeSalida(): permanece en espera de mensajes de salida provenientes de objetos de tipo Manipulador, es decir permanece en espera de la respuesta de los servicios.

- procesaMensajeSalida(MensajeSalida msjS): envía el mensaje de salida al componente de transporte.

Implementación:

```
/*
 * Control.java
 *
 */

package control;
import utilerias.*;
import java.io.*;
import java.util.*;
import manipulador.Manipulador;
import mensaje.MensajeEntrada;
import mensaje.MensajeSalida;
import transporte.Transporte;
/**
 *
 * @author Abraham Valle Alvarez
*/

public class Control {

    /** Creates a new instance of Control */
    public Control() {
    }

    /*
     *
     * Detecta el envió de un mensaje por parte de la clase Transporte
     *  Conceptualmente, esta función trabaja como un demonio en un ciclo
     * infinito
     * @param mensajeEntrada - Objeto que envuelve el mensaje de entrada
     */

    public void recibeEntrada(MensajeEntrada mensajeEntrada){

        procesaMensajeEntrada(mensajeEntrada);
    }

    /*

     * Organiza el proceso de un mensaje de entrada:
     * 1. Debe de almacenar los detalles generales relativos al mensaje de entrada
     * (hora de llegada, identificador del TR, longitud, identificador interno, etc).
     * 2. Ver que se almacene.
     * 3. Establecer el tipo de servicio solicitado en el mensaje.
     * 4. Ponerse en contacto con el componente relevante para brindar el servicio.
     *  Para este diseño, se hará diferencia entre servicios
     *  numerándolos consecutivamente, y la información para el servicio
     *  estará separada por |
     * @param mensajeEntrada - Objeto que envuelve el mensaje de entrada
```

```
     */


    public void procesaMensajeEntrada(MensajeEntrada mensajeEntrada){

        String nameOut="/tmp/send/send"+System.currentTimeMillis()+".txt";

        Manipulador
man=creaServicio(mensajeEntrada.getServicio(),mensajeEntrada,nameOut);//implementacion de un
manipulador
        man.ejecutarServicio();

    }

    /*
     * Método auxiliar que crea un manipulador en base al servicio correspondiente
     * @param tipoServicio - número que identifica el tipo de servicio
     * @param MensajeEntrada - Objeto que envuelve el mensaje de entrada.
     * @param nameOut - nombre del archivo de salida para este servicio
     * @return un Manipulador especializado para un servicio particular
     */
    private Manipulador creaServicio(int tipoServicio,final MensajeEntrada
mensajeEntrada,final String nameOut){
        //aquí se hace el switch con el tipo del servicio

        switch(tipoServicio){
            case 1://servicio de envio de correo
                return null;

            case 2://servicio de búsqueda
                //return null;///esto es una situación temporal, el siguiente código, es una
versión de prueba en el reporte.
                Manipulador man2=new Manipulador(){
                    public boolean ejecutarServicio(){
                        imprimeSalida(null);
                        return true;
                    }
                    public String regresaLog(){
                        return null;
                    }

                    public void imprimeSalida(String pathFile){
                        try {
                            BufferedWriter out = new BufferedWriter(new
FileWriter(nameOut));

                            out.write("sms://");
                            out.write(mensajeEntrada.getnumeroTelefono()+":");
                            out.write(mensajeEntrada.getPuertoMensaje()+":");
                            out.write("|Este es el numero de servicio
"+mensajeEntrada.getServicio()+" ");
                            out.write(" este servicio no ha sido identificado");
                            //in.close();
                            out.close();
                            File fileMsjE=new File(mensajeEntrada.getPathFile());
                             if(fileMsjE.renameTo(new
File("/tmp/logReceive/"+System.currentTimeMillis()+fileMsjE.getName())))
                                 System.out.println("El mensaje fue procesado");
                             else
                                 System.out.println("El mensaje no fue procesado");
                        } catch (IOException e) {

                        }

                    }
                };

                return man2;
            case 3://obtener correo
                return null;
```

```
            case 4://ejecutar comando
                return null;

            default://este caso nunca debe de pasar, a excepción de que no se tenga
implementado ningún servicio
                Manipulador man=new Manipulador(){
                    public bolean ejecutarServicio(){
                        imprimeSalida(null);
                        return true;
                    }
                    public String regresaLog(){
                        return null;
                    }

                    public void imprimeSalida(String pathFile){
                        try {
                            BufferedWriter out = new BufferedWriter(new
FileWriter(nameOut));

                            out.write("sms://");
                            out.write(mensajeEntrada.getnumeroTelefono()+":");
                            out.write(mensajeEntrada.getPuertoMensaje()+":");
                            out.write("|Este es el numero de servicio
"+mensajeEntrada.getServicio()+" ");
                            out.write(" este servicio no ha sido identificado");
                            //in.close();
                            out.close();
                        } catch (IOException e) {

                        }

                    }
                };
                return man;

        }

    }

    /*
     *

     * Detecta que algún componente de manipulación a terminado de elaborar
     * un mensaje de salida, por lo que este está listo para ser enviado
     * conceptualmente, esta función trabaja como un demonio en un ciclo
     * infinito.
     * @param mensajeSalida objeto que envuelve el mensaje de salida
     */

     public void recibeSalida(){
        //if(msjS!=null)
          // procesaMensajeSalida(msjS);
    Timer timer = new Timer();
        TimerTask task = new DirWatcher("/tmp/send", "send.*txt" ) {
            protected void onChange( File file, String action ) {
                if(action.equals("add")){
                    //System.out.println( "File "+ file.getName() +" action: " + action );
                    MensajeSalida ms=new MensajeSalida(file.getAbsolutePath());
                    procesaMensajeSalida(ms);

                }
            }
        };
        timer.schedule( task , new Date(), 5000 );

    }

    /*
     *
     * Organiza el proceso de producción de un mensaje de salida:
     * 1. Almacena los detalles generales del mensaje de salida (componente que
```

```
     * lo produjo, hora, identificador del mensaje).
     * 2. Almacena el mensaje de salida (esto es, trasfiere una copia del mensaje
     * que llego del Manipulador al Almacenamiento).
     * 3. Establece los detalles del TR receptor.
     * 4. Formatea el mensaje en su forma final y lo envía al Transporte.
     * @param mensajeSalida objeto que envuelve el mensaje de salida
     */
    public void procesaMensajeSalida(MensajeSalida msjS){
        Transporte t=new Transporte();
        t.recibeSalida(msjS);

    }

}
```

4.1.1.2.2 Paquete utilerías

4.1.1.2.2.1 DirFilterWatcher

Clase cuyo objetivo es filtrar archivos al momento de hacer una lista de los archivos contenidos en un directorio.

Implementación:

```
/**
 *
 * @author Abraham Valle Alvarez
**/


package utilerias;


import java.io.*;
import java.util.*;



/**Clase interna auxiliar que implementa el filtrador de archivos
 *Permite filtrar el tipo de archivo que se quiere al momento de listar archivos.
 */
public class DirFilterWatcher implements FileFilter {
    private String filter;

    public DirFilterWatcher() {
        this.filter = "";
    }

    public DirFilterWatcher(String filter) {
        this.filter = filter;
    }

    public boolean accept(File file) {
        if ("".equals(filter)) {
            return true;
        }
        return (file.getName().toLowerCase().matches(filter.toLowerCase()));
    }
}
```

4.1.1.2.2.2 DirWatcher

Clase abstracta que tiene como finalidad detectar eventos desde un directorio sobre archivos que estén a su alcance.

Por ser una clase abstracta permite definir la acción a realizar al detectar algún cambio sobre el archivo que dispara el evento.

Implementación:

```
/**
 *
 * @author Abraham Valle Alvarez
**/

package utilerias;


import java.io.*;
import java.util.*;

/**Clase que tiene dos funcionalidades:
 *1.- Permite detectar eventos desde un directorio sobre archivos que estén en a su alcance.
 *2.-Al ser una clase abstracta permite declarar la acción a realizar al detectar algún
cambio sobre un archivo
 */
public abstract class DirWatcher extends TimerTask {
    /**Ruta del directorio de inicio de búsqueda*/
    private String path;
    /**Arreglo de archivos que coinciden con el filtro dado*/
    private File filesArray [];
    /**Hash que nos permite relacionar un archivo y su ultima fecha de modificación*/
    private HashMap<File,Long> dir = new HashMap<File,Long>();

    /**Filtro de archivos*/
    private DirFilterWatcher dfw;

    /**Constructor que crea el objeto y no aplica ningún filtro a los archivos listados
     *es decir lista todos los archivos encontrados
     */
    public DirWatcher(String path) {
        this(path, "");
    }

    /**Este constructor permite especificar el directorio de inicio de búsqueda
     * así como el filtro con el que se filtrara el listado de archivos */
    public DirWatcher(String path, String filter) {
        this.path = path;
        dfw = new DirFilterWatcher(filter);
    }


    /**Método que examina recursivamente y regresa todos los archivos
     *que existen desde el directorio que se pasa como inicio y que coinciden con el
*filtro
     *especificado en el constructor de la clase
     *@param aStartingDir- directorio desde el cual buscar archivos
     *@return  ArrayList - lista con todos losn archivos encontrados
     */
    public ArrayList <File>getFileListing(File aStartingDir)  {
        ArrayList <File>result = new ArrayList<File>();
        File[] tmp=aStartingDir.listFiles();
        File[] filesAndDirs = tmp==null?new File[]{}:tmp;
        java.util.List<File> filesDirs = Arrays.asList(filesAndDirs);

        if(aStartingDir.isDirectory()&&aStartingDir.canRead()){
             File[] files=aStartingDir.listFiles(dfw);
            result.addAll(Arrays.asList(files==null?new File[]{}:files));
            Collections.sort(result);
        }
```

```
        for(File file:filesDirs){
                result.addAll(getFileListing(file));
        }
        //Collections.sort(result);
        return result;
    }

    /**Este método es ejecutado por el timer para ser calendarizado*/
    public  final void run() {
        HashSet<File> checkedFiles = new HashSet<File>();
        ArrayList<File> af=getFileListing(new File(path));
        filesArray= af.toArray(new File[af.size()]);

        System.out.println("runn "+af);
        // Escaneamos los archivos  y verificamos si hay modificaciones o creacion de nuevos
archivos
        for(int i = 0; i < filesArray.length; i++) {
            Long current = (Long)dir.get(filesArray[i]);
            checkedFiles.add(filesArray[i]);

            if (current == null) {
                // new file
                dir.put(filesArray[i], new Long(filesArray[i].lastModified()));
                onChange(filesArray[i], "add");
            } else if (current.longValue() != filesArray[i].lastModified()){
                // modified file
                dir.put(filesArray[i], new Long(filesArray[i].lastModified()));
                onChange(filesArray[i], "modify");
            }
        }

        TreeSet <File>ref = new TreeSet<File>(dir.keySet());

        ref.removeAll(checkedFiles);
        Iterator it = ref.iterator();
        while (it.hasNext()) {
            File deletedFile = (File)it.next();
            dir.remove(deletedFile);
            onChange(deletedFile, "delete");
        }
    }

    /**Este método se ejecuta cuando se detecta un cambio en un directorio, debe
     *ser implementado por una subclase de DirWatcher para realizar ciertas operaciones en
caso
     *de ocurrir algún cambio.
     *@param file - archivo en el que se detecto cambio
     *@action - accion realizada en el directorio, es decir add, modify, delete
     */
    protected abstract void onChange( File file, String action );

}
```

4.1.1.2.2.3 StackTraceUtil

Esta clase regresa el conjunto de excepciones de Java en un formato de cadena.

Implementación:

```
package utilerias;

import java.io.*;
import java.util.*;

/**Clase de utilerías que nos permite regresar el stack trace de Throwable  una cadena y
darle un formato a esta
*/
```

```
public final class StackTraceUtil {

    /**Regresa el stackTrace de aThrowable en una cadena
     *@param aThrowable objeto del cual se tomara el stack trace
     *@return la cadena que contiene el el stacktrace
     */
  public static String getStackTrace(Throwable aThrowable) {
    final Writer result = new StringWriter();
    final PrintWriter printWriter = new PrintWriter(result);
    aThrowable.printStackTrace(printWriter);
    return result.toString();
  }

  /**
  * Define un formato para la cadena de salida
  */
  public static String getCustomStackTrace(Throwable aThrowable) {
    final StringBuilder result = new StringBuilder( "SRII : " );
    result.append(aThrowable.toString());
    final String NEW_LINE = "\n";
    result.append(NEW_LINE);

    final List<StackTraceElement> traceElements = Arrays.asList(
       aThrowable.getStackTrace()
    );
    for (StackTraceElement element : traceElements ){
      result.append( element );
      result.append( NEW_LINE );
    }
    return result.toString();
  }
}
```

4.2 Implementación de la arquitectura diseñada para proveer servicios

Implementación de la arquitectura diseñada por el Dr. Benjamín Macías Pimentel para proveer servicios, se encuentra en el subproyecto SRII-SRV.

El objetivo de este subproyecto es la recepción de la información enviada por SRII-TRS, manipulación de dicha información para atender el servicio solkicitado y el envío de respuesta a la terminal remota cliente.

En esta sección solo se describe el núcleo de la arquitectura y la implementación del servicio de búsqueda de información en el Internet, particularmente de la wikipedia(http://www.wikipedia.org/).

4.2.1 Componente de manipulación

Este componente es el encargado de proveer los servicios solicitados, por ejemplo si el servicio que se pide es la recepción de un correo electrónico, este componente debe contener una clase específica para este servicio, si la petición es ejecutar un comando en un servidor, el componente debe contener

una clase que efectué dicho servicio y así para cada tipo de servicio que se pretenda brindar.

4.2.1.1 Paquete manipulador

Este paquete va a contener todas y cada una de las implementaciones de servicios que se propone brindar.

4.2.1.1.1 Manipulador

Es una interfaz que debe implementarse por cualquier clase que quiera brindar un servicio.

Los métodos que esta interfaz tiene son:

- ejecutarServicio(): este método permite la ejecución de un servicio, el cual debe generar un Log del estatus de la operación y debe generar un archivo de respuesta al servicio.
- imprimeSalida(String pathFile): genera el archivo de respuesta del servicio.
- regresaLog(): escribe en la bitácora el estado de la operación

Implementación:

```
package manipulador;

/**
 *
 * @author Abraham Valle Alvarez
* @author Huilver Nolasco Aguilar
 */
public interface Manipulador {

    /*
     *
     *  Este método es invocado para ejecutar el servicio
     *  @return true si fue ejecutado correctamente, false en otro caso
     */
    public boolean ejecutarServicio();

    /*
     *
     *Regresa el log de salida del servicio
     *@return El log de salida del servicio
     */

    public String getLog();

    /*
     *Este método implementa la escritura de un archivo con ubicacion prefedefinida
     *@param la ruta absoluta del archivo de salida
     *
     */
    public void imprimeSalida(String pathFile);

}
```

4.2.1.2 Paquete manipulador.mail

4.2.1.2.1 EnviadorCorreo

Esta clase implementa a la interfaz Manipulador como un servicio para el envío de correos electrónicos específicamente al servidor de Gmail.

Implementación:

```
package manipulador;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.util.Vector;
import mensaje.MensajeEntrada;
import manipulador.mail.SmtpSsl;
import utilerias.StackTraceUtil;

public class EnviadorCorreo implements Manipulador{
    /**Datos del servidor a donde se debe conectar para enviar correo
     *en este caso el enviador solo funciona para Gmail*/
    private final String SERVER="smtp.gmail.com";
    private final String PORT="465";
    /**mensaje de entrada del cual obtenemos la información para enviar el correo*/
    private MensajeEntrada mensajeEntrada;
    /**contiene el log de ejecucion de proceso*/
    private StringBuffer Log=new StringBuffer();
    protected StackTraceUtil stacktrace= new StackTraceUtil();
    public EnviadorCorreo(MensajeEntrada ms) {
        mensajeEntrada=ms;
    }

    /**Implementación del método que ejecuta el servicio de envió de email
     */
    public boolean ejecutarServicio() {
        SmtpSsl mailer=new SmtpSsl(SERVER,PORT,true,true);
        Vector<String> rec=new Vector<String>();
        String [] datos=mensajeEntrada.getMensaje().split("\\|");
        String remitente=datos[0];
        String password=datos[1];
        String destinatario=datos[2];
        String mensaje=datos[3];
        String subject="mensaje de"+mensajeEntrada.getnumeroTelefono();
        rec.add(destinatario);
        boolean answer=mailer.sendMessage(remitente,password,rec,subject,mensaje);
        Log.append(mailer.getLog());
        return answer;
    }

    /**Este método es usado en combinación con ejecutar servicio, ya que este método escribe
a un archivo el fin de
     *la operación de ejecutaServicio, el archivo que este método genera es tomado por
trasporte para enviar la respuesta
     *al usuario.
     *@param pathFile - la ruta en donde se guardará el archivo de salida
     */
    public void imprimeSalida(String pathFile){
        try{
            BufferedWriter out = new BufferedWriter(new FileWriter(pathFile));

            if(getLog()!=null){
                String dest=mensajeEntrada.getMensaje().split("\\|")[2];
                out.write("sms://");
                out.write(mensajeEntrada.getnumeroTelefono()+":");
                out.write(mensajeEntrada.getPuertoMensaje()+":");

                if(getLog().length()==0)
```

```
                    out.write("Email enviado a "+dest+" correctamente");
                else
                    out.write("Fallo el envio de email  a "+dest);

                out.close();
            }
        }catch(Exception ioe){
            Log.append(stacktrace.getCustomStackTrace(ioe));
            System.out.println(Log);
        }

    }

    public String getLog() {
        return Log.toString();
    }

}
```

4.2.1.2. SmtpSsl

Clase que implementa un cliente para el envío de información a través del protocolo SMTP mediante un canal cifrado SSL (*Security Socket Layer*).

Implementación:

```
/**Clase que implementa un cliente SMTP con soporte SSL
 * @author Abraham Valle Alvarez
 */
package manipulador.mail;

import java.io.IOException;
import java.util.Properties;
import java.util.Vector;

import javax.mail.Authenticator;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.SendFailedException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import utilerias.StackTraceUtil;

public class SmtpSsl {
    private final String SSL_FACTORY = "javax.net.ssl.SSLSocketFactory";
    private String HOST;
    private String PORT;
    private boolean auth;
    private boolean usetls;
    private Session     session;
    private StringBuffer Log=new StringBuffer();
    private StackTraceUtil stackTrace=new StackTraceUtil();

    public SmtpSsl(String host, String port, boolean usetls,boolean auth) {
        HOST = host;
        PORT = port;
        this.auth = auth;
        this.usetls=auth;
    }

    public String getLog(){
        return Log.toString();
    }
```

```
    public  Session getSession(String user,String pwd) throws MessagingException{
        Session     ses;
        Properties props = new Properties();
        props.put("mail.from",user);
        props.put("mail.smtp.host", HOST);
        props.put("mail.smtp.port",PORT);
        props.put("mail.transport.protocol", "smtp");

        if(usetls){
            props.put("mail.smtp.socketFactory.class",    SSL_FACTORY);
            props.put("mail.smtp.socketFactory.port",     PORT);
            props.put("mail.smtp.socketFactory.fallback", "false");
            props.put("mail.smtp.starttls.enable",        "true");
            props.put("mail.smtp.ssl",                    "true");
        }
        if (auth) {
            Authenticator   authenticator = new Autenticador(user, pwd);
            props.put("mail.user",      user);
            props.put("mail.smtp.auth", "true");
            ses = Session.getInstance(props, authenticator);
        } else
            ses = Session.getInstance(props);
        return ses;
    }

    public boolean sendMessage(String user,String pwd,Vector<String> recipients, String
subject, String messageText) {
        boolean respuesta=false;
        if (subject == null)
            subject = "(no subject)";

        if (messageText == null)
            messageText = "";

        try {
            session = getSession(user,pwd);

            MimeMessage   message = new MimeMessage(session);

            for (String addressee : recipients)
                message.addRecipient(Message.RecipientType.TO, new
InternetAddress(addressee));

            message.setFrom(new InternetAddress(user));
            message.setText(messageText);
            message.setSubject(subject);
            Transport.send(message);
            respuesta=true;
        } catch (SendFailedException e) {
            Log.append(stackTrace.getCustomStackTrace(e));
            System.out.println(Log);
        } catch (AddressException e) {
            Log.append(stackTrace.getCustomStackTrace(e));
            System.out.println(Log);
        } catch (MessagingException e) {
            Log.append(stackTrace.getCustomStackTrace(e));
            System.out.println(Log);
        }
        return respuesta;
    }


    private class Autenticador extends Authenticator {
        private String    username;
        private String    password;

        public Autenticador(String username, String password) {
            this.username = username;
            this.password = password;
        }
```

```
        public PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(username, password);
        }
    }
}
```

5 Referencias

[1]	Diseño e implementación de un sistema para envío de información de una terminal remota a un servidor. Huilver Nolasco Aguilar. Agosto 2008.
[2]	Sistemas Remotos de Interacción con Internet. Benjamín Macías Pimentel. PAPIIT 2006
[3]	Unified Modeling Language: Superstructure version 2.1.1. OMG. Febrero 2007

6 BIBLIOGRAFIA

- Bluetooth profiles: the definitive guide
  Autor(es): Gratton, Dean A.;
  Editor(es): Prentice Hall (Upper Saddle River, New Jersey)
  Año Pub.: 2003
- J2ME Bluetooth programming
  Autor(es): André N. Klingsheim
  Editor (es): Department of Informatics University of Bergen
  Año Pub.: 2004

- http://www.modems.com/general/extendat.html
- http://en.wikipedia.org/wiki/Short_message_service
- http://gatling.ikk.sztaki.hu/~kissg/gsm/at+c.html
- http://www.iec.org/online/tutorials/gsm/
- http://www.omg.org/gettingstarted/what_is_uml.htm

Conclusión

Para llevar a cabo la creación del sistema, que cumpliera con el envío de información desde un dispositivo móvil hacia un servidor, se decidió utilizar SMS para la comunicación entre las terminales remotas y Bluetooth para la comunicación entre una terminal remota y un servidor, esto dado su bajo costo.

El uso de comunicaciones asíncronas nos da la facilidad de realizar una petición a un servidor y dejar que este haga la tarea, mientras nuestra terminal remota puede estar haciendo otras cosas, incluso apagada. El servidor regresara una respuesta a la terminal remota con el estado final de la tarea, terminando así el proceso de una petición.

Se lograron dos diseños para realizar el sistema capaz de hacer lo antes mencionado. Cabe decir que el diseño realizado por el profesor titular de la investigación fue el que se tomó como parámetro principal en el desarrollo, pero éste se complementó con el diseño realizado por los becarios integrados en el proyecto.

La utilidad de este sistema se basa en la importancia de las comunicaciones asíncronas desde una terminal remota con Internet sin que dicho dispositivo tenga que encontrarse conectada a la misma ya que esto consume recursos de hardware y tiene un alto costo.

Las pruebas que se plantearon exitosamente fueron el envío de correos electrónicos así como búsquedas en internet.

En mi participación en el proyecto se cumplieron con todas las metas planteadas al principio del año: revisar e implementar la arquitectura sugerida. Esto se llevó a cabo y la implementación fue probada y verificada. Además llevé a cabo una especificación, análisis y diseños alternativos. Este segundo trabajo también cumplió con los objetivos planteados de antemano.

Apéndice A

CONFIGURACIÓN DE BLUEZ

Para que el stack de Bluetooth BlueZ funcione correctamente se deben de configurar un par de archivos, los cuales son rfcomm.conf y hcid.conf y van en el directorio /etc/bluetooth/ o /etc/.

El archivo rfcomm.conf es para la configuración del dispositivo móvil al cual nos vamos a conectar.

El archivo hcid.conf es para configurar el dispositivo Bluetooth del servidor.

Para entender la configuración de los archivos no hay mejor manera que dar un ejemplo.

rfcomm.conf

```
rfcomm0 {
 #indica que al inicio del servicio se liga el dispositivo
 bind yes;

 # dirección Bluetooth del dispositivo es decir el teléfono
celular
 device 00:17:E2:27:CA:D2;

 # Canal de conexión del teléfono para el servicio
 # "Dial-up networking Gateway"
 channel   1;

 # La descripción de la conexión
 comment "Example Bluetooth device";
}
```

hcid.conf

```
options {
 # Inicializamos automáticamente nuevos dispositivos
 autoinit yes;

 # Manejador de modalidad de la seguridad
 #   none - deshabilitado
 #   auto - Uso de PIN para conexiones entrantes
 #   user - Preguntar siempre PIN al usuario
 security auto;

 # Modo de apareamiento
 #   none  - Deshabilitado
 #     multi  -  Permitir  apareamiento  con  dispositivos  ya
apareados
 #   once  - Aparear una vez y denegar intentos sucesivos
 pairing multi;
```

```
     # PIN helper
     #  programa  que  maneja  el  PIN  para  aparear  con  el
dispositivo
     pin_helper /opt/kde3/sbin/kbluepin;

    }

    # Default settings for HCI devices
    device {
     # Nombre del dispositivo local
     #   %d - identificador del dispositivo
     #   %h - nombre del host
     name "%h (%d)";


     # Clase del dispositivo
     class 0xd30108;


     # Habilitar el escaneo de servicios
     iscan enable;
     pscan enable;

     # Modo de enlace
     #   none   - ninguna
     #   accept - siempre aceptar conexiones
     #    master - se convierte en maestro sobre conexiones de
entrada, se          #   niega las conexiones de salida
     lm accept;

     # política de enlace
     lp rswitch,hold,sniff,park;

    }
```

Para escanear los dispositivos bluetooth que están alrededor del servidor se ejecuta lo siguiente.

"hcitool scan"

La salida debe ser algo parecido a lo siguiente.

```
Scanning ...
        00:19:2C:DB:57:3D       Abraham
```

Para buscar los servicios de un dispositivo se ejecuta lo siguiente

"sdptool browse 00:19:2C:DB:57:3D"

La salida debe ser algo parecido a lo siguiente.

```
Browsing 00:19:2C:DB:57:3D ...
Service RecHandle: 0x0
Service Class ID List:
  "SDP Server" (0x1000)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "SDP" (0x0001)
Profile Descriptor List:
  "SDP Server" (0x1000)
    Version: 0x0100

Service Name: Dial-up networking Gateway
Service Description: Dial-up networking Gateway
Service Provider: /a/mobile/system/cl.gif
Service RecHandle: 0x10001
Service Class ID List:
  "Dialup Networking" (0x1103)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 1
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
  code_ISO639: 0x6672
  encoding:    0x6a
  base_offset: 0xd800
  code_ISO639: 0x6573
  encoding:    0x6a
  base_offset: 0xd803
  code_ISO639: 0x7074
  encoding:    0x6a
  base_offset: 0xd806
Profile Descriptor List:
  "Dialup Networking" (0x1103)
    Version: 0x0100

Service Name: Voice Gateway
Service Description: Headset Audio Gateway
Service Provider: /a/mobile/system/cl.gif
Service RecHandle: 0x10003
Service Class ID List:
  "Headset Audio Gateway" (0x1112)
  "Generic Audio" (0x1203)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 3
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
```

```
  code_ISO639: 0x6672
  encoding:    0x6a
  base_offset: 0xd800
  code_ISO639: 0x6573
  encoding:    0x6a
  base_offset: 0xd803
  code_ISO639: 0x7074
  encoding:    0x6a
  base_offset: 0xd806
Profile Descriptor List:
  "Headset" (0x1108)
    Version: 0x0100

Service Name: Hands-Free voice gateway
Service Description: Hands-Free voice gateway
Service Provider: /a/mobile/system/cl.gif
Service RecHandle: 0x10007
Service Class ID List:
  "Handfree Audio Gateway" (0x111f)
  "Generic Audio" (0x1203)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 7
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
  code_ISO639: 0x6672
  encoding:    0x6a
  base_offset: 0xd800
  code_ISO639: 0x6573
  encoding:    0x6a
  base_offset: 0xd803
  code_ISO639: 0x7074
  encoding:    0x6a
  base_offset: 0xd806
Profile Descriptor List:
  "Handsfree" (0x111e)
    Version: 0x0101

Service Name: OBEX Object Push
Service Description: OBEX Object Push
Service Provider: /a/mobile/system/cl.gif
Service RecHandle: 0x10008
Service Class ID List:
  "OBEX Object Push" (0x1105)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 8
  "OBEX" (0x0008)
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
```

```
    code_ISO639: 0x6672
    encoding:    0x6a
    base_offset: 0xd800
    code_ISO639: 0x6573
    encoding:    0x6a
    base_offset: 0xd803
    code_ISO639: 0x7074
    encoding:    0x6a
    base_offset: 0xd806
  Profile Descriptor List:
    "OBEX Object Push" (0x1105)
      Version: 0x0100

  Service Name: OBEX File Transfer
  Service Description: OBEX File Transfer
  Service Provider: /a/mobile/system/cl.gif
  Service RecHandle: 0x10009
  Service Class ID List:
    "OBEX File Transfer" (0x1106)
  Protocol Descriptor List:
    "L2CAP" (0x0100)
    "RFCOMM" (0x0003)
      Channel: 9
    "OBEX" (0x0008)
  Language Base Attr List:
    code_ISO639: 0x656e
    encoding:    0x6a
    base_offset: 0x100
    code_ISO639: 0x6672
    encoding:    0x6a
    base_offset: 0xd800
    code_ISO639: 0x6573
    encoding:    0x6a
    base_offset: 0xd803
    code_ISO639: 0x7074
    encoding:    0x6a
    base_offset: 0xd806
  Profile Descriptor List:
    "OBEX File Transfer" (0x1106)
      Version: 0x0100
```

BIBLIOTECAS

Las bibliotecas utilizadas para el envío de mensajes SMS son rxtx y SMSLib; las primera para el envío de información vía Bluetooth hacia TRS y la segunda para la codificación del mensaje como un comando AT.

Las bibliotecas SMSLib requieren de un archivo de propiedades llamado "gnu.io.rxtx.properties" el cual debe ser ubicado en el directorio $JAVA_HOME/jre/lib/ext.

El archivo "gnu.io.rxtx.properties" debe contener:

gnu.io.rxtx.SerialPorts=/dev/rfcomm0
os.name=linux

La primera indica el puerto por el cual nos vamos a conectar a la TRS y la segunda el sistema operativo.

Programa por lotes mataproceso.sh

```
#!/bin/bash

for i in $( ps -fea | grep $1 | awk   '{c=split($2, s); for(n=1; n<=c;
++n) print s[n] }');
do kill -9  $i;
break;
done
```

Apéndice B

Especificación de un Sistema Informático para el Manejo de Información a Través de Terminales Remotas

Autor: Benjamín Macías
Proyecto: PAPIIT UNAM ("Sistemas Remotos de Interacción con Internet")
Entregable: IS-3 1
Elaboración: 10 de julio 2006
Última revisión: 30 de agosto de 2006
Estatus: Pendiente

0. Introducción

Este documento presenta una especificación de los requerimientos básicos del sistema de transporte de información asociado con el proyecto. Se usa para ello la metodología de estudio de escenarios ("use cases") de orientación a objetos (e. g. Jacobson et al. 1999). De acuerdo a esta metodología, se espera que el proyecto se desarrolle en varios ciclos. Por ello, más que presentar una especificación completa, nos concentramos en los casos centrales del sistema, con el propósito de producir un prototipo del sistema entero funcionando a la brevedad posible.

Seguiremos al proyecto original en identificar informalmente a los componentes principales del sistema como "transporte", "control", "almacenamiento", y así sucesivamente. En el documento de análisis y diseño daremos una descripción más adecuada de ellos.

1. Requerimientos Funcionales

1.1 Escenario 1.

Transferencia de mensajes de correo electrónico del servidor al cliente. En su presentación, el proyecto esboza el siguiente escenario de uso:

La usuaria final envía un mensaje SMS o de correo electrónico solicitando la lectura de su buzón de correo electrónico: "LEE CORREO NUEVO"
El componente de Transporte recibe el mensaje, y lo almacena.
El componente de Transporte notifica a Control la llegada del mensaje.
Control recupera los datos del usuario e indica al Manipulador que inicie un diálogo con el servidor de correo electrónico y baje los correos nuevos (i.e. comportándose como un cliente POP3).

El Manipulador transfiere los datos al Archivista.

Control solicita al Archivista la producción de sumarios para pasar a la salida.
Dependiendo del caso y de la configuración del sistema, la respuesta del Archivista

puede ser alternativamente:

"NO HAY MENSAJES NUEVOS"
lista de encabezados, ordenados tal vez por importancia del que los envió.
Contenido sintetizado de un único, o más importante, mensaje.

Control instruye al Transportista, quien envía el mensaje SMS final a la usuaria.

De esta descripción se deriva el siguiente escenario (cada enunciado en cursivas representa un enunciado del esbozo recién mencionado; las líneas en paréntesis completan la descripción pero se consideran como no relevantes para propósitos de esta especificación):

1.1.1 Paso 1

La usuaria final envía un mensaje SMS o de correo electrónico solicitando la lectura de su buzón de correo electrónico.

La usuaria enciende su TR.
La usuaria abre la ventana en su TR que activa el sistema de intercambio de información.
La usuaria selecciona la opción de lectura de buzón de correo.
La usuaria completa los datos necesarios.
La usuaria selecciona la opción de enviar el comando.
La ventana envía el mensaje en forma de mensaje SMS.
La usuaria apaga su TR.

1.1.2 Paso 2

El componente de Transporte recibe el mensaje, lo almacena, y notifica a Control la llegada del mensaje.

El componente de transporte se encuentra activado, y listo para recibir mensajes de un cliente.
El componente de transporte recibe un mensaje.
El componente de transporte notifica al componente de control el mensaje recibido.
El componente de transporte regresa a su estado inicial, listo para recibir mensajes.

1.1.3 Paso 3

El componente de control entra en un diálogo con los componentes de manipulación y de almacenamiento para elaborar una respuesta;

eventualmente, el componente de control es notificado que la respuesta ha sido generada. El contenido de la respuesta está en el archivista.

El componente de control se encuentra activado, y listo para recibir mensajes del componente de transporte.
El componente de control solicita al componente de almacenamiento que guarde el mensaje recibido.
El componente de control instruye al componente de manipulación sobre las acciones a realizar sobre el mensaje recibido.
El componente de control regresa a su estado inicial, listo para recibir nuevos mensajes.
Los componentes de manipulación colaboran para producir una respuesta. Cuando la respuesta está lista, el componente de manipulación notifica al componente de control.

1.1.4 Paso 4

Control instruye al componente de transporte, quien envía el mensaje SMS final a la usuaria.

El componente de control se encuentra activado, y listo para recibir mensajes del componente de manipulación.
El componente de control recibe el mensaje del componente de manipulación de que el mensaje está listo para ser entregado.
El componente de control recupera los detalles de la dirección de destino del mensaje.
El componente de control le da al mensaje de salida su formato final (usando c.).
El componente de control envía el mensaje de respuesta en forma de mensaje SMS a la dirección deseada.
La usuaria recibe el mensaje SMS con la respuesta.
La usuaria enciende su TR y arranca la ventana para recibir mensajes.
La usuaria lee el mensaje de respuesta.

Clase Usuario

Esta clase representa al dueño del TR; no es necesario modelarlo.

Clase Ventana

Esta clase representa al navegador ejecutando en el TR del cliente. Se introduce esta clase para propósitos de análisis; es posible que su funcionalidad sea redefinida para simplificarla en el diseño. Esta clase es la única clase ejecutando en el TR del cliente.

Nota: La implementación de esta clase es opcional. Para el caso de los teléfonos celulares, éstos ya cuentan con software para enviar mensajes. Sin embargo, es conveniente

contar con un cliente en el TR del usuario para poder formatear mensajes,hacer parsing sobre los mensajes recibidos, etc.

Métodos

abrir():activa el sistema de intercambio de información cliente que pide el servicio; solicita: Usuario; aparece: en 1.b.

int seleccionaFuncion(): se selecciona el tipo de función a realizar; solicita: Usuario; aparece: 1.c.

String[] completaDatos(): se completa los datos necesarios; solicita: Usuario; aparece: 1.d.

solicitaEnvio(String): ordenar el envío del mensaje; solicita: Usuario; aparece: 1.e.

envia(String): envío de mensaje SMS al componente de control; solicita: Ventana; aparece: 1.f

recibe(): despliega un mensaje recibido en el TR; solicita: Usuario.

Nota 1: el mensaje ha llegado a través de la red telefónica; la función de este método (que se ejecuta una vez que se ha abierto la ventana y seleccionado la recepción de un mensaje) sucede una vez que el TR ha recibido el mensaje y lo almacenado.

Nota 2: Se usan clases terminales como String en este punto porque se presume que la funcionalidad del TR impide o hace impráctico usar clases para envolver datos tales como mensajes.

Encadenamiento(pendiente).

Clase Receptor

Esta clase instancia la primera parte del componente de transporte. Su misión es recibir mensajes remotos de TRs de usuarios a través de telefonía pública y canalizarlos a la otra parte del componente de transporte. Esta clase reside en el teléfono celular conectado al servidor. La clase Transporte (ver abajo) instancia el resto de la funcionalidad del componente de transporte.

Métodos

recibeEntrada(): detecta el envío de un mensaje por parte de un TR enviado por la red de telefonía; invoca a enviaEntrada(...); solicita: Ventana; aparece: 2.a, 2.b.

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

Nota 2: esta función aparece mencionada una vez en la especificación; ha sido desdoblada en dos para propósitos del análisis; ver Transporte.recibe(...).

enviaEntrada(MensajeEntrada): envía físicamente el mensaje recibido a la máquina servidor; invoca a Transporte.recibeEntrada(...); solicita: recibeEntrada(...); aparece: no aparece

Nota 1: esta función conceptualmente trabaja como un evento que se dispara cada vez que recibeEntrada(...) se activa.

Nota 2: el envío se hace físicamente conectando el celular con el puerto USB del servidor a través de un cable o de tecnología *Bluetooth.*

recibeSalida(MensajeSalida): detecta el envío de un mensaje por parte de Transporte; invoca a enviaSalida(...); solicita: Transporte.enviaSalida(...); aparece: 4.e

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

Nota 2: esta función recibe físicamente el mensaje conectando el celular con el puerto USB del servidor a través de un cable o de tecnología *Bluetooth.*

enviaSalida(MensajeSalida): envía físicamente el mensaje por la red de telefonía pública; solicita: recibeSalida(...); aparece: 4.e.

Clase Transporte

Esta clase instancia la segunda parte del componente de transporte. Su misión es recibir mensajes de la clase Receptor y canalizarlos al resto de los componentes del sistema.

Esta clase reside en la máquina servidor.

Métodos

recibeEntrada(MensajeEntrada): detecta el envío de un mensaje de entrada por parte de la clase Receptor; invoca a enviaEntrada(...); solicita: Receptor.envia(...); aparece: 2.a, 2.b

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

Nota 2: esta función aparece mencionada una vez en la especificación; ha sido desdoblada en dos para propósitos del análisis; ver Receptor.recibe().

enviaEntrada(MensajeEntrada): envía el mensaje recibido al componente de control para su proceso; invoca a

Control.recibeEntrada(...); solicita: recibeEntrada(...); aparece: 2.c.

Nota 1: esta función conceptualmente trabaja como un evento que se dispara cada vez que recibeEntrada(...) se activa.

recibeSalida(MensajeSalida): detecta el envío de un mensaje de salida por parte de la clase Control; invoca a enviaSalida(...); solicita: Control.procesaMensajeSalida(...); aparece: 4.e.

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

Nota 2: la funcionalidad descrita en 4.e esta repartida entre Control.procesaMensajeSalida(...), este método, recibeSalida(...) y Receptor.recibeSalida(...).

enviaSalida(MensajeSalida)
función: envía el mensaje recibido al Receptor; invoca a Receptor.recibeSalida(); solicita: recibeSalida(...); aparece: 4.e.

Clase Control

Esta clase instancia el componente de control. Su tarea es coordinar a los componentes que almacenan el mensaje, lo procesan, y producen la respuesta. Puede pensarse en él como el control del tráfico que fluye hacia el servidor desde los TRs y desde los TRs hacia el servidor. Esta clase reside en la máquina servidor.

Métodos

recibeEntrada(MensajeEntrada): detecta el envío de un mensaje por parte de la clase Transporte; invoca a procesaMensajeEntrada(...); solicita: Transporte.envia(...); aparece: 2.c.

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

procesaMensajeEntrada(MensajeEntrada): organiza el proceso de un mensaje de entrada:

1. Debe de almacenar los detalles generales relativos al mensaje de entrada (hora de llegada, identificador del TR, longitud, identificador interno, etc).
2. Ver que se almacene.
3. Establecer el tipo de servicio solicitado en el mensaje.
4. Ponerse en contacto con el componente relevante para brindar el servicio.

Solicita: recibeEntrada(...).
Aparece: 3.b, 3.c.

Nota 1: esta función conceptualmente trabaja como un evento que se dispara cada vez que recibe(...) se activa.

Nota 2: la especificación menciona solamente un componente de manipulación. Para propósitos de análisis podemos suponer que hay un componente para cada tipo servicio que se presta, todos implementando la misma interfase.

Nota3: hay una clase extra implícita para representar el mensaje después de que ha sido procesada. La llamaremos MensajeEntradaInterno.

recibeSalida(MensajeSalidaInterno): detecta que algún componente de manipulación a terminado de elaborar un mensaje de salida, por lo que éste está listo para ser enviado; invoca a procesaMensajeSalida(...); solicita: Manipulador.envia(...); aparece: 3.d.

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

procesaMensajeSalida(MensajeSalidaInterno):organiza el proceso de producción de un mensaje de salida:

1. Almacena los detalles generales del mensaje de salida (componente que lo produjo, hora, identificador del mensaje).
2. Almacena el mensaje de salida (esto es, transfiere una copia del mensaje que llego del Manipulador al Almacenamiento).
3. Establece los detalles del TR receptor.
4. Formatea el mensaje en su forma final y lo envía al Transporte. Invoca a Transporte.recibeSalida(...).

Solicita: recibeSalida().
Aparece: 4.b, 4.c, 4.d, 4.e.

Clase Almacenamiento

Esta clase es una clase auxiliar de Control; su función es servir como la memoria permanente del sistema.

Métodos
(pend)

Interface Manipulador

Esta clase abstracta define el API entre Control y cada una de sus subclases. Cada subclase implementa un servicio (lectura de correo, agenda, etc) específico del sistema.

Métodos
(pend)

Otras clases (pend):

MensajeEntrada
MensajeSalida
MensajeEntradaInterno
MensajeSalidaInterno

Apéndice C.

Análisis y Diseño de un Sistema Informático para el Manejo de Información a Través de Terminales Remotas

Autor: Benjamín Macías
Proyecto: PAPIIT UNAM ("Sistemas Remotos de Interacción con Internet")
Entregable: IS-3 2
Elaboración: 31 de julio 2006
Última revisión: 30 de agosto de 2006

0. Introducción

Este documento presenta el análisis y diseño del sistema de transporte de información asociado con el proyecto. Este documento está basado en la especificación de requerimientos del sistema contenida en el documento IS-3 1. Como dicho documento, el análisis y diseño usan de manera general la metodología de orientación a objetos (e. g. Jacbson et al. 1999), aunque adaptada a nuestras necesidades.

1. Análisis

1.1 Clases y su funcionalidad

El elemento principal del análisis queda dado por las clases que se derivan de la especificación. Nótese que estas clases se han introducido para propósitos de replantear la descripción de la especificación en términos computacionales. Al momento del diseño, algunas de ellas pueden desaparecer o redefinirse en otras clases.

1.1.1 Clase Usuario

Esta clase representa al dueño del TR; no es necesario modelarlo.

1.1.2 Clase Ventana

Esta clase representa al navegador ejecutando en el TR del cliente. Se introduce esta clase para propósitos de análisis; es posible que su funcionalidad sea redefinida para simplificarla en el diseño. Esta clase es la única clase ejecutando en el TR del cliente.

Nota: La implementación de esta clase es opcional. Para el caso de los teléfonos celulares, éstos ya cuentan con software para enviar mensajes. Sin embargo, es conveniente

contar con un cliente en el TR del usuario para poder formatear mensajes, hacer parsing sobre los mensajes recibidos, etc.

Métodos

abrir(): activa el sistema de intercambio de información del cliente que solicita el servicio: Usuario; aparece mencionada en la especificación en: 1.b

int seleccionaFuncion()
función: se selecciona el tipo de función a realizar solicita: Usuario aparece: 1.c.

String[] completaDatos()
función: se completa los datos necesarios; solicita: Usuario; aparece: 1.d

solicitaEnvio(String)
función: ordenar el envío del mensaje; solicita: Usuario; aparece: 1.e

envia(String)
función: envío de mensaje SMS al componente de control; solicita: Ventana; aparece: 1.f.

recibe()
función: despliega un mensaje recibido en el TR; solicita: Usuario.

Nota1: el mensaje ha llegado a través de la red telefónica; la función de este método (que se ejecuta una vez que se ha abierto la ventana y seleccionado la recepción de un mensaje) sucede una vez que el TR ha recibido el mensaje y lo almacenado.

Nota2: Se usan clases terminales como String en este punto porque se presume que la funcionalidad del TR impide o hace impráctico usar clases para envolver datos tales como mensajes.

1.1.3 Clase Receptor

Esta clase instancia la primera parte del componente de transporte. Su misión es recibir mensajes remotos de TRs de usuarios a través de telefonía pública y canalizarlos a la otra parte del componente de transporte. Esta clase reside en el teléfono celular conectado al servidor. La clase Transporte (ver abajo) instancia el resto de la funcionalidad del componente de transporte.

Métodos

recibeEntrada()

función: detecta el envío de un mensaje por parte de un TR enviado por la red de telefonía; invoca a enviaEntrada(...); solicita: Ventana; aparece: 2.a, 2.b.

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

Nota 2: esta función aparece mencionada una vez en la especificación; ha sido desdoblada en dos para propósitos del análisis; ver Transporte.recibe(...).

enviaEntrada(MensajeEntrada): envía físicamente el mensaje recibido a la máquina servidor; invoca a Transporte.recibeEntrada(...); solicita: recibeEntrada(...); aparece: no aparece.

Nota 1: esta función conceptualmente trabaja como un evento que se dispara cada vez que recibeEntrada(...) se activa.

Nota 2: el envío se hace físicamente conectando el celular con el puerto USB del servidor a través de un cable o de tecnología *Bluetooth.*

recibeSalida(MensajeSalida): detecta el envío de un mensaje por parte de Transporte; invoca a enviaSalida(...);solicita:Transporte.enviaSalida(...); aparece: 4.e.

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

Nota 2: esta función recibe físicamente el mensaje conectando el celular con el puerto USB del servidor a través de un cable o de tecnología *Bluetooth.*

enviaSalida(MensajeSalida): envía físicamente el mensaje por la red de telefonía pública solicita: recibeSalida(...); aparece: 4.e.

1.1.4 Clase Transporte

Esta clase instancia la segunda parte del componente de transporte. Su misión es recibir mensajes de la clase Receptor y canalizarlos al resto de los componentes del sistema. Esta clase reside en la máquina servidor.

Métodos

recibeEntrada(MensajeEntrada): detecta el envío de un mensaje de entrada por parte de la clase Receptor; invoca a enviaEntrada(...); solicita: Receptor.envia(...); aparece: 2.a, 2.b.

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

Nota 2: esta función aparece mencionada una vez en la especificación; ha sido desdoblada en dos para propósitos del análisis; ver Receptor.recibe().

enviaEntrada(MensajeEntrada)
función: envía el mensaje recibido al componente de control para su proceso; invoca a Control.recibeEntrada(...); solicita: recibeEntrada(...); aparece: 2.c

Nota 1: esta función conceptualmente trabaja como un evento que se dispara cada vez que recibeEntrada(...) se activa.

recibeSalida(MensajeSalida): detecta el envío de un mensaje de salida por parte de la clase Control; invoca a enviaSalida(...); solicita: Control.procesaMensajeSalida(...); aparece: 4.e.

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

Nota 2: la funcionalidad descrita en 4.e esta repartida entre Control.procesaMensajeSalida(...), este método, recibeSalida(...) y Receptor.recibeSalida(...).

enviaSalida(MensajeSalida): envía el mensaje recibido al Receptor; invoca a Receptor.recibeSalida(); solicita: recibeSalida(...); aparece: 4.e.

1.1.5 Clase Control

Esta clase instancia el componente de control. Su tarea es coordinar a los componentes que almacenan el mensaje, lo procesan, y producen la respuesta. Puede pensarse en él como el control del tráfico que fluye hacia el servidor desde los TRs y desde los TRs hacia el servidor. Esta clase reside en la máquina servidor.

Métodos

recibeEntrada(MensajeEntrada): detecta el envío de un mensaje por parte de la clase Transporte; invoca a procesaMensajeEntrada(...); solicita: Transporte.envia(...); aparece: 2.c.

Nota 1: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

procesaMensajeEntrada(MensajeEntrada): organiza el proceso de un mensaje de entrada:

1. Debe de almacenar los detalles generales relativos al mensaje de entrada (hora de llegada, identificador del TR,

longitud, identificador interno, etc).
2. Ver que se almacene.
3. Establecer el tipo de servicio solicitado en el mensaje.
4. Ponerse en contacto con el componente relevante para brindar el servicio.

Solicita: recibeEntrada(...).
Aparece: 3.b, 3.c.

Nota 1: esta función conceptualmente trabaja como un evento que se dispara cada vez que recibe(...) se activa.

Nota 2: la especificación menciona solamente un componente de manipulación. Para propósitos de análisis podemos suponer que hay un componente para cada tipo servicio que se presta, todos implementando la misma interfase.

Nota3: hay una clase extra implícita para representar el mensaje después de que ha sido procesada. La llamaremos MensajeEntradaInterno.

recibeSalida(MensajeSalidaInterno): detecta que algún componente de manipulación a terminado de elaborar un mensaje de salida, por lo que éste está listo para ser enviado; invoca a procesaMensajeSalida(...); solicita: Manipulador.envia(...); aparece: 3.d.

Nota: conceptualmente, esta función trabaja como un demonio en un ciclo infinito.

procesaMensajeSalida(MensajeSalidaInterno): organiza el proceso de producción de un mensaje de salida:
1. Almacena los detalles generales del mensaje de salida (componente que lo produjo, hora, identificador del mensaje).
2. Almacena el mensaje de salida (esto es, transfiere una copia del mensaje que llego del Manipulador al Almacenamiento).
3. Establece los detalles del TR receptor.
4. Formatea el mensaje en su forma final y lo envía al Transporte. Invoca a Transporte.recibeSalida(...)

Solicita: recibeSalida().
Sparece: 4.b, 4.c, 4.d, 4.e.

1.1.6 Clase Almacenamiento

Esta clase es una clase auxiliar de Control; su función es servir como la memoria permanente del sistema.

Métodos
(pend)

1.1.7 Interface Manipulador

Esta clase abstracta define el API entre Control y cada una de sus subclases. Cada subclase implementa un servicio (lectura de correo, agenda, etc.) específico del sistema.

Métodos
(pend)

1.1.8 Otras clases (pend):

MensajeEntrada
MensajeSalida
MensajeEntradaInterno
MensajeSalidaInterno

1.2 Flujo de Control (PEND)

2. Diseño

El siguiente código en pseudo-Java especifica la manera de implementar el análisis de la sección 1. Algunos detalles han quedado sin definir, pendiente a los detalles de la implementación del teléfono celular con el que estaremos trabajando, tanto para implementar la clase Ventana en el TR del usuario, como el método Receptor en el celular asociado al servidor.

Sin embargo, debe de haber suficiente detalle aquí para que la implementación sea inmediata.

2.1 Código

```
/*
      Ejemplifica definición de ventana en el TR.
      Detalles de widgets, eventos etc, son ilustrativos, no exactos.
      Reside en el celular del usuario.
*/

Ventana {

      Window v; // ventana
            Botton b; // boton etc

            // Llamada externa
            abrir() {
                  v = new Window();
                  b = new Botton();
                  b.event( accionEvento() ); // ata evento a boton
                  v.attach( b );                // ata boton a ventana
                  ...
                  v.display();                  // abre ventana
            }

            // Obtiene id función de ventana
            int seleccionaFuncion() {
                  return Integer.parseInt(v.value("FUNCION"));
            }

            // Obtiene datos de ventana
            String seleccionaDatos() {
                  return Integer.parseInt(v.value("DATO"));
            }
```

```
		...

		// Se acciona con el boton de la ventana
		accionEvento() {
			int i = v.seleccionaFuncion();
			// selecciona accion con base a valor función
			switch (i) {
				case Constantes.ENVIA_CORREO_E:
					String mensaje = v.seleccionaDatos();
					envia( cifrado (i,mensaje) );
				break;
				...
			}
		}

		// Cifrado: función opcional para formatear mensaje etc
		String cifrado(int,String) {
			...
		}

		// Envía un mensaje vía red telefónica
		envia(String) {
			// código nativo
			...
		}

		...
	}

	/*
		Modela 1a parte del componente de transporte.
		Reside en el celular del servidor.
	*/
Receptor {

	Sistema s;

	Receptor(Sistema s) {
		this.s = s;
		recibeEntrada();			// se activa al arrancar el sistema
	}

	// Crea un evento escuchando a nuevos mensajes
	// Ilustrativo, detalles no son exactos
	recibeEntrada() {
		Event e = new Event( System.incomingMessage(), enviaEntrada() );
		e.start();
	}

	// Se activa con nuevo mensaje
	enviaEntrada() {
		MensajeEntrada mensaje =
			new MensajeEntrada (System.incomingMessage().contents() );
		s.t.enviaEntrada(mensaje);
	}

	...

}

/*
	Modela 2a. parte del componente de transporte.
	Reside en el servidor.
*/
Transporte {

	Sistema s;

	Transporte(Sistema s) { this.s = s;}

	recibeEntrada(...){}			// innecesario

	enviaEntrada(MensajeEntrada m) {
		s.c.procesaMensajeEntrada(m);
	}
```

```
        ...
}

/*
        Modela componente de control.
        Reside en el servidor.
*/
Control {

        Sistema s;

        Control (Sistema s) { this.s = s;}

        recibeEntrada(MensajeEntrada){}             // innecesario

        procesaMensajeEntrada(MensajeEntrada m) {
                // almacena mensaje entrada
                Hora t1 = System.timeIs();
                Emisor s1 = m.EmisorEs();
                int long1 = m.longIs();
                ...
                MensajeEntradaInterno m2 =
                        new MensajeEntradaInterno(t1,s1,long1,...);
                s.a.guarda(m2);

                // decide tipo proceso
                int i = m.accionEs();
                switch (i) {
                        case Constantes.ENVIA_CORREO_E:
                                s.mS[0].servicio(m);          // no espera por
                                                                // respuesta!
                        break;
                        ...
                }
        }

        ...

}

/*
        Modela el componente de permanencia ("persistence") del sistema.
        Reside en el servidor.
*/
Almacen {

        Sistema s;

        Almacen (Sistema s) { this.s = s;}

        guarda(MensajeEntradaInterno) {...}          // almacena mensaje

        ...

}

/*
        Clase auxiliar; wrapper de todos los componentes y ayuda a intercomunicarlos.
        Reside en el servidor; se arranca antes de comenzar el proceso.
        Nota: en este diseño, la clase Receptor ya esta andando al arrancar el sistema
        en el servidor.
*/
Sistema {

        Receptor r;
        Transporte t;
        Control c;
        Almacen a;
        Manipulador[] mS;

        Sistema() {
                r = conectaConReceptor(); // Receptor esta en el celular
                t = new Transporte(this);
                c = new Control(this);
                a = new Almacen(this);
                mS = new Manipulador[] {
                                new ManipuladorCorreo(this), ...
```

```
                        };
        }

        main() {
                ...
        }
}
/*
        Fin de diseño
*/
```