*Process MeNtOR 3.o*

*Uni-SEP*

Bunty Communicator

**Design Document**

|  |  |
| --- | --- |
| Version: | 1.0 |
| Print Date: | 3/13/2019 |
| Release Date: | 3/13/2019 |
| Release State: | Final |
| Approval State: | Approved |
| Approved by: | G001 |
| Prepared by: | G001: NAMES HERE |
| Reviewed by: | G001 |
| File Name: | G001SDD.doc |

Document Change Control

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Authors** | **Summary of Changes** |
| 0.6 | 3/13/2019 | G001 | Added Class Diagrams added |
| 0.7 | 3/13/2019 | G001 | Component and Deployment Diagrams added |
| 0.8 | 3/13/2019 | G001 | Formatting |
| 0.9 | 3/13/2019 | G001 | Call State Diagram added |
| 1.0 | 3/13/2019 | G001 | Updated Psuedocode |
| 1.1 | 3/13/2019 | G001 | Intro and minor modifications to diagrams |

# Document Sign-Off

|  |  |  |
| --- | --- | --- |
| **Name (Position)** | **Signature** | **Date** |
| N/A | N/A | N/A |

Table of Contents

[1 Introduction 4](#_Toc120469061)

[1.1 Overview 4](#_Toc120469062)

[1.2 Resources - References 5](#_Toc120469063)

[2 Major Design Decisions 6](#_Toc120469064)

[3 Architecture 7](#_Toc120469065)

[4 Detailed Class Diagrams 9](#_Toc120469066)

[4.1 UML Class Diagrams 9](#_Toc120469067)

[4.2 Method Details 11](#_Toc120469068)

[5 Detailed Class Diagrams 17](#_Toc120469071)

[6 Open Issues 18](#_Toc120469072)

[6.1 Gathering Organization Name 18](#_Toc120469073)

[6.2 Empty Service Name Invoke 18](#_Toc120469074)

[6.3 Web Services Querying 18](#_Toc120469075)

[6.4 Other Open Issues 18](#_Toc120469076)

[7 Domain Dictionary 19](#_Toc120469077)

[7.1 Terms and Abbreviations 19](#_Toc120469078)

Introduction

## Overview

In this project, a prototype system will be implemented that allows for the invocation of Web Services from within a SIP protocol environment. This document is divided into six sections that outline some details of the design phase. The resources and their references are cited accordingly.

This second and the third section of the document discusses in detail the design decisions that were made followed by a discussion of the architecture that includes component and deployment diagrams. Only the JAIN-SIP and SIP Communicator packages are considered. The rest of the components are treated as gross components. The deployment diagram assumes the current run-time configuration that will be used to test our system.

The fourth section is followed by class diagrams that present the details of the classes that will be written and/or modified. The class diagrams are divided per module such as Proxy Server, SIP Agent and the Wrapper class. The class diagrams contain the classes in UML notation and a table for each class with its data members and methods with appropriate signatures. This section also includes the pseudo code for all the major methods in the classes that will be written and/or modified.

The fifth section contains a state diagram for the Call class. Any issues with the system are discussed in detail in the final section of this document.

## Resources - References

SIP overview [Understanding SIP](http://www.iptel.org/sip/siptutorial.pdf), [iptel.org](http://www.iptel.org/)

SIP protocol [SIP Protocol Overview](http://www.radvision.com/NR/rdonlyres/0F6C2E81-72B3-4C42-B33C-DDBC9115AC3D/234/SIPProtocolOverview.pdf),

RADVISION SIP Protocol [RCF 3261 - Sip: Session Initiation Protocol](http://rfc.net/rfc3261.html)

JAIN SIP project [NIST JAIN-SIP project home page](https://jain-sip.dev.java.net/) Sun Microsystems

JAIN SIP design tutorial [JAIN SIP Tutorial](http://dns.antd.nist.gov/proj/iptel/tutorial/JAIN-SIP-Tutorialv2.pdf), Phelim O'Doherty, Mudumbai Ranganathan

JAVA XML Registry API <http://java.sun.com/webservices/jaxr/index.jsp>

UDDI and JAXR <http://www.onjava.com/pub/a/onjava/2002/02/27/uddi.html>

WSDP: <http://java.sun.com/webservices/jwsdp/index.jsp>

SIP-COMMUNICATOR: <https://sip-communicator.dev.java.net/>

ebXML: <http://www.ebxml.org/>

UDDI: <http://www.uddi.org/>

Open Issues:

SIP Communicator <https://sip-communicator.dev.java.net/servlets/Search;jsessionid=09354DD40134E76080972490E64D6B11?scope=projectAndSubs&resultsPerPage=40&query=Open+Issues&Button=Go>

SIP Proxy <https://jain-sip-presence-proxy.dev.java.net/servlets/Search?scope=projectAndSubs&resultsPerPage=40&query=Open+Issues&Button=Go>

Major Design Decisions

The design decisions that were made during this project spanned across separate modules. Firstly, the Jain-SIP API uses message passing via buffered strings in different sections of the SIP packets. For example, while building a decline message, an optional message content field can be used to hold extra information such as a list of web services. This information must be in textual format to use the Jain SIP API. A problem with such a scheme is that the text must be formatted in a specific order and then decoded on the receiving end to provide meaningful information. An alternative approach could be used that involves using an XML based structure to standardize the message formatting and passing between entities. This scheme allows easier parsing and encoding of data on either end and thus makes it more scalable.

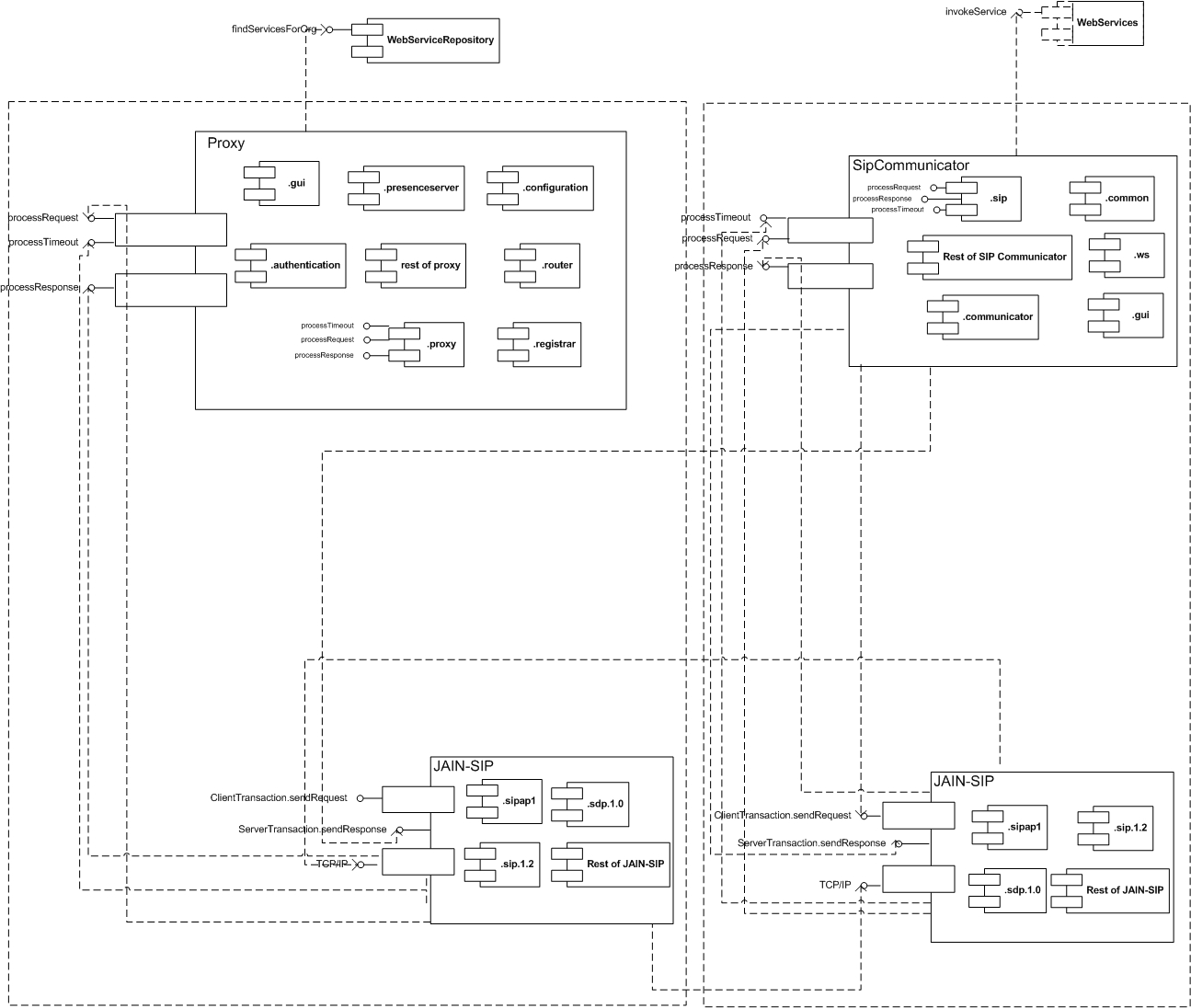
Another design decision that was made was regarding the creating of various SIP messages using the Factory Design pattern. The *MessageFactory* class uses this design pattern to create SIP messages based on a context. Another design pattern that was used in this project was the Singleton design pattern in the design of the Wrapper class. Since there is a requirement that only a single instance of the wrapper be created, this design pattern was used to satisfy this constraint.

To invoke web services from the *Sip Communicator*, the strategy design pattern could be used to create specific classes that have a strategy associated with different types of web services. This allows future expansions in the system to support new web services dynamically.

# Architecture

This project uses the client-server architecture style. This can be illustrated by the Proxy Server interacting with the Sip Communicator and the SIP communicator interacting with Web Services via the wrapper. In either case, the user interacts with Web Services and the Proxy Server via the SIP Communicator that acts as the client.

### The Component Diagram for the system is show below:



*Fig 3.1: Component Diagram of System*

### The Deployment Diagram for the system is shown below:



*Fig 3.2: Deployment Diagram of System*

Note: View the above diagram in MS Visio. All lines are “dotted”.

# Detailed Class Diagrams

## UML Class Diagrams

The major classes that were modified and their Class Diagrams are provided:

* + 1. Proxy
    2. WSEvent

### The Proxy Class diagram is shown below:

 *Fig 4.1: Class Diagram for Proxy Class*

### The WSEvent Class diagram is shown below:



*Fig 4.2: Class Diagram for WSEvent Class*

In addition to the above design patterns used in the classes shown, the **Strategy Design** pattern is used in *WebServicesInvocationWrapper* Class.

### The tables below show the list of data members

***WSEvent***

Data Members

|  |
| --- |
| Protected WSListData Wsld |

Methods

|  |
| --- |
| Public WSEvent(String source) |
| Private void processMessageBody() |
| Public WSListData getWSList() |

***Proxy***

Data Members

|  |
| --- |
| protected LinkedList listeningPoints |
| protected SipStack sipStack |
| protected SipProvider defaultProvider |
| protected MessageFactory messageFactory |
| protected HeaderFactory headerFactory |
| protected AddressFactory addressFactory |
| protected Configuration configuration |
| protected PresenceServer presenceServer |
| protected Registrar registrar |
| protected ProxyUtilities proxyUtilities |
| protected Authentication authentication |
| protected RequestForwarding requestForwarding |
| protected ResponseForwarding responseForwarding |

Methods:

|  |
| --- |
| Public RequestForwarding getRequestForwarding)() |
| Public ResponseForwarding getResponseForwarding() |
| Public AddressFactory getAddressFactory() |
| Public MessageFactory getMessageFactory() |
| Public HeaderFactory getHeaderFactory() |
| Public Registrar getRegistrar() |
| Public Boolean isPresenceServer() |
| Public PresenceServer getPresenceServer() |
| Public ProxyUtilities getProxyUtilities() |
| Public SipStack getSipStack() |
| Public Configuration getConfiguration() |
| Public SipProvider getSipProvider() |
| Public Authentication getAuthentication() |
| Public Boolean managesDomain (String domainAddress) |
| Public Proxy(String confFile) throws Exception |
| Public void processRequest (RequestEvent requestEvent) |
| Public void processResponse(ResponseEvent responseEvent) |
| Public void processTimeout(TimeoutEvent timeOutEvent) |
| Public void start() throws Exception |
| Public void stop() throws Exception |
| Public void exit() throws Exception |
| Public ViaHeader getStackViaHeader() |
| Public ContactHeader getStackContactHeader() |
| Public static void main(String args[]) |

## Method Details

There are six files that have been modified:

* + 1. Proxy.java 2) WebServicesInvocationWrapper.java

3) WSEvent.java 4) MailServiceInvoker.java

5) SipCommunicator.java

The following is the pseudocode for the *processRequest* method in *proxy.java*:

public void processRequest(RequestEvent requestEvent) {

.

.

.

if ( !requestValidation.validateRequest

(sipProvider,request,serverTransaction) ) {

.

.

.

}

if ( request.getMethod().equals(Request.ACK) ) {

.

.

.

}

if (serverTransaction==null) {

.

.

.

}

ListIterator routes = request.getHeaders(RouteHeader.NAME);

if (routes!=null) {

.

.

.

}

if (requestURI.isSipURI()) {

.

.

.

}

else {

.

.

.

}

if (requestURI.isSipURI()) {

.

.

.

}

if (requestURI.isSipURI()) {

.

.

.

}

// we use a SIP registrar:

if ( request.getMethod().equals(Request.REGISTER) ) {

.

.

.

}

if ( isPresenceServer() &&

(request.getMethod().equals(Request.SUBSCRIBE))) {

.

.

.

}

if ( isPresenceServer() && (request.getMethod().equals

(Request.NOTIFY) )) {

.

.

.

}

if ( isPresenceServer() && (

request.getMethod().equalsIgnoreCase("PUBLISH"))) {

.

.

.

}

if (request.getMethod().equals(Request.BYE) ) {

.

.

.

}

if (requestURI.isSipURI()) {

.

.

.

}

if ( registrar.hasRegistration(request) ) {

if (targetURIList!=null && !targetURIList.isEmpty()) {

String orgName = recepient.getName()

WebServicesQuery wsq = WebServicesQuery.getInstance()

Collection serviceInfoColl = wsq.getServices(orgName)

String messageBody = serviceInfoColl.size()

if (serviceInfoColl is not empty ){

Iterator servIter = serviceInfoColl.iterator()

while (servIter.hasNext()) {

ServiceInfo servInfo = (ServiceInfo)servIter.next()

messageBody += “ -- “

messageBody += servInfo.getDescription()+ “ “

messageBody += servInfo.getWsdluri() + " "

messageBody += servInfo.getEndPoint()

}

serviceInfoCall.clear()

Response response=messageFactory.createResponse(Decline, messageBody)

if (serverTransaction!=null) then

serverTransaction.sendResponse(response)

else

sipProvider.sendResponse(response)

}

}

else{

requestForwarding.forwardRequest(request)

return;

}

}

}

.

.

.

}

The following is the pseudocode for the *WSEvent.java file* :

*void processMessageBody(){*

*String messageContent = (String) getSource();*

*for (int i = 1; i < wsListArray.length; i++)*

*{*

*serviceArray = wsListArray[i].split(" ")*

*endPoint = serviceArray[serviceArray.length - 1]*

*URI = serviceArray[serviceArray.length - 2]*

*for(int j = 0 ; j < serviceArray.length -2 ; j++)*

*{desc += serviceArray[j] + " "}*

*listdata.addWSItem( URI,desc.trim(),endPoint)*

*}*

*wsld = listdata;*

*}*

The following is the pseudocode for the *WebServicesQuery*.java file:

*Public Collection findServicesForOrg(String orgName){*

*serviceInfoColl.clear() // Added This Line*

*services.clear() //Added This Line*

*init()*

*makeConnection()*

*executeQuery(orgName)*

*return serviceInfoColl*

}

The following is the pseudocode for the MailServiceInvoker.java file :

*public MailServiceInvoker(String mailWSDLUrl, String serviceEndPoint, String serviceName) {*

*this.nameSpace = serviceEndPoint*

*this.serviceEndPoint = serviceEndPoint*

*this.serviceName = serviceName + "Service”*

*serviceFactory = ServiceFactory.newInstance(;*

*service = serviceFactory.createService(new URL(mailWSDLUrl)*

*new QName(this.nameSpace, this.serviceName))*

*}*

# The following changes were made in the SipCommunicator.java file:

public void handleInvokeRequest(int serviceIndex) {

String address = Utils.getProperty("net.java.sip.communicator.sip.PUBLIC\_ADDRESS")

Object[] params = {address}

wsiw.invokeService(wsList.getWSDL(serviceIndex),

wsList.getEndPoint(serviceIndex), wsList.getDescription(serviceIndex),

params)

}

# The following changes were made in the WebServicesInvocationWrapper.java file:

public Object[] invokeService(String wsdlURL, String endPoint, String

serviceName,

Object[] params) throws Exception

{

WebServiceInvoker impl = new MailServiceInvoker(wsdlURL, endPoint

,serviceName)

return impl.invokeService(params)

}

# 

# Detailed Class Diagrams

The following is the State Diagram for the Call Class:



**Call Class: State Design Pattern**

*Fig 5.1: State Diagram for the Call Class*

# Open Issues

## Gathering Organization Name

Currently we have hard-coded the parsing of the receiving party into the code. If the format of the Header was changed, it would be possible that our code will fail. It is desirable that the functionality of the Header Class be modified to return the appropriate name of the recipient and not simply the SIP address. There is a property called *displayName* which does not seem to be getting initialized properly that could be used for this purpose.

## Empty Service Name Invoke

When trying to invoke a service with the dialog box empty (eg.. “\_\_\_\_\_\_“), the application returns an exception instead of throwing a friendlier message reminding the user to select a specific service from the list. This could be modified by taking care of this case and making the system more stable to unexpected input.

## Web Services Querying

Currently, the setup environment requires the organization to be online on the proxy in order for other users to utilize the web services under this organization. This seems rather unnecessary, since if a user would like to use the services provide for by the organization, the organization would itself have to be logged online. This functionality could be modified such that the organization does not need to stay online on the proxy for other users to utilize web services through them.

On the contrary, we can also understand that the reason this is currently the case is because this setup acts as a easy switch for the organizations to turn their Web Services on or off at any given time.

## Other Open Issues

The Issues above included issues to the extended SIP application. For more Open Issues with SIP communicator and Proxy, it is recommended to browse the Resources section under “***Open Issues***” link.

# Domain Dictionary

## Terms and Abbreviations

|  |  |
| --- | --- |
| Term | Definition |
| JAIN-SIP | Backbone of communication processes between all components of the Sip Communicator. Sends responses and processing requests using TCP/IP protocol. |
| Sip Communicator | The GUI which handles all the SIP calls, including sending requests to retrieve web services. |
| WebServiceRepository | Contains the list of services and their details registered by the Sip Communicator users. |
| UDDI Service Registry | This node contains the WebServiceRepository. |
| Web Service | A Web service is any piece of software that makes itself available over the Internet and uses a standardized XML messaging system. |
| UML | Is an industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems standardized by the Object Management Group. |
| Wrapper | Wrappers are a type of software "glueware" that is used to attach together other software components. A wrapper encapsulates a single data source to make it usable in a more convenient fashion than the original unwrapped source; this distinguishes wrappers from another kind of glue-ware, mediators, that combine data from different data sources. |
| XML | is a W3C initiative that allows information and services to be encoded with meaningful structure and semantics that computers and humans can understand. |