NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

SOFTWARE DESIGN DOCUMENT

Implement extensions
on
JAIN-SIP-PRESENCE-PROXY
and
SIP Communicator

Release:	1.0
Date of Print:	
Date of Release:	December, 2016
State of Release:	Initial
State of Approval:	Draft
Approved by:	Kontogiannis Konstantinos
Prepared by:	Kallas Konstantinos Karampasi Aikaterini Tzinis Eythymios
Inspected by:	Papaspyrou Nikolaos
Archive's Name:	Soft_Eng_SDD.pdf
Form's Number:	

Contents

1 Introduction

1.1 Overview

This document presents in detail the software design specifications of the subsystems Forwarding, Blocking and Charging a Call, which are extensions on the SiP Communicator. On chapter 2 we provide information about major design policies. Next, we refer in detail to the architecture of the subsystems we are going to implement and we create the component diagrams of these subsystems along with the deployment diagrams. On chapter 4 we provide the UML diagrams along with the details to explain them. On chapter 5 we provide some of the most important state diagrams of our classes. Finally, a domain dictionary is given.

1.2 References

1. Lecture notes
2. RFC 3261 - SIP: Session Initiation Protocol: http://www.ietf.org/rfc/rfc3261.txt

2 Major Design Decisions

Good Scenarios

2.1 Blocking

Each user has the ability of blocking other user if he wishes to. In order to do so, he has to add the user he wants to block into his blocking list, which consists of the users' names he has blocked. If one of the blocked user wants to call the user who has blocked him, he won't be able as he will see the other user to be unavailable.

As far as it concerns the list of the blocked user someone has, it will be also saved as a list at the Proxy, but instead of containing the users' names, it will contain their ID's.

Someone who will be blocked by another user, won't be able to see that the other user has blocked him.

The component which will handle this extension will be BlockCall.java which also will be interacting with other classes we 're going to modify.

2.2 Call Charging

When the call comes to an end, it means that either user A(who started the call) or user B(who accepts the call) hangs up. Then, the Proxy calculates how much our call cost and he adds this amount to the billing account to the user who started the call.

The component which will handle this extension will be ChargeCall.java which also will be interacting with other classes we 're going to modify.

2.3 Call Forwarding

In this case, user B, who is the one who's been called, has chosen to forward his calls to someone else and thus his phone won't be ringing at all. In the end, user A who started a call will end up talking to user X who is the user that user B has chosen to accept his calls. For this particular case, we have to handle 3 scenarios:

- User B forwards his calls to user X.
- User B forwards his calls but user X forwards the call back to him.

- User B forwards the call to user X who forwards it to other users but it ends up being forwarded back to him.

Pathological Scenarios

2.1 User B Not Connected

According to this scenario, Location Server isn't able to identify user's B info and ID. We will have to handle this scenario according to the RFC 3261 protocol.

2.2 User A SIP Crash

According to this scenario, user's A SIP Communicator crashes after he's called user . We will have to handle this scenario according to the RFC 3261 protocol.

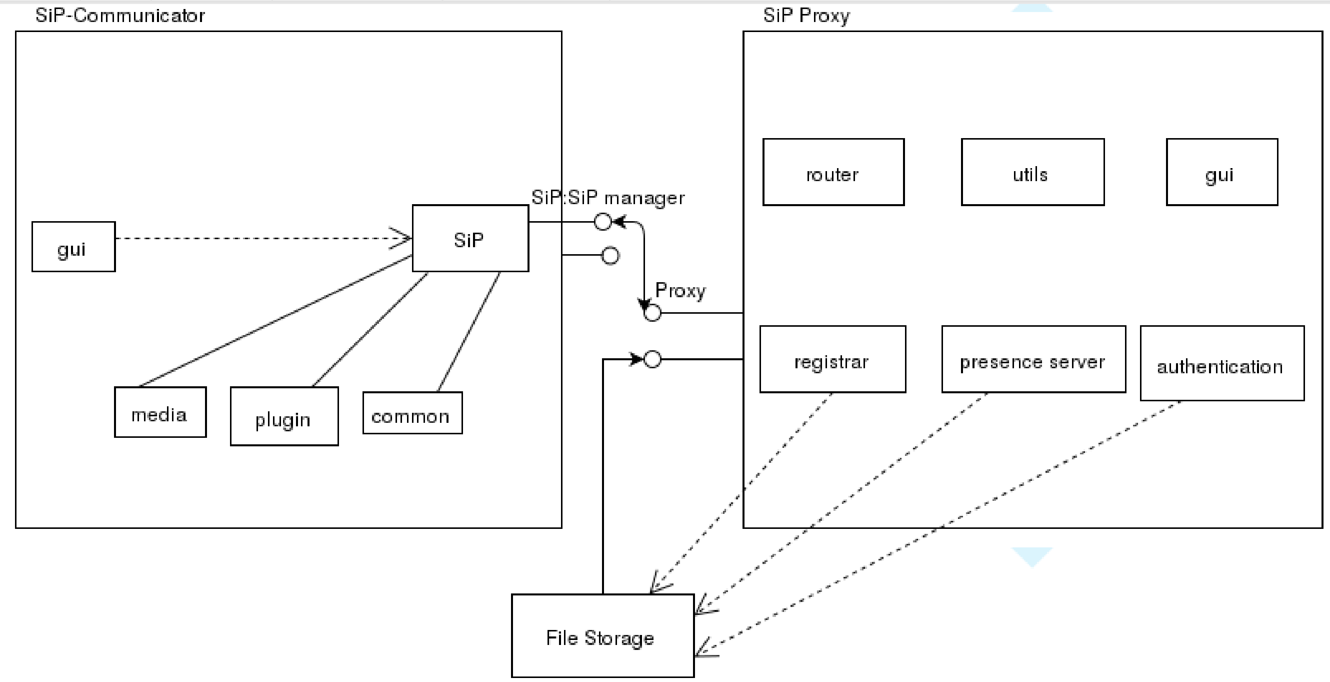
2.3 User B SIP Crash

According to this scenario, user's B SIP Communicator crashes after he's called user . We will have to handle this scenario according to the RFC 3261 protocol.

2.4 Proxy Server Crash

According to this scenario, Proxy Server crashes right after user A has made his registration. We will have to handle this scenario according to the RFC 3261 protocol.

3 Architecture



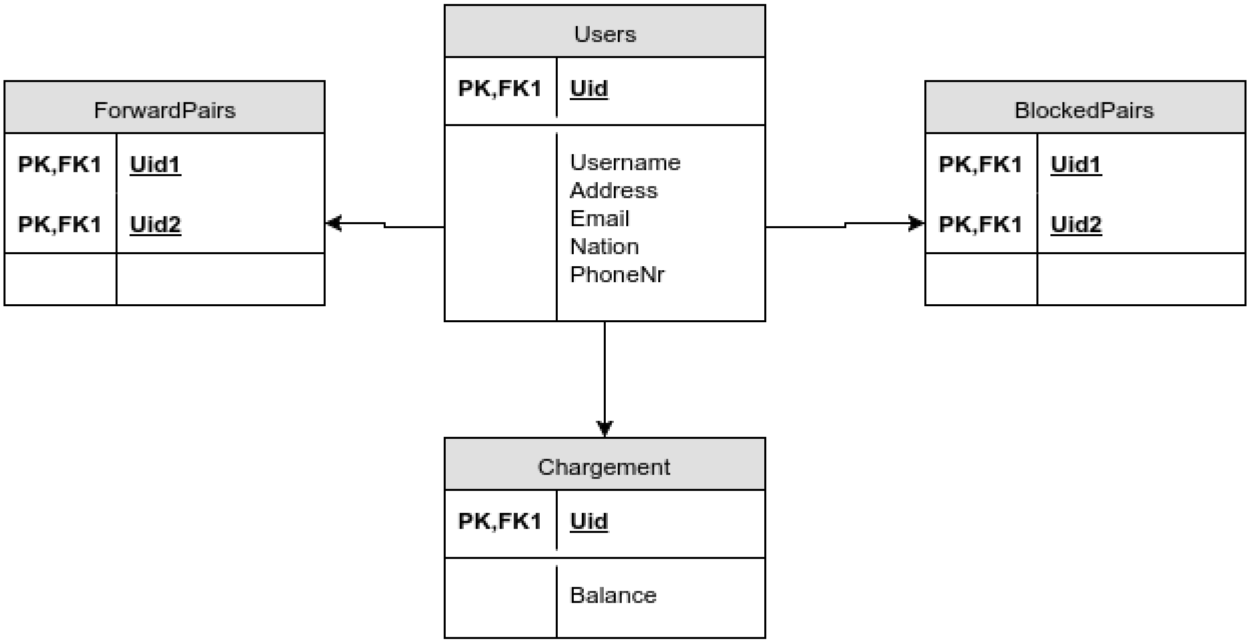
This is the diagram that shows us the connection between the clients and the server who connect through the SIP. First of all, the SiP Communicator is the client who uses the communication system in order to perform some actions such as calling, forward a call, block a user so he cannot call us and get the charge for our call.

As we can see above, each client, who is the SiP Communicator has an interface. Also, we can see that in order for the user to make use of the communication system some infrastructure components are provided, such as plugin, media and common.

On the other hand, we can see that SiP Proxy, also, has an interface. The “router” component is directing the requests either from the Proxy side or the Client's. Moreover, we can see some infrastructure components that need information which are provided from the File Storage.

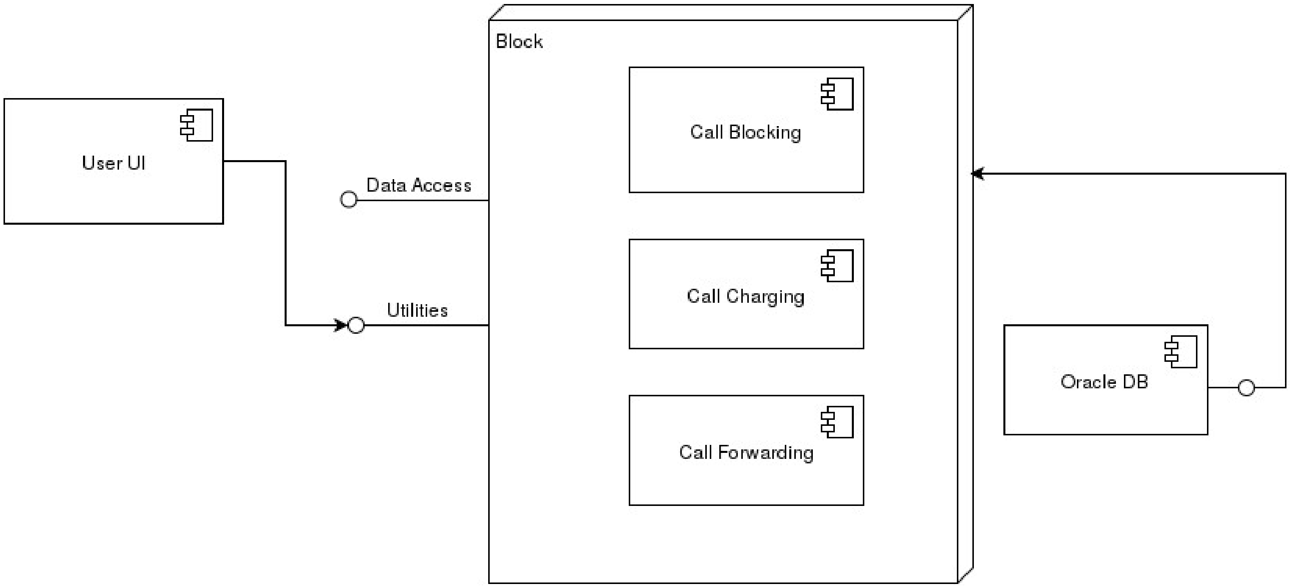
File Storage for our system serves like a database which keeps track of the information needed for the users which are authenticated and make use of the SiP Communication System. As we can see, file storage contains information about lots of users, information which are provided to the Proxy so that the Server comprehends in what condition its system is. As seen above, some of the Proxy infrastructural components, such as registrar, presence server and authentication receives information from the file storage.

Finally, the Communicators and the Proxy are connected through the SiP manager from the former side and through the Proxy from the latter side. This gives the opportunity for the user to communicate through this program and for the Proxy to keep track of the users on each moment.



Form of our DataBase

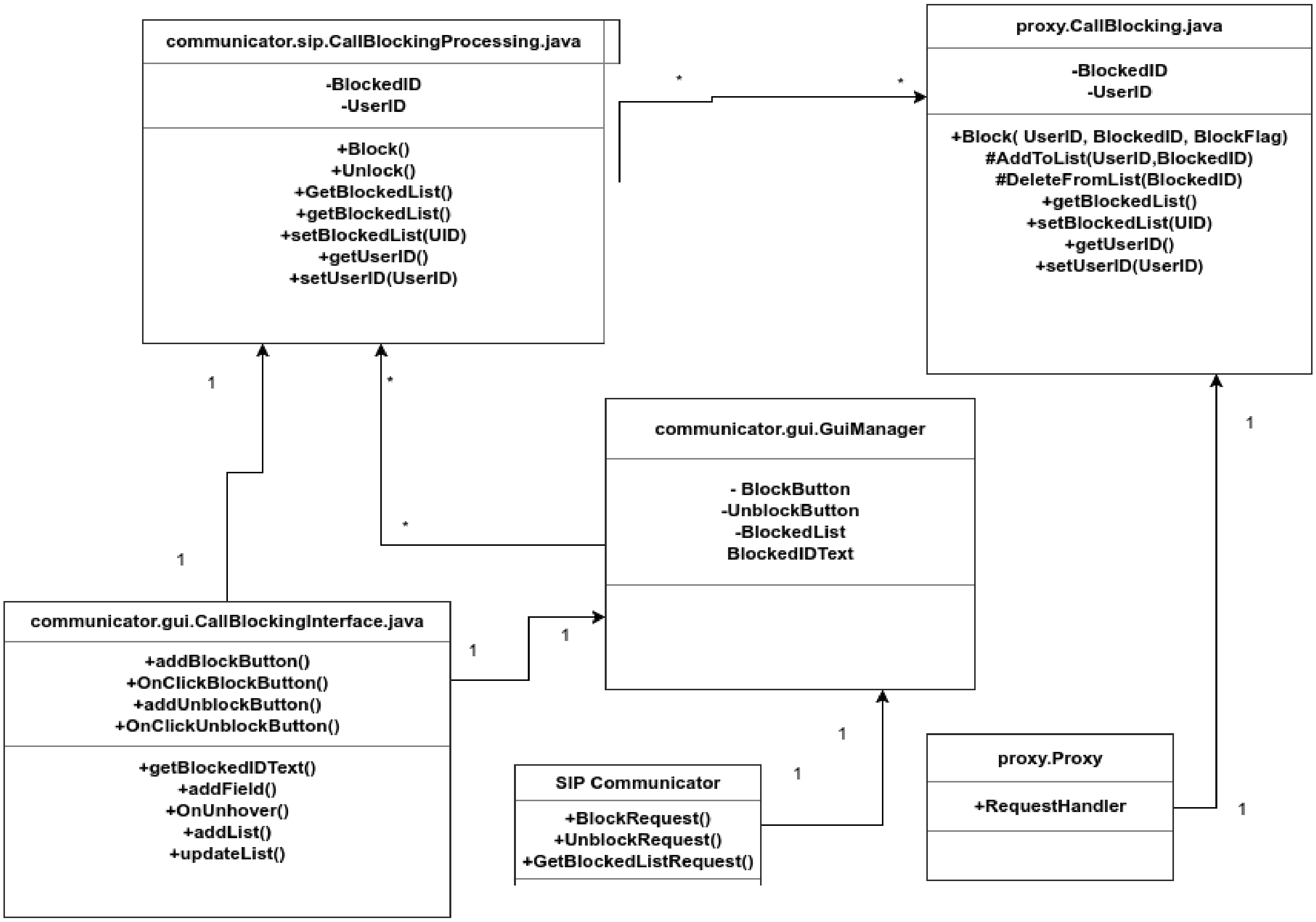
Component Diagram



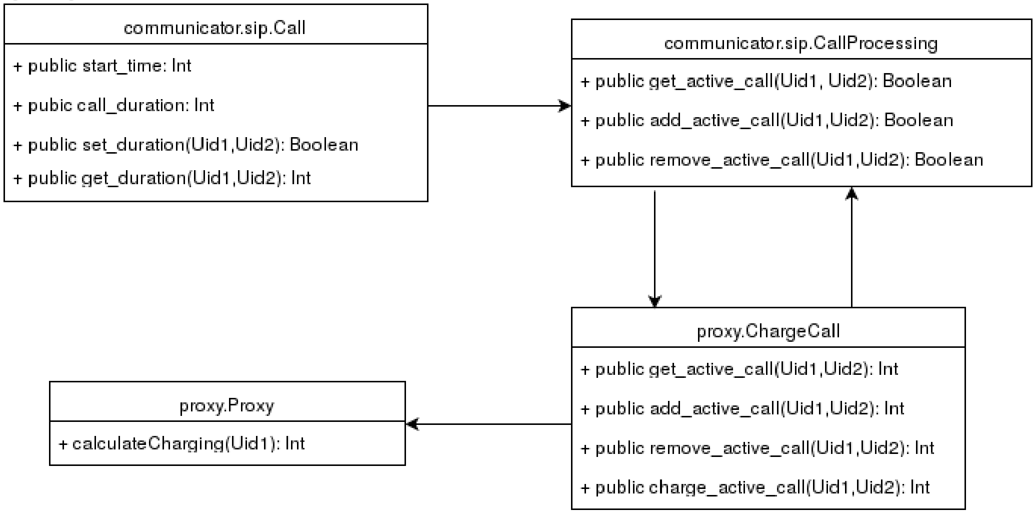
4 Detailed Class Diagrams

UML Class Diagrams

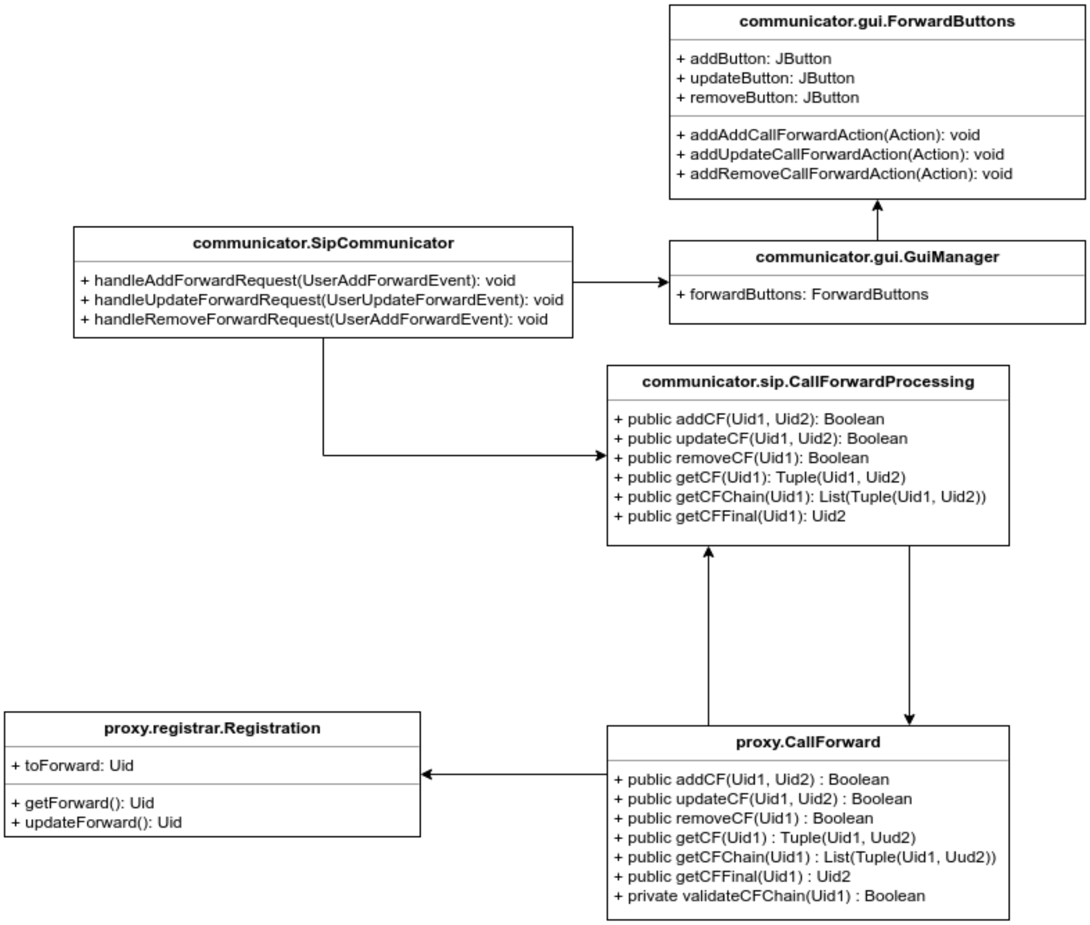
Call Blocking



Call Charging


communicator.sip.Call
+ public start_time: Int
+ pubic call_duration: Int
+ public set_duration(Uid1,Uid2): Boolean
+ public get_duration(Uid1,Uid2): Int
communicator.sip.CallProcessing
+ public get_active_call(Uid1, Uid2): Boolean
+ public add_active_call(Uid1,Uid2): Boolean
+ public remove_active_call(Uid1,Uid2): Boolean
proxy.ChargeCall
+ public get_active_call(Uid1,Uid2): Int
+ public add_active_call(Uid1,Uid2): Int
+ public remove_active_call(Uid1,Uid2): Int
+ public charge_active_call(Uid1,Uid2): Int
proxy.Proxy
+ calculateCharging(Uid1): Int

Forward Call


communicator.gui.ForwardButtons
+ addButton: JButton
+ updateButton: JButton
+ removeButton: JButton
+ addAddCallForwardAction(Action): void
+ addUpdateCallForwardAction(Action): void
+ addRemoveCallForwardAction(Action): void
communicator.SipCommunicator
+ handleAddForwardRequest(UserAddForwardEvent): void
+ handleUpdateForwardRequest(UserUpdateForwardEvent): void
+ handleRemoveForwardRequest(UserAddForwardEvent): void
communicator.gui.GuiManager
+ forwardButtons: ForwardButtons
communicator.sip.CallForwardProcessing
+ public addCF(Uid1, Uid2): Boolean
+ public updateCF(Uid1, Uid2): Boolean
+ public removeCF(Uid1): Boolean
+ public getCF(Uid1): Tuple(Uid1, Uid2)
+ public getCFChain(Uid1): List(Tuple(Uid1, Uid2))
+ public getCFFinal(Uid1): Uid2
proxy.registrar.Registration
+ toForward: Uid
+ getForward(): Uid
+ updateForward(): Uid
proxy.CallForward
+ public addCF(Uid1, Uid2) : Boolean
+ public updateCF(Uid1, Uid2) : Boolean
+ public removeCF(Uid1) : Boolean
+ public getCF(Uid1) : Tuple(Uid1, Uud2)
+ public getCFChain(Uid1) : List(Tuple(Uid1, Uud2))
+ public getCFFinal(Uid1) : Uid2
+ private validateCFChain(Uid1) : Boolean

Pseudocode

Call Blocking

We will briefly describe the changes we have to make in order to enable the Block Call Functionality. First of all, we will import a JList for displaying other Users that the logged user designates to block their calls, a Text Field in order he can properly define a designated user to block or to unblock and of course the respective JButtons for "Block" and "Unblock". Thus, we need to create the "comunicator.gui.CallBlock" and edit "communicator.gui.GuiManager". The default state for our system when the user selects CallBlocking in GUI will be to send a request to Proxy in order we can acquire the List of Blocked Users for the Logged User. After that the latter can insert the username of a user he wants to perform an action and can simply click "Block" or "Unblock" buttons. The Button will send the respective requests to Proxy in order to Add or Remove from the user's blocked list the selected user. For this purpose we will need to modify "communicator.SipCommunicator" and "proxy.Proxy" in order to be able to communicate by sending and listening to messages respectively. Both Proxy and Communicator will have their own "BlockCall" java class for manipulating the request. All the private fields of each one of these classes have their respective get() and set() methods. The Blocking mechanism is completely independent from the Calling one, by this we mean that if a blocked user tries to call the a user who blocked him, he would get a busy signal or something equivalent from the call mechanism. We will use the response processBusyHere() from the sip.communicator.Processing. Some of the code below could be changed throughout the process of implementation. This is only a hard estimation of what the code would be like.

proxy.CallBlocking.java

```
private UserID

private BlockedID


protected AddToList(UserID, BlockedID):

        q_user_reg = execute_query("select count(*) from users where Uid = UserID")
```

```
    if q_user_reg == 0 then Exception(Unregistered User)


    q_blocked_reg = execute_query("select count(*) from users where Uid = BlockedID")
    if q_blocked_reg == 0 then Exception(Designated Blocked User Unregistered)


    q_already_blocked = execute_query("select count(*) from BLOCKEDPAIRS where Uid1 = UserID and Uid2 = BlockedID")
    if q_already_blocked == 1 then Exception(Designated Blocked User Already Blocked)



    q = execute_query("insert into BLOCKEDPAIRS (Uid1 = UserID, Uid2 = BlockedID)")


    return



protected DeleteFromList(UserID, BlockedID):
    q = execute_query("delete from BLOCKEDPAIRS where Uid1 = UserID and Uid2 = BlockedID")


    return


public getBlockedList(UserID):
    return execute_query("select BlockedID from BLOCKEDPAIRS where Uid1 = UserID")


// Must be called everytime user wants to block or unblock someone
public Block(UserID, BlockedID, BlockFlag)
```

```
    setUserID(UserID)
    setBlockedID(BlockedID)
    if (BlockFlag):
        AddToList()
    else:
        DeleteFromList()
    return 0

public setUserID(Uid):
    UserID = Uid
    return

public setBlockedID(Uid):
    BlockedID = Uid
    return

public getUserID():
    return UserID

public getBlockedID():
    return BlockedID
```

proxy.Proxy.java

We have to add the following lines:

```
public RequestHandler:
        if (request{0} is Block):
                return Block(request{1}, request{2}, 1)
        if (request{0} is Unblock):
                return Block(request{1}, request{2}, 0)
        if (request{0} is GetBlockedList):
                return getBlockedList(request{1})
```

communicator.sip.CallBlockingProcessing.java

```
private UserID
private BlockedID

public setUserID(Uid):
        UserID = Uid
        return

public setBlockedID(Uid):
        BlockedID = Uid
        return

public getUserID():
        return UserID
```

```
public getBlockedID():
    return BlockedID

public Block():
    send_request({Block, UserID, BlockedID})
    return response

public Unblock():
    send_request({Unblock, UserID, BlockedID})
    return response

public GetBlockedList():
    send_request({GetBlockedList, UserID})
    return response
```

communicator.SipCommunicator.java

```
public BlockRequest():
    CallBlockingProcessing.Block()
    return response

public UnblockRequest():
    CallBlockingProcessing.Unblock()
    return response
```

```
public GetBlockedListRequest():
        CallBlockingProcessing.GetBlockedList()
        return response
```

communicator.gui.GuiManager.java

```
private JButtons BlockButton
private JButtons UnblockButton
private JList BlockedList
private JTextField BlockedIDText
```

communicator.gui.CallBlockingInterface.java

Block Button

```
private void addBlockButton
private void OnClickBlockButton
```

```
public addBlockButton():
        SetBlockButton
```

```
public OnClickBlockButton():
        communicator.SipCommunicator.BlockRequest(getUserID(), getBlockedID())
        communicator.SipCommunicator.GetBlockedListRequest(getUserID(), getBlockedID())
```

Unblock Button

```
private void addUnblockButton
private void OnClickUnblockButton

public addUnblockButton():
        SetUnblockButton

public OnClickUnblockButton():
        communicator.SipCommunicator.UnblockRequest(getUserID(), getBlockedID())
        updateList()
```

Text

```
private void addField
private void OnUnhover
private String BlockedIDText

public getBlockedIDText():
        return BlockedIDText.CanvasText

public addField():
        communicator.sip.CallBlockingProcessing.setUserID(Internal.Login.UserID)
        SetBlockedIDText
```

```
public OnUnhover():
        communicator.sip.CallBlockingProcessing.setBlockedID(getBlockedIDText())
        updateList()
```

List

```
private void addList
private void updateList
```

```
public addList():
        SetBlockedList
        updateList()
```

```
public updateList():
        communicator.SipCommunicator.GetBlockedListRequest()
```

Call Charging

Brief Description

We will briefly describe the changes we have to make in order to enable the Charge Call Functionality.First of all, when a call will be starting we will save the time when it begun and the duration so far (We are going to modify "communicator.sip.Call"). Then, the default state for our system when a call begins is to send the proxy a request. The proxy server should have a listener that will handle each request from a communicator program(We will edit the "proxy.Proxy"). The request handler will check each moment if the call is active - and therefore there is no need for us to make any calculations at that time - or if the call has come to an end and thus the proxy will calculate the cost of our call and add it to the billing account of the user who called (We will create the "communicator.sip.ChargeCall" and also we will modify "communicator.sip.CallProcessing"). All the above provide us an easy way of handling the call charging, but there is a catch. Our making changes in the communicator's classes can give the opportunity to potentially evil users to send to the proxy false call duration (=0) and thus is no such secure.

proxy.ChargeCall.java

We will create this class so that the proxy will calculate the cost of the call

```
public List active_tuples;

public get_active_call(Uid1, Uid2):
    active = query(call_tuple)
    if(active!=null)
        return 1
    else
        return -1

public add_active_call(Uid1, Uid2):
    tup = query(get_active_call exists_in active_tuples)
    if(tup)
        return -1
    else
        add to active_tuples
        return 1

public remove_active_call(Uid1, Uid2):
    tup = query(is the tuple Uid1,Uid2 an active call?)
    if(tup)
        remove from active_tuples
```

```
                return 1
        else
                return -1

public charge_active_call(Uid1, Uid2):
        dur = query(get duration for tuple(Uid1,Uid2)
        calc = query(calculate charging)
        return calc
```

communicator.sip.CallProcessing

We 're going to make changes in endCall class

```
public get_active_call(Uid1,Uid2):
        send_request({get, Uid1, Uid2})
        return

public add_active_call(Uid1, Uid2)
        send_request({add, Uid1, Uid2})
        return

public remove_active_call(Uid1, Uid2)
        send_request({remove, Uid1, Uid2})
        return
```

proxy.Proxy.java

We have to add the following lines:

```
public RequestHandler:
    if request is ChargeCall call ChargeCall.request
```

communicator.sip.Call

We 'll have to add some new fields and some new methods

```
public int start_time, duration;

public set_duration(Uid1,Uid2):
    call_dur = query(get duration for tuple Uid1,Uid2)

public get_duration(Uid1, Uid2):
    temp = query(get duration for tuple Uid1,Uid2)
    return temp
```

Call Forwarding

Brief Description

We will briefly describe the changes we have to make in order to enable the Call Forwarding Functionality. First of all we need to add the button in the user interface of the communicator(We will create "communicator.gui.ForwardButtons" and edit "communicator.gui.GuiManager"). After adding the buttons we will need to associate an action with each button(We will edit "communicator.SipCommunicator"). The actions will send requests to the proxy(We will create the "communicator.sip.CallForwadProcessing"). The proxy server should have a listener that will handle each request from a communicator program(We will edit the "proxy.Proxy"). The request handler will call the methods that will make and persist changes in the database(We will create the "proxy.CallForward"). We will also need to add a new field at each Registration model (and also each setter and getter) so that the call forward information can be visible at the proxy server interface(We will edit the "proxy.registrar.Registration").

proxy.registrar.Registration.java

We will add the following:

```
private Integer toForwardUid = -1;

public getForward():
    return toForwardUid

public setForward(Uid):
    toForwardUid = Uid
```

proxy.CallForward.java

We will create this class so everything will be added:

```
public addCF(Uid1, Uid2):
    q = query(does Uid1 have a Forward Uid?)
    if q == True then exception
    else
        updatedb(Uid1.forward = Uid2)
        if validateCFChain(Uid1) return true
        else
            removedb(Uid1.forward = Uid2)
            return false

public updateCF(Uid1, Uid2):
    prev = query(Uid1.forward)
    updatedb(Uid1.forward = Uid2)
```

```
        if validateCFChain(Uid1) return true
                else
                        updatedb(Uid1.forward = q)
                        return false

public removeCF(Uid1):
        return removedb(Uid1.forward)

public getCF(Uid1):
        q = query(Uid1.forward)
        return q

public getCFChain(Uid1):
        if not validateCFChain(Uid1)
                return False
        result = [Uid1]
        curr = Uid1
        while(1):
                curr = query(curr.forward)
                if(valid(curr))
                        result += curr
                else
                        return result

public getCFFinal(Uid1):
        if not validateCFChain(Uid1)
                return False
        curr = Uid1
        while(1):
                temp = query(curr.forward)
        if(valid(temp))
            curr = temp
        else
            return curr

// Must be called everytime a call is made and every time a forward chain is updated
private validateCFChain(Uid1):
    visited = Set(Uid1)
    curr = Uid1
    while(1):
        curr = query(curr.forward)
        if(not valid(curr)) return True
        if curr in visited:
            // Cycle
            return False
```

proxy.Proxy.java

we have to add the following lines:

```
public RequestHandler:
    if request is CallForward call CallForward.request
    if request is Call(Uid1, Uid2) call Call(Uid1, getCF(Uid2))
```

communicator.sip.CallForwardProcessing.java

```
public addCF(Uid1, Uid2):
    send_request({add, Uid1, Uid2})
    handle response
    return
```

```
public updateCF(Uid1, Uid2):
    send_request({update, Uid1, Uid2})
    handle response
    return

public removeCF(Uid1):
    send_request({remove, Uid1})
    handle response
    return

public getCF(Uid1):
    send_request({getCF, Uid1})
    handle response
    return

public getCFChain(Uid1):
    send_request({getCFChain, Uid1})
    handle response
    return

public getCFFinal(Uid1):
    send_request({getCFFinal, Uid1, Uid2})
    handle response
    return
```

communicator.SipCommunicator.java

```
public handleAddForwardRequest(Uid2):
    CallForwardProcessing.addCF(getCurrUser(Uid), Uid2)
    show response
    return

public handleUpdateForwardRequest(Uid2):
    CallForwardProcessing.updateCF(getCurrUser(Uid), Uid2)
    show response
    return

public handleRemoveForwardRequest(Uid2):
    CallForwardProcessing.RemoveCF(getCurrUser(Uid))
    show response
    return
```

communicator.gui.GuiManager.java

```
private JButtons CallForwardButtons
```

communicator.gui.ForwardButtons.java

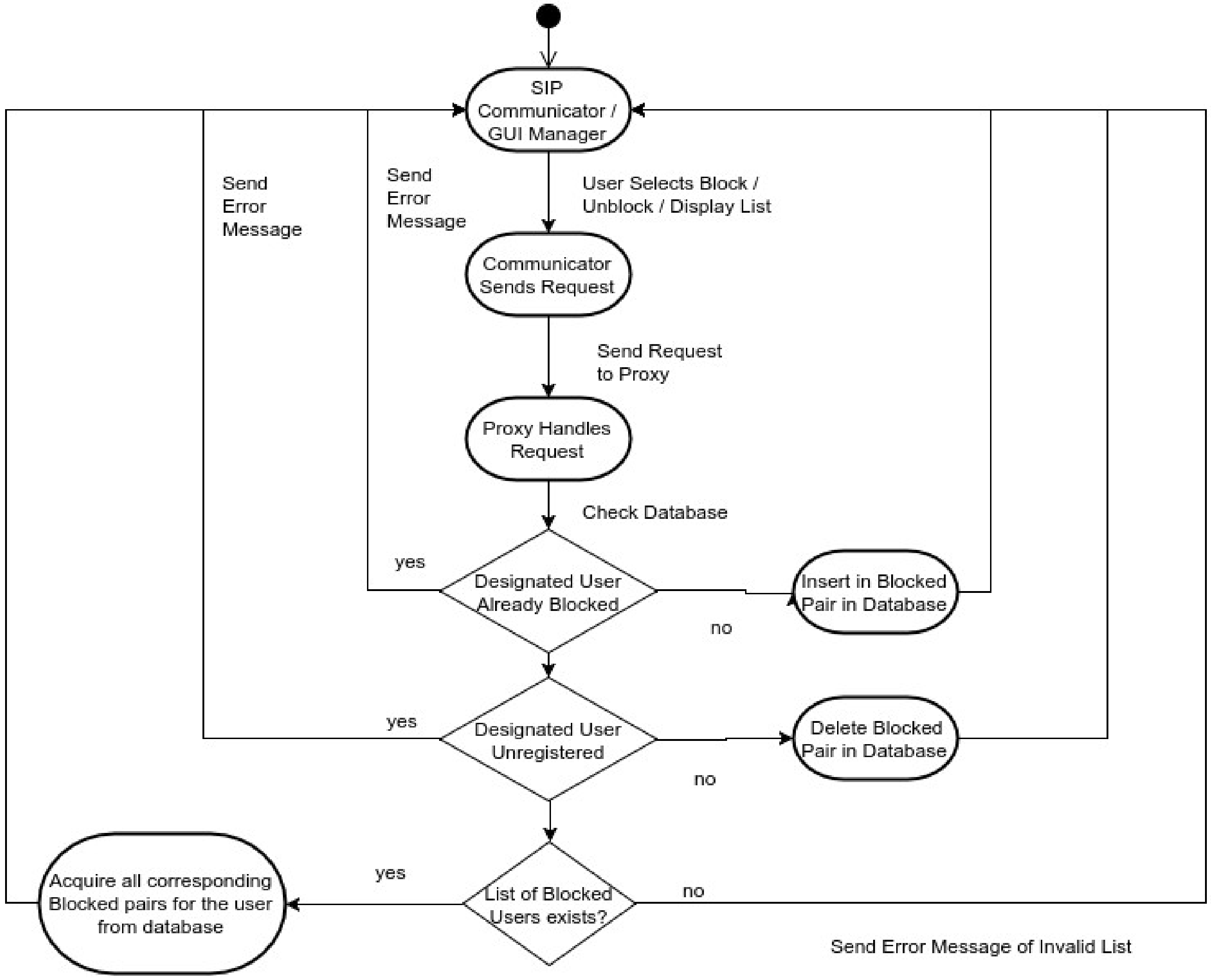
```
private Jbutton addForwardButton
private Jbutton updateForwardButton
private Jbutton removeForwardButton

public addAddCallForwardAction(Action):
    SetupButtonAction

public addUpdateCallForwardAction(Action):
    SetupButtonAction
```

public addRemoveCallForwardAction(Action):

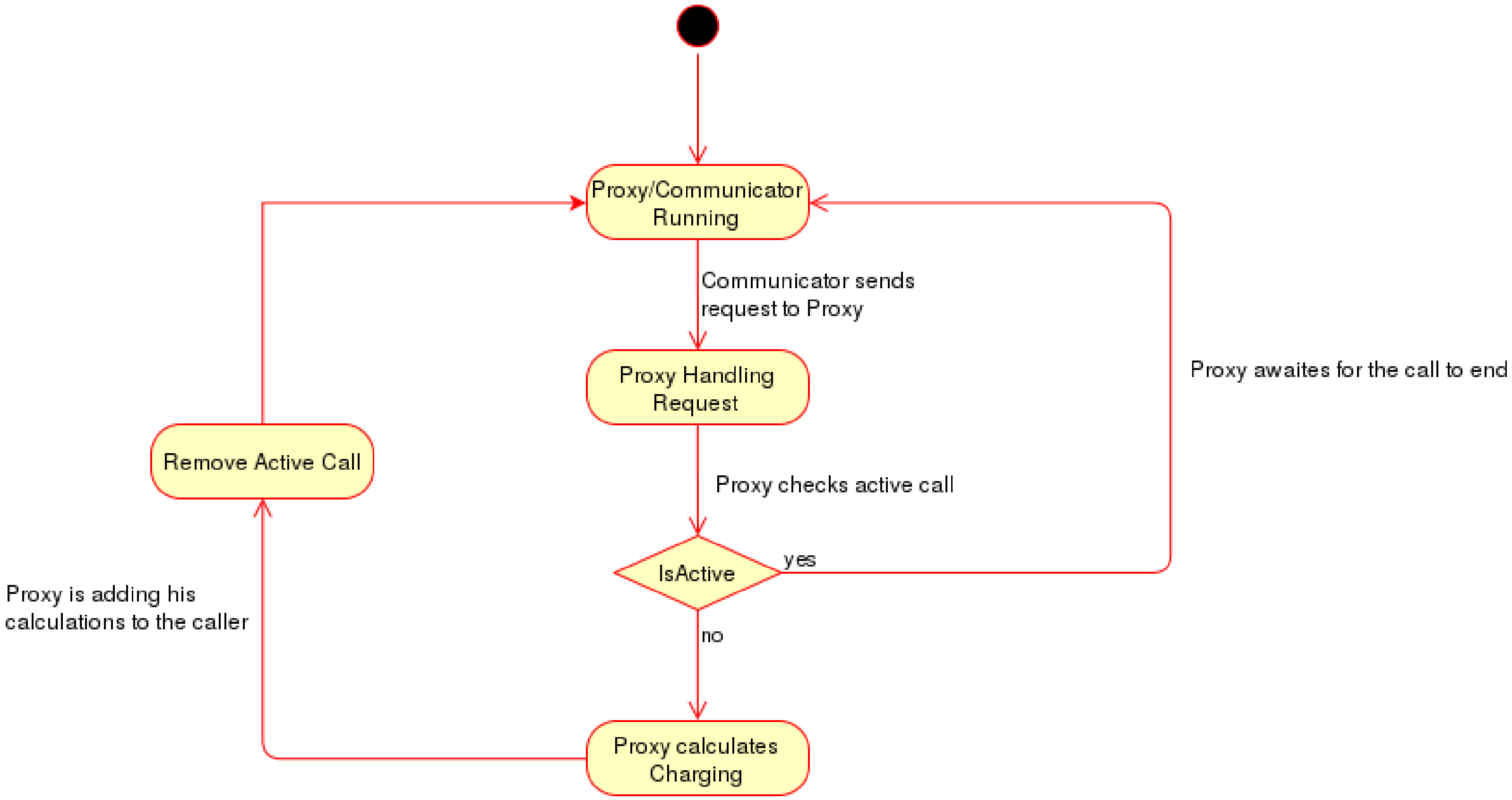
SetupButtonAction

5 State Diagrams

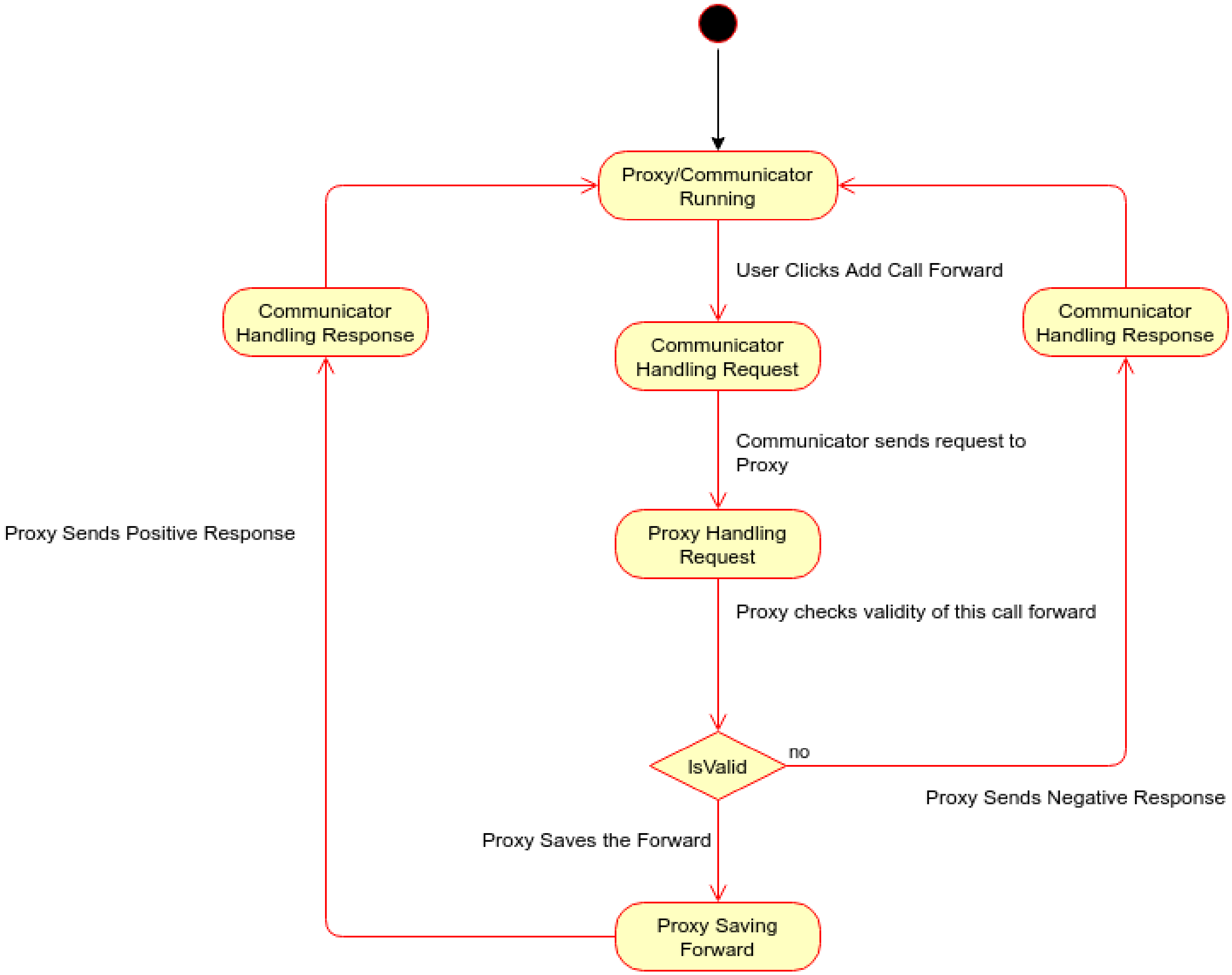
Call Blocking


SIP Communicator / GUI Manager
User Selects Block / Unblock / Display List
Communicator Sends Request
Send Request to Proxy
Proxy Handles Request
Check Database
Designated User Already Blocked
yes
Send Error Message
no
Insert in Blocked Pair in Database
Designated User Unregistered
yes
Send Error Message
no
Delete Blocked Pair in Database
List of Blocked Users exists?
yes
Acquire all corresponding Blocked pairs for the user from database
no
Send Error Message of Invalid List

Call Charging


Proxy/Communicator Running
Communicator sends request to Proxy
Proxy Handling Request
Proxy checks active call
IsActive
yes
no
Proxy awaites for the call to end
Proxy calculates Charging
Proxy is adding his calculations to the caller
Remove Active Call

Call Forwarding


Proxy/Communicator Running
User Clicks Add Call Forward
Communicator Handling Request
Communicator sends request to Proxy
Proxy Handling Request
Proxy checks validity of this call forward
IsValid
no
Proxy Sends Negative Response
Communicator Handling Response
Proxy Saves the Forward
Proxy Saving Forward
Proxy Sends Positive Response
Communicator Handling Response